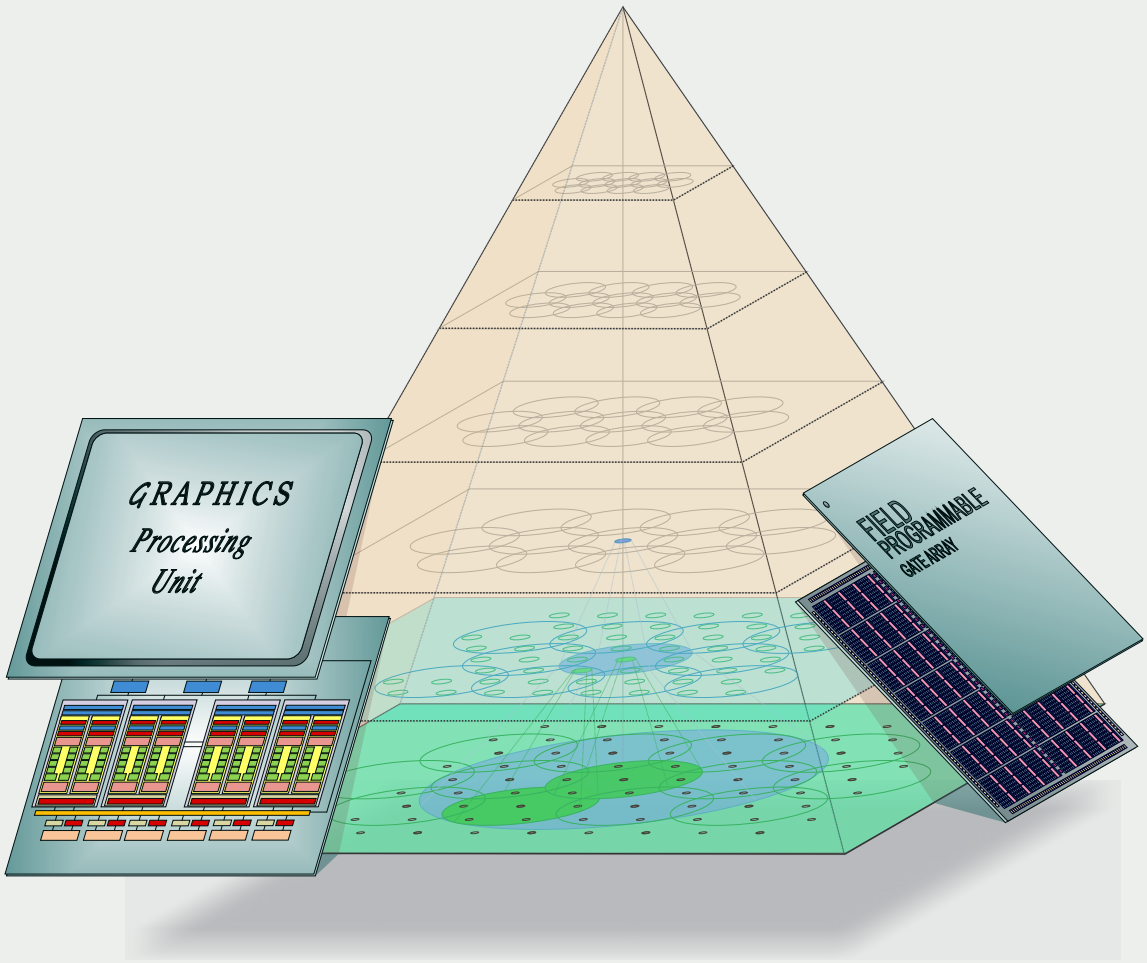


Parallelisation potential of image segmentation in hierarchical island structures on hardware-accelerated platforms in real-time applications

Sergey Suslov

JÜLICH
FORSCHUNGSZENTRUM

Member of the Helmholtz Association

Schriften des Forschungszentrums Jülich
Reihe Information / Information Band / Volume 30

Forschungszentrum Jülich GmbH
Zentralinstitut für Engineering, Elektronik und Analytik (ZEA)
Systeme der Elektronik (ZEA-2)

Parallelisation potential of image segmentation in hierarchical island structures on hardware-accelerated platforms in real-time applications

Sergey Suslov

Schriften des Forschungszentrums Jülich
Reihe Information / Information Band / Volume 30

ISSN 1866-1777 ISBN 978-3-89336-914-0

Bibliographic information published by the Deutsche Nationalbibliothek.
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Publisher and Distributor:	Forschungszentrum Jülich GmbH Zentralbibliothek 52425 Jülich Tel: +49 2461 61-5368 Fax: +49 2461 61-6103 Email: zb-publikation@fz-juelich.de www.fz-juelich.de/zb
Cover Design:	Grafische Medien, Forschungszentrum Jülich GmbH
Printer:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2013

Schriften des Forschungszentrums Jülich
Reihe Information / Information, Band / Volume 30

D 180 (Diss., Mannheim, Univ., 2012)

ISSN 1866-1777
ISBN 978-3-89336-914-0

The complete volume is freely available on the Internet on the Jülicher Open Access Server (JUWEL) at www.fz-juelich.de/zb/juwel

Abstract

The aspects of application development on parallel computing platforms are highly acute today. With the intensive increase in the integration level of silicon devices enabling parallel computing technologies on a single chip the power of supercomputers became available for computing systems of the compact class. Thus, more sophisticated and calculation intensive computing methods have become broadly available for the scientific society and industry. However, the inevitable drawback for this computing boost is often a cardinal change in the application design and development approach.

The present thesis addresses two types of compact High Performance Computing (HPC) platforms found to be most successful nowadays: Field Programmabe Gate Array (FPGA) based expansion cards and Graphics Processing Unit (GPU) coprocessing boards. To define their place in the parallel platform domain a profound overview of modern digital single-chip systems is presented. The characteristic features of FPGA and GPU architectures are discussed to identify the major aspects of the application design for these platforms.

A special attention is paid to the methodological aspects of both application design concepts. A thorough literature study has been carried out to systematise the complete application development cycle starting with the comprehensive system-level analysis and finishing with the workflow of the implementation on the target compute architectures. Several new ideas and interpretations have been introduced in this work.

The application in focus is a fast automated image segmentation method based on hierarchical island structures, the so called GSC (Grey Value Structure Code) worked out by Vogelbruch [2]. This complex segmentation method is feasible for different application areas and provides high-quality segmentation results by the combination of both local precision and global connectivity.

Steered by the proposed methodological guideline a comprehensive analysis of the parallelisation potential of the applied method is carried out in this work. Relying on many statistical measurements and results of versatile system models the GSC algorithm is specially reelaborated for the implementation on the two massive parallel computation platforms to achieve a high performance of the segmentation needed for real-time application set-ups. A special attention has been paid to the question of an effective computation organisation for the target platforms. The details of the implementations on both platforms are discussed thoroughly. Finally, the two implementations are compared to highlight their relative merits and downsides for this complex and computation intensive application.

The results of the work show that even having a considerably longer development cycle the FPGA-based solution on the Xilinx Virtex II Pro architecture can compete with the implementation on the specialised nVidia GT200 GPGPU card of the next technological generation and can even notably outperform it for image resolutions below 1024^2, while the nVidia G80 GPU of the same technological evolution cycle cannot be considered as a competitor. Compared to the processing speed on a single CPU (Opteron 2.6 GHz) the FPGA accelerates the application in dependence on the image resolution by a factor of about 23, while the GPU outperforms with factors of 13 to 20.

Zusammenfassung

Der Forschungsbereich Applikationsentwicklung für parallele Rechnerplattformen ist hoch aktuell. Mit dem rasanten Anstieg der Integrationsdichte von Halbleiterbauelementen, welche parallele Rechnertechnologien auf einem einzigen Baustein ermöglichte, wurde die Rechenleistung von Hochleistungsrechnern auch für Rechnersysteme der Kompaktklasse verfügbar. Dies machte anspruchsvollere und rechenintensivere Methoden für Wissenschaft und Industrie breiter einsetzbar. Allerdings setzten diese Rechnertechnologien in der Regel eine grundsätzliche Veränderung des Applikationsentwurfs und des Entwicklungsansatzes voraus.

Die vorliegende Arbeit befasst sich mit den beiden zur Zeit erfolgreichsten kompakten HPC-Plattformen (HPC ~ High Performance Computing): FPGA-Erweiterungskarten (FPGA ~ Field Programmable Gate Array) und graphische Co-Prozessorkarten (GPU). Zu deren Einordnung innerhalb der parallelen Plattformen wird in dieser Arbeit ein umfassender Überblick über moderne digitale Einzelchipsysteme gegeben. Die charakteristischen Eigenschaften von FPGA- und GPU-Architekturen werden diskutiert, um die wichtigsten Aspekte des Applikationsentwurfs für diese Plattformen zu identifizieren.

Darüber hinaus wird besondere Aufmerksamkeit den methodologischen Aspekten beider Applikationsentwurfskonzepte gewidmet. Basierend auf einer umfassenden Literaturrecherche, wurde hierzu der komplette Applikationsentwicklungszyklus systematisiert, von der umfassenden Analyse auf Systemebene bis hin zum Implementierungs-Workflow für die Zielrechenarchitektur.

Eine schnelle und automatisierte Bildsegmentierungsmethode, der sogenannte GSC (Grey Value Structure Code), welcher auf hierarchischen Inselstrukturen basiert und u.a. von Vogelbruch [2] erarbeitet wurde, steht als Anwendung im Fokus. Diese komplexe Segmentierungsmethode ist für verschiedene Anwendungsgebiete einsetzbar und liefert durch die Kombination von lokaler Genauigkeit und globaler Konnektivität qualitativ hochwertige Segmentierungsergebnisse.

Entsprechend der vorgeschlagenen methodologischen Vorgehensweise wird in dieser Arbeit das Parallelisierungspotential des GSC umfassend analysiert. Basierend auf den Ergebnissen vielfältiger Systemmodelle und der Auswertung von statistischen Messungen, wurde der GSC Algorithmus für die Implementierung auf den beiden Plattformen in Hinblick auf eine hohe Rechengeschwindigkeit, wie sie in Echtzeitapplikationen benötigt wird, neu adaptiert. Hierbei wurde besonderer Wert auf die effektive Organisation der Rechenschritte gelegt. Die beiden Implementierungen werden eingehend beschrieben. Abschließend werden die jeweiligen Vor- und Nachteile der beiden Implementierungen für diesen komplexen und rechenintensiven Anwendungsfall vergleichend herausgearbeitet.

Die Ergebnisse der Arbeit zeigen, dass, ungeachtet des wesentlich längeren Entwicklungszyklus, die FPGA basierte Lösung auf der Xilinx Virtex II Pro Architektur mit der Implementierung auf der spezialisierten nVidia GT200 GPGPU Karte konkurrieren und diese sogar für Bildauflösungen unter 1024^2 deutlich übertreffen kann. Die nVidia G80 GPU des gleichen technologischen Evolutionszyklus kann dagegen nicht als Konkurrent angesehen werden. Verglichen mit der Rechengeschwindigkeit auf einer Einzel-CPU (Opteron 2,6 GHz), beschleunigt die FPGA-Implementierung die Anwendung in Abhängigkeit von der Bildauflösung um einen Faktor von ungefähr 23, während die GPU diesen mit Faktoren von 13 bis 20 übertrifft.

Посвящается
Светлане Сергеевне Балакиревой (Богдановой)
и Николаю Ивановичу Балакиреву

Dedicated to
Svetlana Sergeevna Balakireva (Bogdanova)
and Nikolaj Ivanovich Balakirev

Table of Contents

List of Figures

List of Tables

1 Introduction

With the rapid development of silicon IC technology over the last two decades more and more sophisticated and calculation intensive computing methods have become broadly available for the scientific society and industry. Around the turn of the century an intensive increase of the computation power of digital systems resulting from the clock frequency increase have come to saturation however [1]. The trend for extensive development of computation systems became dominating. This alternative trend is the computation parallelism.

Parallel computations look natural for processing data of many systems. Elements of such systems exist and interact “in parallel” defining the qualities of the systems. However, not all computation methods are suitable for parallelisation. The main reason for that is relatively simple – the majority of existent methods are initially developed for sequential technologies. Moreover, even taking into account that a human naturally perceives the parallelism of concurrent processes, this parallelism is not always consciously handled by the person. The human mindset is oriented to work sequentially, which hampers the comprehension of the parallel aspects of computations. That is why parallel application design imposes special demands on development means and approaches to allow efficient design and maintenance of parallel computations.

This means that new hardware capacities alone are not enough for a cardinal breakthrough in computations. The exploitation of the computing parallelism faces at least two significant challenges: the problem of revealing parallelism in computation methods and the need for adequate development means for parallel application design. This work addresses both the analytical and the instrumental aspects of parallel computations for two representatives of parallel computer architectures.

One of the application fields that have a good potential for computation parallelisation is image processing. Due to the spatial distribution of the image data, elements or regions of an image can often be processed concurrently on different computing elements. The present work focuses on the parallelisation potential of one of the methods for image segmentation. This method is based on hierarchical island structures and referred to as the Grey-value Structure Code (GSC) segmentation method [2]. The method has been shown to be effective for different application areas to provide remarkable segmentation quality [3].

Up to now the method has been realised as a computer program for standard x86 CPUs and has been parallelised for symmetric multiprocessing (SMP) using OpenMP. The parallelisation of the method is done by dividing initial data sets into parts, processing each part as an individual image, and subsequently joining the results. However, due to its regular hierarchical structure the GSC has potentials for massive parallelisation enabled by finer granularity available in the GSC. These potentials are revealed in the analytical part of this work.

Due to its high segmentation quality and universality the method can be attractive for automated visualisation and image processing systems e.g. in medical applications, robot vision, industrial defect and quality control, transport automation, etc. In these application fields processing time often plays a key role. Thus, boosting the method by parallel computation justifies extra efforts for scrupulous parallelisation analysis and parallel implementation studies.

The work targets primarily at the performance efficiency of the parallel implementation rather than the qualitative aspects of the method in general, which has been addressed in the previous works [3]. Although the GSC method may vary in realisation of its algorithmic parts, these particular implementations are not always suitable for parallel solutions. Considering the fact that these algorithmic variants do have a minor impact on the segmentation quality the preferences are given to those algorithmic solutions best suited for the target platforms.

Apart from the particular application the work focuses on both aspects of parallel application design. The first is the parallelisation analysis of the computations carried out in compliance with the methodology presented in this work. The second aspect covers the peculiarities of parallel application development on two different target massively parallel computation platforms. These peculiarities are emphasised by comparative analysis of two implementations of the GSC on these platforms. The comparison considers the design process specifics and the relative implementation efficiency of the parallel GSC. By comparing the implementations the work aims to broaden the view on two specific parallel computation technologies in general.

The computation technologies under consideration are the configurable logic of Field-Programmable Gate Arrays (FPGA) and the stream processing architecture of Graphical Processing Units (GPU). Both parallel processing solutions became popular in the scientific society and industry for their low initial set-up investments, low development costs and relatively high computation power. Both technologies are often considered as rivals in the low-cost solution market for high performance computing (HPC).

The choice of these two platforms is not accidental. The method's specifics require the systems that are equipped with shared global memory and can benefit from high level of computation parallelism and relatively wide range of computation granularity. Therefore, the systems like Massive Parallel Processing (MPP) must be too ponderous due to their distributed memory organisation and the message passing mechanism; the vector systems are too awkward to maintain control flow flexibility in parallel processes; and the multiprocessor systems with symmetric memory access (SMP) may

suffer from poor scalability limiting the massiveness of parallel processing. That is why only two single chip processor[1] solutions have been selected to satisfy the GSC prerequisites. They are the FPGA and the GPU board systems. For the studies the parallel GSC was implemented using a specially designed FPGA (Xilinx Virtex-II) board for memory intensive high performance computing and an nVidia's Compute Unified Device Architecture compatible GPU card (Tesla C1060).

Although both solutions share the same customer niche having comparable hardware prices, power consumption, and computing power parameters, the design process of a target application on these two platforms differs fundamentally. The substance of this difference lies in the way of representing computation algorithms – a configurable logic structure on the one hand or a sequence of operations realised in a particular instruction set on the other hand. The different approaches to algorithm implementation result in characteristic benefits and downsides for each technology. The relative merits and limitations of these two solutions being revealed during the GSC application design are described in this work.

The applications and the projects the GSC is used in (e.g. industrial inspection, medical analysis) demand significantly higher performance compared to what a CPU solution can offer. A cardinal breakthrough in the applications' performance can be very welcomed by industry and medicine. Hence, the topicality of the work for the field of image processing arises from the fact that the GSC has ever been implemented neither in a form of application specific hardware due to the complexity of the method nor using the general purpose computing on GPU (GPGPU) technology due to its relative novelty. These implementations can significantly improve performance of this highly sophisticated segmentation method, while the results of the profound parallelisation analysis of the method presented in this work can be used in future implementations of the parallel GSC on the prospective parallel platforms.

Meanwhile, the two technologies causes a strong curiosity among the experts with regard to the competition between FPGAs and GPUs in the area of compact HPC. Hence, another topicality of the work lies in the instrumental and the technological aspects of developing parallel computing applications for the two technologies.

High-end FPGA boards have been deservedly considered as the leaders of compact supercomputing since many years. However, due to significantly long development cycles in application design these platforms better serve the class of high performance embedded computing. The long development cycle is a contradictory problem for general purpose computing. This problem has been tried to be solved by applying well-developed functional libraries or by resorting to high-level synthesis. Notwithstanding the prospects of the ideas in general, these approaches are not very promising in this particular application area due to the high requirements to the qualification of an end user, who needs to adjust and tune generic functional blocks during application assembling anyway. That is why, when general purpose computing on GPUs became broadly available it was considered as universal remedy for HPC at hand.

[1] The term *processor* is used in a broad sense of the word here.

At the same time, the streaming architecture of GPUs has its own serious limitations, which are revealed in this thesis. These limitations notably restrict the scope of application types that can be efficiently implemented on GPUs and do not generally allow GPUs to substitute FPGAs in the compact supercomputing field.

Nonetheless, the continuous increase in the integration level of modern processors will most probably add a new flavour to GPU computing making its flexible mature programming technology applicable for a broad range of tasks. The new stable tendency towards hybridisation of processor architectures (i.e. combination of several CPU and GPU cores in one silicon device) promises to broaden the GPU's computing capabilities significantly, making the GPU a streaming coprocessing unit in a hybrid multicore system. Hence, in general the stream processing technology has a chance to expand even into the embedded application area of FPGAs, forcing both technologies to become competitors for a huge number of applications. For this reason the comparison of these two technologies is an exciting topic for the computing society.

Structurally the work consists of eight chapters. Chapter 2 shows the research background for the work to draw a general picture of the application field and to introduce the reader to the specific application and the instrumental part of the work. It starts with a brief description of segmentation methods grouping them into several general classes of segmentation techniques. Beside the background of image segmentation, the chapter gives an introduction to the essence of the GSC segmentation approach describing the basic concepts of the method. The section describing the instrumental part discusses tendencies in modern digital systems and gives an overview of the major parallel computation technologies. The conception of algorithm interpretation in modern computing is regarded in this chapter, which brings the previously mentioned algorithmic and instrumental parts together. Finally, the two implementation platforms considered in the work are described in detail.

With the growth of system complexity the design methodology becomes of special importance. This is explained by the fact that at a certain level of complexity a design carried out beyond a methodological framework not only risks losing the result quality, it becomes simply not handleable. The methodological constituent gets especially important for parallel applications as they require qualitatively new approaches to development. This is why an undivided attention is focused on the design methodologies in Chapter 3. Although the methodology aspects of the design process do not contribute any quantitative results to the studies in the application area, it gives a clear picture of development complexity for the two target platforms and helps to compare the implementation approaches qualitatively.

Chapter 3 constitutes a form of survey. The material in this chapter generalises, structures, and refines a broad range of works related to hardware and programming design. Many ideas in this chapter are introduced for the first time or were the object of profound reconsideration. In this chapter the top-down approach is advocated as the most appropriate technique for development of systems of middle and high complexities. The design flow is shown as gradual refinement of the system specifications. A special attention is paid to the verification. The peculiarities of the GPU

architecture are described separately to highlight targets for optimisation of the GSC implementation. The methodology given in this chapter serves as a guideline for the GSC parallelisation analysis and implementation on the target platforms described in the next two chapters.

The results of the GSC analysis and details of its realisation on a precise FPGA platform and a GPU card are represented in Chapter 4 and Chapter 5, respectively. Chapter 4 is laid out in compliance with the complete chain of top-down design and depicts gradual specification of the application at different levels of abstraction. It starts at the functional analysis of the method and concludes with the results of its implementation on the target FPGA platform. At the same time, Chapter 5 focuses solely on the latter steps of a top-down design process of SW development – the partitioning of an application to a target computing architecture. The chapter focuses on the peculiarities of the GSC implementation on the given GPU platform, which determine the implementation process. Although the system level studies of the GSC are represented in the hardware-related chapter, the technology independent results of analysis at the higher levels of abstraction given in Chapter 4 are useful for evaluating the method's potentials for implementation on the GPU platform.

In Chapter 6 the two parallel solutions are compared each with the other using the findings acquired through the GSC implementation. A generic comparison of FPGAs and GPUs is a difficult task as the concepts and the paradigms, including design technologies, behind these device classes are completely different. Moreover, these classes of devices generally come from different application areas, which hinders finding the comparison criteria. That is why it is important to identify a set of metrics that are relevant to the specific application field. This chapter collects a number of criteria that can characterise this type of systems with regard to the challenges of compact high performance computing. The GSC implementations were compared by computing performance, scalability, implementation efforts, and functional modification capability.

Chapter 7 summarises the studies. It highlights the major points of the work, outlines the specifics of the GSC under the aspect of parallel computing, discusses the peculiarities of the implementations of the GSC on the particular computation platforms, and compares the relative merits and downsides of these technologies.

Chapter 8 presents a generalisation of the experience gained from the GSC implementation, and suggests some recommendations for implementation of other algorithms on GPU and FPGA platforms. This chapter also offer an analysis of the trends in FPGA and GPU technologies and gives an outlook on what can be expected from the prospective chips with regard to performance of the parallel GSC. Furthermore, it discusses briefly which potentially interesting questions were not covered in this work and might be subject for future studies.

2 Fundamentals and research background

2.1 Segmentation in image processing

2.1.1 Overview on segmentation methods

Segmentation takes an important place in the image processing pipeline, being a bridge between the preprocessing of the raw image (acquisition, transforming and positioning, quality improvement, filtering) and the intellectual computer analysis (object recognition and classification, scene measurements and analysis). The aim of the segmentation is to partition the image into subsets/regions, each pixel in which is similar to the rest of the pixels in a region by a certain criteria. Segmentation methods are various in applicability and usage. They can be widely applicable (relatively universal) or sufficient (relevant) only to images of a specific nature, automatic or supervised by an operator. The underlying mathematical and physical principles that segmentation methods are based on are numerous [4-10], meanwhile they mainly rely on two basic characteristics of a spatial signal. These are discontinuity (detection of segment boundaries) and homogeneity (similarity of pixels within a segment). The most widely recognised and settled classes of algorithms are briefly described here.

Threshold-based or *histogram-based techniques* [11-12] sorts out the elements of images into some segment classes (e.g. object and background) with respect to appropriate thresholds, which can be global, local or adaptive. These techniques have the advantage of being relatively simple and fast, but are rather sensitive to the image quality (e.g. illumination spread, contrast).

Edge-based methods [13-14] exploit discontinuities of the signal in the images treating them as segment boundaries. Edge-based techniques primarily use differential operators for detecting those boundaries. These methods can be subdivided in sequential edge finding methods [15-16] and parallel methods [17-19]. The challenge often met in those methods is that the edges distinguished in images are not necessarily joint, thus special algorithms are needed to form solid borders of segments [20-22]. The major problems of these methods are their noise sensitivity and their result dependence on the mathematical approximation methods in multidimensional derivatives' computation. A particular case of edge-based methods is the watershed technique [23-25]. This

method regards the image as a topological relief (with the third dimension being pixel features), which is flooded with liquid, thus forming basins, i.e. image segments, enclosed with ridges of the relief. Normally, morphological operators are used for the detection of basin boundaries. One great disadvantage of the watershed technique is its oversegmentation tendency which may be reduced by several approaches [26].

Region-based methods consist in joining/disjoining image elements to/from newly forming segments. In particular graph-based techniques can be used for segments forming, which represent the image as a graph with image elements being nodes, while edges reflect the similarity of nodes as a distance between the regions in some metric space. The strategies for segments' forming can be region merging [27-29], region splitting, splitting and merging [30-31], or pyramid linking [32], [33]. The potential problem with region-based approaches is selecting appropriate homogeneity criteria and in some cases the selection of points for the region inception.

Deformable models [9, 34-41] exploit the idea of the physical characteristics of materials. The underlying idea of the methods is in modelling the behaviour of some imaginary contour having some physical properties (elasticity, stiffness, torsion ability) under internal and external forces. These forces are constantly applied to the contour until they deform the contour so that it fills some image segment. The evolution equation posed as an energy minimisation problem is solved by methods of finite difference and finite elements [42]. Another method to realise a deformable model is to use curve evolution theory [34] and level-set methods [43].

Classification methods [44] can be used to perform an implicit segmentation of a data set. For this, typically several characteristic features need to be extracted from the data set. The assignment of pixels to segments normally is performed in a multidimensional feature space by identifying clusters of similar objects and using a classifier for the affiliation decision. These classifiers comprise e.g. geometric classifier, minimum distance classifier, k-nearest-neighbour classifier, decision tree classifier, and statistical classifiers. Also neural networks [45] can be used to implement a classifier. By the use of a classification method for segmentation the contiguity conditions of segments can get lost. To obtain satisfying results, additional spatial merging algorithms need to be used or enough characteristic features should be extractable from the data set.

In addition to the classical method groups some *hybrid methods* have been proposed to aggregate the advantages of those methods. Most of them combine edge- and region-based methods [46-47], [48] e.g. by applying a watershed algorithm on the image gradient magnitude and complete the segmentation process by a hierarchical region merging process [49].

The GSC method [2] focused in this work is based on the CSC (Colour Structure Code) [50-51] and realises an isotropic inspection of the 2D data set using hierarchical island structures for the image representation. It belongs to the category of automated region-based techniques operating with monochromatic features. The advantage of this method is that it merges local precision and global view of an image by reevaluating homogeneity decisions taken on a lower hierarchical level on the

basis of the global view. This is realised by subsequently splitting originally merged regions down to the lowest hierarchical layer. As it is shown in [3] the GSC can be used for different application areas providing promising segmentation results and remarkable a segmentation quality.

2.1.2 Grey Scale Code method

The GSC is based on the mathematical abstraction of a hierarchical overlapping island structure [50-51]. The 2D-GSC operates on a hexagonal Bravais lattice, which is covered with central symmetric islands of seven neighbouring nodes with one node in the centre. The islands are placed on the lattice so that every island overlaps with each of its six neighbours in a single lattice node (referred to as *an overlapping node* or *point*). Their centres form the hexagonal Bravais lattice of the next upper hierarchical level, with the correspondent islands being nodes of the hexagonal Bravais lattice of the upper hierarchical level. The groups of seven nodes at this upper hierarchical level again form islands on this hierarchical level, which are referred to as macroislands[2], in the same topological rule (Figure 1a). To apply this lattice to a 2D picture the GSC suppose an imaginary shift of even and odd pixel lines, so that the pixels are in an alternate order (Figure 1b).

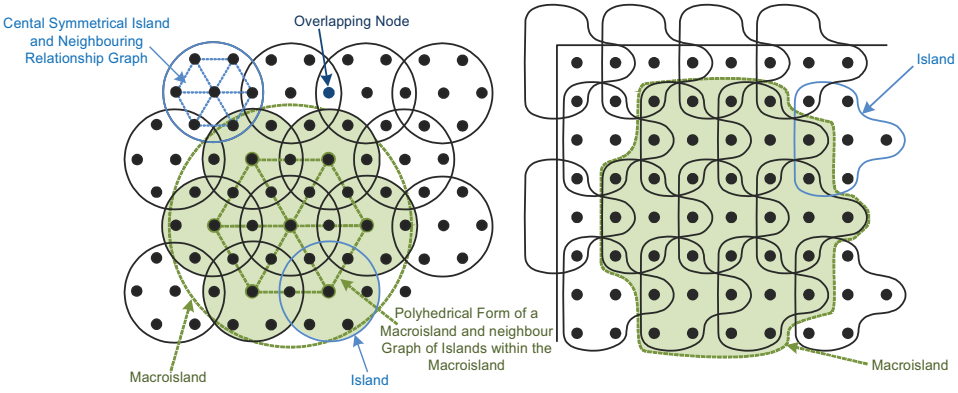


***Figure 1**: Forming hierarchical island structure*
a) Overlapping central symmetric islands on the hexagonal Bravias lattice
b) Mapping of hierarchical overlapping island structure to a pixel image

Considering the island organisation rule the total topological representation of an image can be seen as a pyramidal hierarchical structure (Figure 2) with direct topological correspondence between islands of upper and lower hierarchical levels. A level resolution of the pyramid is defined as

$$W_i = W_{i-1}/2+1 \quad ,$$

where W is the level width in islands, i is the level number, and the division operator is an integer division.

[2] Islands on the upper level are referred to as macroislands, while islands on the lower level are referred as subislands in respect with islands of a current level.

This hierarchical island structure is a framework for the region forming process. On a specific hierarchical level *(HL)* regions grow only inside the islands of this level.

At the lowest level the pixels that are covered by an island (referred to as *a pixel island*) may be grouped into *initial* or *pixel regions*. Two pixels may belong to a region if they are neighbours and if they satisfy some threshold condition. The pixels that are not covered by any region are called singularities. An image pixel can belong to two neighbouring islands and, therefore, can be covered by two different regions. In this case these regions are referred to as overlapping. This overlapping is the criterion for the region forming at the next hierarchical level.

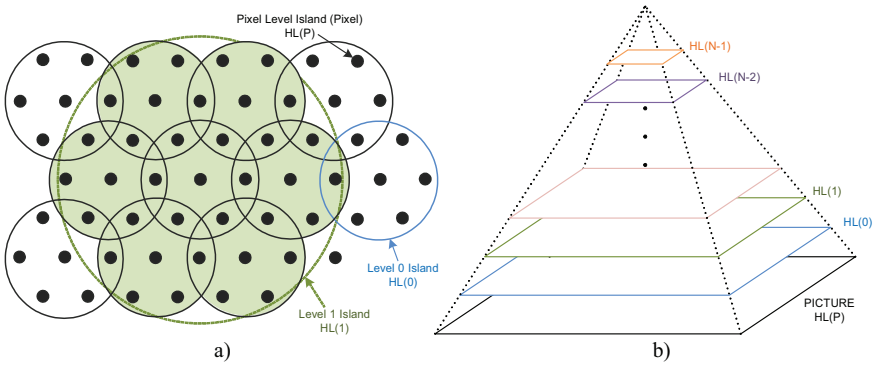


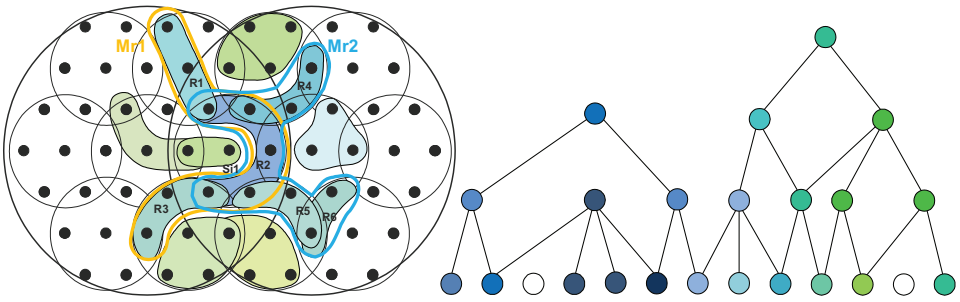
***Figure 2**: Pyramidal hierarchical-layer structure (HL(i) – hierarchical level, i – level index)*
a) Island hierarchy relationship; b) GSC relationship pyramid

One level higher the *pixel islands*' level, the *initial regions* may form new macroregions[3] in correspondent hierarchical macroislands if they overlap and satisfy the threshold condition. As in the case of the *pixel islands*' level, the same region may be included into two macroregions of two neighbouring macroislands, making those macroregions overlapping. Again, this overlapping condition together with the threshold satisfaction are the necessary criteria for forming higher rank regions inside the macroislands on the next higher hierarchical level and for all further levels through the pyramid in general. This means that a region of a hierarchical level consists of overlapping and similar regions of the hierarchical level underneath (Figure 3a).

The process of gradual region linking through the hierarchy represents a simultaneous segment grow over a picture with averaging joint region values. If a region cannot be linked further, it reaches its limits and forms a segment. Hence, the corresponding segment label or the mean value can be assigned to the original image pixels it covers. For this, the GSC requires some relationship information between regions of higher and lower hierarchical levels. This relationship indicates the inclusion of regions of a lower level in regions of a higher level. An upward, downward or two-way relationship can be established if appropriate. This hierarchical region relationship can be repre-

[3] For terminology convenience for the levels higher than pixel island levels the regions that are formed at a current hierarchical level are referred to as macroregions, while the regions that they are built of (i.e. one hierarchical level lower) are referred to as just regions, and the regions being two levels below the current level are referred as subregons.

sented as a hierarchical region coverage tree that establishes the parent – child relationship between regions of adjacent hierarchical levels. Considering this, the whole picture can be represented by the forest of these graphs (Figure 3b), with each tree representing a segment. A tree root will contain a mean value of the whole segment, while the root index may be treated as a segment label.



a) b)

***Figure 3**: Region forming and region relationship tree*
a) Region forming example: R1,R2,R3 and R4,R2,R5,R6 form two macro-regions Mr1 and Mr2 respectively, as they have common overlapping points, satisfy a threshold condition and are covered by common hierarchical islands; the macro regions Mr1 and Mr2 have a chance to be linked at the next hierarchical islands if they are covered by a common macro-island as they have a common sub-region R2 and seem to be similar.
b) Hierarchical relationship of regions represented as a region relationship tree

The GSC method can be represented as the following three subsequent phases. In *the coding phase* neighbouring and similar pixels are combined to local regions of the lowest hierarchical level. During the following *linking phase* these regions are linked hierarchically to global segments up to the highest hierarchical level, forming the hierarchical region coverage forest. In the following *result generation phase* the segments' labels or the mean values stored in the coverage tree roots are assigned to pixels. The image coordinates of a segment's pixel can be unambiguously determined by the topological coordinates of the *pixel islands* and the *initial regions*' pixel positions in the islands.

It is important to note that during *the linking phase* two regions can be overlapping, but nonsimilar. Therefore, in order to obtain a disjoint segmentation result, the overlapping area need to be separated afterwards. This separation is referred to as *region splitting*. The splitting procedure can be integrated either in *the linking* or *the result generation phase*.

2.2 Digital platforms

At the end of the last century it became obvious that the semiconductor industry is not capable to keep the pace of innovation within “Moore’s Law” progression any longer [1]. Moreover each next step in making microelectronic devices faster needs critically greater investments into the interdisciplinary research and development (R&D) cycle than it was ever before [52-53]. As a result the focus of the computer industry shifted notably towards the development of more effective

hardware architectural solutions and sophisticated calculation process organisation [54]. Computation parallelisation and application specialisation of hardware became the global tendency. Although these computation efficiency boosters are not new in computer science, these approaches tend to migrate from the upmarket supercomputers to middle and low-end market computation systems.

Computation parallelism primarily characterises instruction-flow driven computers such as those that are based on conventional processors[4]. For these systems different parallelism levels can be put in correspondence, in contrast to application specific hardware, for which parallelism is its intrinsic and implicit quality and thus it is difficult to define some distinct borders of parallelism granularity. In processor-based systems several levels of parallelism can be distinguished: the parallelism of operations, instructions, tasks or threads, and applications. The nowadays processor architectures try to address all the types. For instance most of the processor vendors extend their traditional instruction set architectures with SIMD (Single Instruction Multiple Data) instruction subsets such as AltiVec by AIM alliance [55], or MMX and SSEx by Intel [56], ARM's mulimedia NEON [57], or encryption AES [58] instructions. These extensions are vector operations and can be treated as a mean for exploiting operation level parallelism. Further perfection of superscalar architectures and adoption of VLIW (Very Large Instruction Word) architecture for variety of processor architectures (e.g. VelociTI architecture by Texas Instruments [59] for digital signal processors, EPIC paradigm introduced by HP and Intel alliance [60] used in Itanium architecture) can be seen as examples of instruction level parallelism. Multicore platform design is an approach that has been focused on by the majority of processor vendors. This approach can be seen as a mean for lower level parallelism exploiting in single chip solutions. The global tendency in multicore chip design is the migration of multiprocessor architecture principles to the level of a VLSI circuit. The multicore platforms vary broadly in a number of correlated characteristics such as the number of cores, inter-core communication organisation, level of architecture integration (common resource sharing), core coarseness, homogeneity, and application domain. The number of cores in a modern multicore processor may count dozens and hundreds (e.g. massively parallel TilePro by Tilera [61], or Ambric's multiprocessors [62] with 2-d mesh interconnect). The number of cores will typically influence inter-core communication organisation (shared memory and message passing) and on-chip network topology (common bus, two dimensional mesh, crossbar, or ring). The cores could be light weighted specialised co-processing units or full potential general purpose processors. An interesting feature of a multicore chip is the core homogeneity. In heterogeneous platforms one core typically play a role of a general purpose host processor and the other(s) act as specialised accelerator co-processors. For example: the OMAP platform by Taxes Instruments [63] contains an ARM host core and one or several digital signal processing (DSP) core(s), or the Cell BE architecture is compound of one PowerPC host processor and a number of so called SPE (Synergistic Processor Elements) streaming coprocessors [64]. In contrast homogeneous graphical

[4] The term is used for a broad class of standard ICs with sequential instruction execution architecture such as microprocessors, digital signal processors, vector processors, specialised co-processors, etc. The term will be used hereafter to refer this type of ICs.

processors contain only a grid of replicated SPMD (Single Program Multiple Data) processing cores [65-66]. The multicore platforms could be found in different application areas such as general purpose computing and embedded, multimedia, and network systems.

Exploiting the architectural parallelism of a system requires special means of computing organisation. These means reside in the hardware programming model, which demands a support from the operating system (e.g. SMP OS) or in optimising compilers (e.g. EPIC technology compilers). Such means make the programming process transparent for programmers. Alternatively the software parallelisation may need special guidance from the coder to discover the potential of a system (e.g. OpenMP, OpenCL, MPI, CUDA or other proprietary SDK).

Thus, the amount of knowledge that has been accumulated over the last century in the field of high-performance upmarket computer architectures become available in middle and low-end markets, making performance demanding computation affordable for a large number of users.

Application specialisation of the hardware is another tendency in the computer industry. This specialisation becomes available due to the following circumstances. With the rapid development of the semiconductor manufacturing technology the cost of a logic gate of matured technologies has become significantly low [67-68]. The level of electronic design automation (EDA) became sufficient for complex system design and in particular for modular system design. The market of hardware intellectual property (IP) cores is well developed and enough saturated, which seriously eases the development of complex systems. All these preconditions make silicon implementation of rather complex algorithms affordable even for micro- and nanocap companies. Thus, it often allows the migration from solutions based on standard VLSI circuits with a unified architecture, such as conventional processors, to customisable and application specific solutions such as Application Specific Standard Products (ASSP), Application Specific Integrated Circuits (ASIC), Programmable Logic Devices (PLD), and coarse-grained customisable multicore SoPs. The mainstream processor vendors are not exceptions in this case. Application specific instruction set extensions and heterogeneous multicore processors are good examples for the hardware specialisation.

In the following section the computation organisation will be seen from a perspective of the target application implementation.

2.2.1 Algorithm implementation

It happened historically that the electronic digital computer is commonly understood as a machine that manipulates pieces of data in sequential order guided by a sequence of intrinsic instructions. This sequence is called a program and is stored in external memory. The total of all intrinsic instructions is an Instruction Set Architecture (ISA) of the machine. A machine built this way is conventionally called a processor. Generally a processor can be represented as a regular set of execution units with a data-path switching circuitry that distributes data among the execution units upon given instructions. This approach in building computer architecture gives two important

characteristics to the computer to be an economically efficient solution. They are regularity of hardware structure and versatility of computations. Changes in an implementation of computations require modification of only a program. Although modern computers are capable of executing multiple instructions at a time they still obey the principles of program algorithm interpretation and imply time-sequential execution.

A program as series of instructions is not the only way of implementing a computation task. A computation task can be realised as a number of data-transformation (or functional) blocks that are connected to each other with data paths to form a computation structure. This understanding of computing corresponds to the data-flow representation of a computation task. It allows a natural exposition of parallelism in data transformation in contrast to the instruction-flow representation, which characterises rather the executor of data transformation than the data transformation itself. Computation systems that are built based on structural interpretation of algorithms are opposed to the unified and sequential processor architectures.

The main technological difference between these two approaches is that the computers based on the principles of structural implementation of algorithms lacks for application versatility compared to the program-driven. Functionality of their data-transformation blocks and interconnection network configuration are application specific. A change in an algorithm implies change in either network configuration or block functionality or both. This becomes a problem if the application specific system is produced as a single VLSI circuit. Although VLSI will typically improve characteristics of a system, such as cost, performance, reliability, power consumption, etc., compared to multichip solutions on standard components, it makes the production of those Application Specific Integrated Circuits (ASIC) more expensive in comparison with unified standard ICs. This rule is true unless the implemented functionality of these ASIC becomes so broadly demanded in the market that this type of ASICs starts being produced massively (Application Specific Standard Product – ASSP).

To reduce production costs the industry once chose the following two strategies: structure regularity and structure programmability. These two approaches brought to life a class of ICs with programmable or configurable structure. The key idea of this type of devices is to produce standard ICs with regular array of functional blocks (mainly configurable) with programmable[5] infrastructure for block interconnection and to let the end user configure the device for a specific application. Together with high level of integration achievable by the modern fabrication these programmable structure devices made hardware implementation of application algorithms broadly available.

2.2.2 Devices with configurable structure

Modern high-integration devices with programmable structure could be divided into two subclasses. They are known as Complex Programmable Logic Devices (CPLD) and Field-Programmable Gate Arrays (FPGA). These two types of devices are principally different in the architecture of their

[5] Terms *configurable* and *programmable* can generally be treated as synonyms. Meanwhile in this chapter *programmable* implies that a device can be configured by an end user without purchasing any additional services from a manufacturer for device configuration.

functional blocks and the organisation of their interconnection system. To understand the fundamental principles and peculiarities of these two programmable device types it is important to look at the roots of these architectures.

2.2.2.1 Simple programmable logic devices

CPLDs are the result of the evolution of more simple programmable logic devices. These simple programmable logic devices (SPLD) are Programmable Logic Arrays (PLA) and Programmable Array Logic (PAL).

Both device types implement disjunctive normal forms (DNF) of switching functions. The structural model of an SPLD can be represented as an AND gate plane and an OR gate plane linked in series. The AND plane may receive m Boolean inputs in direct or inverse form and produces q conjunctive terms. Connection of each input signal (direct or inverse) to an AND gate is programmable, while the number of conjunctive terms is device specific. The outputs from the conjunctive terms are fed to the OR plane. The OR planes for devices of PLA and PAL types are different. In a PLA device the OR plane is programmable, whereas in a PAL it is fixed[6].

The first generation of SPLD was quite trivial in functionality and could implement only combinatorial logic with simple input/output buffer schemes. The further development of this device type was targeted to higher functionality and application flexibility. It was bidirectional or high-impedance pins introduction, output inversion and internal signal feedback, configurable trigger memory elements and resource sharing. A good retrospective of SPLD evolution could be found in [69]. With increase in complexity of PLDs a new class of programmable logic devices has been distinguished. This new architecture built on arrays of PLA/PAL functional blocks connected by a common programmable interconnection array is referred to as Complex Programmable Device.

2.2.2.2 Complex programmable logic devices

A classical CPLD is a device consisting of multiple simple PLD-like Function Blocks[7] (FBs) and I/O Blocks (IOBs) highly or fully interconnected by a switch matrix referred to as Programmable Interconnect Array or Matrix (PIA/PIM). IOBs provide the buffering of device inputs and outputs and can be configured to be complaint with different electrical standards. Each FB provides a programmable logic capability for the device. It consists of a Sum of Product (SoP) plane[8] linked to an array of macrocells. The outputs generated by FBs may be routed back to the switch matrix or used

[6] Programmable OR plane in a PLA consists of a number of disjunctors, each OR gate having q inputs. By programming the OR plane each disjunction may receive any combination of available conjunctive terms. Each conjunctive term can be fed to several disjunctors. The number of disjunctors in the OR plain is defined by the number of device outputs. Given n outputs, a PLA can implement a system of n switching functions, which depend on no more than m variables and contain no more than q terms. The reason why in a PAL device the OR plane is fixed is that in many cases there is no need for such flexible interconnection capability of AND terms. The inputs to an OR gate can be hardwired from a fixed set of conjunctive terms making PAL device simpler and faster. Those AND terms that happened to be fed to more than one OR gates can be elaborated in the AND plane several times.

[7] In Xilinx terminology. Altera, for instance, uses the term Logic Array Block (LAB) to refer to analogous unit.

[8] The SoP plane of a functional block can be PLA-like (i.e. both AND and OR planes being fully programmable), PAL-like (i.e. only AND plane is fully programmable, while OR plane is fixed), or a hybrid. An example of CPLD with a hybrid SoP plane is Xilinx XC9500 [70]. Function blocks of this CPLD have fully-programmable AND planes linked to Product Term Allocator (PTA) blocks, which allow product term exchange between neighbour macrocells.

to drive correspondent IO blocks (Figure 4a). The switch matrix connects all FB outputs and device input signals to FB inputs.

Function blocks of a CPLD may contain local feedback paths that allow the outputs of FBs to be driven back into programmable AND array without going outside the FBs. These paths are used for creating very fast sequential logic where all state registers are within the same FB. CPLDs may include dedicated resources for signal exchange between neighbouring FBs without routing them through the PIA, which allow complex logic functions to be allocated in adjacent functional block increasing user design performance.

Each function block contains an array of macrocells. Further, each macrocell comprises a combinatorial or a registered logic path for one of sum-of-product functions[9] generated in the SoP plane. A macrocell typically contains a combinatorial logic cloud followed by a flip-flop. Flip-flops of macrocells can be used to register input signals of a CPLD device directly without routing through PIA and SoP plane of FBs. The macrocell flip-flop may accommodate asynchronous presets and resets as well as power-on initial states. It can typically be configured as either D or T type register, or as a latch.

Outputs of functional blocks are connected to vertical unsegmented lines in the Programmable Interconnect Array. It is typical that each FB output has its own dedicated vertical line. FB inputs are connected to horizontal lines. Horizontal lines intersect all the vertical lines providing programmable interconnection points. Therefore, any input of FB can be connected to any output, thus offering full connectivity of blocks. General scheme of PIA is represented in Figure 4a.

The advantage of this kind of interconnect is high-speed and deterministic signal propagation within PIA, due to identity of connection paths and a small number of programmable point. In some cases PIA does not contain programmable interconnection points at all (e.g. Altera's MAX3000) allowing unused input signal to be masked prior to letting them run into functional blocks.

Programmable structure devices with a common unsegmented switch matrix have their own structural limitations. The number of connection lines increases dramatically with increase in number of elements to be connected, which hampers FB number scaling for CPDLs. A modern CPLD contains up to some hundreds of functional blocks and I/O pins. This fact together with rich combinatorial logic capability and fast internal connectivity makes CPLD architecture perfect for such applications as interface bridging or I/O expansion, but not for a real System on a Chip. A more solid application requires more resources and other architectural approaches, like those of FPGA-type devices.

2.2.2.3 Application-specific integration circuits

FPGA devices have their architectural roots in ASICs technology. ASIC devices can be classified as full-custom or semi-custom devices. The difference between those classes is in the IC element optimisation and the component reuse.

[9] In a real CPLD architecture it is difficult to distinguish a border between the SoP plane and a macrocell. Many datasheets shows the OR plane gates as a part of macrocells.

An IC can be viewed as a set of transistors and connection wiring. A full-custom ASIC implies customisation of all the components of an IC including transistor parameters, component placement and routing. This approach in IC manufacturing is the most efficient in power consumption, IC performance, and size, yet the most expensive, error-prone, and laborious. That is why it is used typically exclusively for designing dies with some critical requirements and for devices made by a new manufacturing technology.

Standard Cell methodology helps to reduce the time-to-market and the design costs. This methodology implies the reuse of IC components. These components are given as a library of predesigned low-level logic functions, such as AND gates, OR gates, multiplexers, flip-flops, etc., known as standard cells. These cells are realised as full custom cells with fixed height and variable width. Full customisation of cells means that each cell in a library may be optimised individually, which allows minimisation of internal delays and area optimisation. The fixed height lets the cells be placed in rows, which eases automated design layout, while the variable width gives the functional flexibility of a cell.

Rows of standard cells may form rectangular blocks with higher-level functionality. These areas of standard-cell rows can be combined with larger predesigned cells like controllers or memory blocks on a chip. These higher-level cells are called megafunctions or system-level macros. A designer is flexible in ASIC component placement. All these allow modular design with much space for chip optimisation, although still requires production of complete set of IC layer masks.

Another approach of ASIC production is to use premanufactured wafers with a regular array of transistor pattern blocks, which are called base cells. Base cells have a fixed logic element set inside. All ASIC resources have a fixed placement and form a regular structure. This is why this kind of ASICs is called Gate-Array-Based ASICs. Customisation of a device is performed exclusively by the interconnection customisation, which let them be called semi-custom ASICs.

Gate-Array ASICs may contain dedicated area for interconnection routing. In this case base cells are typically organised in columns. This type of GA is called channelled. The other type of GA does not have special routing areas. Wiring is performed through unconfigured base cells. This type of ASIC is called channelless GA or Sea-of-Gates (SoG). GA can be homogeneous or containing blocks of base cells of different type or dedicated megafunction cores. In the latter case this type of ASIC is called structured GA. A base cell can be fine-grained, containing only transistors and resistors, or can be partially preconfigured (coarse-grained), containing low-level logic functionality such as multiplexors, registers, lookup tables, etc.

A good overview of ASIC technology can be found in [69, 71]; more detailed information on ASIC design is given in [72].

2.2.2.4 Field-programmable gate arrays

The concept of Gate-Array ASIC architecture was adopted by FPGA devices. A typical FPGA architecture consists of an array of identical function blocks, a distributed interconnection system and a number of I/O blocks placed in the peripheral areas of a die. The main difference between GA ASICs and FPGAs is that all elements of an FPGA including the connectivity infrastructure are pre-manufactured and end user programmable. A typical FPGA is organised as a rectangular matrix of function blocks with its interconnection system placed between rows and columns of the matrix Figure 4b.

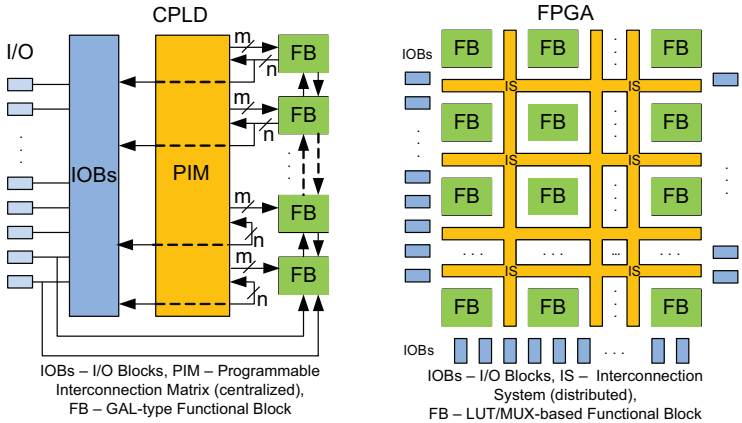


***Figure 4**: CPLD vs FPGA principle architectures*
a) CPLD architecture; b) FPGA architecture

When programming an FPGA, its function blocks are configured to perform the required data transformation, while its interconnection system is programmed to make connections between those function blocks. As a result, the inner parts of an FPGA form the circuits that implement the desired application functionality, and its I/O blocks on the edges of the die provide an interface for the integration with the rest of computation system. I/O blocks of modern FPGAs are programmable to be complaint with a wide variety of I/O signalling standards.

A typical function block model can be represented as a data path consisting of a function generator block, implemented as a lookup table (LUT)[10] or a number of them, a cascade of multiplexers for data-path configuration, and a register block, which includes one or several flip-flops.

A large number of function blocks make highly connected systems built on continuous connection lines ineffective. As the number of blocks grows, high connectivity leads to an irrational usage of

[10] Devices with LUT-based function blocks currently predominate in the market. Meanwhile, there are alternatives (mostly matured) e.g. multiplexer-based FB devices and fine-grained Simple Logic Cell (SLC) architectures. The author of the work is aware only of one modern alternative multiplexor-based architecture, which is QuickLogic PolarProII FPGA [73]. A good survey of alternative matured FPGA architectures could be found in [69, 74-75].

the die space, as the most of the chip area is needed for wiring. That is why the interconnect system of modern high-density FPGA architectures is organised as hierarchical segmented networks.

Such interconnection systems consist of segmented lines of different length and programmable switching matrices at the intersection of horizontal and vertical interconnection traces. The segments are linked together with programmable connection points inside the switching matrices to make the desired connectivity of the logic blocks.

Every programmable connection point adds a significant delay to signal propagation time in the interconnection network. To minimise the number of hops hierarchical interconnection systems are built with line segments of different lengths. The short lines connect mainly neighbouring or nearby function blocks of an FPGA. They are used to glue functional blocks with high logic integration (dense logic groups), which are reasonably placed close together. The longer segments connect function blocks in a certain distance from each other. They are used to connect logic groups which have a small number of signals to exchange.

Hierarchical network organisation together with accurate resource placement and connection routing may allow keeping the number of programmable connection point minimal. Therefore, a hierarchical interconnection system offers a balanced solution between density of interconnection network wiring and signal propagation time in a complex, resource intensive design.

Another example of hierarchical organisation in FPGA devices is integration of a number of function blocks within a new kind of architectural blocks (Configurable Logic Block in Xilinx terminology and Logic Array Block in terminology of Altera). The function blocks inside these unions can share logic resources and exchange signals without exploiting global interconnection network system. These architectural blocks form new singularity in a global interconnect system. This grouping allows efficient implementation of both simple and complex logic functions on the same types of function blocks, keeping granularity of function blocks relatively low.

A special attention in modern FPGAs is paid to the synchronisation system. With the increase of the clocking frequencies the length of a trace becomes a critical factor. The difference in distances between a synchronisation signal source and sinks can lead to desynchronisation and metastability effects in a system. Thus, clocking networks have a tree-like topology to minimise this effect. Synchronisation systems are built using dedicated resources of a die such as detached trace layers, specialised pads, buffers and clock control circuits. Clock control circuits are used for clock deskew, jitter filtering, phase shifting, frequency synthesis, etc.

Apart from the basic architecture components mentioned above, modern FPGA devices are typically equipped with dedicated arithmetic blocks, such as adders of matrix multipliers, blocks of RAM, dedicated connection resources, such as fast carry chains or wide logic chains, and even hard cores of popular interface controllers or even microprocessor cores[11].

[11] The most typical architectural features that characterise each type of modern CPLD and FPGA devices are architectures of logic blocks and interconnection systems. Meanwhile, this difference in modern devices becomes dilute. Starting from FLEX10K architecture Altera Corp. tried to combine advantages of both devices types. It combined network of unsegmented horizontal and vertical channels with LUT-based function block architecture [76]. In the latest MAX II CPLD family Altera migrated from SoG-based to LUT-based logic

The integration level of modern FPGAs may be measured in hundreds of thousands of function blocks, have megabytes of onboard RAM and hundreds of I/O pins. This integration level is sufficient for building complex system on a single chip.

2.2.2.5 Virtex II Pro architecture

The Virtex-II Pro is a high-density FPGA architecture with an SRAM-based in-system configuration. It is built on a matrix of configurable logic elements and embedded blocks surrounded by input/output blocks. The connectivity of the blocks is provided by a hierarchical distributed interconnection system. The internal logic is implemented by configurable elements of three types organised in a regular array of:

- Configurable Logic Blocks (CLB) consisting of four slices or eight function blocks each[12] (up to 11.024 CLBs),
- static dual-port RAM blocks with total capacity of 7.992Kb (up to 444 BRAMs),
- 18-bit x 18-bit matrix multiplier blocks (444 maximum).

The interior of the die contains up to two embedded IBM PowerPC 405 RISC processor hard-cores, providing further flexibility in System on a Chip design.

The clock management is provided by Digital Clock Manager (DCM) blocks that includes self-calibrating clock distribution delay compensation, clock multiplication and division, and widely programmable clock phase shifting. The most powerful device in the family contains twelve of these blocks.

The I/O system of a chip provides up to 1.164 user pins. The configurable I/O blocks of Virtex II Pro are complaint with twenty-two single-ended standards and ten differential standards. The I/O clocks can be configured for unidirectional or bidirectional signalling and for single or double data rates.

The Virtex II Pro contains up to twenty gigabit serial transceiver with a maximum data transfer rate of 3.125 Gb/s each, providing a high-bandwidth interconnection between chips, backplanes, or other subsystems. The programmable routing resources provide the interconnection for all of these elements. The interconnection system is hierarchical. The network consists of vertical and horizontal segmented lines of different lengths. The lines are connected through programmable switch matrices, allowing flexible and balanced routing of signals.

All programmable logic blocks and routing resources are controlled by static configuration memory cells. Thus, devices are capable for unlimited reprogrammability. Partial configuration of a die is

elements grouped in larger logic blocks. Additionally Altera changed the organisation of MAX II interconnection system from common programmable interconnection array to a network organised in vertical and horizontal unsegmented tracks [77]. In fact the architecture principles of FLEX10K and MAX II families are very similar. It seems that architectural approaches in early Altera FPGA families are resurrected now in the latest CPLDs.

[12] A slice is a fines structural unit of Virtex architecture. Each slice roughly consists of two functional blocks, i.e. two LUTs and two flip-flops with path configuration logic.

available as well. Compact profound description of elements of Virtex II architecture can be found in Appendix A.1.

2.2.3 Graphics processing units

Although the architecture of graphics processing units (GPU) has drastically evolved from accelerators with fixed graphic processing pipeline stages of fixed or minor configurable functionality to massively parallel computation coprocessors capable for high performance computation (HPC), it is very important to understand the graphics background of this type of platform to effectively exploit its computation power.

Awareness of 3D processing basics helps in comprehension of modern GPU architecture peculiarities and assists in understanding what types of algorithms are well suited for GPU computing or how an algorithm should be optimised to gain performance on such platform.

2.2.3.1 Graphic pipeline

Graphics processing units have been designed to relieve the CPU of the increasing burden of the real-time rendering of virtual scenes for 3D games or animation filming. The introduction of the object abstraction model into graphic processing allowed the decomposition of the visualisation process into a number of unified visualisation stages independent on a scene and its characteristics. The decomposition lets a virtual world designer concentrate on specific properties of instances intrinsic for each specific abstraction level, while the stage unification gives a capability for the automation of the visualisation process. At the same time the decomposition helps to optimal organise the visualisation processing, allowing task parallelisation and even distribution of data flows over the graphics pipeline stages.

A scene in a 3D graphics world is a composition of virtual objects and light sources. Objects are characterised by their shapes and surfaces attributes. Surfaces of complex shapes are normally represented as a collection of plane primitives (typically triangles) to simplify the computation of the light beam and surface interaction. The composition of geometry, light and surface qualities defines the visualisation of a scene.

A canonical graphic pipeline is a hardware/software abstraction representing stages of the visualisation process realised in a GPU. The exact number of stages and the function set of each stage varied over the evolution history of GPUs [78-80]. Meanwhile, a general graphics pipeline can be represented as the following:

a) The vertex processing stage operates on the vertexes of the surface primitives. The basic task of this stage is the lighting of each individual vertex depending on the relative position to a light source and the projection of 3D coordinates in the virtual space to the plane of observer monitor[13].

[13] Meanwhile, depth information of a scene is not discarded, it is in latter stages for element visibility detection.

b) The primitive assembling stage performs the grouping of correspondent vertexes that comprise a geometry primitive.

c) The geometry processing stage allows new geometry generation out of former geometry primitives (e.g. primitive tessellation).

d) The rasterisation stage transforms the geometric primitives to groups of pixels.

e) The fragment processing stage calculates visual parameters (generally colour and opacity) for each individual pixel in a fragment (group of pixels comprising a primitive). It shares common calculations within the fragment and then derives individual values for each pixel in a fragment.

f) The pixel operation stage performs the mapping of all the pixels with the same x, y coordinates, but different z-depth to the screen producing the resulting scene view.

Three generation of GPU architectures are distinguished in this work. Initially the GPU architecture was represented as a hardware pipeline with stages of fixed or configurable[14] functionality [78]. This rigidity in architecture tied the visualisation capability of software to functional capacity of certain device families, forcing the programmers to reserve several execution paths for different platform. At the same time it bound a designer to the limited set of processing functionality provided by hardware vendors, which limited the progress of the visualisation technique.

As the time passed, some of the stages that are more diverse in implementation become programmable, while more straightforward operations remained to be executed by fast dedicated blocks [81-83]. These programmable stages were the vertex and fragment processing stages[15]. The introduction of programmable nodes into the graphics pipeline marked the second generation of GPU architectures. Programmability gave a higher potential to the visualisation technology and allowed more impressive visual effects. The visualisation technique now relied more on creativeness and craftiness of graphic programmers.

At first the instruction set architecture (ISA) of those programmable stage processors were quite specific and were programmed in specialised assembly languages. The programs for these processors are called shaders. However, to unbind programmers from specific hardware architectures unified graphics application interfaces (API), such as Direct3D and OpenGL, were introduced to function as a gasket between hardware and software, shifting the problem with software compatibility onto GPU producers.

Meanwhile, the hardware task parallelism (i.e. hardware task pipeline) gave several major disadvantages. First of all is the problem of distributing the computing load, which is dependent on the nature of a scene. So, for example, a scene with complex geometry, but simple surface attributes will shift the load imbalance to the vertex processing stage, while a scene with small number of

[14] Functionality could be selected or adjusted, but not created like in programmable units.

[15] Due to its complexity the geometry processing is a relatively new stage, which was implemented in shaders starting with the unified shader processor architecture. Before that the geometry processing stage appeared in some rare GPUs, in which it was realised as fixed tessellation blocks, but was more an exception than a rule in GPU architectures.

polygons and complex textures will load more the fragment processing stage. The second problem is a fixed data-flow path configuration which is specific for different graphics processing stages due to hardware optimisation reasons. It restricts the usage of shader processors of certain graphical hardware pipeline stages to particular graphical subtask. It implies that each new kind of shader introduced in visualisation process should change the graphics hardware pipeline. These reasons led to the idea of unified shader processor architectures – the third GPU architecture generation.

The third GPU architecture generation is characterised by shader processors with a generalised ISA, which are built in a unified data-flow path implemented as a unified memory access model [84-88]. This architecture allows executing different types of shader (even those that are not yet invented) on the same piece of hardware. Meanwhile, it does not imply that the dedicated function blocks are removed from GPUs. Those blocks now play the role of auxiliary processing units and are not the inevitable stages in a data path.

The concept of the third generation architecture could be illustrated by the high level structure diagram of the nVidia G80 architecture (Figure 5). One can notice that instead of linear data forwarding through a hardware graphics pipeline the data circulates through a unified shader array programmed for a certain graphic pipeline stage execution under the control of a dispatch unit.

2.2.3.2 Peculiarities of graphics processing organisation

When programming modern GPU for general purpose computing, one do not need adapt the problem to the graphics pipeline, while now a modern GPU is far more than a graphics accelerator. Meanwhile, graphics processing tasks have a number of distinct characteristics in common, whose peculiarities defined the architectural concepts of GPU processors. Understanding architecture specifics of graphical processors is the clue to effective implementation of general purpose computations on a GPU.

The nature of graphic processing implies high data parallelism, as the majority of the data pieces of the graphic pipeline (vertexes, primitives, fragments, pixels) could be processed in each pipeline stage independent on each other [78-79]. This makes the architecture of GPUs highly parallel and scalable[16]. The multicore organisation of GPU is a notable feature of GPU architectures.

Meanwhile, the graphic data element parallelism is not the only intrinsic data parallelism in the graphics processing world. The majority of intrinsic data types of graphics processing are vectors, which adds finer granularity of parallel computations to GPU architectures[17].

Another noticeable feature of graphics processing is that the execution flow on nearby data-stream elements often results in the same execution path. This instruction stream sharing together with the

[16] This peculiarity always drove extensive development of GPU architecture by increasing the number of parallel processor elements at each stage of graphics pipeline.

[17] That is why processing elements in GPU architectures were equipped with vector ALUs(with vector component permutation) up to the third generation of GPUs. Latter this tendency split into two direction. ATI chose concept of SIMD operations on VLIW ALUs, which could be seen in case of GPU architecture as further development of vector operation concept that allows combined type operation on vectors (needed in some graphic transformations). nVidia took more cardinal way of SIMD processing elements built on scalar ALUs.

vector orientation of the calculation brought up the idea of SIMD orientation into graphics processing. SIMD organisation is in reality the core paradigm of modern GPU architectures.

The idea of SIMD is even more strengthened by the fact that the graphics algorithms are calculation intensive instead of a control branching orientation, which makes computations on SIMD architecture less sensitive to divergence in control.

The independence in the processing and strong spatial locality of data elements brings to life another noticeable feature of the computation organisation in modern GPUs – hardware multithreading. The dynamical branching capabilities of the shader processors, which are driven by newer shader standard requirements, pose the problem of execution stalls in the processor cores. These stalls are caused by the latency of a memory access if the required data cannot be prebuffered due to dynamic branching uncertainty[18]. The spatial proximity of graphic elements allows to load a solid bulk of data close to a processor core and to bridge the memory access stalls by performing the computations for prefetched data elements using thread execution context switching until the data requested from the memory become available to the core. All modern GPUs maintain large numbers of execution contexts on the chip to provide maximal memory latency-hiding ability.

This approach to memory latency hiding available in graphics processing together with other peculiarities of graphic pipeline organisation make the memory system of GPUs quite specific. The memory system is predominantly oriented to high throughput bulky accesses instead of latency hiding typical for memory hierarchies of CPUs.

As noted above GPUs are characterised by a massively parallel multicore organisation based on SIMD computation principles and oriented to a high data streaming throughput. These characteristics are posed by the peculiarities of the graphic processing organisation. Meanwhile, those intrinsic features of the computation organisation are not exclusive for graphic computation but are common for a whole class of computation tasks with data parallelism, which were shown in a number of works in the fields of physics, biology, finances and so forth. This makes the power of modern GPUs available for general purpose computing.

2.2.3.3 Modern GPU architecture

The principles of architecture are the same for all the GPUs of the third generation [87-95]. A GPU consists of a grid of unified shader processor cores with SIMD organisation with a broad ISA that executes the general computation routines supported by a number of specialised graphic function blocks. The workload distribution between the processor cores is performed by a thread blocks scheduler (Ultrathreaded Dispatch Processor in ATI terminology and Thread Processor in terms of nVidia). Each SIMD processor core consists of an array of similar execution units (basically ALUs with some execution context circuits). The principle difference between GPU architectures of the

[18] Massive parallelism of a computations does not allow speculatively prefetch of large amounts of data needed for feeding all processing elements working in parallel.

main two GPU vendors – ATI and nVidia – is that nVidia has scalar ALUs and ATI builds its SIMD processor cores on ALUs with VLIW architecture.

The data parallelism does not mean total computing stream isolation in GPUs. Shader algorithms often require a data exchange between streams. To realise this each SIMD processor core is equipped with a local storage shared by all the thread cores inside a SIMD processor core.

A typical example of such architecture is the nVidia GeForce 8800 GTX chip (G80 architecture). The resources for general purpose computing of this device are shown at Figure 5.

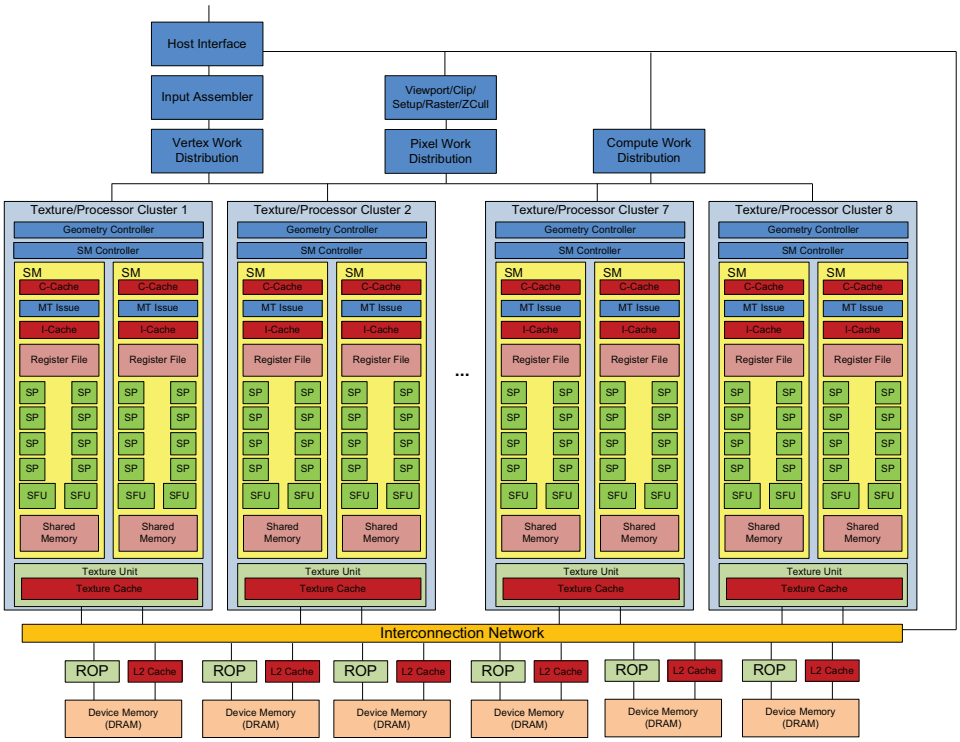


***Figure 5**: nVidia G80 architecture*

nVidia GeForce 8800 GTX comprises 16 SIMD processing cores called “Streaming Multiprocessors” (SM) governed by a computing thread blocks scheduler. The processing cores are grouped in eight Thread Processing Clusters (TPC) with two SM each. The two SM share a read-only L1 cache and a number of specialised graphic resources (Texture Units – TUs). All eight TPCs share six global memory access channels using an intra-chip crosspoint switch. Each channel comprises a

read-only L2 cache (load stream, load accesses may bypass L2 cache)[19] and a render output[20] unit (store stream) coupled with a 64-bit GDDR3 controller.

A Streaming Multiprocessor consists of eight scalar execution units, which are called Thread Cores or CUDA cores. The execution units share a common storage for thread interactions, the Shared Memory. Each execution unit owns a private register space in a common register file and consists of an ALU equipped with a floating-point unit and an individual branch unit with an instruction pointer for the individual execution flow control. Meanwhile, an execution unit is not a mature processor as it lacks a complete front end that can fetch and schedule instructions independently. The mechanism of execution flow control is provided by a common instruction fetch/dispatch block, which issues the SIMD instructions to all the execution units in an SM (see Figure 5).

A single Streaming Multiprocessor operates with computing thread blocks. A thread block is a number of execution threads, driven by a single program[21] over a number of different data sets (or data streams), which are able to interact with each other via a shared memory common to all execution units in a Streaming Multiprocessor.

The number of execution threads in a block is flexible and may be greater than the number of execution units[22]. Moreover several thread blocks can be executed on one SM concurrently. Heavy computing thread population of a multiprocessor is needed to support the hardware multithreading mechanism mentioned earlier.

Hardware multithreading is realised by microarchitectural abstraction called a warp. A warp is thread grouping that form a hardware multithreading context in a Streaming Multiprocessor. Each SM is able to maintain several warps in a time realising context switching. The size of a warp is fixed for a given architecture, but not visible in a programming concept.

This leads to the idea that nVidia G80 architecture is well suited for massive parallel scalable computation, meanwhile one noticeable downside of the SIMD computing organisation implemented in GPU multiprocessors needs to be mentioned. In case of divergence in execution threads within a warp a multiprocessor have to execute all current conditional branches of a program, which leads to serialisation of control flow tree execution in this thread group. This downside is mentioned here as it is an intrinsic peculiarity of the stream processor architecture. G80 has some other pitfalls, but these pitfalls are related to memory subsystem hardware implementation and not the architectural peculiarities. Those pitfalls will be described in Section 3.2.

[19] L2 cache as well as L1 cache in the devices of the given architecture is used for data buffers of certain types allocated in the global memory and can be bypassed for the load accesses for data of different types.
[20] Render output (ROP) blocks correspond to the last pixel operation stage of the graphics pipeline.
[21] The same program may be executed with different execution paths depending on data it processes, these different paths with correspondent data streams are called execution threads.
[22] This programming concept of SIMD architectures where the data dimensions are not directly bounded to hardware resources in a program context is called stream processing paradigm.

2.2.3.4 GPU computing

Initially general purpose computing on GPUs were realised by mapping the computation algorithms to the graphics pipelines by means of graphics APIs and later higher level shader languages [96], such as nVidia's Cg, OpenGL GLSL or DirectX HLSL, which was obviously an awkward approach of the algorithm description. As the interest to GPU-powered computation arose in the scientific world the need for abstract languages for general purpose GPU programming became tangible in the academic environment. This need gave the birth to a number of academic projects for general purpose programming on GPU, which were primarily built on the programming principles of stream processing. BrookGPU and Sh were most broadly recognised among them.

The success of GPU computing shown in scientific community attracted the attention of the business society and triggered a number of commercial third-party projects, e.g. RapidMind (commercialised Sh, later acquired by Intel) and PeakStream (commercialised Brook, later purchased by Google). The next step in GPGPU computing was made by the GPU vendors, when they realised the marketing advantages of targeting a new application field for their GPUs. Both AMD and nVidia launched their own GPGPU programming systems (Close-To-Metal and CUDA, respectively)[80].

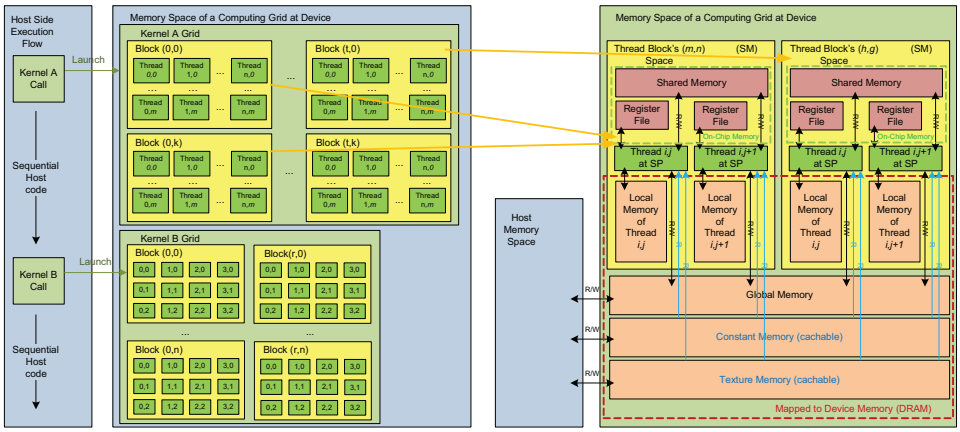
The latest stage of GPGPU language evolution is the standardisation of architecture-free stream-processing languages. The current candidates to become the defacto standard are OpenCL [97], initially developed by Apple Inc. in cooperation with AMD, Intel, IBM, and nVidia; and DirectCompute, a part of Microsoft DirectX. Both OpenCL and DirectCompute together with nVidia's CUDA [98] (the most matured GPU computing framework) will be determining the market landscape of GPU computing in the medium-term perspective. While all the modern GPGPU programming frameworks are conceptually very similar, the market dominance most probably will not be determined by the peculiarities of the programming concepts but by the marketing policy of the companies behind those platforms.

The programming model of stream processing being common for all modern GPGPU frameworks is based on the two main concepts of data streams (or just "streams") and function kernels (or simply "kernels")[93 ,99]. A kernel is a program executed virtually in parallel over a set of data streams, whereas a stream is a collection of data to be processed by a kernel.

The Compute Unified Device Architecture (CUDA) developed by nVidia is a particular example of GPGPU programming framework. A CUDA application consists of a code executed on a central processor (host) and of asynchronous function calls for kernels executed on a GPU (device).

The execution threads (or simply "threads") comprising a function kernel are grouped in an array of thread blocks, called a "grid". A block is an array of threads that are grouped together to be processed on a single SIMD multiprocessor (Figure 6a). All thread indexes are available inside the kernel functions via special type variables. Typically they are used to bind a particular portion of the common data arrays as individual input and output data streams for each execution thread.

A kernel is executed in CUDA memory model shown at Figure 6b.



a) b)

***Figure 6**: CUDA architecture*
a) CUDA application model; b) CUDA memory model

There are four types of variables according to the current CUDA memory model: local, shared, global, texture and constant. The local memory is a private memory space of a thread. Local variables are in general placed in the external memory, but can also be placed in the private register file of a multiprocessor. This allocation is performed by the CUDA compiler and generally cannot be controlled by a programmer. Global, constant and texture memory spaces are allocated in onboard memory. Constant and texture memories are the only spaces that are cached in the current generation of nVidia acceleration boards. Shared memory is the memory space allocated in on-chip storage local to each multiprocessor. The shared memory is accessible to all threads within a thread block and used for fast interthread communication within a block. The variables of all memory types except the local memory variables can be treated as references to some common storage allocations if seen from inside of a thread. Allocation of variables in different memory spaces is defined by a programmer via variable type qualifiers except local memory variables, which are classified as local by default.

3 Methodology

The methodology proposed in this chapter serves as a guideline for the GSC implementation. It describes the theoretical aspects of the technological workflow for bridging the gap between the mathematical abstraction of the computation task and the target implementation platforms. The concept presented in this chapter relies on the current state of the art in the fields of application specific hardware design and GPU programming.

The chapter consists of two separate parts reflecting the two-fold nature of the work. The major part is dedicated to the design of FPGA based systems, while the minor focuses on the peculiarities of GPU computing and related optimisation strategies. This disproportion reflects the difference in the amount of efforts normally required for implementing an algorithm in hardware and software.

As the matters addressed in this chapter are considerably capacious and cannot be fully covered in one chapter the problems are described briefly, and a number of references to the most remarkable supplementary materials are given for deeper contemplation.

3.1 FPGA-system design methodology

Since FPGA devices are prefabricated standard ICs, FPGA based system design mainly consists in the logic design, not going significantly deeper into the physical design phase[23]. This drastically reduces and simplifies the development cycle being one of the favourable features of FPGA design.

With growing complexity of digital systems, component reuse became an indispensable practice in design process. Although component reuse generally increases the economic efficiency of a project, it imposes stricter requirements on each individual reusable component and thus increases the initial costs of the component design [100]. Those requirements are higher adjustability, reliability, and thorough documentation of the components.

Although modern digital design addresses a variety of physical aspects of digital systems, such as power efficiency or signal integrity [101-103], this work primary focuses on the logic design process. A good introduction to system design on FPGAs can be found in [100, 104].

[23] FPGA design implementation includes elements of physical design, such as technology mapping, placement, and routing, but they can hardly compare with the physical design phase of, for example, full-custom ICs in complexity. Due to the prefabricated structure of the device these procedures are highly automated and require human assistance only in some critical cases.

3.1.1 General system design concept

3.1.1.1 Spiral approach to system development

The system design process lies in the gradual refinement of a system model. The process starts with the specification of general system requirements and constraints, providing the initial outline of the system. The system development path goes through a number of refinement stages of the system model down to a description level sufficient for hardware realisation by means of automated computer-aided instrumentation. Each refinement stage itself consists of a number of development cycles including a specification, an implementation, a verification and an analysis. These cycles form a top-down spiral like design flow (Figure 7).

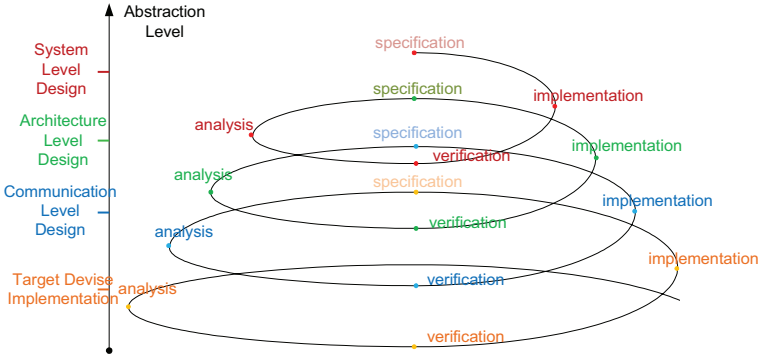


***Figure** 7: Top-down cyclic design flow*

The initial specification of design requirements and constraints is followed by the exploration of all possible solutions to the computation problem and the examination of available technical means for realisation of the producible system. The ideas are translated into a model or a number of models, constituting the initial implementation step. In general, implementation is a description process of the system at a certain abstraction level in compliance with the corresponding model specification. The implementation process is necessarily followed by verification, which examines the implemented design models for compliance with the specifications imposed on the system and determines the flaw source in case of inconsistency. The subsequent analysis steers the development process to a certain decision direction for the next design process iteration. The model analysis typically leads to the elaboration of additional design constraints resulting in an updated model specification for the next abstraction level.

The top-down design approach[24] is generally the only way for complex system development, as it enables a designer to develop global understanding of a design at early stages of the project. It reveals potential challenges and bottlenecks of a system realisation before these problems manifest in lower level implementation phases. This approach also serves as guidance for a designer in

[24] The top-down approach is not a strictly descending process. It permits both iterative cycles in design and early implementation phases for critical system elements to assure satisfiability/performability of a system within a given specification and design time constraints.

finding particular design solutions optimal for a given computing task in the multifactorial resource-performance-cost space.

The process of refinement of system components does not have to be concurrent and synchronous across the whole system model. The modern design techniques let different parts of a system be described at different abstraction levels in a single model in the same time. This implies that the main goal of the gradual refinement process is persistent consistency of a system model, avoiding hazardous ruptures in model development. This asynchrony gives a high flexibility in the project management allowing designers to concentrate on the most critical parts of a system and to work on different project parts in parallel and independently.

3.1.1.2 Three-phase view on system design process

The design process can be represented in another view, which does not abolish the spiral-like approach but split the development workflow in three coarse-grained phases characterised by their development purpose and design technique.

In particular the term *implementation* in the framework of on-chip system design implies the description of a system model on a particular technological and elementary basis subsequently followed by the physical design phase. All activities on higher levels of abstraction targeted to the proactive analysis of a design before the implementation are referred to as *modelling*. The *verification* term in this workflow scheme implies almost exclusively the functional verification of an implementation model, while the verification of other system features is referred to as the *analysis* in the design process terminology (e.g. *power*, *area*, or *performance analysis*). Distinguishing the functional verification as a separate workflow is due to the complexity of the functional verification problem requiring specific engineering skills and techniques.

The term of modelling here implies the process of the realisation of a design specification for the early analysis of significant design features. The goal of the modelling process is to approve the conformance of the results attained with a proposed computing approach with the required or expected results in terms of quality of computations (functional modelling). Another essential aim of the modelling is an estimation of the data-flow distribution inside a system.

The goal of implementation is describing the system at an abstraction level sufficient for automated hardware realisation by computer-aided tools. The implementation process is focused on the aspects of efficiency (i.e. power, area, performance, resource consumption) of precise realisations of particular data manipulations under given technological constraints. Implementation can be seen as a further refinement of models for preceding functional analysis. Implementation and modelling are influencing each other. The global data flow distribution pattern and the concrete computing methods selected during modelling substantially determine the structure of system components and interfaces between them. Equally, the computation and throughput characteristics of the implemented components and purchased IP cores may adjust or constrain the higher-level models.

The aim of the verification process is to check the consistency of expected and actual system responses. The challenge of the functional verification lies in the appearance of a consistency gap, which can occur when transitioning from functional to logic description of a design. The developer is not always able to take into account the side effects of the functioning of each discrete logic element separately, as well as the cumulative effect of these side effects on the overall functionality of the system. Therefore, the design intent realised by the developer and the resulting real functionality of the system do not always match. The verification process is insensitive to the result quality, the implementation efficiency of a computation solution, or other model aspects emphasised in the preceding workflows.

Verification models used as verification references typically interact with implementation models assisting the developer in detecting, qualifying, and localising the mistakes made during the design implementation. The verification workflow may have subordinate relationship to the implementation process if white-box or grey-box verification strategies, in which verification models are built in accordance with a particular design implementation [105], are adopted, especially in the case of assertion-based verification using the means of formalised design property specification. Equally, the verification workflow is connected with the modelling activity, since the system features are inherited directly from the model specifications at each abstraction level in the spiral design view when the system verification plan is built. Moreover, some high-level models can be taken as golden references in some verification strategies, or their infrastructure could be used for individual component verification. Inconsistency of the results may indicate equally a fault in a model under test or in a verification model or both. At the same time, the absence of inconsistency indications does not guarantee a fault-free design.

3.1.2 Modelling

3.1.2.1 Static and dynamic models

In digital system design static and dynamic classes of models for the preimplementation analysis can be distinguished [103]. A static model is a mathematical description of a system, which determines the relationships between a system feature of interest and associated prognostic input parameters. The main difficulty in constructing a static model is in determining a set of relevant significant input parameters and finding functional dependencies between them and the target feature. A dynamic or executable model describes temporal behaviour rules in a system. Using these models the features of interest are derived from statistical observations of system response. Therefore, the creation of a static model is a scientific problem, while creating a dynamic model is a technical task. Moreover, dynamic models have several advantages over the static models, which include higher accuracy, versatility, and practicality.

The following sections primarily focus on dynamic models due to more formality and determinism in their construction as well as higher relevance to functional logic design.

3.1.2.2 Model accuracy

For the model refinement process it is necessary to define the key model aspects and their resolution levels. A solid work on the generalisation of the electronic system development was performed under the aegis of the Virtual Socket Interface Alliance (VSIA), which elaborated a classification system for digital system models for the current technology level [106-107] shown in Figure 8.

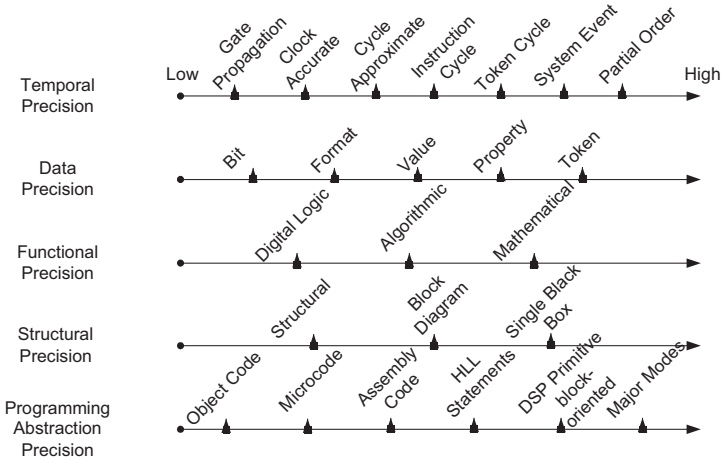


***Figure 8**: VSIA digital system taxonomy*

Since this representation is continuously being re-elaborated and refined (e.g. generalisation for electronic system level design proposed in [108]), four main constituents are recognised in this work for the accuracy specification of hardware logic design models, which slightly differ from the VSIA understanding:

- time accuracy determines the degree of the event ordering in a model; models may vary from untimed and approximate accurate (e.g. event ordered) models to models operating with physical signal propagation delays;
- data abstraction represents the information load carried by the data objects; models may vary from models operating with featureless data units to models refined to the signal level;
- computation accuracy defines how precisely a data manipulation method is described; the description may vary from abstract computation description stating only what result should be achieved without specifying the way how it should be achieved (e.g. functional languages) to a precise implementation of a selected method in terms of technological elements acting as an implementation constraint;
- topology precision defines the system's elemental abstraction recognising the function (role) of the elements in a system, detailing of the connectivity of the elements and the nature of the links; models may vary from opaque interface models, which provides only the service lists to the surrounding environment, to fine-grained placed and routed gate-level models.

The more accurate a model is the more efforts it requires from a designer to be implemented and the more resources it demands to be executed at a computer. Thus, for modelling more general and global processes in a system it is reasonable to stay at a higher level of abstraction. Equally, this implies that the costs for correcting a mistake resulted from taking a wrong principle decision in the design process can be the more tangible the higher the decision has been taken and the lower the mistake is detected. Moreover, getting more global knowledge of a system using models of a high detailing level often requires additional efforts for aggregating higher precision data and deriving the global level features.

3.1.2.3 Partial model refinement

At the early stages of a design process it is often required to make the characteristics of an individual component more exact. This may be needed for the forward analysis of a critical part of a design or for the integration of a new component to an already implemented design.

Gradually changing the abstraction level of the components is often used for the transition of the whole system from one abstraction level to another during the development process. It helps to avoid serious model inconsistency gaps that may be caused by a momentary transition of the whole design.

This kind of modelling/implementation technique is enabled by the separation of the interface and functional parts of the model components and by the detachment of the transport and computational infrastructure of a model.

3.1.2.4 Transaction-level modeling

The modelling technique that realises the transport and computation separation is referred to as Transaction Level Modelling (TLM). The TLM design is known to be a design concept on a level higher than the register transfer level (RTL) paradigm. It represents a system as a number of computation blocks (modules) interconnected with communication components (channels), which offer the transport medium for the message exchange (transactions).

TLM channels provide the communication interfaces to the computation blocks for message exchange. An interface in the TLM concept is seen as a set of services provided by method calls of a channel object. The types of the channel objects virtually define the abstraction level of the interface. A channel may provide several different interfaces even at different abstraction levels. In the latter case such a channel is called a transactor. The way a pair of interfaces is coupled is the matter of the internal realisation of the message conversion inside a channel object.

The bus functional model (BFM) is a type of transaction level models which has a special significance in modern design: it emulates the signal-accurate cycle-accurate activity of a communication channel via the mechanism of function calls thus providing means for cross-level interaction between synthesisable RTL and higher abstraction levels or for highly accurate transaction modelling.

3.1.2.5 Modelling analysis

Functional model

The earliest stage of modelling a system is the functional analysis. The functional model is constructed strictly aloof from possible implementation variants or target platforms in an infinite or flexible resource environment to reveal the highest possible potentials of computing or control systems or computation methods. The model is represented as a number of interacting processes, which communicate with each other via data streams, not focusing on realisation of those streams. Functional modelling can analyse the complexity and parallelisability of the data processing, the quality and accuracy of the computation methods, the interdependence of computational processes and the interactions between them.

Architectural model

At the next stage of modelling the system is divided into elements with clearly distinguishable functions: data-processing or control units, communication and storage elements[25]. The decisive criteria for designing this architectural model are the current level of the technological development[26], the project budget, the development time, the available IPs and reuse components.

Functional implementation of data-processing units is not necessarily accurate – the model can operate on symbolic objects, while the amount of data and the intensity of the flows in the system are more important for this analysis stage than the quality of data and the quality of operations performed on them. Component interfaces and communication channels do not necessarily have to be specified at the signal level – communication delays and latencies, as well as arbitration, may be approximate. Communication in this case is carried out usually at the level of atomic transactions, while data processing is described at the level of computation processes that are distributed over the functional blocks of system architecture (process partitioning).

An important task in the analysis of architectural models is in performing well-balanced functional decomposition, so that data streams are uniformly distributed in the system. Of a primary importance become evaluation and prediction of parameters of throughput of communication channels, as well as computational complexity and foreseen architecture of individual data processing blocks.

Interface model

A more detailed analysis of the system performance and the optimal distribution of information flows can be done using an interface model. In this case the component interfaces are defined on signal level. At this stage the question about the feasibility and effectiveness of the proposed communication system arises. This model is built to find out the precise values of the communication medium parameters, such as latency, delay, arbitration, and bandwidth, and even

[25] It is worth noting, however, that the memory elements may also serve as a communication means (shared memory, buffers).
[26] The technological development level is the integration level that defines the types and storage capacity of memories, available on-chip resources for building the communication system and implementing the functional blocks.

the questions of signal integrity and energy efficiency are addressed [103]. The importance of an early interface specification on signal level is reasoned on their strong impact on the hardware implementation of system components at lower levels. This means that an interface acts as a specification for a functional block. Early specification of interfaces allows independent and parallel development of individual components and simplifies the replacement of components in cases of revision of their hardware implementation.

At the interface analysis stage the internal implementation of the functional components are generally realised at the level of interacting processes that interact with the signalling interface through transactor function calls.

Component model and early implementation

Typically, after the elaboration and analysis of the interface, model a designer concentrates on interior organisation of the functional blocks, describing their functionality at the behavioural level (BL), which binds the functionality to signalling protocols. This type of model is a component model. The analysis of the model aims at the planning and optimisation of the data and control flows within the components and a more accurate valuation of the operating parameters of the functional blocks.

For the components critical to the hardware implementation, the architectural model may be refined with the early implementation of functional blocks prior to developing a complete interface model. This implementation is performed at the behavioural level or at register transfer level. The early implementation of the functional blocks should answer the question whether it is possible to implement the component with the given parameters of performance, area or energy consumption.

Similarly to RTL a behavioural model describes the timed cycle accurate functionality inside blocks, but hides some peculiarities of the hardware implementation and avoids the precise description of the structure details and the deterministic behaviour of every schematic element. The main advantage of the behavioural description comparing to RTL is that the RTL model may be significantly more time consuming both for the implementation and for the execution during simulation.

3.1.3 Implementation

While the modelling workflow can be considered as a search process for the optimal system organisation by the analysis of different system configurations, the implementation realises the selected approaches in a concrete element basis on a given processing platform and under strict operational constraints (resource, time, power, and constructive constraints). Thus, the technical constraints in the implementation workflow pop up to the foreground.

The modern implementation process is a description of the logic circuit that realises a computational method, rather than a description of the method itself. The translation of this

description to the target elemental basis, i.e. the synthesis[27], and the layout (placement and routing) of the design is performed automatically by CADs guided by the designer's implementation constraints input[28]. Meanwhile, EDA manufacturers do not leave the hope for inventing effective high-level synthesis means[29].

3.1.3.1 Register transfer level

The level of the circuit description mentioned above is the register transfer level. The RTL description paradigm implies that the state of any element of the system is explicitly determined at any time. This makes the RTL description advantageous for automatic synthesis.

A significant merit of RTL descriptions, comparing to the two lower level abstractions (logic gate level or transistor switch level), is its portability between target platforms with different elemental basis. The description at the RTL defines a device as a set of memory elements linked with data paths. The data paths are built out of combinatorial logic lumps, in which data transformations are performed. In this way a digital system of any complexity can be represented as a network of data propagation and transformation paths with intermediate storage elements scattered along them. This network operates under the control of finite state machines that implement the flow of the control over the data streams in the system.

When designing the system, the designer needs to take into account the propagation time of the signals along the data propagation paths and clearly define the complexity of the combinational logic between adjacent memory elements on this path. This is needed to ensure the data integrity in the system. In a synchronous design, which is an indubitable rule for FPGA designs nowadays, the constraint on the signal propagation time is determined by the clocking period (or its multiples).

3.1.3.2 Language-based description

Hardware description languages (HDL) are almost exclusive means for RTL description[30]. The main advantage of these languages is the capability to define the way the data are transformed inside the combinatorial logic nodes by a procedural description convenient for human perception. Other advantages of HDLs are the description flexibility (design templatisation and parameterisation), the data abstraction, and the modularity.

Hardware description languages can operate on both higher and lower levels of abstraction. The latter may be useful for the manual optimisation of the project, e.g. tailoring design to a certain FPGA family. The lower level description allows using directly the peculiarities of the die architecture by building up the design structure using basic architectural (technological) primitives.

---

[27] Synthesis in logic design in general is an automatic translation of a model description from one level of abstraction to another lower (not necessarily adjacent).

[28] A solid amount of literature is devoted to technique of the logical design and implementation on FPGA. That is why this topic is not covered in this paper in details but considered only in general its conception.

[29] To get to the level of progress in this area the reader is invited to refer to the following sources.

[30] Normally manufacturers offer and a graphical input for better perception for a human, but even the graphical input tools lately transform a graphical representation to linguistic for further elaboration and storage.

Such an approach can improve the project characteristics both in clocking frequency as well as in area occupancy. Yet this approach will reduce the portability of the design, so that it ought to be applied for the optimisation of critical parts.

3.1.3.3 Component reuse

The component reuse approach broadens the application area of components and reduces the development effort for a project, but imposes additional requirements to the design process. Naturally, the movement towards a greater universality of the component architecture generally leads to a drop in its performance. So to satisfy the demands of different systems special attention is paid to the tunability of components. With its use also the fast and easy adjustment of a component for the current design is enabled.

The component customisation is carried out either by a configuration, i.e. adjusting the functionality of a component while the system is operating, or by parameterisation, i.e. tuning the functionality of a component at a production phase. Both approaches have their drawbacks: a configurable component utilises more resources of a die, while a parameterisable component requires a separate production cycle. Of particular concern is the functional verification of such tunable components due to the variety of their operational modes.

3.1.4 Verification

Verification is a set of arrangements for post-implementation[31] analysis of the constructed model. It targets to the compliance checking between obtained and intended qualities of a design and the discovering of the potential flaw sources that led to inconsistency.

3.1.4.1 Functional verification

The central point in logic design is naturally taken by evaluating the functionality correctness. The fact that functional verification is an inevitable and complicated procedure for any digital design makes it most methodologically developed and formalised in the area of design verification.

Each step of a model refinement brings a new bulk of implementation errors. The amount of the accumulated errors can be so high that the verification of a model as a whole will not be feasible, because the localisation of the errors is not possible any longer. For this reason the verification strategy uses a divide-and-conquer approach based on the system's hierarchy.

This implies an intensive and separate testing of all components of the system, necessarily followed by integration tests, which will normally reveal unforeseen contingencies in the components interaction. Due to the increasing complexity of designs, the automation of the verification process is inevitable.

[31] Implementation here is interpreted in a broad sense meaning that in general verification can and generally should be done at any level of abstraction.

The importance of the role of the verification increases with a broad adoption of the component reuse approach, which requires exhaustive verification coverage of each individual component. This scrupulousness is reasoned by potential bugs in different component configurations, which didn't appear in the original design.

Theoretically two principle verification models can be distinguished in functional verification. White-box verification models are those that have access to the internal structure (signals) and are built with the assumption of a particular implementation of a device under test. In contrast, the black-box models are built without any knowledge of internal organisation of a component or a system and designs under test can be observed only through its interfaces.

These two models are antipodes in their qualities. Black-box models are reusable for any realisation of the design functionality. They can be treated as a golden reference while they express the pure functionality of a design without the influence of a designer's implementation approach. That cannot be said about white-box models that require changes in the testbenches after changes in the implementation. However, if the problem of observing and localising flaws is in the focus, the white-box model is the only solution. In practice a testbench is a combination of the two approaches.

Meanwhile, it is important to organise the verification environment in such a way that the means for functionality checking and flaw localising are separated from each other. It is crucial that the testbench for checking the functional correctness does not repeat the design implementation and implementation faults. Thus, this testbench should be written at another abstraction level to focus on the design intent and not on the way it is particularly realised. The localisation of mistakes is performed by checking the implementation model against the design specifications and characteristic properties of the system at the implementation level.

It needs to be noted in advance that traditionally the hardware description languages (HDL) have in fact conspicuous functional language trait. The characteristic description style of functional languages does not allow elegantly conveying dependencies between the states of a system, which is critical for the description of dynamic systems properties[32]. For addressing that problem special syntax languages, referred as property specification languages, were introduced and latter incorporated into modern hardware oriented languages. A solid portion of tasks targeted to flaw observation is currently being shifted to these languages.

The verification task requires the list of what is to be check during the verification both for feature checking and flaw localisation. It is essential to identify the scope of the properties to be verified and to put it down into a verification plan otherwise the verification process is likely to be mosaic and badly organised. The implementation of test cases for items in a verification plan forms a verification model, which can be refined and supplemented with new test cases during the design refinement.

[32] Although it is possible to do by deriving additional substances for property specifications. The most famous example for is Open Verification Library for property assertion that was written in pure Verilog HDL syntax.

By the criteria of subject in focus, verification tasks can idealistically be divided into task of three verification levels: application level, level of components and integration level. Tasks belonging to application level are focused on checking the functional correctness to the declared intent. Meanwhile, application level tasks do not necessarily imply functionality of the total system – the concept can be applied to subsystems or an individual component, the point is only functionality, i.e. black box approach. Tasks belonging to component level verification are focused on flaw localisation, i.e. implementation of a component, meaning white- or grey-box approach. This group of tasks requires most intensive coverage of a design properties, as it covers the aspects of system at the abstraction level of implementation, i.e. the exhaustive detailing of features of a system. Integration verification tasks combine two concepts of above levels. The verification process is organised as if the higher-level functionality is exercised, but the purpose of that is to check the correctness of components interaction, which could be, for instance, damaged by interface functionality misinterpretation inside a design group, or if some interface features were not taken into account during component level verification.

There are two major techniques for a design functional verification, which differ in the way they handle models under test. Static verification exploits formal mathematical techniques that do not require dynamic simulation of a model. The second is dynamic verification, which checks model behaviour by stimulating a system and observing its response over a time.

Verification plan

The verification plan is a document that aims to capture the scope of the verification problems and the prospective approaches to their solution. Often the task of creating a verification plan seems to be intuitive, but the formalisation of this process is a good practice for improving the efficiency of the verification procedure. The verification plan is the basis for formal property specifications, testbench design, assertion based verification and functional coverage. A good overview of verification planning and some methodological guidelines can be found in [108-109].

A verification plan is to be composed of the items that cover a set of properties that a device should have to meet the requirements of the specification. The properties need to be associated with methods that allow revealing these properties in the system. The methods, in turn, should be bound to the relevant attributes allowing the proposed methods to be realised. The attributes are abstract entities that allow the verification plan to be used independently from a particular device implementation and for models of different levels of abstraction. Finally, the attributes need to be associated with a set of implementation elements to bind the properties of the design to a particular implementation. This final step is not needed for the verification of application-level properties, as the application-level tasks can be isolated from the level of implementation with the transport stack of verification environment infrastructure (see below).

Verification problems can typically be discovered by analysing a design specification in general and model specifications at each abstraction level in particular. This analysis gives a feature set of a

design and a number of derived corner case conditions. A verification plan is typically refined during the design process as more design details appear. Thus, a verification plan composing is a dynamic iterative process.

In addition to properties that are directly defined in a specification, a verification plan needs to be complemented by implied properties. These properties are associated with the dynamic characteristics of a system. They describe the situations that lead to the transitions between the stationary modes and to exceptional situations. These properties are usually referred to as «corner cases». Corner cases are effectively transitions between system states that mark substantial changes in operational modes of the device. This is why corner cases are of special interest for verification as they allow the delicate behaviour for a system in transient mode to be checked.

Attributes themselves can be divided into two types: determinative and subordinate. Determinative attributes are the substances that determine the manifestation of the related properties. Auxiliary attributes are those entities that help to observe manifestations of the properties. They are attributes-markers. This division of attributes is important in dynamic and static verification as they play different roles in different kind of verification realisations.

Formal verification

Static verification is a relatively new approach and is far less spread than the traditional dynamic verification technique. Static verification basically includes model checking and theorem proving methods. These methods rely solely on formal analysis to demonstrate that certain features expressed as design properties in a specification language are implemented properly. Model checking methods comprise an exhaustive exploration of all states and transitions in a model and the search for a certain state. Theorem proving relies on mathematical and logical reasoning to prove the correctness of a design.

The major advantage of formal methods is that they can provide the complete coverage of a given property specification. Meanwhile, model checking approaches are bound by capacity constraints of the tools with respect to the amount of design states. These constrains limit the application of formal methods to designs of a certain complexity. To find more details on formal verification a reader may refer to [110-115].

Dynamic verification

Dynamic verification is the verification procedure that is carried out by monitoring the changes of the states in a system over time, which is aimed to cover all the characteristic properties of the system specified in the verification plan. The state's changes are driven by the external stimulation of the system model by a verification program (testbench). The observed changes are called the response of a system. Thus, dynamic verification is based on three basic concepts: stimulus, response and coverage.

In order to verify the correct functionality of a system, it is not necessary to check the response of a system to all possible combinations and sequences of stimuli. This is explained by the fact that not all groups and sequences of stimuli are relevant to the properties under examination. Even for the relevant stimuli there are indifferent sets of values that do not make any difference in system response. However even the number of meaningful combinations for complex systems can be so numerous that the problem of automating the stimulus generation becomes imminent.

Constrained randomisation and directed stimulus verification

In case of complex stimuli combinations the generation of values and sequences of stimuli is performed by randomisation mechanism. The underlying assumption is that if at a certain number of random input combinations are processed correctly all other combinations are handled faultlessly as well.

One remarkable feature of dynamic verification in contrast to the static technique is that design features not foreseen in the verification plan can be unintentionally exercised. In particular, intensive sequence randomisation creates unexpected conditions for pseudo-sporadic system events, which often cannot be identified for the verification plan. Thus, random generation helps to discover complex dependencies in a design, which cannot be deducted by a human.

However, not all inputs but those associated with determinative attributes of the verification plan are typical candidates for randomisation. The entities related to subordinate attributes may also be under randomisation to create uniqueness of identification markers.

The mechanism of constraint randomisation can improve the efficiency of the automatic verification by a more intensive input generation in certain value or pattern ranges that are significant for the manifestation of a given property. Constraint randomisation in the modern verification languages is implemented using declarative syntactic constructions similar to the description style of functional languages. They enable a designer to describe complex statistical regularities including their dependency on other variables and parameters. This cross-dependent randomisation is defined by equations or systems of equations, which are solved during runtime.

The more intensive the constraint randomisation is used and the stricter the constraints are, the closer the verification process is to directed tests driven by pregenerated values and sequence patterns. This type of verification is called directed stimulus verification.

Moving from lower to higher levels of abstraction there is a tendency to shift from random to directed verification, due to the simplification of the combination search.

Response prediction models

The automated stimulus generation implies in turn the automated checking of the system response. The generation of reference responses for comparison is performed by a reference model. The reference model can be specifically designed or an already existing model of the device can be used. In the latter case, such a model is called a golden reference.

The advantage of using a golden reference is that it is much less likely to produce incorrect reference response, since the correctness of the reference model can be presumed. In the case of a specially created reference model the probability for incorrect responses is generally higher. However, describing it on a higher level of abstraction greatly increases the reliability of the reference responses.

A reference model can focus either exclusively on the correctness of the functional transformation of data (important for data-dominated systems) or on the system operating details related to the order and the time of data transformations (important for control-dominated systems, interfaces, protocols, execution flows and data-path consistency). In the first case, the model is called the transfer function model. In the second case there is no established terminology yet, but the term *checker*[33] is likely to be adopted for this purpose.

Functional coverage

In the case of dynamic verification the question of verification completeness and thoroughness measurement arises. The problem of assessing the verification quality is solved by the mechanism of functional coverage of the implemented model. This mechanism provides means for the registration and the statistical counting of the number of exercised system properties gathered in the verification plan. The functional coverage is a set of behavioural patterns of a system, corresponding to the situations where manifestations of examined properties are expected. Those patterns are typically described as dependencies between the states of system over the time.

The mechanism of functional coverage allows feedback to a verification program for dynamical change of its execution flow or for taking a decision on the successful completion or the aborting of the verification due to the lack of functional coverage progress.

It is often that the application of property specification languages for the purpose of coverage evaluation is a highly convenient solution, which becomes an increasingly popular modern verification technique at the register-transfer level.

Assertion based design

The concepts of sequence[34] (relationship of states over time) and property (logical relationship between system states) are the basics of modern property languages. They describe relationships of states of a system or its individual components between and among each other, which allows an engineer to directly address the design intent in a flexible and expressive way and not the verification process arrangements.

In order to apply the properties to a system model, three basic operations are defined on properties: *assertion*, *coverage* and *assumption*. Property assertion means that the design has to comply with the rule formulated by the property. With the property coverage the manifestation of the system

[33] The term is used in modern open verification library, the standard TLM and SystemVergilog to describe the modules aimed at verification of temporal logic of a system. Meanwhile, the term has not yet found a precise definition.
[34] In particular, a sequence of states over time can be perceived as a state of a higher level, i.e. a macrostate.

behaviour described by a property is registered. Property assumption is used to constrain possible system behaviour to instruct static verification engines.

Despite all the obvious advantages of using formal specification of properties, modern specification languages are not a universal verification instrument, because the level of abstraction remains relatively low. The sequence and property specifications use Boolean operations on signal values and are bound to clock domains. As a result modern languages do not allow a developer to build a full-fledged verification model of a complex system exclusively. As practice shows, the most convenient exploitation of this vehicle is use it at the register transfer level for flaw observation and localisation and for functional coverage in combination with the classical verification technique at the higher levels of abstraction.

Verification environment infrastructure

The verification environment itself is a complex system. The increasing complexity of digital systems leads to a dramatic increase in verification burden. A complex verification environment requires a developed infrastructure that lets an engineer concentrate on the verification task itself and abstract from auxiliary underlying means. It allows the ease of maintenance and modification, the ability for tests and verification environment components reuse. The object-oriented approach augmented by the abstraction layering of the verification environment is the technique that addresses these objectives.

A typical verification environment may be represented as a pile of the DUV integration, the communication stack, the scenario layers and the layer of tests. Environment layering solves the most important requisite for a complex verification task – focus on a problem. Each layer is isolated by transactors that allow an easy exchange of the verification components at one layer without influencing the rest of verification environment.

The verification environment nomenclature includes the following types of infrastructure components: drivers, monitors, checkers, analysers, generators, coverage monitors, and testcases. The first two types of components perform transaction disassembling and reassembling to forward them down or up between abstraction levels. Checkers are responsible for the design properties check within the same abstraction level. Analysers are components that perform the comparison of the expected and the actual design response. Generators are the agents that generate various behaviour scenarios (sequences) that are targeted to cover different corner cases. Coverage monitors guard the verification progress. Finally, the testcases are the agents that generate device exercises for different functioning mode testing.

The lowest layer of the environment is the DUV (device under verification) integration layer represented by the transport layer. On the top of the DUV integration layer there is the transport layer followed by the functional layer, the scenario layer and the testcase layer on the top of hierarchy. The DUV integration layer implements means for device model integration into the verification environment. The components of this layer are expressed at the same level of

abstraction as the device model external interface. Important components of this layer are checkers. The transport layer performs the translation of data objects on the higher abstraction level (functional level) to objects of the lower abstraction level (integration level) and vice versa. This layer necessarily contains drivers, monitors and optionally protocol checkers. The functional layer typically contains analysers, the scenario layer generators and the testcase layer testcases.

Introduction of concepts of object-oriented design and specification properties allowed a significant breakthrough in verification techniques. All major market participants focused their attention on the development of methodologies for the design of the verification environment. In addition to developing the concepts, the industry members make serious efforts in designing standard template libraries for enabling the proposed concepts [116-118].

3.1.4.2 Timing verification

Although timing verification goes beyond the frame of the logical design, it is of special importance for checking the functional correctness of a system, as signal propagation delays in real circuits have a crucial effect on the design functionality. Like in functional verification the timing analysis may be performed with dynamic and static methods [119-120]. The timing verification is applied to netlists (gate-level models) of a design with delays assigned to each contributing element by EDA tools during a back-annotation process. Those delays may be specified in a separate file (e.g. in the Standard Delay Format [121]), so that the netlist file is not additionally modified. Those models can be very accurate, allowing the specification of all critical timing characteristics for different types of electronic components such as sensible signal pulse width, trigger setup time, signal hold time etc. The functional correctness of the design in dynamic verification may be verified by applying the same test cases as for functional verification during simulation. In static timing analysis process back-annotated models are checked by comparing each path delays against a set of timing constraint. Dynamic timing models are capable for a higher accuracy and are more universal (applicable for both synchronous and asynchronous designs), static verification requires additional effort from the designer for constraints specification. Static timing analysis took strong position in timing verification and often dominates in FPGA development due to its high verification coverage and high level of observability with sufficient accuracy. Meanwhile, considering how high can be the cost of a flaw both approaches are normally used together.

3.2 GPU implementation optimisation

A CUDA program consists of execution sections of two kinds: sequential code sections are interleaved with calls of massive parallel computing kernels. It is important to emphasise that the parallel kernels require massive data parallelism; otherwise, the costs for organisation of computations may exceed the profits from parallel execution. Furthermore due to the hardware multithreading organisation of modern stream processors, the workload intensity of the thread cores

influences the density of the processed data streams. The maximal utilisation of the available bandwidth of the global memory bus indicates an optimal utilisation of stream processing systems.

Therefore, taking into account the peculiarities of stream processing architectures, mapping an algorithm on a CUDA platform requires the identification of computation subproblems, which satisfy the criteria of an explicit data parallelism, or can be turned into parallel algorithm without loss in computation accuracy.

Parallel algorithms and algorithm transformation techniques can be found in many works, e.g. [122-125]. In particular, a summary of parallel algorithm patterns for GPU computing can be found in [96].

Even if massive parallelism is picked out, it does not guarantee high performance of the system, if the massiveness of concurrent execution leads to a deficit of common resources, which can result in access conflicts. Mechanisms for resolving massive access conflicts may be time consuming and dramatically lower down efficiency of the parallel execution. Thus, a careful resource planning is the central issue of a parallel programming methodology. Assuming architectural peculiarities of stream processing systems one may consider three most obvious aspects to concern (in order of priority):

- efficiency of global memory bandwidth utilisation;
- conflict-free multiprocessor shared buffer management;
- minimisation of SIMD execution path resulted from branch divergence.

Many performance sensitive aspects, such as local variables allocation, memory traffic planning, kernel execution scheduling, etc., are not controlled by the developer but done by the compiler and the underlying hardware dynamically. This makes a careful optimisation on assembly language level not very effective for CUDA platforms. Meanwhile, a developer is able to assist the compiler and hardware to optimise the task execution.

The optimal execution configuration for better application performance is the matter of heuristic search and can be hinted by profiler information. More specific recommendations on the device configuration may be found in technical documentation for a device [98, 126].

3.2.1 Global memory throughput optimisation

Achieving efficiency in global memory bandwidth utilisation implies the minimisation of the global memory bus traffic and the maintenance of a constant data flow to hide the memory access latency.

One principle approach to reduce the traffic in stream processing systems is buffering the data to the local multiprocessor memory. Data are loaded to shared memory, processed locally without generating external memory bus activity and forwarded back to the global memory.

Another possibility is to exploit the cached memory spaces, the constant and texture memory, where applicable. The texture cache has a special feature inherited from graphic applications. It may be

configured to optimally operate with two-dimensional spatial locality, which can be helpful for some general purpose applications.

Automatic variables of a thread are allocated by a compiler and could be placed either in register file or in global memory (typically large arrays or structures). The later can influence an application performance. That is why some local variables could be thought to be placed in some specific memory types, e.g. in shared memory or cashed constant or texture memories.

There are two points that should be mentioned about the organisation of the memory transactions in nVidia GPUs, which may seriously influence the performance of data-intensive systems.

The memory controllers typically operate with some definite formats of machine words. In particular nVidia GPUs operate with 32-, 64-, and 128-bit words. This means that if data is not aligned in memory to the borders of these words, it is possible that extra memory access cycles will be generated. Therefore, one can increase efficiency of memory bandwidth utilisation by alignment of data structures to the word borders.

The second point is in the way how memory accesses are generated for simultaneously executed threads within a multiprocessor. The bandwidth of modern synchronous dynamic memory most effectively used in burst accesses. To exploit this feature a CUDA device memory controller can detect coalescing memory accesses within a group of simultaneously executed threads in a multiprocessor and organise a single memory transaction of 32, 64 or 128 bytes. The extent of this capability is dependent on the different hardware platform of NVIDIA.

A better memory layout for taking advantages of coalesced memory accesses can be achieved by plain data structure organisation, e.g. by reorganising an array of structures into a structure of arrays, which is a common technique for SIMD systems.

To maintain a continuous data stream between the global memory and the multiprocessors, the multiprocessors should be inhabited with a sufficient amount of computation threads to hide the memory access latency via mechanism of computation context switching. The main limitation for the number of threads concurrently executed on a multiprocessor is the total amount of resources shared by all threads.

3.2.2 Shared memory

The shared memory, as a resource common for all the thread cores in a multiprocessor, may cause a concurrent access conflict. It is organised in several banks, in which the number of banks corresponds to the number of threads simultaneously executed in a double core clock cycle, i.e. half warp for CUDA device capability 1.x. If several simultaneously executed threads try to access different locations in one bank, these access are serialised. Simultaneous accesses to different banks are conflict free. Concurrent read accesses to the same location may be broadcasted. Concurrent write accesses will generally lead to a single write by an undefined thread. Thus, special attention should be paid to the organisation of data in the shared memory.

3.2.3 Dynamic branching

Due to the SIMD organisation of multiprocessors an ideal algorithm for a stream processing architecture is an algorithm that allows the same execution path for all threads in a warp. Thus, the algorithm and its program implementation should be tailed the way that the number of branching (including loop conditions) is minimal. Meanwhile, if dynamic branching is inevitable the following guideline should be minded to minimise the cost of divergence:

- branches should be primitive, i.e. without subordinate branching nodes;
- branches should be short;
- alternative branch should be idle, i.e. should lead directly to a join point.

Those requirements can often be fulfilled by rebalancing the control flow graph of an algorithm and operations drifting.

More detailed information on optimised application development on stream processing architectures can be found in [79-80, 96, 99, 127-130].

4 Hardware implementation

4.1 Functional analysis

4.1.1 Reference software implementation

The software sequential implementation [3] described below is the reference point for the parallel GSC implemented on two massively parallel processing platforms. It has been used as a base for performance and quality comparison, for the experiments with different algorithmic approaches in different GSC processing phases, and for statistical measurements.

4.1.1.1 Data structure

The basic data element of the algorithm is a so-called *code element (CE)*. A CE_i[35] provides an efficient way to store information about a *region* (R_i) at a *hierarchical level i (HL(i))* and its relative location in the hierarchical island structure. It contains information on the mean grey value of the region, the number of covered subregions R_{i-1}s and their positions inside the island (I_i) the region belongs to. A *code element* CE_i also contains the addresses of its subregions' CE_{i-1}s and the addresses of its two parent regions' CE_{i+1}s in main memory. With these pointers the *CE* structure is ready for building two-directional link lists, which represent hierarchical relationship trees. These link lists form a so-called *GSC database (DB)* that represents the total hierarchical island structure (Figure 9). The roots of the trees contain information on the mean grey value of the segments of an image, whereas the leaves correspond to the coded regions R_0 formed by the pixels that comprise the segments. A region's *CE* does not have a fixed size in the software implementation, as the number of subregions the region may cover is not limited. The number of the newly produced regions in an island is not known a priori as well. This implies that the total number of regions is not known in advance. For this reasons the database is operated as a dynamic data structure, using an indirect addressing scheme to provide a compact and lossless storage form. This indirect addressing is realised by the means of a sequential key table that carries the topological information of the hierarchical island structure. Each island of each hierarchical level is

[35] The first subscript index for the GSC data element abbreviations always indicates a hierarchical level number, unless otherwise stated.

represented with one entry in the key table. An entry stores the start address of the first region and the length of the total region coding of an island in the region database.

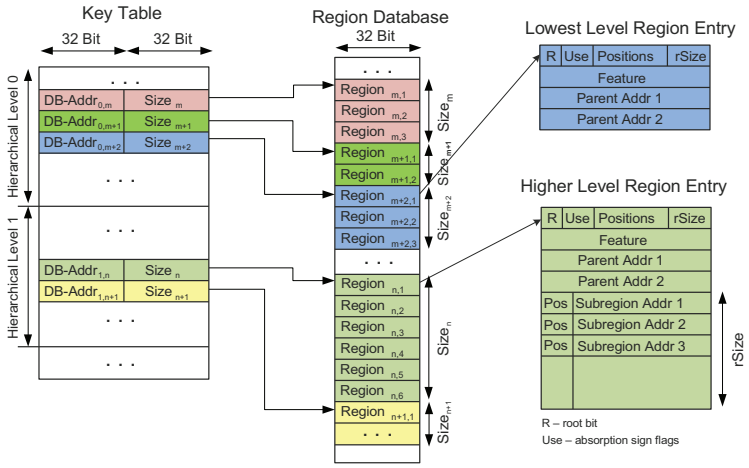


***Figure 9**: Software GSC database structure*

4.1.1.2 Implementation of the different GSC phases

The software implementation tries to form new regions inside islands by going sequentially island-by-island through all hierarchical levels in the *coding* and *linking phases*. During the initial *coding phase* similar and neighboured pixels are grouped into *initial regions* R_0. The process is realised as a recursive graph traversing applied to the neighbouring pixel graph of an island. The similarity measure is defined by the grey value distance between two pixels, which has to be smaller than a selectable threshold.

During *the linking phase* subregions R_{i-1} of a level *i-1* are grouped (linked) to form regions R_i of a level *i* for all hierarchical levels. The linking is performed recursively in a similar way as in *the coding phase*, except for applying the additional region overlapping criterion instead of using the explicit pixel graph relationship to detect the neighbourhood condition. In an island I_i of a level *i*, comprising seven subislands I_{i-1} of a level *i-1*, subregions R_{i-1} are joint, if they are similar and overlapping. To detect the overlapping condition every sub-subCE (CE_{i-2}) of a currently linked subCE (CE_{i-1}) is accessed using downward *DB* pointers and checked for existence of a second parent (upward *DB* pointer). If the second parent exists and the corresponding subregion is located in the current island I_i they are treated as overlapping regions. Therefore, to get the overlapping subregion partner for similarity comparison the method needs two database redirection accesses.

Several linking strategies have been implemented for measuring their impact on the segmentation quality. These linking methods differ in the region mean value computation, the linking starting point in the island topology (the centre or a border node), and the direction of region growth. The summary of the studied linking methods is given in Table 1.

***Table 1**: Linking method summary*

Linking Method	*Simple*	*Centroid*	*Weighted*	*BestFit*
homogeneity criteria	$\lvert g_i^{(h)} - g_j^{(h)} \rvert \le t$	$\lvert g_{act}^{(h+1)} - g_j^{(h)} \rvert \le t$	$\lvert g_{act}^{(h+1)} - g_j^{(h)} \rvert \le t$	$\lvert g_{act}^{(h+1)} - g_j^{(h)} \rvert \le t$
mean feature value	$g^{(h+1)} = \frac{\sum_{i=1}^{N} g_i^{(h)}}{N}$	$g^{(h+1)} = \frac{\sum_{i=1}^{N} g_i^{(h)}}{N}$	$g^{(h+1)} = \frac{\sum_{i=1}^{N} s_i \bullet g_i^{(h)}}{\sum_{i=1}^{N} s_i}$	$g^{(h+1)} = \frac{\sum_{i=1}^{N} s_i \bullet g_i^{(h)}}{\sum_{i=1}^{N} s_i}$
neighbour analysis sequence	$v_0 : v_1 \ldots v_6$ $v_1 \ldots v_6 : \circlearrowright$	$v_0 : v_1 \ldots v_6$ $v_1 \ldots v_6 : \circlearrowright$	$v_0 : v_1 \ldots v_6$ $v_1 \ldots v_6 : \circlearrowright$	$\uparrow \lvert g_{seed} - g_{neighbour_i} \rvert$
characteristic features	– susceptible to chaining error – low implementation cost	– reduces chaining errors (comparison against intermediate region feature)	– correction of mean value by weighting with size of linked subregions	– most similar candidates linked first – mean value corrected by weighting – highest implementation cost due to candidate sorting

$\circlearrowright$: clockwise analysis of candidates $g_i^{(h)}$: feature of a region i on hierarchical level h N : number of regions linked t : threashold
s_i : number of subregions of a region i $\uparrow\lvert x \rvert$: accending order according to feature distance $g_{act}^{(h+1)}$: current feature of the region on level $h+1$

The peculiarity of the *Simple Linking* method is that it is based exclusively on the similarity of the original features of overlapping subregion pairs, without considering the intermediate mean feature of the currently linked region as it is done in the other linking strategies. Therefore, this method may lead to a so-called linking error, if the mean value constantly shifts from the initial region value during sequential linking (chain effect).

The *Centroid Linking* tries to avoid this linking error by comparing the mean value of already joint region against the value of a candidate subregion to be linked. The mean value of the region is calculated as the arithmetic average of the included subregions.

A more sophisticated method for computing mean values is introduced in the *Weighted Linking* where the contribution of a newly linked subregion is additionally determined by the size of the subregion, i.e. the larger subregion has the higher feature weight in computing the new feature of a region. The size of a subregion is determined by the number of sub-subregions it contains. The introduction of this method increases the accuracy of linking.

The methods described before analyse the neighbouring subregions in a predefined search order. The first candidate that matches the threshold condition in the clockwise search is linked to the region first. In the *BestFit Linking* the growth direction method is further refined. The method chooses the most similar neighbour (the lowest feature difference) to be connected first. Using this method provides a higher separability of segments (less faulty connections).

Due to the chain errors the *Simple Linking* may lead to forming large fused segments. On the contrary, the *Centroid Linking* and the *Weighted Linking* provide more differentiated segments. In its turn the *BestFit Linking* improves the segment boundaries and, because of the ascending neighbourhood feature sorting, leads to more accurate segment features.

The linking methods have been qualitatively rated with the multi class error measuring the correct matching of classified segmented images with corresponding reference images [3]. The best results were shown by the *BestFit Linking* approach, while the worst were shown by the *Simple Linking*.

The experiments indicated as well that the start position for the linking procedure matters: starting from the central node of an island is preferable.

Generation of segmented images is performed during *the result generation phase* by traversing the hierarchical relationship trees from the lowest level regions R_0 up to the roots of the trees. When a root is found the related segment feature or the segment label is assigned to the correspondent pixels of a region R_0. The image coordinates of the pixels are unambiguously determined by the topological coordinates of the island I_0 the region belongs to and the information on relative positions of the region's pixels within the island coded in the region's CE_0.

Region splitting is implemented in two generally different variants. The first approach is to perform the splitting during the linking (two methods are realised in this way – the *Simple* and the *Complex Splitting*). Splitting is initiated whenever a splitting condition is detected: the method tries to separate all underlying subregions recursively down a hierarchical region tree. This approach preserves the spatial contiguity by modifying region relationship information in the database. The method is recursive and notably resource intensive. The second approach (referred to as the *BestFit Splitting*) tries to avoid implementation overheads by ignoring spatial contiguity of segments and by operating without database modification. Assigning particular segments to dubious initial regions is done during the result generation by simply selecting the most appropriate route to a tree root.

The comparative evaluation of segmentation quality of the two approaches does not reveal an unequivocal preference for one of them. Considering the qualitative aspect the approach in which splitting is performed during the result generation may perform worse than those in which it is performed during the linking. This can be reasoned by the fact that on higher hierarchical levels a region can represent a very large area of a data set. Therefore, a simple assignment of subregions to false segments without global view may lead to bad segment surfaces or even to spatially disjoint segment fragments.

Nonetheless, in reality the segment fragmentation occurs relatively rare. As it could be seen from the measurements, the number of decaying segments did not normally exceed 0.1% of all segments in 2D. Moreover, the correction of the fragmentation may partially be carried out at a postprocessing stage, e.g. during subsequent classification.

Additionally, the quantitative evaluation based on multi class error measurements for real scenes (as opposed to artificial phantoms) even showed the results that are contrary to the theoretical reasoning, giving the preference to splitting during the result generation.

The software model examines also approaches to pixel splitting and singularity correction. The need for splitting pixels arises from the fact that two pixel regions of neighbouring islands may overlap in one common pixel; hence, the pixel needs to be decided for one of the regions. This can be realised by applying the similarity criterion to the original feature of the pixel in the initial image and the features of the two rivals. The singularity correction also operates on features of pixels from the original image, but it compares them with features of nearby elements of the result.

The splitting methods described earlier only work on regions in the *DB*, thus R_0s may still overlap in pixels after the conventional splitting. However, the *pixel splitting* can be realised during *GSC DB* generation by expanding the *Simple/Complex Splitting* to the pixel level resulting in deletion of the dubious pixels from the less similar R_0s in the *DB*. The alternative approach implies deciding between rival segments during result image generation after the *DB* has been completely formed. The measurements show the better segmentation quality when the second method is applied.

The need for correcting singularities appears if not all pixels are included in pixel regions in *the coding phase.* Meanwhile, the classical task of segmentation requires an image to be fully covered by segments. One possible approach to the problem is in eliminating singularities directly in the DB guided by the GSC island topology. Alternatively, this problem can be solved after result generation by assigning the segment features of most similar neighbour pixels to the singular pixels using its features in the original image. Due to the less degree of freedom in neighbour analysis the database method leads to worse quality results, having higher computation complexity at the same time.

4.1.2 Study premises

The primary intention for the research work on the GSC is to explore the potentials for implementing it on high performance massively parallel computing systems of expansion board class and images of up to 2048^2 pixel resolution with the value depth of eight bits. For examining the scaling capacity of the GSC method for images of very high resolution the study is expanded on the 4096^2 pixel images.

A device with programmable structure of FPGA type and a streaming processor of GPU type are considered as the computing platforms for realisation of the segmentation method.

4.1.3 Algorithm analysis for computation parallelism

The GSC has a strong potential for parallelisation. At *the coding/linking phase* all islands of the same level can be processed concurrently. Data dependency exists only between two superimposed hierarchical levels. Therefore, the whole *coding/linking phase* can be seen as a pipeline, each stage of which is correspondent to a hierarchical level of the GSC topological pyramid (*GSC LP: GSC Layer*[36] *Pipeline*) and consists of processes performing linking inside the islands of this hierarchical level.

Nonetheless, even an approximate assessment of the volume of the total hierarchical data structure for the claimed resolutions shows that implementation of the complete GSC pipeline in a structural representation is not possible for the current integration level of ICs. Moreover, the stages of the *GSC LP* correspondent to the lower levels of the hierarchy are not realisable on a die as a whole. This generally means that each layer should be processed in parts. Fortunately, the process of linking is similar for all islands throughout the hierarchy (except the pixel level). Hence, the same

[36] The term *layer* with respect to the GSC pyramid is introduced to convey connotations with the amount of data associated with each hierarchical level, whereas the term *level* carries a sense of hierarchical relationship; in some context both terms can be equivalent.

hardware can be used for implementing *the layer pipeline* without a significant reconfiguration[37], while the intermediate results of *the linking phase* are to be stored in the GSC database on external storage. Using the same hardware for linking different islands in parallel implies that *the GSC DB* has to be a shared resource with symmetric access for all concurrent linking processes.

Therefore, the data stored in the GSC database should be protected from data inconsistency caused by simultaneous access by different concurrent linking processes. This can be realised either by arbitrating the memory accesses or by assigning each island in the hierarchy a specific data location. Due to the complexity of the arbitration task for scalable massively parallel processing the second approach is preferable for exploiting in the GSC implementation. This means that each hierarchical layer in *the GSC DB* should be realised as a regular array with constant-length elements.

The linking can be divided into two classes. In the first one, the region[38] inclusion depends on threshold condition satisfaction for two neighbours exclusively, noted as the *Simple Linking* above. This means that the threshold criterion can be examined for all neighbour region pairs independently. The sequence and the starting point of linking do not influence the result of linking. This simplifies the combinatorial function for concurrent region creation within an island and makes this class attractive for structural hardware realisation for small spatial extensions, such as pixel islands.

The second class implies a stepwise region linking, interpreting a region as a chain of subregions, for which a decision on subregion inclusion is taken at each step on the basis of an intermediate region mean (*Centroid*, *Weighted*, *BestFit Linking*). In this case parallelisation can be efficiently applied to simultaneous analysis of all possible neighbour candidates for linking at each linking step in a chain. Simultaneous stepwise growth of several regions from different starting points in an island is feasible as well (multithread linking); however, the starting point of growth influences the segmentation quality, as mentioned before. Additionally, the problem of competing for subregion absorption between any two linking threads may arise. As the experiments showed, the need for conflict resolution between processes makes the approach inefficient at implementation. Moreover, the longer runtime for single-thread linking can be compensated by processing islands in parallel.

The neighbourhood criterion, which is at higher levels defined as regions' overlapping, is substantial for the linking procedure. The initially proposed linking algorithm for the software implementation realises the overlapping detection of regions by analysing all the subregions of a region for existence of a second parent. Thus, a linking process needs to have access to the islands of two hierarchical levels below the current. This approach is not attractive for implementing on massively parallel processing systems due to excessive data access[39]. The amount of data forwarded to a linking process can be significantly reduced by retrieving only the specific information that is exclusively relevant to the linking procedure (Appendix A.2.1). These data need to provide only the

[37] This can be applied both to HW and to SW implementations as a program of an instruction-driven processor can be treated as a configuration series for underlying hardware.
[38] For the pixel level islands pixels can be treated as primitive regions of the ground level.
[39] This approach leads to higher demand for local memory for data prebuffering, which becomes crucial in multiprocess execution schemes, or to sporadic memory access to external storage otherwise.

information about the overlapping subregion pairs within the islands of the current level (connectivity data). Such detection of the overlappings can be performed during the linking in neighboured islands on the previous hierarchical level by exchanging data between parallel linking processes.

To implement this interaction the concurrent linking processes can exchange the information about the resulting regions via shared structures referred to as *hierarchy relationship structures (HRS)*. These structures correspond to the subislands I_{i-1} being the overlapping nodes for adjacent islands I_i of the current level. An *HRS* can be best implemented in a form of an associated array with the number of entries equal to the number of subregions R_{i-1} in a subisland I_{i-1}. If a subregion R_{i-1} belonging to the subisland I_{i-1} is linked to the newly created region R_i, the information about this region R_i should be recorded to the correspondent entry in the shared structure in association with the R_{i-1}. After both neighbouring islands I_i are processed, the information in the *HRS* is analysed to identify *the overlapping region pairs* (ORP_i). The results are recorded on a global storage structure that contains one list of all corresponding overlapping pairs per island I_{i+1} of the next hierarchical level. These lists are referred to as *overlap-list structures* (OVL_{i+1}). At the same time, to make the massively parallel processing of the *HRS*s efficient it is reasonable to allocate an individual storage for each overlapping node in the lattice ensuring the data consistency in the common storage for parallel access. The data structure for storing all the *ORP*s associated with an overlapping node in the global storage is called *an overlapping node entry (OLN)*. This implies that the connectivity information for each next hierarchical level should be organised in a regular array of fixed-sized overlap-list structures OVL_{i+1} with fixed offsets to each overlapping node entry OLN_{i+1}. The number of overlapping nodes in a macroisland is fixed and equal to twelve as shown in Figure 10a.

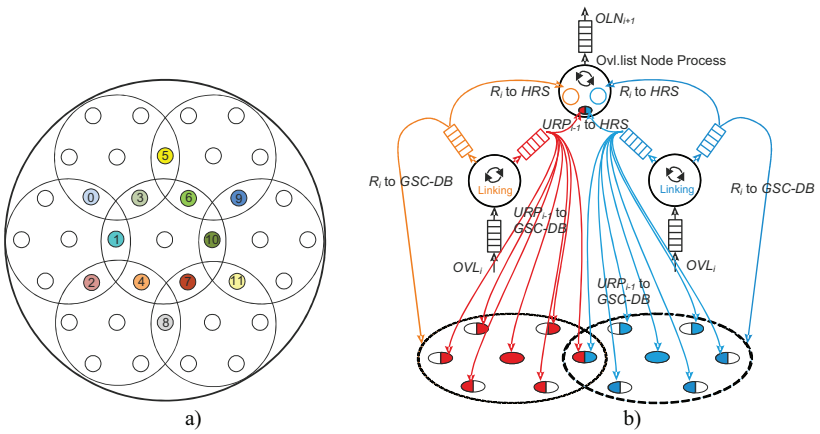


***Figure 10**: Linking data-flow model*
a) Fixed mapping scheme of overlapping nodes in a macroisland; b) Data-flow model of linking process

The linking procedure also has to establish relationship between the regions of two superimposed hierarchical levels to form the forest of segments' trees. There are two possible types of the

relationship: *parent – child* and *child – parent*, which are effectively equivalent for the parallel GSC method. However, the parallel method on the regular *GSC DB* structure does not really require an absolute addressing scheme to be applied for maintaining this relationship inside the GSC database, as it exploits the topology of the relative placement of islands within the pyramid. In this case the *child – parent* relationship can be described more laconic because it requires referencing a parental region within a parental island only, while establishing the *parent – child* connection additionally requires the child island to be specified within a parental macroisland. Moreover, processing expenses for establishing *child – parent* connections are minimal as the same upward relationship is effectively used for detecting overlapping pairs in the overlap-list creation process. Hence, the same *HRS* structures can be used for all subislands I_{i-1} (including those that correspond to the central nodes of islands I_i) in linking procedure for establishing hierarchical relationship between regions.

Summarising the theoretical reasoning above the linking procedure coupled with the overlap-list creation can be represented as the following data-flow model shown at Figure 10b. A linking process takes the overlap information stream of OVL_is in and produces two data streams for two adjacent levels: the region information for the current level R_i and the *child – parent* region relationship information for the next lower level in the form of *upward region pointers* (URP_{i-1}). These input and output flows of different linking processes are independent within a single *GSC LP* stage. At the same time the output flows of a single process have no feedback to the process.

The overlap-list generation process consumes the output flows of two linking processes of two adjacent islands I_i and generates an output stream with the connectivity data for linking on the next upper level (*OLN*s for OVL_{i+1}s). Such output streams are also independent within an island layer. Equally, the overlap-list generation processes have no feedback dependencies on their own output streams.

After the *GSC DB* is formed, there exit two possible approaches to generating a segmented result. They are the upward and the downward tree traversing. *The result generation phase* can be realised as simultaneous segment formation by traversing hierarchical region trees concurrently from the leaves up to the roots in the GSC pyramid. However, for high resolution pictures the simultaneous traversing of all upward traces in the forest is not practicable. Thus, the leaves labelling has to be done in parts by a limited number of concurrent processes.

One feasible approach to the parallel leaves labelling consists in assigning sets of nearby leaves to a pool of traversing processes for root tracking. This approach may benefit from the topological relationship of islands in different hierarchical layers, as the data from the upper layers may be scheduled for prefetching and shared among the whole pool of processes due to the persistence of the spatial locality of islands throughout the hierarchy. This may raise efficiency of data access. However, the data scheduling and data distribution among the processes throughout the whole hierarchy is a complex task due to reconvergent structure of the pyramid. Additionally, due to a wide variance in the trace lengths from the leaves to their roots, the problem of efficient load balancing across the total pool of tree traversing processes arises. The dynamic leaves distribution

among the processes, when a new leaf is assigned to a process directly after the process has finished its previous search, might increase the overall performance. However, the dynamic approach is bound to the problem of arbitrating the leaves' distribution among the processes. Moreover, the independent parallel forest processing without an intermediate process synchronisation makes the data access policy eventually inefficient: many traversing processes may enter the same traces multiple times lacking for data sharing capability as the data scheduling for the total pool becoming unrealisable in the dynamic scheme (in contrast to synchronised leaves distribution, where neighbouring processes may plan accesses to the upper level tree nodes and share the data). Taking it all together it can be seen that the bottom-up approach is not attractive for parallel implementations.

An alternative to bottom-up tree traversing is the result generation by downpropagating the root values to leaves. Downpropagating a root value by tracing a complete hierarchical relationship tree in a single pass by a dedicated process is not favourable for parallel implementations due to the recursive nature of the algorithm. Alternatively, the downpropagation can be realised in a layer-wise manner, in which all values from an upper layer are propagated down to the next lower layer at each iteration. This approach benefits from a plain data scheduling and distribution scheme as it works with two neighbouring layers only, which makes the topological relationship patterns between parental and child islands rather simple. The problem of dynamic load balancing is not relevant for this method, as the processes do not stay idle for long if they have no current data to process. Although this method does not solve the problem of maximal computing capacity utilisation over the whole hierarchy, it makes load balancing more efficient due to a significantly smaller quantum of computation (a single island versus a tree). The obvious downside of this method is the need for write operations on global memory to the correspondent regions for downpropagating the values. Nonetheless, the systematic layer-wise downpropagation reduces the number of accesses to global storage almost twice compared to the bottom-up approach as experimental results showed. Together with its computation simplicity this makes the downpropagation approach most favourable in view of performance efficiency. Moreover the method has the qualitative advantage in comparison to upward traversing approach as the decision for a parent is made based on more global segment view (i.e. closer to roots).

During result generation the initial content of the GSC database can be modified with downpropagated segment information (*the GSC DB modification implementation*) or a separate temporary island-oriented data array can be used to store downpropagated segments as an option (*the temporary segment layer implementation*) depending on the memory resources available. Region hierarchical relationship information does not need to be retained after segment assignment and can be removed or overwritten in the case of the GSC DB modification (*GSC DB decay*).

In general[40] a downpropagation process, viewed as a data-flow model, consumes two types of data streams from two adjacent hierarchical levels: one from a current level (the target for assignment)

[40] A special case appears when processing higher resolution images with the particular data format used in the HW GSC implementation. See Section 4.2.2.

with region features and parental pointers and one from the upper level with propagated segment values. It produces one outgoing stream for the current level with the propagated values, which may (but not necessary have to) modify the GSC database. The data flows into and out of the processes are independent within one hierarchical layer (*a downpropagation stage*). Equally, the input and the output streams of a single process are decoupled.

4.1.4 Data organisation

For raising efficiency of parallel processing each hierarchical island is assigned a fixed location in the GSC database for storing data associated with the island. This allows the linking and the result generation processes to work on individual islands[41] without data contention and may give benefits of direct addressing for data access scheduling. To fix the position of an island in the GSC database the size of island data structures should be predefined. This implies determining a static number of regions inside an island and using static data structures for region representation.

As the number of pixels in a pixel island is seven, with minimum two pixels per region a pixel island may comprise three regions at the most. The theoretical maximum of regions in an island of the next higher level is nine[42]. The theoretical number of regions in islands of all other higher levels is excessively high. That is why experimental data were used for defining the optimal number of regions for representing an island at these levels with regard to implementation efficiency and segmentation accuracy. The statistical analysis showed that a number of twelve is appropriate.

The data abstraction for region description known as *a code element* needs to include a feature value and two upward parent pointers. As the topology of the island structure and regularity of the GSC database unambiguously identify both parental islands, the two parent pointers need only to identify parental *CE*s within these parental islands. Additionally, due to the fact that a linking process uses *overlap lists* for region creation, there is no need for maintaining bidirectional links between *CE*s of superimposed levels. Hence, the *parent – child* pointers do not need to be implemented on the higher levels. However, the *parent – child* relationship is required at the lowest level of the hierarchy for determining pixels covered by regions. Again, the topological determinism of the GSC hierarchical structure helps to identify a region's pixel without using its absolute image coordinates. Hence, to benefit by more compact region coding *CE*s of the lowest level need to include a structure that unambiguously defines the positions of the region pixels inside the pixel islands.

The connectivity information for the linking is collected in a regular island-oriented data array. Each element at an island location is a list with the capacity of 36 overlapping subregion pairs (*ORP*s). The list is divided into 12 groups (*OLN*s) of three to four pairs, which correspond to the 12 overlapping nodes. The fixed number of the overlapping pairs was defined on the basis of statistical data as well. The first subregion in a pair always belongs to the north-west-most subisland.

[41] The term *island* indicates both the topological element of the hierarchical island structure and the aggregate of data associated with it, as well as a particular data structure for representing the data.

[42] The number is derived from the number of overlapping nodes, the number of islands in a macroisland, and the maximal number of regions in a pixel island.

Hence, the position of a subregion in *the overlap-list structure* unambiguously defines which sub-island in the island this subregion belongs to. Every region in a pair must be described by its feature, its region weight (the number of subregions it covers), and by a region index inside its island.

The segment-assigned regions – the basic data elements for downpropagation process – grouped in islands can be represented with segment labels and segment features (optionally) propagated from upper levels. The structure for storing segment data should be compliant with the data organisation within a hierarchical layer in *the GSC DB*.

Additionally, for labelled image generation the structure called *a segment key table* can be introduced for extended segment characterisation and fast feature lookup. The absolute addresses in the table can be used as label identifiers, while its entries can contain information about image segments (segment features and optional segment characteristics such as segment coordinates and segment area).

4.1.5 Functional executable model

An executable model based on the above propositions and implemented by means of SystemVerilog and SystemC modelling languages helps in validating the introduced parallel computation approaches and in assessing the computation quality. It is used as a basic model for further hardware system refinement and as a reference for verification of subsequent detailed system models of the FPGA- and the GPU-based designs.

Used for gathering statistical information the functional model reveals substantial knowledge about the GSC cursorily discussed above. The most significant information derived from the statistical measurements for particular algorithmic implementations is investigated further together with the results of the profiling of cycle accurate models in Section 4.2.5.5.

4.2 Architectural model

4.2.1 Computation platform

An FPGA board was specially designed for prototyping the hardware realisations of the GSC. This board contains two independent DDR-SDRAM banks and two independent ZBT-SRAM banks of external random access memory. The core of the board is a Xilinx Virtex II-Pro FPGA. A PCI bus is bridged to a local board bus (i960 Local Bus compatible with extended functionality) by PLX PCI 9656 chip to interface the FPGA with a host system (see Figure 11a). The DDR-SDRAM banks are realised as two socketed DIMM modules (up to 1 GB) with 266 MSamples/sec. The data buses of these modules are operated at 220 MHz maximum to provide double-word 128-bit data at 110 MHz at the FPGA ports (about 1.6 GByte/sec per channel). Two additional ZBT-SRAM (zero bus turnaround) memory banks (up to 8 MB) can be used for fast and random data access at 110 MHz via 128-bit data buses. All four banks achieve an overall data bandwidth of about 6.4 GByte/sec.

The onboard system operates on 128-bit memory words as the architectural data unit (no masking is available for write operation on memory banks); hence, the capacity of each SRAM bank can be treated as 512K of addressable memory words, whereas the capacity of each DRAM bank is 64M words. For integration to industrial applications an external camera can be connected directly to the FPGA board using a Camera Link interface. The manufactured board can be seen at Figure 11b.

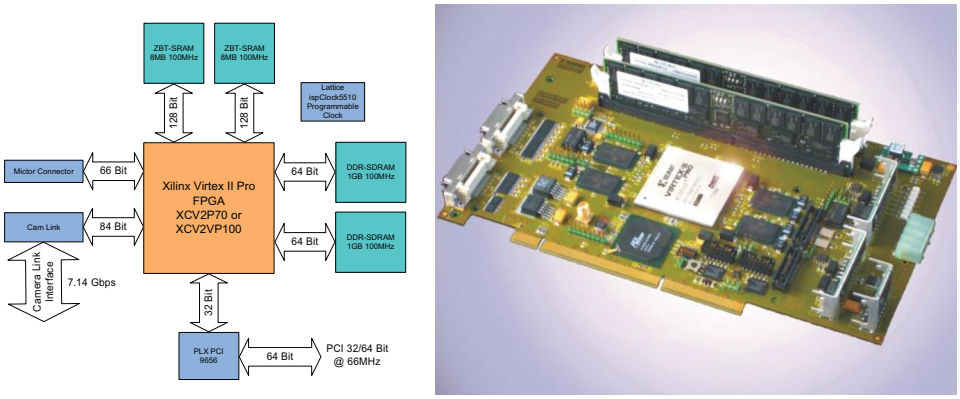


***Figure 11**: FPGA platform*
a) FPGA board diagram; b) AE100Pro expansion board

The multibank architecture of the platform is highly beneficial for the parallel GSC as it can be seen from the functional analysis. Data of two adjacent hierarchical layers can be placed separately, thus maximising the total memory throughput of the system and reducing the spread of memory accesses inside a bank. The latter makes the memory access pattern more predictable (linear access).

A user front-end software, which runs on a host PC, controls the system on FPGA via the PCI interface and carries out any required preprocessing of the image data. The segmented image data are transferred back to the application software for further processing, visualising, and storing.

4.2.2 Hardware data-structure layout

The initial 8-bit greyscale image is packed into 128-bit memory words by 16 lined pixels per word and stored in the onboard memory for segmentation. The result image is either a grey value image of the same size or a labelled image of the same layout, but packed in the memory by four 32-bit label pixels in a word.

The GSC database (see Figure 12a) is split into arrays of *island data structures (IS)* of a fixed size to represent each hierarchical layer of the GSC pyramid. The absolute address of an island data structure is determined by x, y-coordinates of the correspondent island in a layer. A layer is stored column-wise in memory to improve the efficiency of buffering schemes as explained in the next section. Each island data structure contains an array of *CE* structures for regions' description.

The number of *CE*s in an *IS* is fixed and dependent on the hierarchical level number *n* (4 for (n=0)[43], 8 for (n=1)[44], 12 for (n>1)). Each *CE* contains a region feature component coded in 12-bit fixed point format for improving the GSC's accuracy and two parent pointer components of four bits each. The parent pointers are also used to identify invalidity of a *CE* when assigned an out-of-range value. *CE*s of the lowest level have an additional region position component to identify pixels covered by a region. It is represented by a 7-bit flag vector with each bit corresponding to a pixel's position within its island (see Figure 12b). Thus, the *IS* of the lowest layer can be packed in one memory word, while the *IS* of the secondary lowest layer fits in two, and the *IS* of the upper layers fits in three memory words. The *CE*s of an island are organised in compact component arrays as shown in Figure 12c, which is done for traffic reduction as explained in the next section.

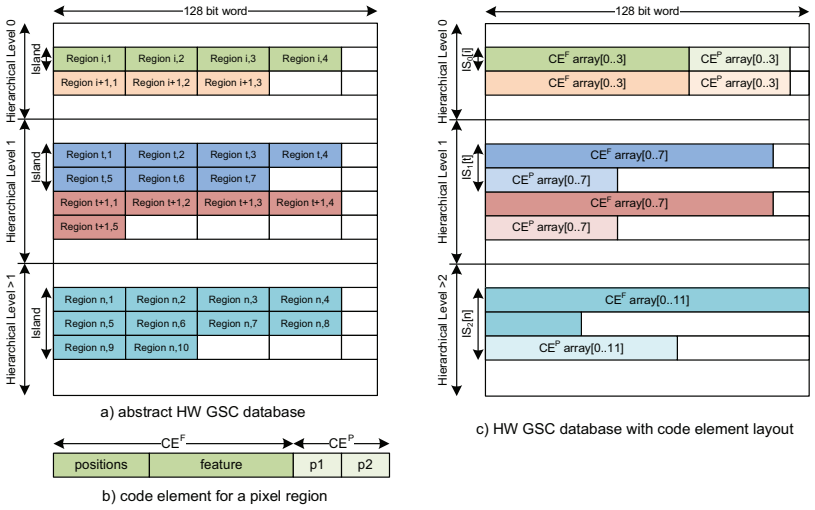


***Figure 12**: Hardware GSC database*

Each of twelve *OLN* entries within the overlap-list structure consists of a fixed number (three or four depending on the GSC configuration) of *ORP*s and is stored in a single memory word (see Figure 13). Every region in a region pair is represented by a 12-bit region feature, a 4-bit region index and an optional 3-bit region weight for weighted linking. The lifetime of an array of overlap-list structures for storing complete connectivity data for a hierarchical layer *i* referred to as *an overlap-list layer*[45] expires after processing two superimposed hierarchical layers *i-1* and *i*. Therefore, *an overlap-list layer* can be treated as a temporary data object in *the linking phase*.

During result generation by downpropagation with *GSC DB decay* the contents of *the GSC DB* are gradually replaced layer by layer with downpropagated parental 12-bit segment features and 20-bit

[43] This number is greater than the theoretical number for a pixel island. This excess is explained by implementation peculiarities of the linking method at the lowest hierarchical level, as shown in the correspondent section of the work.
[44] The theoretical limit of nine regions per island is rarely achieved in statistical measurements, hence the number of regions in the secondary lowest level islands is limited to eight for uniformity in labelled data organisation.
[45] The term *layer* is further specialised to refer to a particular data organisation matching topology of islands arrangement at a certain *HL*.

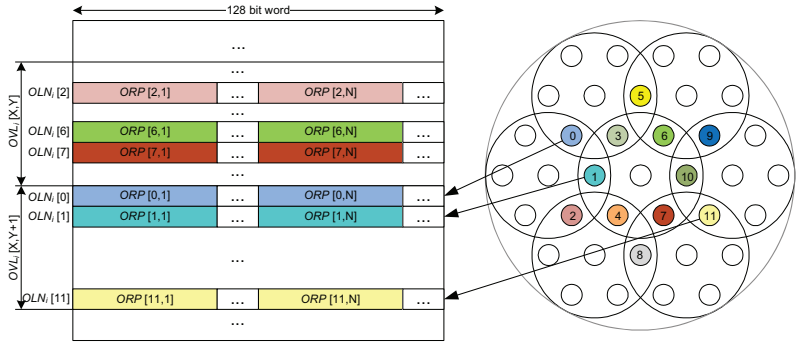


***Figure 13**: Hardware Overlap-list layer*

segment labels assigned to all valid *CEs* in island data structures. The layout of *the DB* is not impaired as the new data structures for representing labelled islands can be packed in the same number of memory words that the original *IS*s fit in. Alternatively, the same data structures can be stored in a temporary island-oriented array that retains the data organisation of a particular hierarchical layer in *the GSC DB*. This approach can be used for increasing memory throughput in the system as discussed in the next section. The lifetime of an array of the labelled island data structures for representing a hierarchical layer *i* is equal to the processing time of two superimposed layers *i* and *i-1*.

Although the proposed data organisation is tempting for the parallel GSC, with the growth of image resolution over 2048^2 the 20-bit label format can theoretically become insufficient for enumerating all the image segments. An approach to solving this problem yet retaining the established *IS* sizes is to use the *IS*s for downpropagating extended 32-bit segment labels exclusively while storing related segment attributes in the associative *segment key table*.

The addressing scheme used in the parallel GSC is based on island coordinates inside a layer lattice; moreover, each GSC process operates on an aggregate of island-associated data seen as a unit from the inside of a process. Thus, data streams to and from external global memory are regarded as streams of island-associated data structures, often called simply *islands*. As it can be seen from the data-flow analysis, the linking process does not ever operate on entire *code element* structures. Hence, for convenience of description of further GSC models the compact component arrays within *the island data structure* are treated as separate data units and are referred to as *an island structure of region features* (IS^F) and *an island structure of parental pointers* (IS^P). To refer to separate components of a *CE* in these structures, the CE^F and CE^P abbreviations are used to identify the region feature field together with the region position vector if applicable and the parent index pair, respectively. A separate position vector is denoted by CE^V. To draw analogy with island data structures in the GSC database *the overlap-list data structure* is referred to as *an overlap-list island* or *an OVL-island structure* and is denoted by OVL^S, while *the overlapping node entries* in an OVL^S are denoted by OLN^S with superscript ^{S} indicating a particular fixed size data structure in the hardware implementation. *Overlapping node entries* are also referred to as *overlap-list nodes* for short.

In *the result generation phase* the parallel GSC operates on the complete linked island data structures incorporating both CE^Fs and CE^Ps and the island-associated data structures with downpropagated segment information. To differentiate these island data units the structures of the first type are referred to as *region islands (island structures[46])* and the structures of the second type as *labelled islands (island structures)* and are denoted by IS^{CE} and IS^L, respectively. To distinguish code elements with assigned segment features they are denoted as CE^L (*labelled CE*s).

4.2.3 Data-stream model

The substantial constraint for computing performance of the given platform is the throughput of its external memory channels. That is why the following targets were given priority in the performance optimisation strategy:

- optimisation of memory channel bandwidth utilisation, i.e. *maximisation* of the *traffic density*;
- general *reduction* of external memory *traffic*.

Performance of the GSC processing units strongly depends on data access efficiency. The efficiency of data accesses is bound to the ability to minimise or to mitigate data latencies in data transactions between processing units and external storages. These latencies are characterised both by propagation delays of memory commands and data words inside data paths within an on-chip system and by intrinsic timings of memory ICs. These data latencies may come up to significant delays, which could directly influences performance of a system. An efficient solutions for compensating the latency impact on the system performance can be found in pipelining computations or in designing a multilevel hierarchical memory subsystem with caching mechanism[47]. The first approach is the best choice for implementing data driven computations, which the parallel GSC method belongs to. Hence, for attaining maximal performance in the GSC implementation the following formula should be kept in mind: to achieve peak productivity of a pipeline a constant and dense data-flow through it needs to be maintained.

Efficiency of a pipelining scheme is strongly dependent on the capability of memory channels for handling transaction concatenation. Transaction concatenation allows a system to maintain dense data flows to and from its processing units. This capability depends on the efficiency of data-path organisation inside communication subsystems on a chip and on the characteristics of the architecture of external memory devices. Maintaining dense data flow is a special challenge for SDRAM architectures, for which access to memory cells is only available within an open row of a memory bank array, while activation and deactivation of a row consume tangible time. Moreover, the data latencies vary for different memory operation types [69, 131]. This means that fully random access to DRAM generally reduces traffic density in memory channels, while sequential[48] memory

[46] The phraseme *island structure* hereafter refers to data structures and can denote any type of data structure associated with a lattice node.
[47] Computation pipelining is not opposed to hierarchical memory subsystem design, however, due to well-schedulable data flows the buffering mechanism in pipelined systems is much less complicated.
[48] To be more precise memory access operations need to be within the same row not necessarily to subsequent locations.

access together with command concatenation techniques and a rational SDRAM bank policy ensure maximum bandwidth utilisation. This peculiarity of SDRAM architecture leads to the idea of using this type of storage as burst-oriented memory, which implies memory traffic scheduling, i.e. exploiting buffering and prefetching schemes as an intermediate layer in a memory subsystem and avoiding stochastic memory activity on SDRAM[49]. ZBT-SRAM architecture is free from these limitations. Although it essentially has some data access latencies, these delays are uniform for all locations and operation types. Hence, these devices provide real random access to their memory arrays with perfect transaction concatenation capabilities. The weakness of these memory devises in comparison with SDRAM with regard to the GSC is that they have relatively small capacity to accommodate all global data of the parallel GSC application. Thus, the questions of proper data allocation, traffic optimisation and scheduling are of a special importance for the parallel GSC design.

Traffic scheduling is feasible for the GSC due to topological orientation of the method and regularity in the proposed data organisation. The scheduling task becomes relatively simple provided that the auxiliary data for the linking, the overlap-list generation, and the downpropagation processes are stored in on-chip memory close to or inside the correspondent functional blocks, which removes the irregular traffic from external buses. This is especially important, if several processing blocks work in parallel, as the traffic tends to become random without a special synchronisation technique applied. In the case of an application-specific hardware design using a high-capacity FPGA a flexible on-chip data-storage organisation becomes practicable.

4.2.3.1 Linking

General traffic reduction in the GSC implementation can be attained by sharing data already loaded on the chip and reducing both the number of transactions and the data volumes for storing intermediate computation results in external memory.

To realise these two concepts efficiently it is reasonable to process the GSC layers row-wise in island by island order. Moreover, it is desirable that several adjacent island rows (block of rows) are processed concurrently, which enables data sharing among neighbours along the *x*- and *y*-axes. For *the linking phase* this scheme together with compact storage of *CE*s in *IS*s (see Figure 12b) allows the system to avoid read-modify-write operations on subislands IS_{i-1} in the external DB, as the linking operation at a layer *i* does not require modification of region features $CE_{i-1}{}^{F}$ at the underlying layer *i-1*, and equally parent relationship data represented by URP_{i-1}s for each subisland I_{i-1} can be first gathered from both parental islands I_i in an internal *HRS* buffer before storing it to *the DB*.

This scheme, in which complete parental information for subislands is generated in a single pass, is beneficial to implementation of overlap-list generation as well, because together with data of newly linked regions $CE^{F}{}_{i}$ parent pointers URP_{i-1} can be forwarded directly to overlap-list node processing elements from the local buffer *without* the need for *intermediate storage in external memory*.

[49] In those systems where random memory access is inevitable the most universal approach to maximise memory throughput is insertion of caches or buffers built on static memories between processing units and DRAMs. This approach treats DRAMs as a burst-oriented memory as data from DRAMs are prefetched to SRAM buffers in solid blocks.

Meanwhile, this processing scheme requires adjacent blocks of island rows to overlap in one island row. Otherwise, subislands in this border rows cannot be filled in with the parental information of their second parents located in the adjacent (lower) island rows, which is essential for the direct generation of overlap-list nodes OLN^{S}_{i+1} (see Figure 14). Thus, the border island rows are linked for only obtaining the parental information without writing *CE*s (IS^{F}_{i} and IS^{P}_{i-1}) to the GSC database. These rows will be linked again during processing the next block of island rows. This relinking does not influence the region result as the linking is a deterministic process provided that the starting conditions are preserved. The need for the row overlapping generally increases the amount of data being passed over the external channel, but the specific share of this traffic in the overall generated data flow significantly decreases with the growth of the number of rows per block. Nonetheless, the row overlapping approach is still more advantageous with respect to overall traffic volume and implementation resources compared to the approaches, in which the intermediate results are stored in external memory.

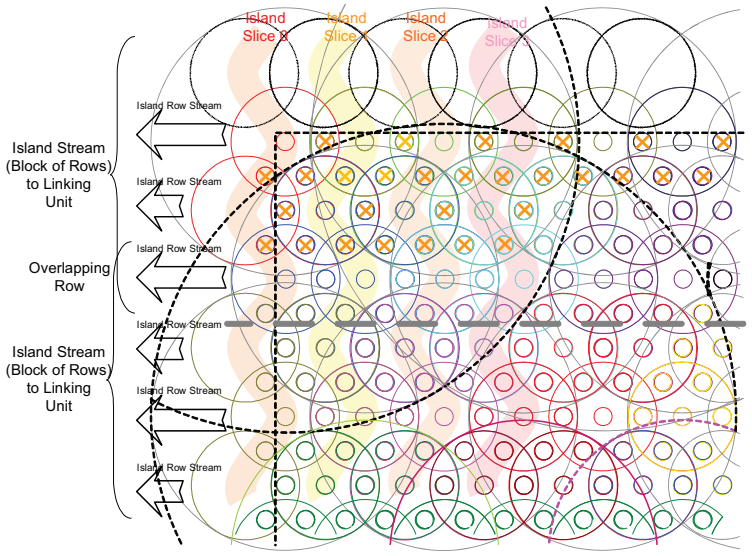


***Figure 14**: Row-wise linking layer processing*

Island processing within a row is performed sequentially island by island resulting in a coarse-grained linking pipeline with data granularity equal to one island structure (OVL^{S}). It is an efficient processing scheme, which can provide a continuous data stream through a processing unit keeping utilisation of implementation resources minimal. Island processing inside a row block is synchronised column-wise. A column of islands is referred to as *an island slice* (see Figure 14). Ultimately, this means that row blocks are processed by a sliding window scheme with the window width of one slice. This scheme allows linking processes in adjacent rows and columns to form their complete *HRS* structures at the corresponding overlapping nodes (marked with orange crosses

at Figure 14) immediately on the chip *using minimal local storage*. Island processing within a slice can be performed either by a single linking processor (column serialisation) or by a single-dimensional array of parallel processors. The latter approach is preferable for upper hierarchical layers *(HL>1)*, because the linking method at these layers implies a state machine realisation, which may take a notable number of clock ticks to complete linking. Moreover linking processes at these layers operate on large data structures and, thus, implementation of internal data stream switching within the linking pipeline may become excessively resource demanding.

The architecture of a linking processing unit, which realises the *linking pipeline*, is shown at Figure 15. The figure illustrates an implementation model of the hardware linking unit configured for processing a block of three island rows.

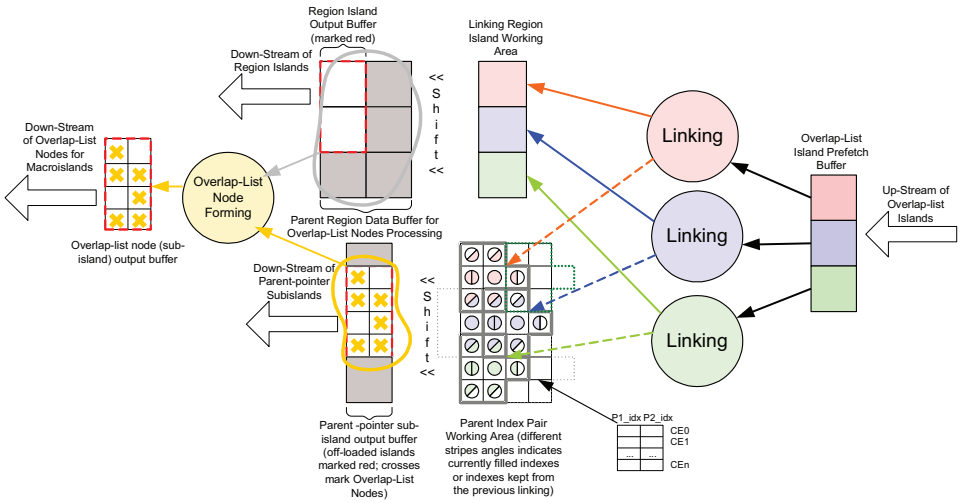


***Figure 15**: Linking Unit pipeline*

The linking processing unit consists of an overlap-list OVL^S_i prefetch buffer, a single-dimensional array of linking processors also referred to as *linkers*, an array of linked regions CE^F_i grouped in islands' IS^F_is together with a matrix of subislands' IS^P_{i-1}s filled with parent relationship URP_{i-1} data forming an *active linking zone*, two correspondent islands' IS^F_i and subislands' IS^P_{i-1} output buffers, an overlap-list node OLN^S_{i+1} generating block, and an output buffer associated with this block. Connectivity data of islands I_i in OVL^S_is are shifted from the overlap-list prefetch buffer into the linkers for generating regions CE^F_i and assigning parent pointers URP_{i-1} to correspondent subregions CE^P_{i-1}. When linking is finished in all linking processors the contents of the active linking zone are shifted to the correspondent IS^F_i and IS^P_{i-1} output buffers, so that some IS^P_{i-1}s still remain inside the active linking zone to be supplemented with parent pointers from the next island slice. The data shifted out from the active zone to the output buffers can be offloaded to *the GSC DB*. While the data are streamed to the external memory, the overlap-list node generating block analyses the data

in overlapping node positions (marked with orange crosses at Figure 15) in the underlying subisland layer and region data of the correspondent parental islands to build up overlap-list nodes OLN^S_{i+1} for islands of the next hierarchical level. These data are placed to the correspondent output buffer and streamed out from this buffer to external memory. It is practical to process all overlapping nodes in the overlap-list node generating block in parallel[50] by several *OLN* generators for all OLN^S_{i+1}s to be complete before all correspondent IS^F_is and IS^P_{i-1}s have been offloaded. This parallel scheme also minimises resource utilisation for routing the data within the block.

Although the mutual coupling of data streams to and from linking processors and *OLN* generators is relatively loose, it is reasonable to organise prefetching and offloading data using synchronised buffers to improve the scheduling of the traffic. To simplify the memory access patterns the island layers are laid out column-wise in external memory so that the island structures can be loaded and stored from/to sequential locations in the physical memory.

Pipeline architecture can be characterised by the maximum intensity of data flow attainable through each stage of a pipeline separately and the intensity of its incoming and outgoing data traffic as the outer constraints for a processing unit in general. This characterisation helps to identify critical sections of a design and to concentrate on particular problems of their implementation to improve the overall pipeline performance. The maximum throughput of the linking unit with the proposed architecture equipped with a single multiplexed external channel can be achieved, if the maximum processing time in the linking processor array, as the slowest processing stage, is less than the aggregate time needed for filling up the prefetch buffer and for emptying all the output buffers. If the unit has several memory channels for different up- and downstreams, the maximum throughput can be achieved if all stages of the pipeline are balanced to be able to maintain maximum possible data flow through the slowest channel.

One overlap-list structure OLV^S_i (12 memory words) in the upstream generates one island structure of region features IS^F_i (one or two memory words depending on the *HL* index), four island structures of parental data IS^P_{i-1} (1x4 memory words), and three overlap-list nodes OLN^S_{i+1} (1x3 memory words) in the downstreams of the linking unit. Assuming that the OLV^S_i layer array is not replicated in external storage, i.e. the upstream can be sourced from only one memory bank in the given architecture, the upper throughput limit for the linking unit is the throughput of the channel canalising the OLV^S_i upstream as the total data volume generated by processing one OLV^S_i is less than the data volume of a OLV^S_i to be uploaded. All the offloading data streams can be reasonably directed through a single downstream channel thus. Considering the throughput of the linking stage it should be noticed that the time of linking is a stochastic value that generally depends on image contents. For the linking stage not to become a bottleneck in the pipeline the linking latency can be hidden in the parallel linker scheme by increasing the number of rows in the row block, thus raising the time needed for filling the prefetch buffer. The number of rows for the balanced configuration of the linking pipeline can be defined by profiling the linking unit with images of different scene types.

[50] Provided that the whole linking unit works in a single clock domain.

4.2.3.2 Coding

Coding is a particular case of linking. Thus, it can inherit the general principle of computation organisation realised in the linking implementation to improve performance and optimise resource utilisation. Coding at the lowest hierarchical level differs from linking at all upper levels in the way it establishes relationship between the basic elements of the layers. During coding this relationship is established based on relative spatial location of pixels in the image. Technically to perform coding at the pixel level initial islands need to be extracted from the image and all predefined connections within the islands should be analysed for similarity for elementary regions to be formed. After initial regions are formed the same spatial relationship is used for establishing connections between newly formed regions. The procedure for overlapping detection at this level is principally analogous to the procedure at the higher levels except that pixels in the image are treated both as overlapping nodes and as common subregions (i.e. subislands of one subregion). Figure 16 shows the structural scheme of a pipeline processing unit that realises coding. The main differences in structure of the coding unit compared to the linking hardware pipeline are the pixel-buffer front end and absence of the parent-pointer output buffer.

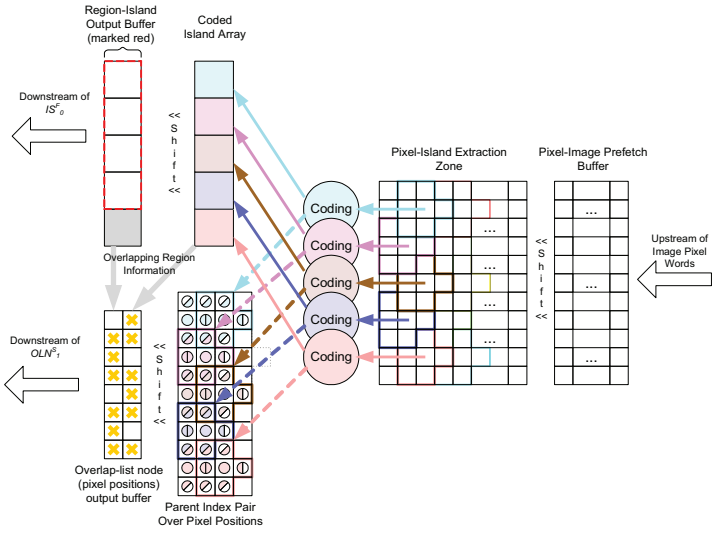


***Figure 16**: Coding Unit pipeline*

For extraction of pixel islands memory words of adjacent pixel lines are loaded to the pixel prefetch buffer. As one memory word of a pixel image contains several pixels, the pixel prefetch buffer represents a rectangular fragment of the image. The number of pixel lines covered by the buffer should be sufficient for extracting an integral number of pixel island rows. When fully filled, this buffer is shifted into the pixel island extraction zone, where the islands' pixels are selected and forwarded to the coding processor array. Coding the pixel layer is performed in blocks of pixel island rows in the same slice-wise manner as in the general linking scheme. Realised in the front-end of the coding

unit such data organisation allows generating a dense flow of pixel islands fed to the array of coders working in parallel.

The results of coding are written to shared local storage realised as an array of coded regions (both CE^{F}_{0} and CE^{V}_{0}) grouped by islands IS^{F}_{0} and a two-dimensional array of URP_{0} pairs in pixel positions. Both are shifted out to the output buffers upon island slice completion. In contrast to the general linking implementation the coding unit does not generate a stream of parent-pointer data for the underlying hierarchical layer, as there exist no true subregions, so the vertical relationship have to be coded in pixel regions themselves therefore. The generation of entries for overlap lists at this layer is a trivial operation, as two regions of adjacent islands can overlap only in a single pixel. Thus, the coding unit produces only two outgoing streams of coded region islands IS^{F}_{0} and of overlap-list entries OLN^{S}_{1}.

Processing one island in this architecture generates an upstream of 0.25 memory words of pixel data and a downstream of 1+3 memory words, one for the coded islands IS^{F}_{0} and three for the overlap-list nodes OLN^{S}_{1}. As it can be seen the potential bottleneck of this pipeline is the back end of the coding unit, hence it can be reasonable to distribute the output traffic to two different external channels separately for the IS^{F}_{0} and the OLN^{S}_{1} data elements. It can also be sensible to route pixel data and coded island structures through a single multiplexed channel, as it should not shift the balance in the coding pipeline. Considering the throughput of the coder stage it should be noticed that structural parallel implementation of the coder can guarantee deterministic coding time, which can be down to one clock per island minimum as it is discussed later in Section 4.2.5.1.

4.2.3.3 Downpropagation and result image generation

Processing island layers during result generation by downpropagation is performed in a similar pipeline scheme to that realised in the linking processing pipeline. Adjacent island rows of labelled island layers and the correspondent rows of underlying subislands as the targets for segment features downpropagation are grouped in blocks for processing. The blocks have special arrangement of the rows of two superimposed layers, which guarantees that complete parental label information for all subislands IS^{CE}_{i} to be labelled in a block can be found in the labelled islands IS^{L}_{i+1} of the upper layer covered by this block. In terms of the topology of the GSC a block consists of an even number n of subisland I_i rows, covered by $m=n/2+1$ rows of labelled islands I_{i+1}; these rows of superimposed islands are disposed towards each other in the way that the first row of subislands I_i in the block is a middle line[51] for the islands I_{i+1} in the first row of labelled islands I_{i+1}. This means that to cover a layer i completely adjacent blocks must overlap in one row of labelled islands I_{i+1} (Figure 17).

Within the block island processing is performed in a sliding window scheme, so that subregions to be labelled in IS^{CE}_{i} can share the label data of parental regions in adjacent IS^{L}_{i+1} using minimal local storage resources. The processing window slides slice-synchronous in both superimposed layers.

[51] The middle line here means the line connecting the central nodes of the labelled islands in one row.

One island slice corresponds to one column of labelled islands IS^{L}_{i+1} and two coherent columns of subislands IS^{CE}_{i} in the underlying layer.

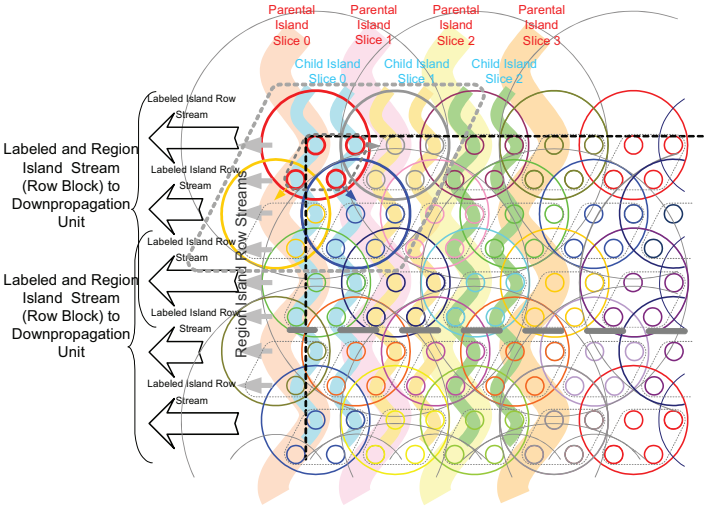


***Figure 17**: Row-wise downpropagation layer processing*

The architecture of a processing unit that realises the proposed computation scheme is shown at Figure 18. It represents a downpropagation unit pipeline configured for labelling subregions in a block of six subisland rows. The downpropagation unit consists of two prefetch buffers for uploading IS^{CE}_{i} and IS^{L}_{i+1} island structures, one matrix of IS^{L}_{i+1} structures for parental information lookup, a twinned matrix of subisland structures holding IS^{CE}_{i} and IS^{L}_{i} for label downpropagation, a single-dimensional array of labelling processors, and an output buffer for offloading labelled subislands IS^{L}_{i}. The data are shifted through this pipeline slice-wise realising the sliding window scheme for processing island blocks. The single-dimensional organisation of synchronous parallel processors is justified by the same performance and resource consumption reasons as those deduced for the linking unit design.

It is convenient to consider one labelled island and the group of the underlying subislands having this labelled islands as the first parent (marked with the same colour at Figure 18) as the basic data aggregate for the downpropagation processing unit. This aggregate is associated with a single labelling processor. Thus, the density of the upstream generated for processing one such group is $3\times(m+1)/m+12$ ($HL>2$), $3\times(m+1)/m+8$ ($HL=2$), or $2\times(m+1)/m+4$ ($HL=1$) memory words depending on the index of the hierarchical layer HL under processing. The first term in the sums indicates the contribution[52] of a labelled island IS^{L}_{i+1} and the second figure is the number of memory words for all subislands IS^{CE}_{i} in the group. It is important to note that the upstreams of IS^{L}_{i+1}s and correspondent IS^{CE}_{i}s can be routed through two different channels as adjacent layers of *the DB* can

[52] The fraction that modifies the sizes of the labelled island structures IS^{L} in the formulas indicated the contribution of a labelled parental island in the border row of the island block.

be placed in different memory banks; thus, the upstream throughput can be limited by only the subislands' channel. Equally, the traffic density of the IS^{L}_{i} downstream for the downpropagation unit is 12 (*HL>2*), 8 (*HL=2*), or 4 (*HL=1*) memory words per group[53].

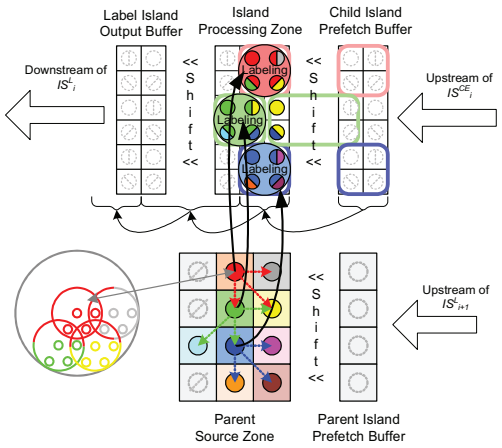


***Figure 18**: Downpropagation Unit pipeline*

The region labelling by one labelling processor within one subisland group can be performed either fully parallel for all regions and all subislands or in a sequential manner. In the case of sequential region-wise processing labelling a single subisland requires 12 (*HL>2*), 8 (*HL=2*), or 4 (*HL=1*) clocks provided both parental islands are directly available to the processor. Therefore, for the fully sequential labelling processor (both subislands and their sub*CE*s processed in series) not to become a bottleneck in the downpropagation unit pipeline, the unit should be configured for processing at least eight subisland rows in a block to hide the processing latency, provided that the upstreams and the downstream are decoupled and do not share the same bandwidth. If the subislands under processing should be modified in the same *DB* storage locations, i.e. the channel bandwidth of a memory bank, in which the current subisland layer is stored, is shared by the upward IS^{CE}_{i} and the downward IS^{L}_{i} streams, the number of rows in a pack can be halved. Alternatively, to eliminate the problem of the processing latency the labelling processors can be configured to operate on all subislands in the group in parallel while keeping the sequential processing organisation of their subregions intact. In this case the resource consumption for the labelling processor implementation increases approximately linearly with the increment of the number of processing elements for sequential subisland labelling within the labelling processor. Full parallelisation of subregion labelling inside these processing elements is not practical as it does not improve the throughput of the complete unit, yet requiring significantly more hardware resources for realising full data connectivity and implementing replicated computations.

[53] During processing the lowest hierarchical level segment labels can be directly assigned to the image pixels with no need for labelling IS_0s in *the GSC DB*. The last figure is computed for the generation of the 32-bit label image and accidentally coincides with the number of memory words hypothetically to be overwritten in *the GSC DB* in the case of the IS_0 level modification.

The assignment of unique labels to newly detected tree roots is a special concern in the parallel processing scheme. Issuing a key can be performed either under the scheme of centralised distribution of the label numbers, which leads to the hardware complication and the shared resource contention, or by assigning an own unique label number space for each processor, which leads to the growth of the memory size for allocating the segment key table. The latter approach is more attractive for the parallel GSC implementation on the FPGA due to sufficient storage available in SDRAM banks.

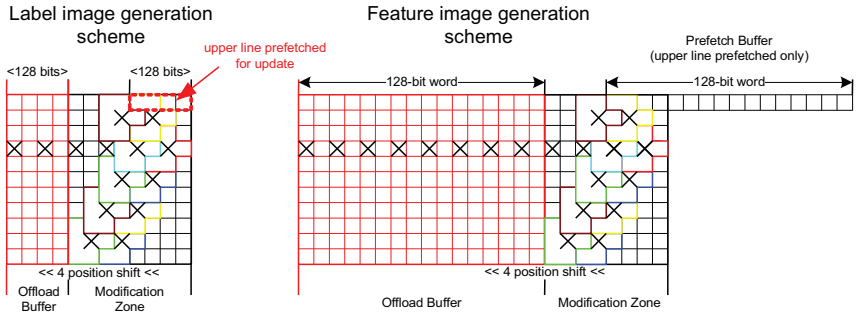


***Figure 19**: Image Generator scheme*

The label and the feature image generation is performed after labelling regions at the lowest hierarchical level without storing the updated island structures IS^{L}_{0} to external memory. To perform image generation an island slice is mapped to the arrays of label or feature pixels. When the arrays are fully inhabited, the data are shifted out gradually to output buffers and then streamed out packed in memory words to external storage after the output buffers are complete (Figure 19). It is important to note that the pixel blocks generated by such scheme overlap in one pixel line, which is shared by adjacent island blocks. That is why the upper pixel line should be loaded to the image generator for updating the pixels. This can be done by storing and loading the line to and from external memory or by organising a ring buffer (queue) of memory words for storing this line on the chip. The last solution is preferable for performance reasons as it simplifies and reduces external traffic.

4.2.3.4 Performance improvement strategies

Considering the particular organisation of the processing pipelines and the data-stream models deduced from it, the GSC implementation can be quantitatively assessed for performance. In the described system architecture the application performance will directly depend on the amount of data moved over external channels. Thus, the performance estimations can be based on the model of the data traffic generated by the parallel GSC, taking in account different bandwidth utilisation ratios of the external channels. This model is referred to as *the Parallel GSC Traffic Model* and is adduced in Appendix A.3[54,55]. The most optimistic scenario assumed in the traffic model of a data-

[54] The *Traffic Model* does not take into account the overhead of the traffic introduced by the overlapping of row blocks as it depends on particular configurations of the processing units (the number of rows in a row block).
[55] The *Traffic Model* does not take into consideration the traffic to/from the label key table as it is image dependent and can be assessed only by profiling the target image.

driven system for the model simplification is that the external traffic is perfectly scheduled and concatenated and thus any processing unit in the system can receive or send a portion of data on every clock tick. The performance rate assessed under this assumption corresponds to the maximal possible performance of the system and can be treated as a measure of efficiency for data traffic organisation in a particular system implementation.

The analysis of the *Traffic Model* of *the parallel GSC* shows that the major bulk of the traffic volume is generated while processing the lowest hierarchical layers. Therefore, it is reasonable to focus on revealing the opportunities for traffic optimisation at these layers to improve the overall application performance. A substantial effect of such optimisation can be observed when the coding stage of *the GSC LP* is integrated with the subsequent linking stage, i.e. the traffic between these two stages does not leave the die. The effect of this integration can be seen at Figure 20. The figure shows the number of memory words transferred during *the linking phase* at each *HL* to and from external storage for a separate (navy bars) and a joint (orange bars) processing of the first two layers. This layer processing integration results in a reduction of 3.88 to 3.98 times of the total external traffic in *the linking phase*. The redundant data volume in the separate processing approach is explained by the need for load-modify-store operations while updating the parent pointers CE^{P}_{0} in the IS^{CE}_{0} structures and by the need for an intermediate load-store operation for passing the connectivity data between the two lowest layers. The integration of the coding and the linking pipelines is feasible for hardware implementation as the coder array does not generate a large amount of connectivity data per island I_1 for the next linking stage[56] and the coded island structures IS^{CE}_{0} are relatively compact to be temporarily stored in the on-chip buffers for subsequent URP_0s assignment in the linker array.

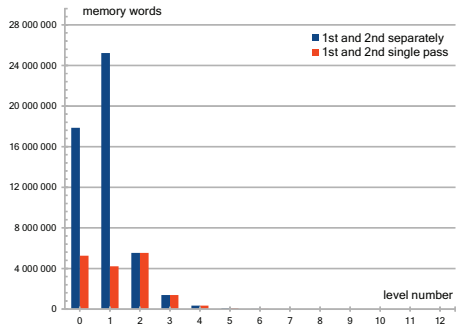


***Figure 20**: Memory traffic per linking level (in memory words)*

The structural scheme of a processing unit for the joint processing is shown at Figure 21. This unit is called an *Initial Linker*. The initial linker pipeline is effectively a concatenation on the coding and linking units. After initial coding the region connectivity information is forwarded directly to the linking processor array and the coded island structures are shifted into the linker's update buffer for

[56] Each overlapping node is represented by a single overlapping region pair only.

parent-pointer assignment[57]. A prerequisite for configuring the initial linking unit is that the number of pixel lines in the front-end buffers should be sufficient to extract an integral number of macroisland I_1 rows for the linking stage. The initial linking unit shown at the figure is configured for coding five pixel island rows and thus for linking two macroisland rows of the next *HL*.

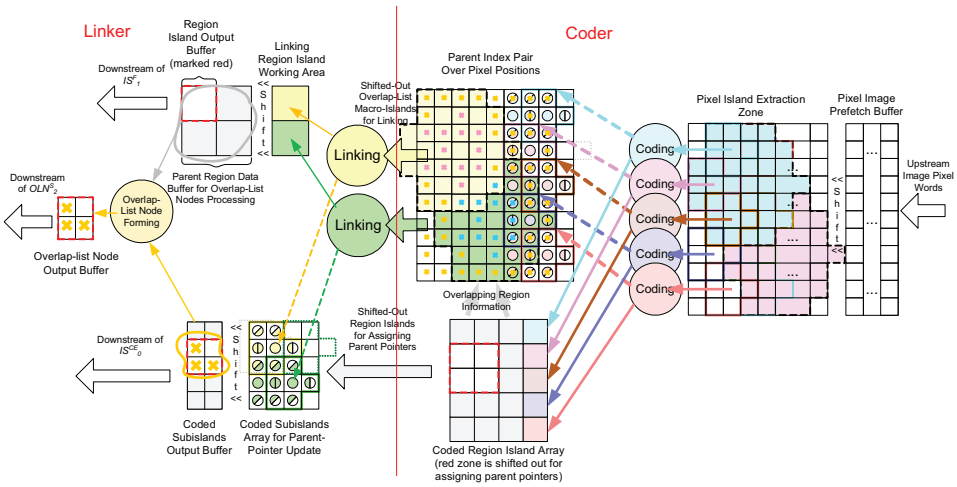


***Figure 21**: Initial Linker Unit structural scheme*

Although linking at different layers is performed based on the same principles, the overlap lists for the secondary lowest and the upper layers differ in complexity (one against three *ORP*s per overlapping node, respectively). Thus, the linking processors in the initial and general linking units can have different implementation for performance and resource optimisation reasons. Therefore, the initial linker can be realised either as an independent hardware unit with specialised linking processors or as an extension of the general linking unit with the coding stage integrated as an add-on with the generic linking pipeline. Thus, in the latter case the *Extended Linking Unit* need to have two front ends – one data path for pixel data prefetching and coding and one for overlap-list structures loading.

As regards the application performance, the first approach is preferable for the reason that each separate linking unit can be tuned individually for the optimal pipeline throughput. This primarily implies the individual configuration of the set of external channels that conforms to the demands on external traffic organisation of each linking unit type and the adjustment of the sizes of the internal processor arrays for balancing the stages of the pipelines – the processing of a single overlap list is performed generally faster on the secondary lowest layer than on higher layers due to the difference in the number of *ORP*s in the overlap lists. The second approach allows hardware reuse and saves the resources of the die, although it complicates the data-flow switching in the front end of the

[57] In fact CE^{F}_{0}s are not affected by the linkers, so the CE^{F}_{0}s are just shifted through the pipeline of the unit and are glued eventually togethher with their CE^{P}_{0} counterparts in the output buffer.

Extended Linking Unit. Meanwhile, increasing complexity of the unit can be critical for implementation on a particular hardware platform as it endangers observance of the timing constraints[58].

Allowing for the gain obtained by applying the joint processing approach in *the linking phase* in the GSC model the system generates the traffic indicated by the layer profile at Figure 22. The graph reflects the traffic distribution over *the linking* and *the result generation phases*.

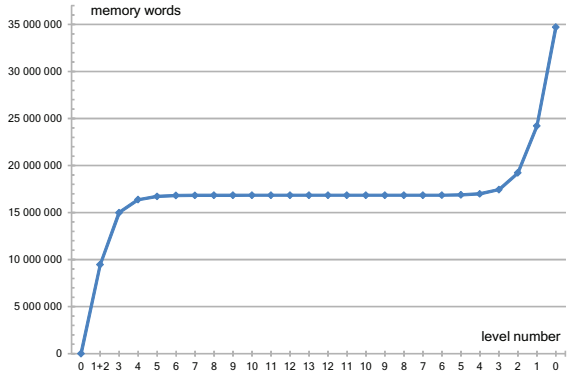


***Figure 22**: Accumulated traffic per GSC level (in memory words for 4096² image resolution)*

As noted above, the utmost performance of each processing unit depends on the configuration of the external channels. In particular customising the units for operating in a single channel configuration makes the processing time of the units dependent on the total traffic volume generated by them. Moreover, channel multiplexing may negatively influence the channel bandwidth utilisation in cases of fragmented memory access to the same SDRAM banks. At the same time, data-flow separation allows eliminating the contribution of the traffic in the alternative parallel channels to the total data forwarding time and thus to the overall processing time. Hence, the upper performance limit for a processing unit is defined by the slowest channel[59]. The effect from the data-stream parallelisation for *the GSC* can be seen in Table 2 and Table 3 built on the basis of the *Traffic Model*. The tables reflect the change in the application performance in different GSC phases depending on external channel configurations of the processing units. The tables provide the totals of only the memory accesses that contribute to the processing time in a given configuration (figures in brackets estimate the processing time for the external channels with a sample rate of a 0.1 memory word per 1 ns or 100MHz data sampling).

Table 2 shows the data traffic of *the* total *linking phase* for different image sizes. The *single channel configuration* is considered to be the reference (*LS* – Linking, Single channel).

In *the parallel channel configuration type-A* (*LA*) the upward and the downward data streams are separated into two channels. In this case the traffic bottleneck of the general linking unit is the

[58] Extended Linking Unit was not implemented on the given Virtex II Pro FPGA due to this reason.
[59] The slowest does not mean a difference in bandwidths of channels but reflects the specific volume of data that should be moved through one channel comparing to another.

overlap-list input channel, whereas the output channel can be totally neglected in the performance estimations due to its lower traffic density. However, it is important to note that the output stream of OLN^S_{i+1}s and the input stream of OVL^S_is need to have different target and source memory banks. Otherwise, both data streams share the same physical channel and the OVL^S_i traffic adds to the processing time. Thus, overlap lists of adjacent layers should be stored in different memory banks. On the contrary, for the initial linker in this configuration the bottleneck is the output channel, which multiplexes the traffic of IS^{CE}_0, IS^F_1, and OLN^S_2 structures. Overall, this parallel configuration gives a 25% reduction in the processing time of *the linking phase* compared to the basic configuration.

***Table 2**: Linking Phase traffic for performance estimation (in memory words)*

	4096^2	2048^2	1024^2	512^2	256^2
Single Channel Configuration LS	16840940 (168.41ms)	4 226 257 (42.26ms)	1 064 630 (10.65ms)	270 235 (2.70ms)	69 632 (0.70ms)
Parallel Channel Configuration LA	12 626 044 (126.26ms)	3 167 338 (31.67ms)	797 278 (7.97ms)	202 066 (2.02ms)	51 910 (0.52ms)
Parallel Channel Configuration LB	8 417 391 (84.17ms)	2 111 581 (21.12ms)	531 537 (5.32ms)	134 725 (1.35ms)	34617 (0.35ms)

The *parallel channel configuration type-B* (*LB*) realises the idea of minimising the impact of the output traffic of the initial linker on the overall performance. It splits the output traffic of this processing unit into two downward streams with similar data-flow intensity and routes them into two output channels: IS^{CE}_0 structures via one and IS^F_1 and OLN^S_2 structures via the other channel. In this case only the coded pixel islands' data are considered in the performance model. The parallelisation of the downstreams for the linking at upper levels does not have an impact on the performance but can be realised as a compromise solution for the extended linking unit. Overall, the *LB configuration* may give another 25% increase in the overall performance for *the linking phase*.

***Table 3**: Result Generation Phase traffic for performance estimation (in memory words)*

	4096^2	2048^2	1024^2	512^2	256^2
Single Channel Configuration PS	17 860 693 (178.61ms)	4 473 932 (44.74ms)	1 122 883 (11.23ms)	282 938 (2.83ms)	71 857 (0.72ms)
Parallel Channel Configuration PA	10 510 393 (105.10ms)	2 633 779 (26.34ms)	661 549 (6.62ms)	166 951 (1.67ms)	42 529 (0.43ms)
Parallel Channel Configuration PB	7 354 397 (73.54ms)	1 842 202 (18.42ms)	462 359 (4.62ms)	116 500 (1.17ms)	29 585 (0.30ms)

Table 3 shows the traffic generated in *the result generation phase* relevant to the GSC application performance for different configurations of the downpropagation unit channels. The maximal performance of the downpropagation unit is achievable, if all three data streams are decoupled: two separate input channels for the IS^L_{i+1} and the IS^{CE}_i upstreams and one output channel for the IS^L_i downstream. It should be noted that, if downpropagation is realised by updating island structures in

the GSC DB, i.e. the source of the IS^{CE}_i upstream and the target of the IS^{L}_i downstream are in the same memory location, both streams will use the same physical channel of a memory bank, which doubles the counted traffic. Thus, it is reasonable to use a dedicated data structure for the intermediate storage of a labelled layer and to allocate this data structure in a different memory bank other than the source bank for the IS^{CE}_i upstream. Moreover, two adjacent layers should have different allocations for their intermediate storage to prevent data inconsistency by overwriting due to the larger dimensions of the lower layer. Additionally, these data structures are recommended to be placed in different memory banks, as the upstreams of IS^{L}_{i+1} structures and the downstreams of IS^{L}_i structures have to share the same physical bank channel otherwise. This situation is equivalent to the configuration in which all sources and targets of the same layer are placed in different memory banks, but the downpropagation processing unit has only a single input channel for both upstreams. The traffic model for this situation is referred to as the *parallel channel configuration type-A* (*PA*). The configuration with fully decoupled streams is described by the *parallel channel configuration type-B* (*PB*) model. This model takes into account the traffic of only a single stream of subislands (eqaully IS^{CE}_i or IS^{L}_i) per layer for the performance estimation as the most dense data-flow for the configuration.

As it can be seen in the table, the realisation of the *PA configuration* can increase the downpropagation performance by 41%, while the introduction of the *PB configuration* may save up to 59% of computing time for this GSC phase in comparison with the single channel configuration (*PS*).

Processing images of resolutions higher than 2048^2 pixels generally requires change in data organisation for the segment labels' downpropagation, as the feature and the label data together do not fit into the conventional island data-structure format. In this case the allocation for one IS^{L}-structure in the external memory can be extended by one extra memory word, or alternatively, the segment features can be stored and looked up in *the segment key table*.

The first approach benefits from the plain traffic scheduling scheme, but loses in the constant storage and traffic overhead. The traffic increase is 16%, 13%, and 19% for the *PS*, *PA*, and *PB configurations*, respectively.

The second approach should lead to the problem of scheduling the traffic of the segment keys as the label features cannot be retrieved from *the segment key table* until labelled parental islands have been loaded to the downpropagation unit. However, the relative weight of the key table traffic in the overall traffic is highly dependent on the image nature (number of segments and fractality). Moreover, this traffic can be seriously reduced by exploiting the spatial locality of the image segments. It can be beneficial to cache the segment features of the parental regions in the downpropagation unit, as they will be reused likely for the region labelling in the neighbouring islands while the processing window slides over a layer. To decrease the common resource access contention and raise the bandwidth efficiency of the data channel to the key table, it is reasonable to implement the two level caching scheme: the first-level feature caches are to be placed in the labelling processors, and the second-level cache is to be shared by all labelling processors in the downpropagation unit.

In this segment key table scheme the downpropagation unit requires two additional up- and downstreams for key-table entries to the shared data resource in an external storage. The up- and downstreams can share one common external channel of the downpropagation unit, as both streams are eventually routed through the same physical channel of a memory bank. The key table traffic should preferably not interfere with other regular streams to the SDRAM banks as the access operations to the key table are completely random and thus may seriously affect the linear access pattern for the island structures. If this decoupling is not feasible, a special attention should be paid to the traffic planning of a multiplexed channel to provide slots for the access requests of each kind.

4.2.4 Application memory space mapping

Considering the results of the data-dependency analysis of the GSC method in Section 4.1.3 and the data-flow intensity estimation in the data stream model in Section 4.2.3, the GSC global-memory data structures are mapped to the onboard memory in a way that allows minimum data-flow interference. To avoid physical channel multiplexing, the data stream distribution across the memory banks in the hardware implementation is performed dynamically depending on the current hierarchical level index and the image resolution. In general the SRAM devices are used as external buffers for temporary storage of data structures for data exchange between the GSC hierarchical levels, while the DRAM, having a higher storage capacity, is used for application lifetime data structures such as *the GSC DB* and *the segment key table* (if applicable).

The initial pixel image is temporarily stored in one of the two static memory banks (base configuration) or in one of the dynamic memory banks (for resolutions higher than 2048^2) and is overwritten with other temporary data by the running GSC application[60]. The *GSC DB* is distributed between the different DRAM banks so that the data of two neighbouring hierarchical layers are always located in different DRAM devices and the IS^{CE}_1 layer is always stored in the opposite bank to the DRAM bank containing the initial image if the image is loaded to DRAM. Temporary OVL^S arrays are placed in the SRAM devices inside two memory allocations made for data of different hierarchical layers. Two OVL^S arrays of two adjacent hierarchical layers are always placed to the opposite SRAM banks[61]. The temporary OVL^S arrays are constantly overwritten by OVL^S arrays of higher hierarchical levels. In the base configuration the OVL^S array generated by the initial linker is always placed in the opposite SRAM bank to the one used for the initial image storage. In the case of the higher resolution images the OVL^S arrays of lower hierarchical levels are placed in DRAM. The OVL^S_0 array is always placed in the same DRAM bank in which the initial image is located.

During the downpropagation IS^L arrays are stored in SRAM. Similar to OVL^S arrays IS^L arrays of two adjacent hierarchical levels are placed in opposite memory banks and are overwritten subsequently. For the higher resolution images the arrays of IS^Ls are stored back to *the GSC DB*

[60] The singularity correction is found not practical for parallel implementation and thus is performed as a post-processing stage of the segmentation, while the pixel splitting is performed in a simplified scheme. Thus the image does not need to be longer retained in memory.

[61] The OVL^S_0 layer for pixel images of 2048^2 and higher does not fit into a single SRAM bank in the current device configuration. For this reason the total SRAM address space is distributed unequally between the two overlap-list layers' allocations – 1.5 SRAM bank are reserved for the one and 0.5 SRAM bank for the other OVL^S layer buffer.

overwriting the original unlabelled islands. The *segment key table* is placed in the DRAM bank that is opposite to the bank that contains the IS^{CE}_{1} layer of *the GSC DB* for the current resolution. Result images are always stored in DRAM in the same bank together with the *segment key table*.

The above data allocation scheme allows a complete separation of all data streams into different physical memory channels for image resolutions up to 2048^2.

Because of the dynamic bank switching the current addresses of the initial and the result images in the board address space are made visible in the application register set of the device. Thus, the precise manipulation of the input and the output data is delegated to the client software at the host side.

4.2.5 Architectural executable model

The data-stream model is implemented as the further refinement of the functional model in several steps. Firstly, the functionality of the GSC executable model referred to in Section 4.1.5 is partitioned into the architectural units described above to build a transaction level model (*GSC TLM*) to analyse the interaction of the architectural components in dynamics. Secondly, the coding, the linking, and the downpropagation methods discussed in Section 4.1.3 are timed (cycle accurate) to gather statistics on the execution latencies and to analyse the contributory operational features by profiling the system with specific data sets. These data can be used to infer an optimal configuration of the processing units, i.e. the number of the parallel processors to balance the stages of the pipelines. To build the timed models the precise computing algorithms are mapped to a number of computation steps (time slots) with each step matching an estimated computing power allocation that is presumably realisable in one clock cycle on the target hardware. Finally, *the GSC TLM* is supplemented with the transactors between the TL core of the model and the bus cycle accurate models of the external memory devices[62] to exercise the system in the precise technological constraints of the hardware platform and in particular to test the influence of different channel configurations on the channel bandwidth utilisation. The results of the timed simulation of the GSC executable model are discussed in the subsections below, and the impact of the hardware platform on the application performance are discussed in Section 4.3.4, where the results of the simulation of *the GSC TLM* refined by introducing the timed interface model of the communication subsystem are reported.

4.2.5.1 Coding algorithm

The functional analysis showed that the impact on the segmentation result quality of the linking at the lowest hierarchical level *HL(0)* by different sequential and parallel structural methods is difficult to assess objectively, and thus an indisputable favourite cannot be distinguished among the approaches. At the same time a manual graph colouring test on a group of people showed that the subjective human perception for the corner cases of hexagonal graph colouring differs significantly. Besides, the difference in the final segmentation results of different linking methods used for coding

[62] A preimplemented DDR SDRAM controller core is used in the on-chip system. For this reason one type of BFM (bus functional model) transactors interfaces a gate model of the core and not the DDR SDRAM bus directly.

initial region is generally compensated by the global-view nature of the hierarchical algorithm. Thus, in choosing an appropriate algorithm for coding the focus is set to the performance and the resource efficiency of the coding method. Parallel structural methods applied to the pixel sets attract by their potential for inexpensive parallelisation and pipelining, which guaranties a short and strictly predictable processing time. Hence, a parallel structural method is preferred for the coding stage.

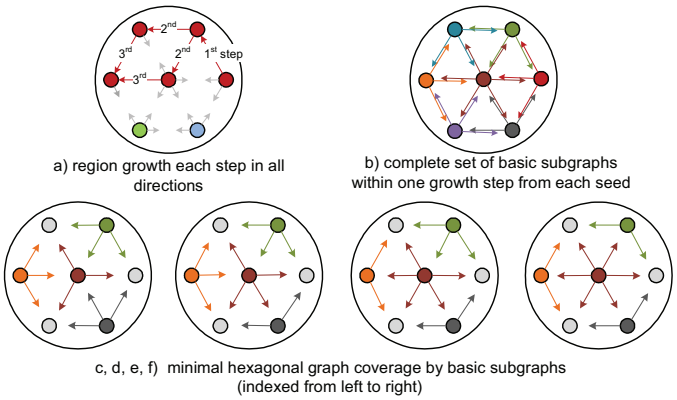


***Figure 23**: Hexagonal graph coverage for parallel structural coding*

The coding is based on the concurrent growth[63] of regions from different seeds (potentially seven). Obviously, the combinatorial functions for encoding all possible region subgraphs associated with each seed are rather complex to be realised in one clocking slot due to a large number of partial sums for computing the region averages. Thus, the coding is split into a number of time steps. Each step realises concurrent single-edge distance growth of all regions in all possible directions from all current (included in the previous step) vertexes of the regions' subgraphs (Figure 23a). Since the inclusion depends exclusively on the feature distance of adjacent vertexes, all vertex connections in the graph can be established before the growth starts. Defining the subgraphs of all the vertixes considered at the first inclusion step as *basic subgraphs* (Figure 23b) and the regions built in this step as *basic regions*, the further linking can be interpreted as stepwise merging of *the basic regions*, subject to common vertex detection. The hexagonal graph is excessively covered with *the basic subgraphs,* as seen at Figure 23b. The minimum number of noncyclic *basic subgraphs* having no common edges is four; all possible combinations are shown at Figure 23c-f. The preferable variant for implementing the coding is the one in which the number of edges in all *basic subgraphs* is three (Figure 23c), which is justified by the following rationales focused on the hardware resources:

a) Avoiding common edges in *basic subgraphs* eliminates duplication/sharing of the hardware for evaluating the threshold condition in the case of a parallel processing of the *basic subgraphs*.

b) The need for reducing the number of *basic subgraphs* is tied to the fact of existence of the maximum possible number of unique basic regions and to the problem of the mapping of coded

[63] The term *growth* does not imply here a time-ordered sequence of computing steps but relates to data propagation in a data-dependency tree of a combinatorial function that describes the logic of subgraph vertex inclusion.

regions to some fixed hardware resources. As this maximum number is three, all initial seeds can eventually produce three unique regions at maximum, all others being either duplicate or empty. The complexity of the task of selecting these unique regions increases with the total of region candidates. This leads either to complicating the priority encoding combinatorial scheme for parallel unique region detection or to a higher number of cycles for sorting the candidate region list. At the same time the four noncyclic *subgraphs* are the minimum vertex cover of the graph that guarantees that no potential result region is omitted at the coding stage.

c) An optimal edge distribution among the *basic subgraphs* minimises resource utilisation for computing region averages in two ways. Firstly, the edge distribution influences the total number of partial sums needed for the total island coding, provided all four *basic regions* are formed in parallel (28, 32, 42, 58 partial sums for configuration *c*, *d*, *e*, and *f* respectively). Secondly, even if the *basic regions* are computed in series reusing the same hardware, the maximum number of edges across all basic subgraphs significantly influences the complexity of the routine for the partial sum selection or the complexity of a combinatorial selector scheme in the case of parallel partial sums opting (7, 15, 29, 49 partial sums for the largest subgraphs in configuration *c*, *d*, *e*, and *f* respectively).

The method of parallel structural coding is realised by two timed algorithms for hardware implementation. The first represents a coding pipeline with the initial data latency of five clocks and the subsequent output rate of one coded island per clock (the *pipelined coder*). The first stage realises data loading and does not perform any computations. At the second stage all *basic regions* are formed in parallel generating pixel position data and the averages for all *basic regions*. At the third stage each *basic region* is compared against all the others to detect common pixel positions and is merged with the respective regions. At the fourth stage the merging repeats to join the basic regions not immediately overlapped but having common basic regions in between. The total averages for all four resulted regions are computed at this stage as well. The last stage is dedicated to forming a special *Region-to-Position Export Table* (*region export table*) for the subsequent linking process. The third and the fourth stages of the pipeline can be merged together, or alternatively, the computation burden can be redistributed between them, subject to the hardware implementation constraints.

The second algorithm represents a coder implementation with reuse of hardware resources (*the sequential coder*). It realises a two stage pipeline, each stage of which is a state machine that processes the basic regions one after another. At the first stage all four basic regions are coded in series and the resulted data representing a whole pixel island are passed to the next stage of the pipeline. At the next stage each basic region is merged with the others in parallel at every clock and the *region export table* is formed at the last clock. The initial data latency of this implementation is 10 clocks and the subsequent data output rate is five clocks.

From the performance viewpoint, none of these two coding implementations have any significant influence to the total throughput of the initial linking pipeline. In the case of the first implementation the grounds for this are evident – the pipelined coding of this type can provide a constant flow of coded islands with the maximal data rate. The second implementation can potentially cause stalls

in the total pipeline. The processing at the next linking stage of the initial linking unit can complete within five clocks (see Section 4.2.5.5), and for oversegmented images with poorly populated overlap lists the occurrence of such cases can be considerably high. However, the simulation of the architectural executable model with realistic data sets indicated that the increase of the processing time for a whole image does not exceed 3% in comparison to *the pipelined coder*. This is explained by the fact that the latency of the next linking stage is defined by the maximal processing time across the array of linking processors, which entails smoothing effect. Introduction of the timed memory the TL model relaxes the coupling of the stages inside the pipeline even more – the output stage becomes slower than the coding stage (see Section 4.3.4 for the modelling results for external channels), which makes the coding stage insensitive for the throughput of the initial linker pipeline.

The number of regions formed with the parallel structural coding approach is equal to the number of basic regions[64]. Hence, at least one of the resulted regions is duplicated (or empty). This in principle implies that the duplicated regions should be eliminated both to reduce the storage space in external memory and to avoid region recomputation at *the result generation phase*. However, the excessive regions are left in the coded island structures to be filtered later at the stage of overlap-list entity creation as well as in *the result generation phase*. This is done without the danger of irrationally using the external memory in the particular hardware implementation as the hardware data format for the lowest level islands can accommodate up to four coded regions. The ambiguity problem in linking pixel regions is resolved by using a priority scheme to assign a region index to a pixel position when creating *Region-to-Position Export Tables*.

The *Region-to-Position Export Tables* generated by the coding processors are data structures introduced to the initial linker scheme for instantaneous exchange of connectivity data within the groups of seven neighboured islands I_0 constituting macroislands I_1 for linking at the subsequent linking stage. Each *export table* consists of six entries that correspond to the outer pixel positions in an I_0. Every entry or a *region descriptor* of these *export tables* contains information about the region which the position belongs to. It includes a validity flag indicating that the position is covered by a region in the island, the index of this region within the island, and the average feature of the region.

4.2.5.2 Linking algorithm

Due to higher complexity of the elements in a lattice of higher hierarchical levels the parallel structural approach for region growing is substituted with a potentially less resource-demanding sequential linking based on traversing *Macroisland Overlap Structures*. The traversing is performed using the topology knowledge of overlapping regions within a macroisland, which drastically hasten sequential linking.

The *Macroisland Overlap Structure* used in the initial linking pipeline consists of seven *Region-to-Position Import Tables* (*import tables*), which correspond to seven coded islands within a macroisland (Figure 24). The *import tables* are used in combination with the correspondent coded island

[64] The *basic region* is a structural unit associated with *a basic subgraph* and thus can be empty.

structures to provide complete information on the regions and their connectivity inside a macro-island. The *import tables* are counterparts to the *Region-to-Position Export Tables*. Having the same structure they represent the regions of the neighbour islands occupying the correspondent outer positions of an island. The region data is taken from the correspondent *export tables* constituting a macroisland (*region descriptors* are exported at the overlapping points of a macroisland to the *import table* structures). The validity flags of the region descriptors within an *import table* form *an overlap position vector* analogous to the position vector in the *CE* structure of coded islands (CE^V). This vector representation allows a simple mechanism for detecting region overlaps by trivial vector operation on *the overlap position vectors* and *the position vectors* of *CE*s of correspondent islands. Each entry in *the import table* of an island is associated with the index of a neighbouring island within the macroisland and the index of the correspondent overlap position, i.e. pixel (or generally, lattice node) position, inside this neighbouring island. All the indexes together form a *transition lookup table* of a macroisland to provide a navigation scheme for traversing. This *transition lookup table* together with the position vectors and region indexes from the region descriptors provide a complete connectivity graph of the regions in a macroisland enabling the topology aware linking.

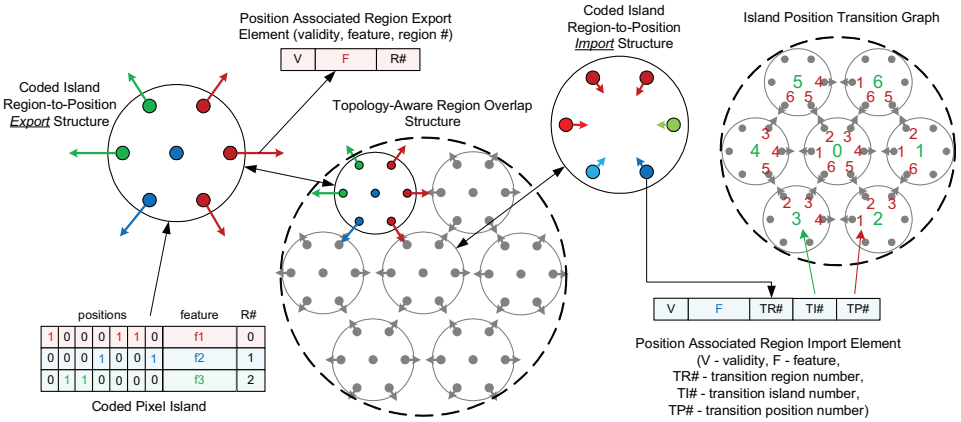


***Figure 24**: Pixel-region macroisland overlap structure*

The initial linking is realised as a state machine for traversing macroisland connectivity graphs and a LIFO (last in, first out) stack for storing information on branching points of macroregion trees for rolling back when the top of a branch is reached. The algorithm for tree traversing is as follows:

a) find a start point of a macroregion linking tree (root):
   - take a valid (nonempty) region that has not been included in any macroregion;
   - initialise the macroregion feature with the feature of the current region (seed region);
   - assign a new region index to the new macroregion;
   - if there are no valid regions left or a certain limit in the number of macroregions is reached, the macroisland linking is complete;

b) select candidates for transition from the seed region:
   - detect all overlapping (candidate) regions in the neighbour islands using the CE^V of the current region and *the overlap position vector* of *the import table* of the current island;
   - test all candidate regions against the merging threshold condition using the region features from the imported *region descriptors* and the *CE* of the current region;
   - if such regions satisfying the linking criteria exist, select one candidate region depending on the chosen analysis method (clock-wise in the current implementation) and prepare for transition;
   - if there is no region to transit to, check if there is a branching point record in the linking tree stack:
     - if there is a branching point record, prepare for roll-back;
     - if there is no roll-back possible and the current region is the root region (the macroregion has not been formed), discard the macroregion and find a new start point;
     - if there is no roll-back possible and the current region is not the root (macro-region is complete), increment the region index for the next macroregion assignment and find a new start point;

c) prepare for transition to the selected candidate region:
   - mark the overlapping point associated with the selected candidate (transition point[65]) as visited in the current island to exclude a transition through the same overlapping point in the case of a branching roll-back (unset the validity flag in *the region descriptor* of *the import table*);
   - mark all overlapping points covered by the seed region as visited in all neighbour islands to exclude looping in the macroregion linking path (unset the correspondent validity flag in *the region descriptor* of *the import table* of each neighbouring island);
   - assign the parent macroregion index to the seed region;
   - assign a new macroregion feature as the mean of the current macroregion feature and the newly linked region;
   - if the seed region has more than one candidate, write the seed region index and its island index within the macroisland to the LIFO storage (branching point record) to roll back when the top of a linking tree branch is reached;
   - mark the current region as linked and transit to the next seed region;

d) prepare for rolling back to the branching point region:
   - mark all overlapping points of the current region as visited in the neighbouring island to avoid branch or tree gluing;

[65] *A transition point* or *a transition position* is the position of an overlapping point within an island.

- assign the parent macroregion index to the current region;
- mark the current region as linked and roll back to the branching point region;

e) transit to the next seed region:

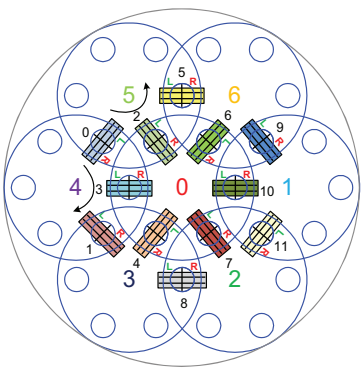
- identify the next seed region using the transition island index and the transition region index associated with the transition point from the import table;
- set the region as the seed region and select candidates for transition;

f) roll back to the branching point region:

- identify a new seed region using the region and the island indexes stored in the LIFO;
- remove the branching point record from the LIFO storage;
- set the region as the seed region and select candidates for transition.

The search for a new root candidate is realised by a separate process working in parallel to the linking. The process goes through all regions in all islands starting from the central island until it finds the next nonempty region – there the process blocks. If this region is being included in the currently linked macroregion, the search is triggered to continue until the next candidate is found.

The linking algorithm described above implies that the decision for the next transition can be made in one time slot as the analysis of all transition candidates (and LIFO state) can be performed in parallel for all overlapping points. Depending on hardware implementation constraints this potentially results in one or two clocks per transition. The executable model is built with the assumption of one clock per transition for the initial linking.



Ovl. List index	Check region index in (Ovl. Ent.#+Side)	Belongs to Island #	Left		Right		Belongs to Island #	Check region index in (Ovl. Ent.#+Side)
0	2L,5L	5	CE#	feature	feature	CE#	4	1L,3L
1	0R,3L	4	CE#	feature	feature	CE#	3	4R,8L
2	0L,5L	5	CE#	feature	feature	CE#	0	3R,4L,7L,10L,6R
3	0R,1L	4	CE#	feature	feature	CE#	0	4L,7L,10L,6R,2R
4	7L,10L,6R,2R,3R	0	CE#	feature	feature	CE#	3	1R,8L
5	0L,2L	5	CE#	feature	feature	CE#	6	6L,9L
6	5R,9L	6	CE#	feature	feature	CE#	0	2R,3R,4L,7L,10L
7	10L,6R,2R,3R,4L	0	CE#	feature	feature	CE#	2	8R,11R
8	1R,4R	3	CE#	feature	feature	CE#	2	7R,11R
9	5R,6L	6	CE#	feature	feature	CE#	1	10R,11L
10	6R,2R,3R,4L,7L	0	CE#	feature	feature	CE#	1	9R,11L
11	9R,10R	1	CE#	feature	feature	CE#	2	7R,8R

x3 pairs per each position

***Figure 25**: Overlap-list topology relationship*

The linking at higher levels is analogous to the initial linking. Changes in the algorithm are related to the difference in the representation of connectivity data in *Macroisland Overlap Structures*. The *Macroisland Overlap Structures* for the linking at higher levels are based on the OVL^S structures stored in the external memory. Although an overlap list does not contain explicit information on spatial location of overlapping regions, the layout of the data aggregate allows restoring the

topology of macroislands (Figure 25). This makes the algorithm of topology-aware traversing applicable to the higher level linking.

The *Macroisland Overlap Structure* in this case consists of a number of *Region Transition Tables,* which correspond to seven overlapping islands within a macroisland. These *transition tables* are equivalent to the *import tables* for the initial linking and similarly guide the traversing in the linking process. Each *transition table* contains a number of entries corresponding to the overlapping positions within an island. Each of them contains a number of overlapping region pairs[66]. The pairs are spatially oriented – the left-hand elements describe the regions belonging to the island associated with *the transition table,* and the right-hand elements represent the transition target regions in the neighbouring islands. A single region having several overlaps with regions in neighbour islands appears in *the transition table* of its island several times. Thus, the search for transition candidates is based not on the region position matching but on the matching of the registered index of a seed region against the indexes of left-hand regions in *the transition table*, which is the only conceptual difference between the linking algorithms. In contrast to the initial linking the linking process at higher levels also records positions of the islands within a macroisland the linked regions belong to. This information is registered in the CE^{V}s of newly created macroregions and is subsequently used exclusively for ranking *ORP*s while generating OLN^{S}s as described in the next subsection.

The peculiarity of the root search in the higher level linking is the traversing order. During the initial linking the search goes through all regions within an island and then similarly over all islands within a macroisland. In the higher level linking the search goes through all first pairs at each transition position of an island and then proceeds with the next island. After the complete round through all islands the search repeats with the next overlapping pair index. This search order is justified by the fact that the overlapping pairs are arranged by the region weights during OLN^{S} generation to improve the quality of linking and to increase the efficiency of the root search process.

The higher number of potential transition candidates and the necessity of region identification by index definitely increase the complexity of candidate selection, but do not make a parallel hardware implementation impracticable. For the cycle accurate simulation two variants of the sequential linking executable model for hardware implementation under optimistic and pessimistic assumptions are built. In the first version each linking step is realised in one clocking slot. The timing assumptions for the second version are two clocks per forward transition step (the first clock is used for parallel feature comparison and the second for prioritised selection) and one clock tick per roll-back transition step, i.e. the situation in which the seed region has no further transition options.

4.2.5.3 Overlap-list creation algorithm

Creation of overlap-list entries is the final processing stage of either linker pipeline. It is performed at several overlapping nodes of the lower level lattice in parallel using the data from the output buffers of the linking stage. In general the OLN^{S} creation process consists in filtering and sorting the

[66] This number naturally corresponds to the number of *ORP*s in the OLN^{S} structures in a given *GSC* implementation.

overlapping region pairs in *HRS*s by sequentially traversing the subislands' region lists in the onchip buffer. The algorithm uses the parental data in the subisland structures and the feature and the position data of the parent regions found in the correspondent parental island structures (Figure 26).

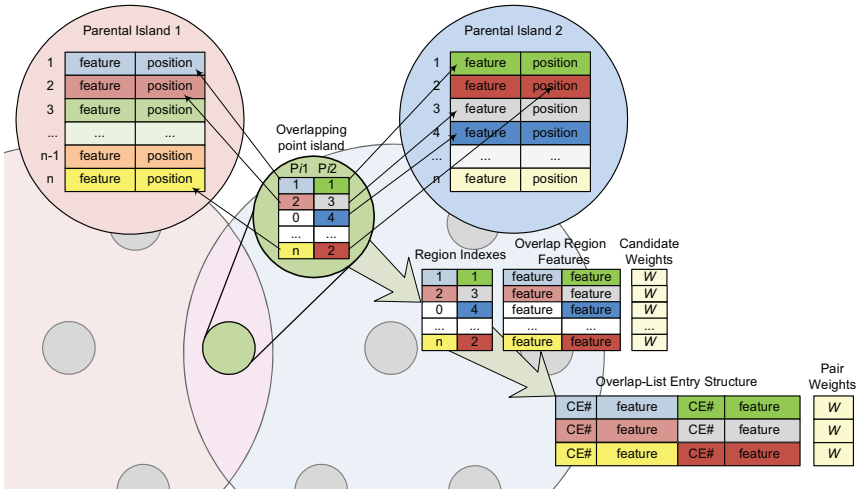


***Figure 26**: Overlap-list entry creation*

The need for a pair sorting is conditioned by the restricted number of overlapping region pairs in the OLN^S structure. To select the most valuable pairs and to increase the efficiency of linking at the levels above the algorithm arrange pairs by their region position weights (*ORP* ranking). At *HL(2)* the ranking and filtering procedure also aims to exclude repeated and empty regions from coded islands[67].

The overlap-list entry creation algorithm for higher levels consists of a number of steps or pipeline stages for sequentially processing all ORP_is within an *HRS* after initial loading of respective *IS*s to the local memory. Each pipeline stage is executed in a single clock slot. The algorithm steps are:

a) Stage one – loading local buffer:
   - locally load the data of respective subislands IS^P_{i-1} (overlapping point islands) and associated parental islands IS^F_i from the linker output buffers to each separate OLN^S creation unit (*OLN generator*);
   - load the first index pair CE^P_{i-1} from the IS^P_{i-1} to an *index pair register*;

b) Stage two – merging condition check and pair weight assignment:
   - check if both parent indexes are valid in the *index pair register* (overlap condition);
   - check the merging threshold condition for the features of parental regions CE^F_i indexed by the *index pair register* (current ORP_i);

[67] The priority encoding scheme used for export table generation at the last stage of the coder pipeline does not allow duplicate regions to appear in any region descriptor. Thus, duplicate region cannot be linked to a macroregion and thus be assigned any parent pointer. Hence, the parent index pairs of such regions in overlapping point islands are empty.

- set a validity flag if both merging conditions are satisfied;
- assign a weight to the current ORP_i depending on the number of positions occupied by the correspondent parental regions CE^V_i;
- record the validity flag, the parental region indexes CE^P_{i-1}, the parent features CE^F_i and the pair weight to an intermediate data structure (*overlap candidate register*);
- load the next index pair CE^P_{i-1} from the IS^P_{i-1} to the *index pair register*;

c) Stage three – overlap candidates ranking:

- compare the pair weight in *the overlap candidate register* against the weights of ORP_is already placed to the overlapping node structure OLN^S_{i+1} (seen as a list of ORP_is);
- if the weight of the candidate is greater or equal to the weight of a pair in the OLN^S_{i+1}, shift the allocated pair and the pairs below down in the list OLN^S_{i+1};
- place the candidate pair to the respective released position in the OLN^S_{i+1} list.

As it can be seen, although the *OLN* creation unit has an internal pipeline organisation with the data forwarding rate of one *OPR* per clock, a complete OLN^S structure is processed with a constant latency equal to the number of regions in the overlapping point islands of the correspondent hierarchical level plus two clock cycles for initial data loading and data forwarding through the internal pipeline.

4.2.5.4 Downpropagation process

Islands within *labelling subisland groups* (Section 4.2.3) are processed by the labelling processors in parallel. Each island is processed sequentially region-wise with the processing tempo of one clock per region. The algorithm for labelling is the following:

- at the first clock a *labelling subisland group* of IS^{CE}_is and four associated parental islands IS^L_{i+1} are loaded to the shared memory of the labelling processor;
- at the second clock each subisland processing line (processing element) is loaded with a subisland's CE^F_i and its two parental CE^L_{i+1}s indexed by the related CE^P_i in a dedicated register (first loaded at the initial step and subsequently updated at each processing clock);
- at the third clock the subisland region CE^L_i is assigned the segment label and feature of the most similar parental region CE^L_{i+1}; if the region has no parents a new *segment key* is generated and the feature of the region becomes the segment's feature;
- the second and the third steps are repeated for every region in a subisland until the region-per-island limit is reached or an empty region is first detected indicating the end of the region list of an island (except for the lowest level islands in which an empty region may appear at any region position).

To generate a unique segment key in the parallel island processing scheme the segment index space is evenly divided among the labelling processors. Each processor is initially assigned a starting

address of its key space. Within a group keys are distributed dynamically under a round-robin discipline. This means that if a new segment key needs to be generated for a region, the correspondent subisland processing line is stalled until the arbiter issues a new key to it.

Newly generated segment descriptors (segment key and feature) are sent to *the segment key table*[68]. This is done with the same round-robin arbitration scheme: when a key is granted to a processing line the segment descriptor is put to a constant length queue. Segment descriptor queues from all labelling processors are forwarded to a single external memory channel using another round-robin arbiter. The queues are used to smooth out the collision effect resulted from the channel sharing.

This processing scheme implies that the time of an island processing at maximum is equal to the number of regions per island IS^{CE}_i plus two clock ticks for the intermediate data buffering and a surplus of clock ticks conditioned by arbitrating the segment key generation and the multiplexed channel granting. All together this results in a data-dependent stochastic number within a certain range, which is the subject of the model profiling.

Image generation is done during the downpropagation at the lowest hierarchical level. A group of pixel processors works on IS^{L}_{0}s in columns mapping features and segment labels of their CE^{L}_{0}s to certain positions of pixels in the pixel output buffers using respective CE^{V}_{0}s. To eliminate the need for dedicated hardware for resolving the conflicts at common pixels of neighbouring islands every pixel processor works in two phases. At the first step the values are assigned to the three upper-left and the centre pixel positions. The remaining three positions are filled at the second step. These two steps are made in different clock slots introducing a latency of two clock ticks to the stage of the pixel buffer populating.

4.2.5.5 Simulation results

Built on precise algorithms the executable model is simulated for profiling the implementations on variety of data sets. To estimate timings of application specific computation blocks special statistic counters are integrated in the architectural model to record time (in clock ticks) between different process synchronisation events. Using these statistics data distribution histograms of processing times are built. Three parameters are used for quantitative evaluation of a standard processing time:

a) a distribution sample number *Tk* corresponding to the processing time of a processing unit in clock ticks that covers 95% of all sampled processing time trails (*k*:

$$\sum_{1}^{k} \bar{p}_i \leqslant 0{,}95 ,$$

where $\bar{p}_i$ is a normalised occurrence frequency of each processing time sample in a distribution, and *i* is a sample number corresponding to the number of clock ticks in the histogram);

b) a distribution sample number *Tn* corresponding to an utmost processing time sample *n* with its probability higher than 1% (n: $\forall\, \bar{p}_i \leqslant 0{,}01 , i \in [n:m]$, where *m* is the maximum number of clocks in a processing time distribution);

[68] Having an impact on the application performance and the resource consumption this mechanism is optional for images below 4096^2.

c) a weighted mean *Tt* of a unit's processing time in clock ticks

$$Tt = \left(\sum_{1}^{m} p_i * i\right) / \sum_{1}^{m} p_i ,$$

where p_i is an occurrence frequency of a processing time sample, and i is a sample number.

A special problem with system profiling is the selection of sample data sets. The quantity of sample data sets does not necessarily influence the quality of the sampling. Realistic extremes of the observed parameters cannot be necessarily found within even a vast number of test images. At the same time artificial (phantom) images specially synthesised for the study of corner cases are not representative for the analysis of a system operating on verisimilar data. Thus, it is desirable to define a set of characteristic features of an image relevant to a parameter under study.

To define characteristic image features relevant to the linking processing time a correlation analysis of spatial distribution of

- the number of linking clocks,
- the regions formed per island,
- the total region chain length, and
- the overlap pairs per island

is done for various hierarchical levels and merging thresholds (see Appendix A.4 for profiling data).

The analysis indicates increasing time of linking in the areas with higher fragmentation (fractal[69] global elements or detail intensive areas with element sizes comparable to the size of the hierarchical island) and high gradient areas (such as segment edges). At the same time in the areas of over-segmentation (the size of regions noticeably lower than the island size at a specific *HL*) the linking time drops appreciably. Thus, the linking time for these areas decreases with rising the *HL* index. Extreme values for the linking time are strongly related to the new roots search when an overlap list is highly populated with overlap pairs, but the linker does not produce long region chains (massive regions). This situation characterises highly fragmented areas. If this fragmentation corresponds to the global fractality, the effect of fractality persists through all hierarchical layers.

The labelling time is obviously correlated with the number of regions within an island. Higher numbers correspond to fractal, highly detailed or gradient areas. The surplus of processing time related to new segment generation is typical for highly detailed areas with the size of elements comparable to the islands' size. For this reason higher processing time values are observed in that areas in contrast to fractal or gradient areas for which segment features are propagated from levels above.

With the above characteristic features a number of images are selected for processors' profiling. The statistics parameters for assessing the elapse time on these data sets are shown in Table 4 together with the derived number of the processors to completely cover the latency of the processing stages in the respective pipelines[70].

[69] The term *fractal* is not used here in a strict mathematical sense but to denote the shape of an edge line of an object having a specifically long border with the background or other elements per square unit or otherwise a winding edge line combining large and small curves.
[70] The estimation already assumes the maximum bandwidth utilisation rate for different external channels discussed in Section 4.3.

***Table 4**: Expected values on elapse time of the Processing Units (in clock ticks)*

	Tk	*Tn*	*Tt*	*Number of Processors*
Linking 1clk/linking step (HL=1)	<24	<17	<12	>3
Linking 2clk/linking step (HL=1)	<27	<28	<20	>4
Linking 1clk/linking step (HL>=2)	<20	<20	<14	>1
Linking 2clk/linking step(HL>=2)	<34	<34	<22	>2
Downpropagation (HL=1)	<10	<11	<7	>2
Downpropagation (HL=2)	<11	<14	<8	>2
Downpropagation (HL>=3)	<15	<15	<9	>2

The distribution characteristics of the elapse time for different processing units indicate that the latencies of the processing stages in the pipelines of different processing units can be completely hidden by increasing the number of processors in their processor arrays up to a certain reasonable number, which balances the front-end or the back-end buffering stages and the processing stages. Of a special importance is that the General Linker, which cannot to be realised in a one clock per linking step scheme in the target device due to timing violations, can be harmlessly implemented with repartitioning the computations of a linking step into two clocking slots. Starting from the number of linking processors equal to three, the difference in the performance of both solutions becomes negligible, which is proven by simulation[71].

4.2.6 Preimplementation resource estimation model

For preliminary evaluation of the realisability and estimation of the resource consumption of the system in certain configurations a *GSC HW Resource Model* is built (Appendix A.5). The outputs of the model together with the results of the cycle accurate models help to steer the implementation of the architectural units in cost[72]/performance space and to define the share of each particular solution in the resource budget of a given device. The model is based on the calculation of the on-chip storage amount required for the realisation of each architectural unit. It especially focuses on the data aggregates that will presumably require on-chip flip-flop memory for their implementation.

Such model benefits from the notable simplicity of its realisation, maintenance, and modification at the preimplementation designing stage. The model realisation requires only the structural analysis of the architectural blocks without a functional analysis of data transformation in the data paths.

This approach is justified by the fact that an explicit assessment of involved combinatorial resources requires manual conversion of high level functional operations into Boolean algebra expressions and putting forward a hypothesis on mapping this logic onto architectural primitives of the target device the way it is performed by a particular EDA tool during the RTL synthesis. Each time an alternative solution is introduced, the laborious procedure needs to be carried out over again.

[71] The difference does not exceed 1% for the processing time of one hierarchical layer.
[72] As the amount of hardware resources for particular implementation.

To lend credibility to the model without a laborious manual translation of the functional operations into the architectural primitives the model is constructed relying on the following RTL design assumptions:

- the RTL implementation of the system conforms to a high-speed synchronous designing rule, which primary implies even distribution of registers in the data paths with moderate complexity of combinatorial logic in between to meet the timing constraints of a design;
- the target device comprises combinatorial resources and flip-flop memory evenly distributed in the device, which is normally achieved in the modern FPGA architectures by designing the functional blocks with practical balance of the logic-to-register ratio to conform with synchronous design requirements at up-to-date technological level.

Table 5 shows the flip-flip resource consumption and the share in the overall resource budget of the Xilinx Virtex XC2VP100 device for the processing units in various processing configurations.

***Table 5**: Processing Unit resource estimation (in bits of register memory)*

rows in a block	*Coding Unit*	*Lowest Level Linking Unit*	*General Linking Unit*	*Initial Linking Unit*	*Extended Linking Unit*	*Down-propagation Unit*
2	2684 (2.94%)	3228 (3.54%)	9778 (10.71%)	11510 (12.61%)	19962 (21.86%)	14024 (15.36%)
3	3984 (4.36%)	5296 (5.80%)	15717 (17.21%)	17530 (19.20%)	31161 (34.13%)	26320 (28.82%)
4	5284 (5.79%)	7364 (8.06%)	21656 (23.72%)	23550 (25.79%)	42360 (46.39%)	38616 (42.29%)
5	6584 (7.21%)	9432 (10.33%)	27595 (30.22%)	29570 (32.38%)	53559 (58.65%)	50912 (55.76%)

Provided that about 10% to 15% of the die is reserved for supplementary infrastructure, the linking and the downpropagation processing units may not exceed four island rows in a block in their processing configurations, and the preference for realising the linking phase is given to the extended general linking solution for the implementation on the given device, all other operational constraints being met.

4.2.7 On-Chip infrastructure

The complete System-on-FPGA is shown at Figure 27. It consists of the auxiliary on-chip infrastructure (peripheral and interconnect resources) described below and the application specific blocks discussed in Section 4.2.3. The peripheral controllers are marked in grey, the interconnect resources in pink, and the GSC components in blue. The black arrows indicate the data paths and the blue arrows the control stimulus for arbitrating the application.

The on-chip infrastructure provides external interfacing (*EIF*) and on-chip communication (*OCC*) functionality. The *EIF* components realise the signal level peripheral interfaces with the onboard

devices and provide an application interface of the board to the host system. The *OCC* subsystem realises data connectivity in the on-chip system and provides flexible data-stream routing to the architectural components.

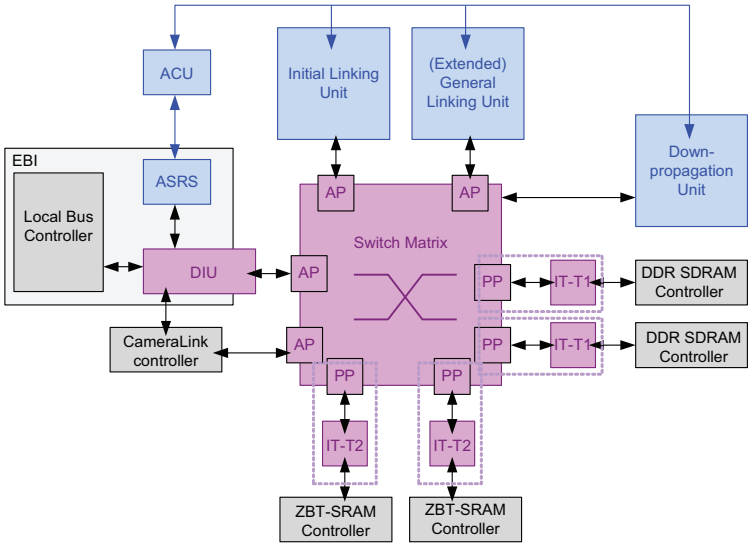


***Figure 27**: System on the FPGA chip*

The *OCC* infrastructure is realised as a configurable network of point-to-point data channels between the network agents. The data connections are established between active (transaction initiating) and passive (transaction serving) clients of the network. The *OCC* is designed to meet the following requirements:

- simplicity of client module connection;
- ease of modifying configuration of the on-chip system to allow reusing the interconnection infrastructure in other projects and to enable fluent modification of the system for experimenting with configurations in the HW GSC implementations;
- conformity with GSC-specific demands for both sufficient performance and facility for integrating the particular application.

To satisfy the partially contradicting demands stated above the communication subsystem is based on the principles of modularity and parametrisability. It consists of a module providing transport medium, port components for traffic management, and interface transactors for integrating the network agents to the network. To relax contention problems for common resource accesse, the switch architecture is preferred to the bus architecture for the on-chip communication.

A scalable switch matrix is the core of the *OCC*. It provides point-to-point links between any two components connected to the switch's ports. The number of ports, either active (*AP*) or passive (*PP*), is easily adjusted by parameterisation. This assures ease of introducing new end-point

components to the system. However, the switch matrix architecture becomes significantly resource intensive at a certain number of end points. Therefore, for frugal resource utilisation a flexible tunability is enabled in the design. The parameterisation allows an individual connection mapping: for each active port in a configuration the number of destination ports can be specified; equally, each channel connected to the port can be defined either as an up, as a down, or as a bidirectional connection. Advantageous is that the design can be configured with a single structure of configuration map parameters, which makes reconfiguration of the system highly automatic.

The interconnection matrix ports perform transaction routing and channel arbitration in the *OCC*. The interface of the active ports is common for all ports of this type, while the passive ports can have either a unified interface or an interface specific to the peripheral controllers connected to them. The latter is the result of integration of the passive ports with the interface transactors for the peripheral controllers of two different types (*IT-T1*, *IT-T2*). The integration is performed for minimising the data-link latency. Alternatively, the interface transactors as standalone components can be connected to the passive port modules with the unified interface. The availability of the unified passive ports gives an advantage for integrating new passive components without the need to modify the communication system kernel. The presence of the interface transactors as standalone components of the communication system can be seen as an extra level of platform-specific system adaptation. Selecting the type of a passive port is also an option in the configuration map of the *OCC*.

The peripheral controllers of the *EIF* infrastructure are the next layer of adaptation of the communication system to a specific hardware platform, which integrates the off-chip devices at one side and the inner part of the system on a chip (SoC) at the other side. These components are realised to be efficient for a wide range of applications, yet at the same time they simplify the signalling protocols of standard IC devices for application-specific components by pulling up the level of the data abstraction towards the application level. The system on a chip contains four types of peripheral controllers: a Local Bus controller (interfaces the PLX PCI chip), memory controllers for ZBT-SRAM and DDR-SDRAM interfaces, and a CameraLink controller.

The *EIF* infrastructure also realises the mechanism for interaction between an application running in the host system and the application executed on the FPGA board. It acts as a bridge between these two systems being an active agent in the *OCC* subsystem and a subordinate part on the onboard bus. This application interface functionality resides in an add-on to the Local Bus controller in an *Expansion Board Interface* (*EBI*) block.

The *EBI* block contains both the application specific and the auxiliary system elements. Beside the Local Bus peripheral controller core it includes the *Device Interface Unit* (*DIU*), which offers the hardware interface for the firmware located on the host and acts as an interface transactor between the Local Bus core and the Switch Matrix. The application specific unit is the *Application Specific Register Set* (*ASRS*) storing the application specific information for configuring the GSC parameters. It interacts with an *Application Control Unit* (*ACU*), which performs the arbitration of the overall application via a control link.

4.2.8 Clock domains

Due to the timing constraints of the external interfaces, the on-chip system must operate at least in three different clock domains. One domain is associated with the Local Bus board-level interface, which is restricted to operating at a clock frequency of 66MHz. The second domain is conditioned by realisation of the double data rate interface of the dynamic RAM. The third domain is the clocking domain of the system kernel, which can be further subdivided into the main kernel domain and subordinate domains to clock the processor arrays inside the linking and the downpropagation units at a multiple of the kernel clock frequency to perform faster sequential computing.

4.3 Interface and communication model

Signal level interfaces play a significant role in system design. On the one hand they determine characteristics of the data streams circulating in a system, thus inducing the system's qualities at the higher abstraction level. On the other hand the interfaces predetermine the interior design of the components of a system at the lower implementation level. From the point of view of the SoC design, peripheral interfaces act as technological constraints imposed by the silicon devices employed in a platform. Interior interfaces of an SoC are relatively free from restrictions on signalling protocols they realise. These interfaces are design for customising the SoC for particular applications and introduce a number of different adaptation layers (from application specific to generic and technology specific) to the SoC infrastructure to gain flexibility and reusability of the system's components.

4.3.1 System Interfaces

The FPGA interfaces three types of silicon devices considered in the design of the computing system. They are PLX PCI-to-Local Bus Bridge, DRAM, and SRAM devices.

The Local Bus interface is a synchronous burst-oriented bus interface with a frame and a strobe signalling control and a wide range of data transaction management mechanisms including:

- suspended, split, and aborted transactions,
- bus holding,
- transaction timeout and recovery,
- interrupt and direct memory access.

The System-on-FPGA operates in a slave mode, while the onboard PLX PCI chip acts as a bus master and an arbiter. Detailed information on the signalling protocol can be found in [132].

The DDR-SDRAM interface is a typical interface to synchronous high-capacity dynamic RAM with a separate column/row addressing and a programmable burst length for atomic memory access transactions. The specific feature of the interface is a source-synchronous double-rate data sampling. The details on the DDR-SDRAM interface can be found in [133].

The ZBT-SRAM interface is a specific synchronous interface to the fully addressable array of static memory cell. The interface is designed for burst atomic transactions of a signal controlled length (within the range of up to four memory locations with turnover). The peculiarity of the ZBT (Zero-Bus Turnaround) interface compared to a conventional synchronous RAM is the absence of idle cycles between read and write access operations. This allows the full utilisation of the available bandwidth of its low latency data channel. The details on the interface can be found in [134].

The interior interface that can be treated as a precondition for the HW GSC system implementation is the interface of the reused DDR-SDRAM controller IP core [135]. The core has a generic user-application interface with separate command and data buses. The interface of this core significantly simplifies the communication with the SDRAM devices, which have a complex signalling control protocol.

Two interfaces were specially designed for the given infrastructure subsystem: an interface to a ZBT-SRAM-Controller core and a Switch Matrix interface.

The requirements to designing the ZBT-SRAM-Controller interface are formulated as follows:

- to simplify the interface of the silicon device, which bears the technological peculiarities of the memory IC, yet keeping the maximal throughput of the ZBT-SRAM interface,
- to make the interface more friendly to computation systems, yet keeping it generic,
- to provide full control on data flows assuring data consistency.

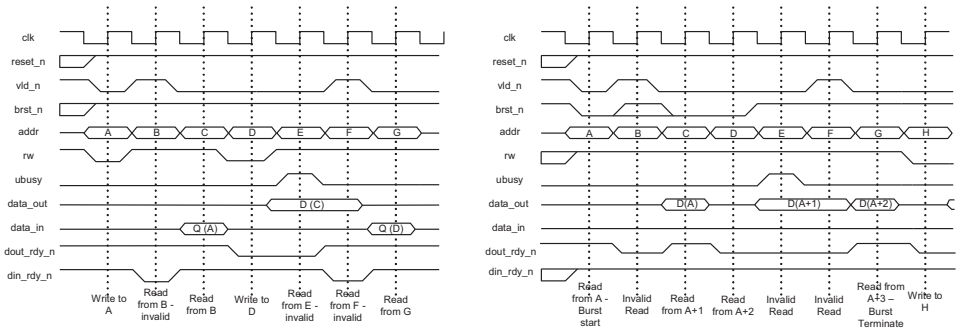


***Figure 28**: ZBT controller interface cycle diagram*

The first aim is achieved by excluding ambiguity in the signals' function (one signal – one function). The second is fulfilled by providing a simple burst mechanism without alignment or burst length constraints using a burst frame signal (*brst_n*). The third is achieved by introducing a handshaking mechanism that guarantees both sides of the communication function with their own data-forwarding pace without data dropping.

A memory transaction via the interface consists of two subsequent phases: transaction initialisation (by a validity signal *vld_n* for access addresses on an address bus *addr* and an access type signal *rw*) and subsequent data transfer (via two unidirectional data channels *data_in*, *data_out*). The second

phase may follow the first one in an arbitrary number of clock ticks, subject to the handshake control (using two controller driven data ready signals *dout_rdy_n*, *din_rdy_n* and a client-side signal *ubusy*). Transactions are atomic, meaning that a transaction that is initiated first is completed first. The initialisation of the next memory transaction is irrelevant to the completion of the previous. This means a memory operation may be initialised each clock cycle regardless to the completion of an operation issued before (transaction are pipelined). Figure 28 illustrates the principles of the interface functioning. The complete interface specification of the ZBT-SRAM controller can be found in Appendix A.7.1.

The interface of the *Switch Matrix* active ports is maximally adapted for the needs of the application. It provides a mechanism for atomic transactions of a fixed length of one to four memory words. The GSC data are mapped to these transaction atoms. For the *Switch Matrix* component reuse in alternative on-chip applications the upper limit of the transaction length can be adjusted by a parameter. The length of each transaction is determined by a field in the transaction descriptor, which additionally includes a destination port identifier, a transaction direction flag (read or write), and an address of the first transaction word in the memory space of a passive *Switch Matrix* client.

The key characteristic of this interface is that it consists of three independent data-asynchronous channels: a transaction descriptor channel and two data channels for upward and downward data streams. Together with a simple handshake mechanism (channel request and channel grant signals) these decoupled data channels provide a maximum flexibility for the data-flow organisation inside processing units connected to the *Switch Matrix*, since their interfaces mimics the interface of a queue – the most simple and popular dynamic data element for data-flow organisation in a computing system. The up and down data-stream channels are additionally equipped with signals for data streaming: a channel forward grant signal indicates the channel availability in the following clock cycle. This signal can be used to assist a data-flow control FSM (finite-state machine) in a client module to maintain continuous data streams.

The passive port interface is analogous to the interface of the active ports of the *Switch Matrix* except for the direction of data streams in relation to the request and grant signals in the respective channels. The detailed description of the *Switch Matrix* interfaces is given in Appendix A.7.2.

4.3.2 System network traffic and data buffering scheme

At any time of the GSC execution there is only one active agent in the system's network. This is why the traffic inside the *Switch Matrix* is fully defined by the buffering policy of a GSC processing unit and the channel connection configuration of the particular unit.

The GSC processing units have several connection schemes to the *Switch Matrix*. Depending on the chosen configuration their up and down data streams can be routed through a single *Switch Matrix* channel, or separated into two channels for the whole incoming and the outgoing traffic apart resulting in a full-fledged data-stream pipeline, or combined in a way that all channels are nearly

balanced with respect to the data flow intensity with the most dense data flow being routed through a dedicated channel. Alternatively, one configuration provides a dedicated channel to each data stream, which is used if all the GSC processing units share common *Switch Matrix*'s connections. The choice of one configuration is the matter of the compromise between the resource availability and the data throughput in the communication subsystem.

The buffering control scheme deployed in the computation units is layered and relies on modular organisation of control components, which simplifies assembling and maintenance of the system in various configurations. At the lowest level (close-to-buffer layer) the control scheme realises an aggressive data policy trying to maintain a constant input or output stream to/from each buffer in portions (bursts) of buffer sizes. The next layer (layer of data streams) is responsible for batching buffer bursts. It is required for protecting the communication subsystem from overflow conditions and for multiplexing or demultiplexing data within a group of data streams of the same direction that share a common *Switch Matrix* connection. The highest control layer (layer of stream groups) manages the data streams at the level of the application preventing logical deadlocks in the GSC processing units.

It is important to note for further simulation analysis that in the present system it is not possible to organise an uninterrupted upstream to a read buffer even in a separated channel configuration. This is explained by the fact that the DDR SDRAM controller has no feedback signal for the data paths' control. This means that in some operating conditions, when the system cannot accept data from the IP core, the data must be dropped, implying a transaction recovery procedure. To keep the infrastructure simple the control logic of the buffers does not allow initiating new memory transactions until the input buffers are free or the output buffers are complete. This entails a certain read data latency being appreciable for the GSC application performance as shown in Section 4.3.4.

4.3.3 Interface temporal logic and bus functional executable models

To isolate possible integration level mistakes the interfaces between the architectural blocks of the system are constrained with signal-level temporal assertions. All the signal protocols deployed in the system are translated to temporal logic interface descriptions using the SystemVerilog Assertion (SVA) syntax executable during simulation. These temporal logic rules guarantee the interface compatibility of any first introduced modules and assure the model consistency in cases of partial system rearrangements or reimplementation of the system's components.

A cycle accurate interface model of the complete communication subsystem is introduced to the GSC TLM as the further refinement of the HW GSC model. This bus functional model of the communication subsystem allows evaluating the impact of the precise signal level interfaces on the GSC performance taking into consideration particular organisation of the on-chip communication infrastructure and implementation parameters of internal elements of its components (sizes of internal queues, arbitration policy, influence of register balancing[73] in the data paths, etc.).

[73] Register balancing is a design technique to meet the timing constraints of a design by (re-)placing auxiliary registers in its data paths.

To realise the model, the components of the communication infrastructure are implemented as functional (behavioural) models. Such models are built on virtual modelling objects referred to as transaction channels and interface the surrounding environment by the mechanism of function calls to realise the communications: the encapsulated channels use function import and export calls for interacting with adjacent functional modules. The calls emulate the signal-level bus activity with cycle accuracy inside the modelling channels and mimic the communication system's latency introduced by each component of the communication system.

From the perspective of hardware implementation, introducing the bus functional model of the communication subsystem with dedicated virtual channels contributes to the gradual transition of the description of the total system from the functional to the register-transfer level, as a function call interface at either end of a virtual channel can be easily substituted with a signal level interface for the channel to become a bus functional to cycle accurate transactor for connecting with RTL blocks.

4.3.4 Modelling results

After the complete translation of the model components into the time domain, the precise throughput of the GSC application and the influence on the GSC performance of the system's infrastructure can be measured. Figure 29 shows a typical profile of the application execution time for the processing of various hierarchical layers depending on the processing power of the GSC processing units. It can be seen that the performance progress is rapidly exhausted with the increase in the number of array processors inside the units. This is explained by the influence of the system's infrastructure.

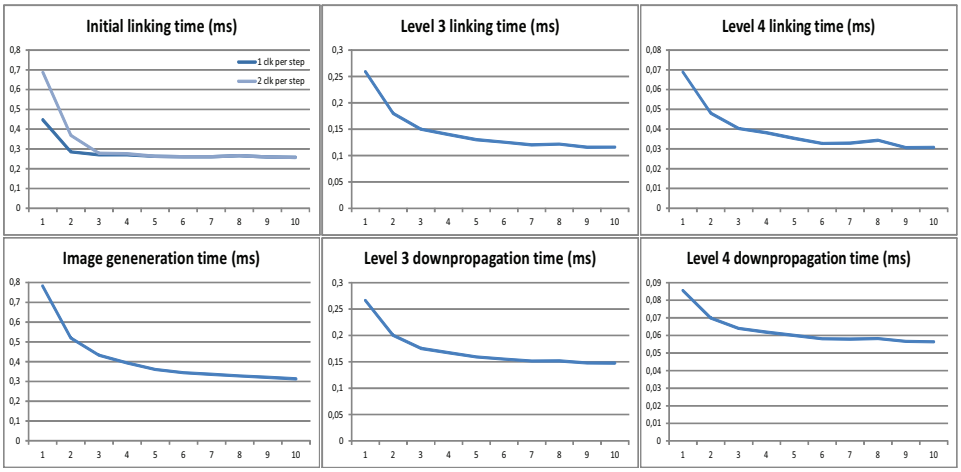


***Figure 29**: Level processing time for different number of linking and downpropagating processors in a processing unit (as an example of a 256^2 image processing)*

A number of data-channel parameters are measured for the communication subsystem using statistics monitors integrated in the communication components to estimate the impact on the application

performance. The measurements show that the throughput of the DDR SDRAM controller interface, which is about 0.49 data samples per clock, and the data latency for read access bursts at the active agent ports, which amounts to 10 clocks for reading data from the external static memory and about 18 clocks for retrieving data from the dynamic memory banks, are critical for the GSC application. The relatively low throughput of the DDR SDRAM controller is explained by the fact that it is pre-configured for accessing the SDRAM devices with the minimum selectable burst length.

The influence of data-link parameters[74] of the communication infrastructure is best illustrated by the bandwidth utilisation ratios for various data streams in the external channels of the GSC processing units. Figure 30 and Figure 31 show the dependency of this ratio on the number of array processors in a unit for all incoming and outgoing streams at different hierarchical levels for *the linking* and *the result generation phases*. The yellow colour denotes the critical data streams which limit the throughput of the GSC processing units' pipelines. Thus the data for the initial linker (Figure 30, upper row) indicate that the most dense data flow of coded islands IS^{CE}_{0} to *the GSC DB* rapidly exhausts the bandwidth of a SDRAM channel with the increasing of the amount of output data from the linking processors. The less dense downstreams of overlap-list entries OLN^{S}_{2} and linked islands IS^{F}_{1} just follow the curve of the limiter. The opposite tendency is demonstrated by the pixel up-stream profile, which indicates a drop in the bandwidth utilisation ratio with the increasing in the number of the linking processors. This effect is typical for the upstreams if they do not act as the limiting factors. This is explained by the fact that the overall incoming traffic decreases with the increasing in the number of island rows in a row block due to the decrease in the overall number of overlapping island rows.

The linking performance at higher hierarchical levels is constrained by the incoming traffic of the overlap-list structures OVL^{S} (Figure 30, middle and lower rows). As it can be seen, the data stream of this type saturates the bandwidth of a channel of the SRAM device the overlap-list array is stored in. The other data streams having lower data density simply follow this curve. It is worth noticing that the saturation curves have positive inclination towards higher numbers of the array processors. This results from the initial delay in arrival of read packets at the front-end buffers of the GSC pipelines: as noted in Section 4.3.2 the front-end buffers do not maintain uninterrupted upstreams. The specific weight of these gaps in the total traffic through a channel determines the behaviour of the utilisation ratio over the number of processors parameter.

The same fundamental rationales for explaining the characteristics of the external data streams apply to the downpropagation unit. The limiting factors for the performance of the unit are the throughput of the SDRAM core's interface and the initial latency for the read buffer packets constraining the upstreams of subislands IS^{CE} from *the GSC DB* (Figure 31).

A significant practical value of the presented data is also in that the bandwidth utilisation graphs distinctly indicates which data streams can be multiplexed in common communication channels without affecting the throughput of the GSC processing units.

[74] Parameters characterising a data connection in a communication system such as data access time, specific data-flow density, etc.

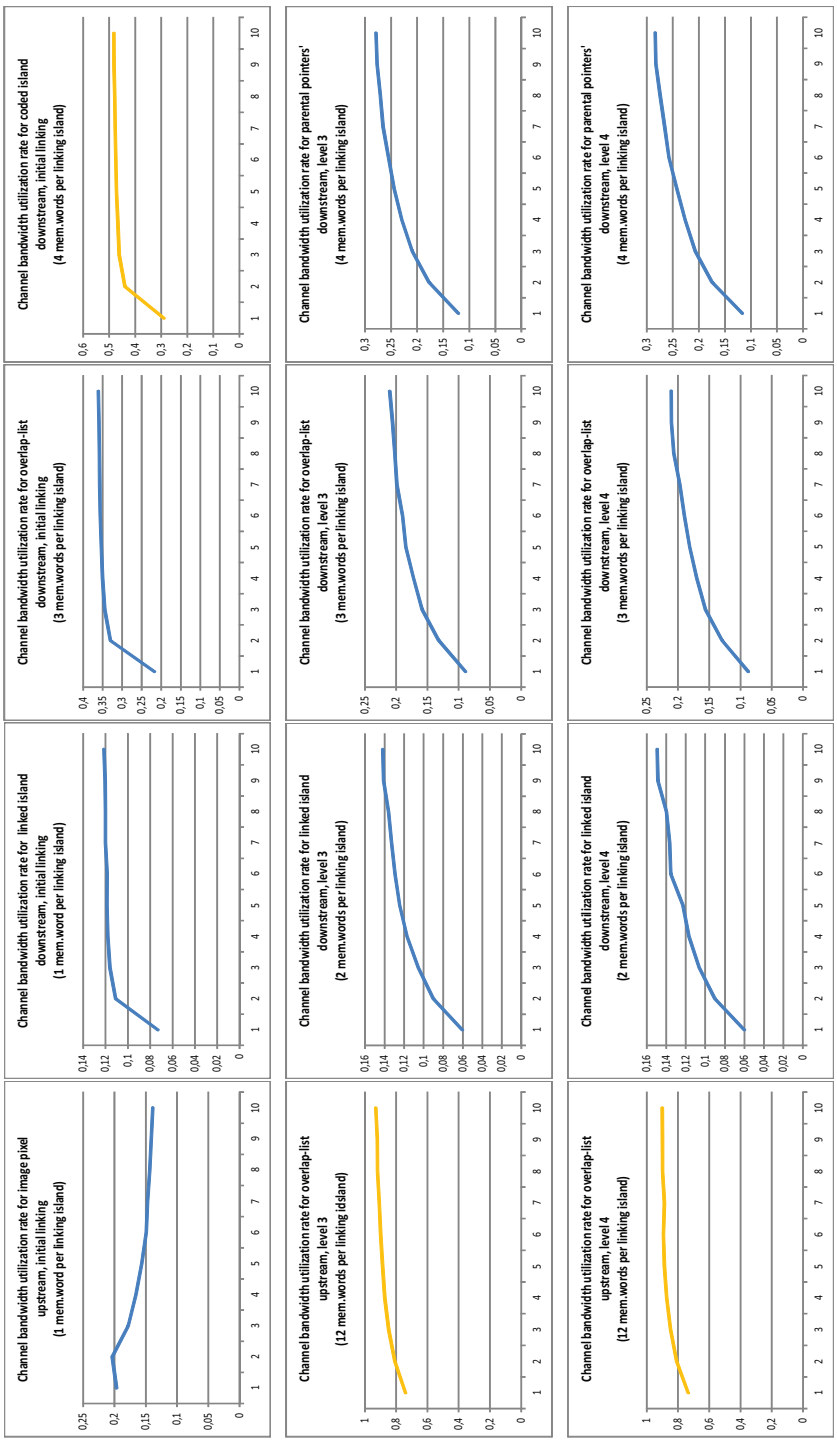


***Figure 30**: Channel bandwidth utilisation ratios for incoming and outgoing data-streams of linking units*

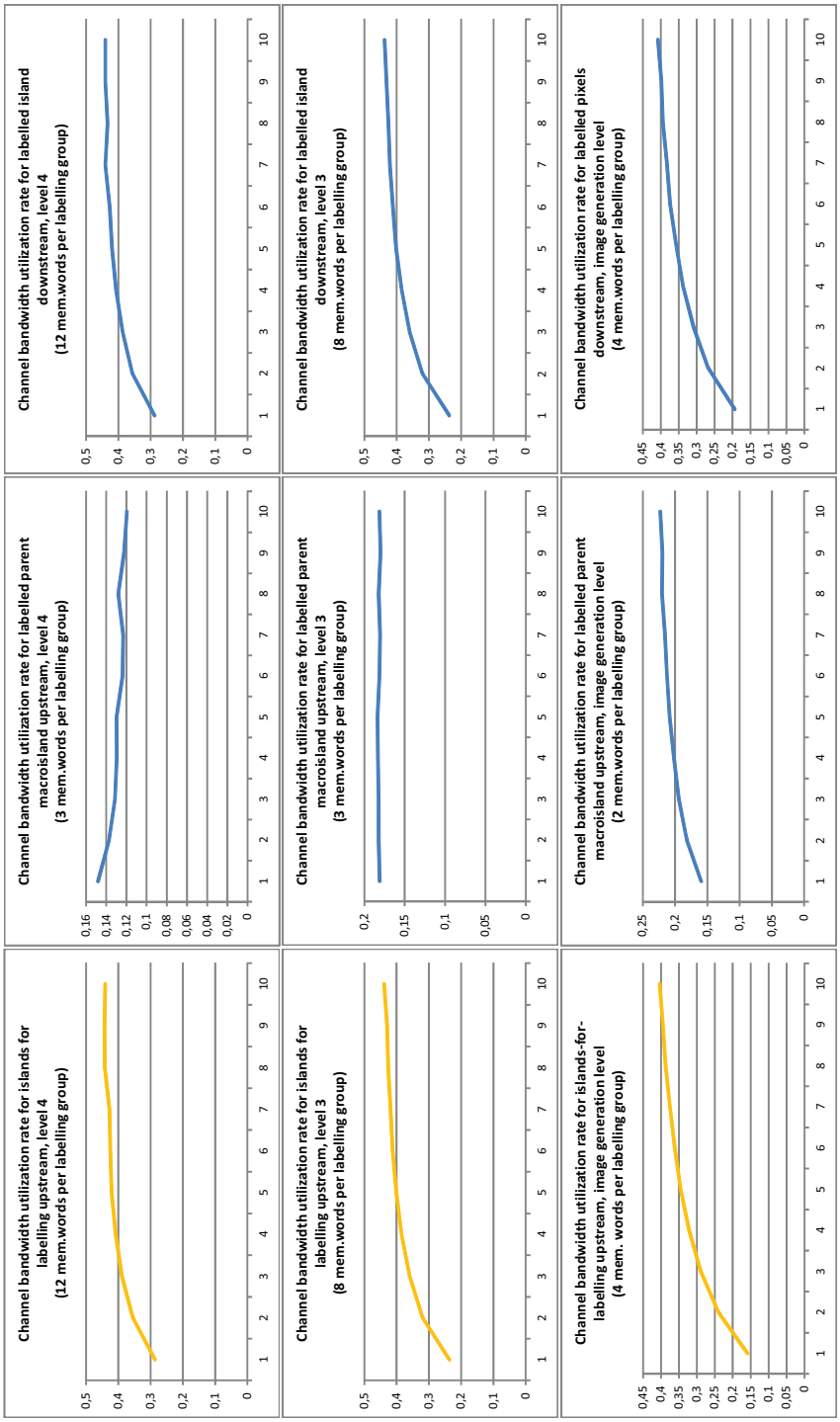


***Figure 31**: Channel bandwidth utilisation ratios for incoming and outgoing data-streams of downpropagation units*

4.4 Register-transfer level implementation

In developing the synthesisable models a special attention is devoted to tunability and portability of the design. A design technique based on modular structural description of the system at the level of functional units is adhered in the RTL implementation. By adopting this approach the coarse architectural blocks of the system are realised as structures built on generic functional units, such as storage, computation, and data-path control components, glued with customised RTL logic described in the procedural language style. These units realise operations of a higher functional level than the logic design primitives such as multiplexers, encoders, counters, and similar do; at the same time, they remain aloof from particular application algorithms being common functional components. Thus, the functional units can be seen as application customisable but not as application specific components of the system. Elaborating flexible general-purpose functional units requires firstly determining design patterns common to a number of application specific blocks, secondly exploring their generalisation capabilities for a wider range of applications, and thirdly providing the measures for maintaining the tunability of the components that realise these design patterns.

Compared to the plain unimodular RTL description which realises structuring by means of procedural constructions, the modular structural approach benefits from a higher level of representation, which, what is practically significant, particularly results in less space for intrusion of design faults. This is technically explained by the fact that the design description to realise higher functional precision relies on the functional units designed as regular components and reused in many architectural blocks and thus requires concentrating on only integrating the components and realising application specific functionality.

Equally important is that the realisation of the abstract operations is optimised at the signal level for a predicted scope of applications: in contrast to design for high-level synthesis intended for attaining comparable abstractness, the abstraction of the data manipulations with this design technique is not achieved by abstracting the operations from their realisation, but by extensive parameterisation of the generic functional units at RTL. However, the design of the functional units requires appreciable initial investments in elaborating the tunability and proving the reliability of the components.

At the same time, structural description at the level of technological primitives is practically avoided in design, as the low level representation leads to lower maintainability and flexibility in introducing changes to the design. Exceptional are the cases in which the architectural elements of the target device are difficult to infer from generic description by current synthesis tools. The examples are DCMs and DDR registers in I/O blocks of the FPGA.

Amongst others the following parametrisable functional blocks are designed:

- handshake controlled queues with a parametrisable data width and queue depth and an optional forward write/read operation grant signals, which can be realised with either built-in RAM blocks or distributed memory and configured for the first word falls through policy;

- a stack memory with a flexible data width and stack depth, which can be realised using either on-chip RAM blocks or distributed memory and can be configured to operate in several modes of data guarding (handshake control, overwrite exception, no data protection);
- a fully associative cache with one or two data tag ports and a selectable data replacement policy, which can be implemented with either built-in RAM blocks or distributed memory;
- serialiser/deserialiser control components designed to be integrated with buffers of adjustable buffer depth to control data uploading/offloading from/to data channels with a simple handshake control mechanism;
- configurable data channel multiplexers/demultiplexers for atomic data transaction handling;
- overwrite register guard circuits, which operate in either a handshake or a posterior overwrite-fault-firing mode, for protecting data in registers of arhythmic data flow paths;
- a signed/unsigned integer pipelined multiplication core[75], which takes two operands of selectable widths as inputs and outputs the result with the initial latency in ticks equal to the multiplier width, provided that the control signal for stalling the pipeline is deasserted;
- a sequential signed/unsigned integer divider with reminder correction, which takes operands of adjustable width and is equipped with various operation control and indication signals;
- a pipelined signed/unsigned integer divider, which is implemented as an automatically configured array of sequential dividers;
- common arithmetic and logic operations, such as a distance against threshold comparator (absolute value of difference), which are implemented to speed up the operations most commonly used in the GSC and are specially optimised at the Boolean logic level;
- nested loop index counters with a selectable number of nested loops and variable index ranges used specifically for data element indexing with regard to the GSC topology.

In implementing the functional units the use of automated IP core generation is rejected to avoid any gate-level description produced by design automation tools (typically in a vendor specific format[76]). Although the automated core generation can significantly reduce the initial development time, it limits the portability of the design by restricting it to a particular device architecture family. However, to benefit from exploiting the hardware cores of the FPGA architecture (in particular RAM and DSP blocks) the RTL models are implemented in such a way that the hard core primitives can be inferred from the functional description by a synthesis tool.

Ubiquitous parameterisation of the constituting components provides flexible tunability of the application-specific and the infrastructural components and thus allows fast modification of the HW GSC implementation and reusing the on-chip auxiliary subsystems. Together with the technologically independent description this makes the design portable between various hardware platforms such as FPGAs and ASICs.

[75] The soft core can be used when hard-core multipliers are not available in the target FPGA platform.
[76] Generated IP cores (firm cores) are usually optimised at the gate level and are realised in a form of netlists for a particular architecture.

4.4.1 Application-specific blocks

4.4.1.1 Coding Unit

Due to the relative inexpensiveness of a coding unit realisation the pipelined coder is chosen for implementing in the FPGA (Figure 32[77]). The coder gets an island of pixels extracted from the pixel buffer and a threshold feature as input to form four basic regions (*BR*) associated with four fixed pixel positions as discussed in Section 4.2.5.1. Depending on the threshold condition satisfaction the pixel positions of the basic regions are set in respective bit-flag vectors (*POS_V*), and selector keys are generated to identify the edge pattern in each basic subgraph. All possible partial sums (*AF*s) are reckoned in parallel, and the selected sums initialise the basic region features by using 8-to-1 muxs.

At the next stage all basic regions are checked for common pixels using bit operations on the position vectors to form the *Linkage Table* (*LT*) identifying which basic regions will contribute to which resulted *CE*s. The position vectors of the basic regions are propagated unchanged to this stage. All partial sums of the basic region averages are calculated and registered to be selected and assigned to the correspondent CE^Fs at the next pipeline stage. At the final stage of the coding the export tables are formed. To do this every pixel position correspondent to the outer pixels of an island is checked for region occupancy in all CE^Vs. Considering that the same pixel can be included in several duplicate *CE*s, a priority encoding scheme is used to select the first occurrence of a bit set in the correspondent position to avoid region ruptures during subsequent linking. The correspondent CE indexes and the region features are assigned to the region descriptors of the export table.

It is important to note that for computing the feature averages of the regions (CE^Fs) the division operation is not used explicitly, instead the digit capacity of the CE^F is extended from 8-bit integer for pixel features to 12-bit fixed point format, realising virtual division to increase the accuracy. It is also notable that the pixels in which the basic regions overlap contribute to a resulted average twice, shifting the averages to their features. This effect can be interpreted as a merit of the coding realisation as it shifts the average to the feature of a central point of a region graph compensating a possible chain effect.

The given design is operable at 148MHz and requires 1.8% of CLB slices (1% DFFs[78]) of the target device for the implementation.

4.4.1.2 Linking Unit

The Linking Processor is realised as a stack machine (Figure 33) for traversing the *Macroisland Overlap Structures* using the stack as the memory for storing branching points of a region tree (*Return Stack*). The goal of this linking stack machine is to reduce the traversing time of a tree. For achieving this all transition points of a current island are analysed in parallel to reduce the time for making a transition decision. To do thus all features of the candidate regions are compared against

[77] For shorter notation the first subscript indexes in Figure 32 and Figure 33 indicate the indexes of region associated variables within *IS*s.
[78] DFF stands for D-type flip-flop

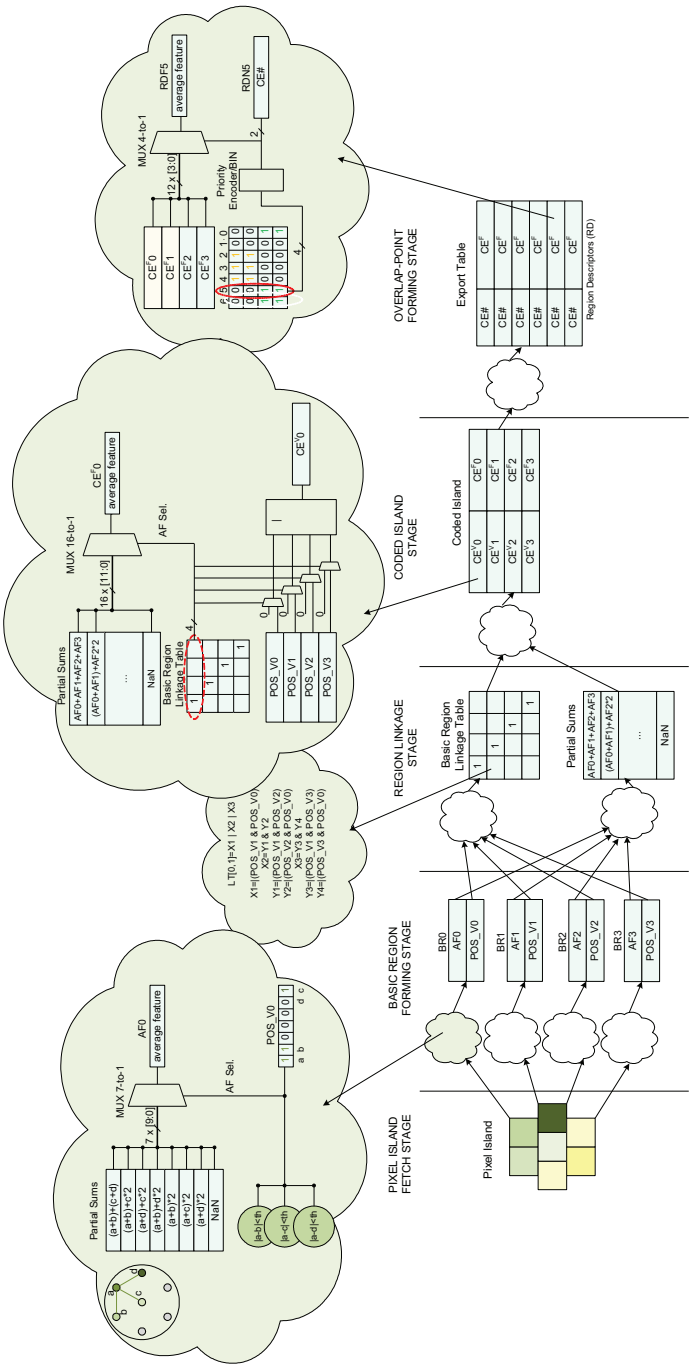


***Figure 32**: Parallel Coder pipeline*

the current macroregion feature forming *a threshold condition bit mask*. This mask is compared to *a position vector* of the seed region, *an overlap position vector* indicating the regions' connectivity from *the Macroisland Overlap Structure*, and a bit mask of the closed transition points being modified during the linking[79] (see Section 4.2.5.2) to form *a region transition mask*. In the case of *the Initial Linker* the seed region's *position vector* is taken directly from the correspondent *CE* of a coded island (CE^V), while in *the General Linker* this vector is formed by comparing the index of the seed region with the region indexes in the import table of the current island.

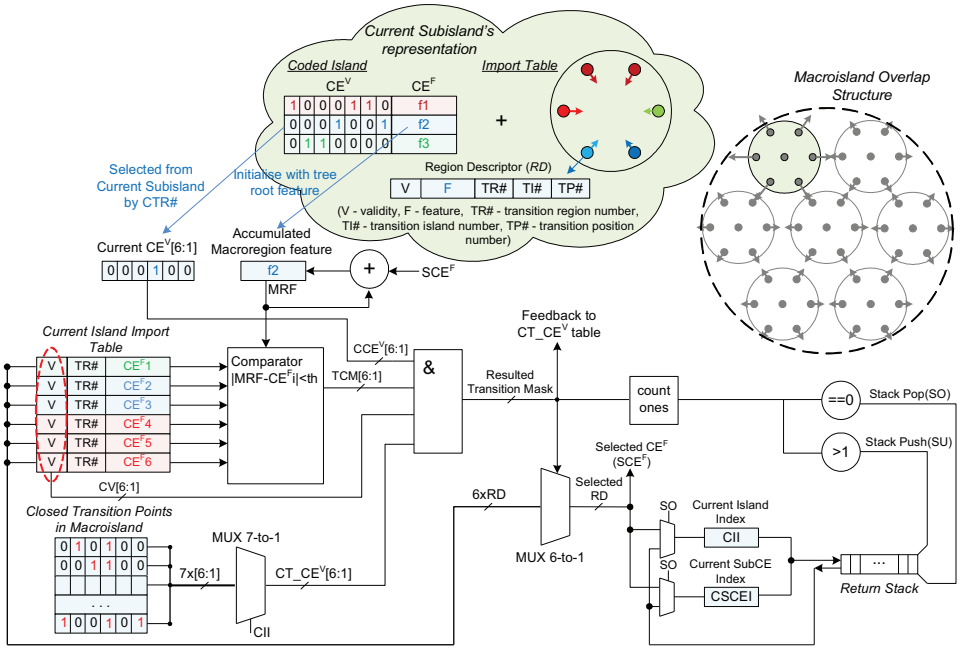


***Figure 33**: Initial Linker stack machine*

Using *the region transition mask* as a selector key after priority encoding, one candidate is selected for transition. Due to the fact that each overlapping point in *a Macroisland Overlap Structure* of the *General Linker* contains three region descriptors the total combinatorial logic for candidate selection becomes too complex to realise this scheme in a single clock step implementation. That is why the logic is split up using intermediate state registers, in contrast to the *Initial Linker* implementation, which is realised with one clock per linking step. As shown before the two-stage linking scheme does not influence the *General Linker*'s processing time in certain configurations.

To reduce the signal propagation delay in the combinatorial logic the digit capacity of the comparator which test the absolute feature difference against the threshold is reduced under the following assumption. Realistically, the range of the merging threshold value does not exceed a certain frac-

[79] In fact the closing of transition points is realised by unsetting bits in the overlap position vectors, so the two last bit vectors are combined in the present implementation.

tion of the range of the region feature to avoid oversegmentation. This means that it is sufficient to compute the difference of the region features in only some least significant bits, comparing the most significant bits only for equivalence. Moreover, the threshold condition for overlapping regions is always checked in the previous linking stage during *OLN* creation; thus, checking the threshold condition during tree traversing is needed only to avoid the chain effect. In practice this means that it is not necessary to compare all most significant bits of the features but only a few bits higher the threshold range digits. The limit of the threshold range is defined as a divisor for the range of feature values in the linking processors. Being a compile time parameter, this leads to the reduction of combinatorial logic in particular hardware implementations.

The *Initial Linking Processor* and the *General Linking Processor* designs are operable at 129MHz and 112MHz and occupy 2.7% and 8.7% of CLB slices (1.5% and 8.2% DFFs) of the target device, respectively.

The *Overlap-Point Processors* load the input data from the output buffers of the linking stage and process the overlap pairs sequentially to arrange them by their weight in the resulted *OLN* structures. It is important to note that the overlap points in the underlying island lattice are arranged in a constant pattern within island slices and this pattern repeats over two adjacent rows of linked islands. That is why to avoid dynamic routing of IS^{F}_{i} and IS^{P}_{i-1} to the *Overlap-Point Processors* array, which leads to complication of the *Linking Unit* hardware, it is reasonable to configure the *Linking Units* to process an odd number of island rows in the island row block.

The *Initial Overlap-Point Processor* and the *General Overlap-Point Processor* designs are operable at 116MHz and 129MHz and occupy 0.5% and 0.6% of CLB slices (0.5% and 0.6% DFFs) of the target device, respectively.

4.4.1.3 Labelling Processors

The *Labelling Processor* consists of four equivalent processing lines correspondent to four subislands in the labelling island group. The labelling algorithm described in Section 4.2.5.4 is implemented in a form of pipeline for processing the CE_is of the subislands in series (Figure 34). The pipeline stages are as follows:

- each subisland processing line is loaded with a new subisland region CE_i;
- two parent regions CE^{L}_{i} indexed by the child's CE^{P}_{i} are loaded;
- the distances between child CE^{F}_{i} and each parental region CE^{F}_{i+1} features are calculated;
- the subisland region CE_i is assigned the segment label and the segment feature of the most similar parent region; if the region has no parents a new segment key is generated and the feature of the region becomes the segment feature.

At the higher levels the pipelines are fed with CE_is until all regions of an IS^{CE}_{i} are forwarded to the pipeline or the first empty region is met. At the lowest level all four CE_0s are forwarded to the pipeline as an empty *CE* does not mean the last *CE* in an IS^{CE}_{0}. Moreover, the lowest level processing

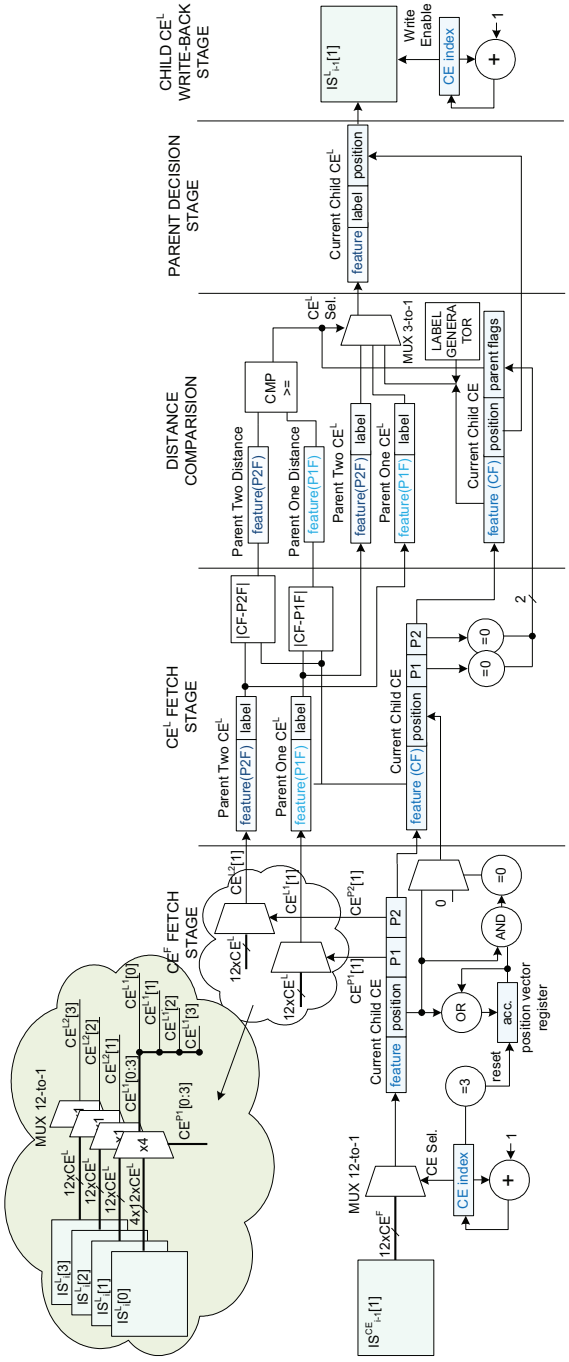


***Figure 34**: Downpropagation Unit single lane pipeline*

requires filtering of the duplicate coded regions not to produce false segments in the resulted image. To exclude this kind of *CE*s each processing line of the *Labelling Processor* records the region positions CE^{V}_{0} of the processed *CE*s in a single position vector register. If the current *CE* matches at least in one position of set bits with the position vector register, this *CE* is treated as duplicate and a segment label is not assigned to it. It is notable that to identify a coded region unambiguously it is sufficient to check only its four pixel positions (the central and its three equidistant outer positions), which is used to reduce the hardware resource consumption for the processing line implementation.

This pipeline implementation extends the processing time of one island by two clocks due to the insertion of two additional processing steps to meet the timing constraints of the design, although it does not influence the overall performance of the *Downpropagation Unit* if the number of labelling processors in the processor array is higher or equal to 4.

The processor design is operable at 127MHz and occupies 6.6% of CLB slices (6.2% DFFs).

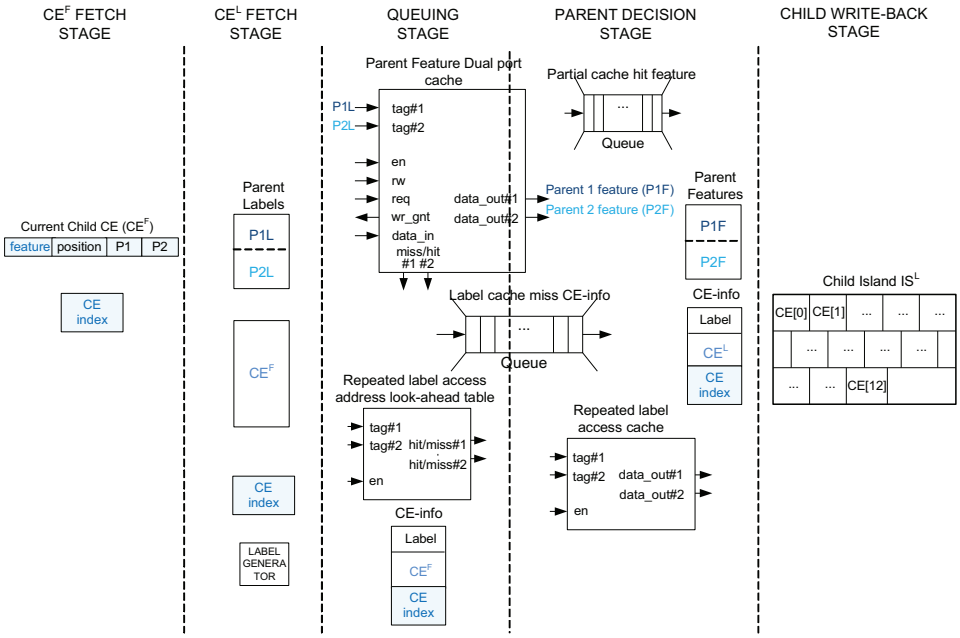


***Figure 35**: Labelling pipeline with out-of-order execution*

The second type of the *Labelling Processor* is designed for processing images of the higher resolutions for which the segment features are stored separately in *the segment key table*. This type of processor relies on two level segment feature caching to reduce the latency of stochastic data access to *the key table* by exploiting the spatial locality of segments in images. The first level cache (24 entities, dual-port) is placed in each island processing line, while the second level cache (64 entities, single-port) is shared among all four processing lines in the processor. This caching scheme showed relatively high efficiency of 0.64 cache hit per key table read access for these low cache

capacities (for cache efficiency measurements refer to Appendix A.6). The second level cache plays an important role in the caching scheme introducing 41% of efficiency gain in comparison with the scheme equipped with only the first level cache. Its significance increases for highly segmented fractal images. It is important to note that the increase in the cache capacities beyond the given values within a reasonable capacity range leads in notably slow increasing in the caching efficiency.

The scheme of the labelling processing pipeline of the second type is shown at Figure 35. This pipeline exploits the mechanism of out-of-order execution to smooth out the pipeline stalls caused by missing segment features in the local cache.

The labelling processor has the following pipeline stages:

- at the first stage the pipeline is loaded with a CE^{F}_{i} and its index within its IS^{CE}_{i};
- at the second, two parental segment labels are extracted from parantal IS^{L}_{i+1}s, and a new segment label is generated and forwarded to the label queue to be written to *the segment table*, if the parental pointers in CE^{P}_{i} are not valid;
- at the third stage the parental segment labels are forwarded to the dual-port label cache of the first level (L1) to get the features of parental segments; the labels are also checked in *a Repeated Label Access Address look-ahead table* to avoid a repeated access to *the segment key table* in the case if the segment information has already been requested from the external storage, but has not yet arrived to the label feature cache;
- in cases of a cache miss all information from the previous stage is pushed to *a Label Cache Miss CE-info queue* for intermediate storage until the parental segment descriptors (segment features) are delivered from the second level (L2) cache or the external storage at the fourth pipeline stage; while the *CE* with cache-missed labels is hold in the queue, the next *CE* can be processed (out-of-order execution), if the associated data are ready for it;
- when the parental segment descriptors are delivered at the next stage, the *CE's* data are retrieved from the queue, the parent features are compared with the child feature, and the correspondent parent segment is chosen for downpropagation; the CE^{L}_{i} is stored to a write-back buffer dependent on the *CE*'s indexes forwarded together with the *CE's* data through the pipeline; the segment information is stored in both the L1 cache and another look-up storage called *a Repeated Label Access Cache* (12 entities, dual-port), which is required for the *CE*s that have been already forwarded to the *Label Cache Miss CE-info queue*.

The design of the *Labelling Processor* of the second type is operable at 103MHz and occupies 18.1% of CLB slices (16.7% DFFs, 3.5% RAM). Due to insufficient resource capacity of the target device this design has not been integrated to the HW GSC system for the final implementation.

To simplify the logic for assigning pixel features and labels an additional pixel decision stage is added to the *Image Generation Block* of the *Downpropagation Unit* which realises the final stage of the segmentation. At this stage the block forms the pixel island by sequentially processing coded regions and assigning the CE^{L} values to corresponding pixel positions of the pixel islands depending

of the position vectors of the coded CE^Vs. This stage requires additional four clocks for the pixel islands to be filled. This latency does not influence the performance of the *Downpropagation Unit* as it is fully covered by the latency of the *Labelling Processor* stage. The image generation back-end of the *Downpropagation Unit pipeline* in the configuration of four island rows for one row block is operable at the clock rate of 118MHz and occupies 14.2% of CLB slices (9.9% DFFs).

4.4.2 On-Chip system infrastructure

4.4.2.1 Expansion Board Interface Unit

As mentioned in Section 4.2.7, the communication of the host system with the board (device) is performed via an *Expansion Board Interface* module, which provides a direct access to the onboard memory through the on-chip communication subsystem and realises the application-level interface of the device. The *EBI* module encloses a Local Bus (*LB*) slave controller for serving the external bus and a *Device Interface Unit*, which processes the transactions received from the *LB* controller and redirects them to either the *OCC* system via an active *Switch Matrix* port or to an application specific register file.

The *LB* controller and the *DIU* are connected by asynchronous queues built on on-chip RAM blocks. The opposite ports of the queues are clocked by two different clock signals asynchronous to each other, which guarantees secure data transition from one clock domain to the other. For generating the two signals the design uses internal FPGA DCMs. The clock from the Local Bus clock domain is used as the carrier for the external bus.

The Local Bus slave controller consists of a number of complex finite state machines interacting with each other. The FSMs generate the response for the external stimulus of the Local Bus and realise the control of three streams of data: a stream of transaction descriptors and up- and down-transaction data streams. The data of all the three streams and the output *LB* control signals are registered in I/O blocks of the chip close to the die's I/O pins to conform to the fully-synchronous design practice, which improves the efficiency of automated layout and raises clocking frequency limits of the design.

The *DIU* at the other end of the asynchronous queues directs the transaction flow to either the *OCC* system or *Application Specific Register Set* depending on the target addresses of the transactions.

The *ASRS* consists of a number of 32-bit registers, which contain both general device and GSC specific information. The general device information registers are a device status register, exception code interrupt registers, and memory descriptor registers needed due to the difference in local on-chip and Local Bus memory space sizes. The GSC specific registers contain application operational information, e.g. image dimensions, thresholds, application status, exception interrupt codes, etc.

In cases of local memory access requests the *DIU* realises a sophisticated mechanism for the data packets conversion due to the difference in data word width of the Local Bus and the *OCC* system.

For efficient utilisation of the bandwidth of the *OCC* channels the *DIU* performs the packing/unpacking of 32-bit Local Bus data words into/out of 128-bit *OCC* data words in cases of linear access pattern detection. Additionally, the *DIU* can provide a buffering mechanism for accumulating the traffic in both directions. In particular the internal logic of the module can detect consecutive write-read access to the same memory locations, the data for which are present in the buffer. In this case the data are read from the buffer directly without forwarding the request to the *OCC* subsystem provided that the content of the onboard device memory has not been locally modified. This caching mechanism can be disabled by a parameter setting.

4.4.2.2 Switch Matrix

A set of hardware queues, which form a communication channel network, is the core of the communication matrix. Depending on parameters setting these queues can be realised either on the FGPA hard-wired RAM blocks (if available) or using conventional distributed logic resources. The queue-based channels are connected to the end-point ports of the *Switch Matrix*. The active port modules perform routing of the incoming transaction streams to different communication channels attached to the ports, while the passive port modules realise arbitration of the incoming channels, both maintaining data links of different end to end connections intact. The arbitration policy is defined by a parameter setting.

Both types of ports have similar realisation of data streams control based on interacting FSMs and transaction request queues. Each port module has separate queues for up and down data-stream transaction requests, which are filled with transaction descriptors (channel identifier and transaction length) at transaction acceptance and are emptied when the correspondent transactions have been served (i.e. the requested amount of data is forwarded to/from a correspondent data channel).

4.4.2.3 Memory peripheral controllers

The ZBT-SRAM Controller core is built on an FSM that realises the coupling of the internal (on-chip) and the peripheral (off-chip) interfaces and provides a flexible data-flow control. To assure operating at high clocking rates, the address and the data signals as well as the output control signals are registered in the flip-flops of I/O blocks.

For secure operating the DDR controller cores their output control signals, the address, input, and output data are registered in I/O blocks as well. For the double-rate data transmission the design exploits the specially coupled flip-flops of the I/O blocks clocked in opposite phases.

4.4.3 Low-level verification aspects

For the functional verification of the RTL implementation of the design a special attention is paid to the reliability of the elementary functional units as the basic building blocks of the system. Each functional unit has been subjected to a thorough verification procedure indispensable for reusable

components. The stimulus-response verification approach is complemented by assertion-based models and guided by functional coverage. The temporal logic property models have a twofold application purpose. In addition to the verification of standalone components, the temporal logic models accompany the RTL implementation blocks in the integrated system through verification tests at higher structural levels. This allows guarding the system functionality locally and ascertains that all critical features of individual components have been properly exercised during the standalone verification, thus providing a feedback to the verification plan if a component feature should have been overlooked. One considerable advantage of the temporal logic constraints for the functional verification of the integrated system is that the temporal logic guards can be easily switched on and off without the need for changing the verification environment of the design.

At the higher structural level special attention is devoted to the verification of the on-chip infrastructure elements and the communication subsystem as a whole. The elements being reusable components designed with generic application intent, the range of their operational conditions and configurations can be significantly wider than the GSC application imposes. Thus, the communication infrastructure requires an exhaustive verification policy. The stimulus-response dynamic verification used to realise this policy focuses on two intrinsic functional features of the communication system: correct traffic routing and lossless data forwarding between active and passive communication parties. A verification scheme based on common memory access patterns is applied to testing the components of the communication system for these purposes.

In this scheme the active verification components resided at one end of the communication channel generate memory access transaction traffic, which is served by the passive verification components at the opposite end. The passive verification components realise the elementary storage function, allowing an active verification component to check the consistency of the data moved through a single communication component, a part, or the whole communication system. The main challenge for the verification with this scheme is the exhaustive traffic pattern generation (including multiple source/target combinations, where applicable) as well as the verification coverage. To increase the verification efficiency, the data paths in the communication subsystem are strictly constrained by the temporal logic checks (based on SVA) used locally in the communication components and the interface rule checkers between the components. This approach helps in localising sources of the functional problems. The common verification scheme for the communication system benefits from reducing effort investments in creating the verification environment, as the most verification components can be reused for verification of various components of the communication system. This requires only replacing one type of signal level interface transactors between the verification environment and a DUV with the other.

The application functionality is checked against the models implemented at the higher abstraction levels. Similar to the verification of the communication subsystem the data paths of the data processing units are guarded with temporal logic constraints, although the assertion-based verification does not play the same important role here, as the application specific units are subject

to more often conceptual modifications. Thus, the temporal logic guards are mainly located in the reusable components of the functional unit. A special procedure is introduced for stress testing of the application level interface as the concluding stage of the logic design verification to verify secure reaction of the device in cases of unconventional behaviour from of the host application.

The postimplementation verification is focused on the observance of timing constraints and the correctness of the synthesis. The gate-level functional simulation is applied to ascertain the synthesis correctness as the synthesiser is not guaranteed to properly translate any high-level language constructions. The design being synchronous (except for three cross-domain data paths realised by asynchronous queues), the timing constraints check is mainly realised with the post place-and-route static timing analysis. The gate-level back-annotated timing simulation is held exclusively for verifying the peripheral controllers to confirm the correct functionality of the on-chip communication subsystem integrated to the total onboard system taking into consideration the signal delays at the board level.

4.4.4 System-on-Chip synthesis and layout summary

The configuration of the system is chosen so that a maximum number of the processing cores can fit in the FPGA resource budget: a three *Linking Processor* array for the *Initial Linker*, a three *Linking Processor* array for the *General Linker*, and a three *Labelling Processor* array for the *Downpropagation Unit*. The interconnection matrix was configured to support two memory channels per processing unit for up- and downstream data traffic to keep resource consumption of the project moderate. The results of synthesis after profound logic optimisation are shown in Table 6.

***Table 6**: Synthesis results (with percentage of hardware resource budget of Xilinx XCV2P100)*

	D-Flip-Flops	*LUTs*	*Block RAM*	*DSP Blocks*	*Operable Frequency (MHz)*
Initial Linking Unit	19031 (20.84%)	24413 (26.74%)	16	6	108.484
General Linking Unit	21360 (23.39%)	17414 (19.07%)	16	7	107.028
Downpropagation Unit	28880 (31.63%)	31607 (34.61%)	32	10	105.667
System Infrastructure	6898 (7.55%)	5375 (5.89%)	116	0	131.010
Complete On-Chip System	77222 (84.57%)	80176 (87.80%)	180 (40.54%)	23 (5.18%)	102.135

5 GPU implementation

5.1 Overview

5.1.1 Function partitioning

Based on the functional model analysis in Section 4.1, the GSC was partitioned and mapped to the Compute Unified Device Architecture (CUDA) for the GPU implementation of the algorithm. The application is represented as a number of sequential invocations of CUDA kernels executed on the computing device. Massive parallelism is exploited for the island processing of each hierarchical level in all stages of the GSC pipeline, while all stages exchange the data via the global device memory. Figure 36 represents the general architecture of the GPU-GSC application.

The GPU-GSC application starts with the allocation of necessary global memory segments on the device for the GSC data-structures and the copy of the initial image from the host to the device. The first stage of the application pipeline – the *Coding* – takes the pixel image as input and generates a down stream of coded regions to the GSC database. These regions are taken by the *Pixel-Level Overlap-list Creation* kernel and are analysed for overlapping at certain nodes in the pixel lattice with the results stored in the *Overlap-list Array* structure. This data structure is further reused by all subsequent stages of the Linking phase. Next the *Initial Linker* kernel generates the upstream of overlap lists and produces the downstream of newly linked regions for the hierarchical level one and the parental pointers for level zero to the GSC database. The higher level *Overlap-list Creation* uses both the parental information of regions from the underlying hierarchical levels *i-1* and the region information from the current hierarchical level *i* to form the overlapping pairs in the *Overlap-list plain*. The next *Linking stage* is similar to the *Initial Linking* except for some functional modification inside the kernel due to the difference in the overlap-list length of the lowest and higher hierarchical level, which is discussed below. The last two stages are sequentially repeated until the application reaches the top of the hierarchy.

The subsequent *Result Downpropagation* phase consists of repeated calls of the *Downpropagation* kernel for the higher levels of the GSC pyramid and a finalising call of the *Downpropagation & Result Generation* kernel for the lowest hierarchical level. Finally, the result images are transferred

to the host. The *Downpropagation* kernel takes the regions with parent pointer information and the segment information of the parents from the GSC database and updates the regions with the segment information propagated from above. The *Downpropagation and Image Generation* kernel does not update the lowest hierarchical level in the database but propagates the segment values directly to the pixels of the result image.

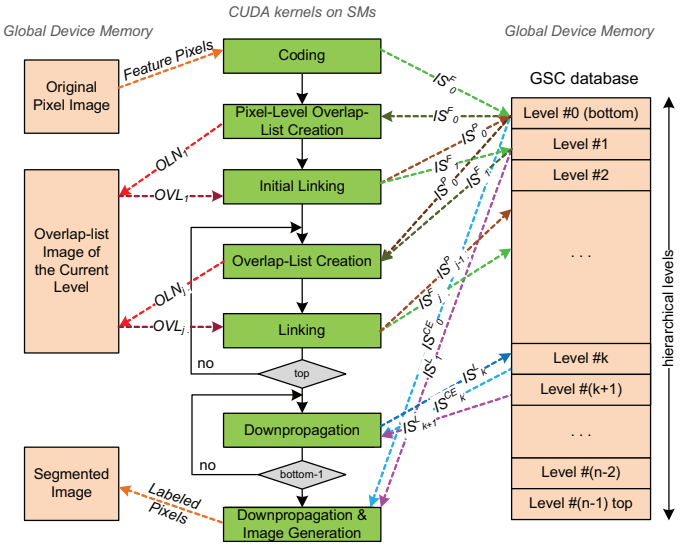


***Figure 36**: GPU GSC application structure*

The principal difference in the computation organisation at the application architecture level compared to the FPGA implementation is that the *Overlap-list Creation* is realised as a separate stage of the GSC processing pipeline, getting its input data from the off-chip global device memory. This is explained by the intention to load each multiprocessor of a CUDA device with a higher number of islands to be processed in parallel to maintain solid up- and downstreams in the device memory channel. The alternative implementation, in which the overlap-list generation was combined with the preceding *Coding/Linking* stage, indicates a significant drop in the overall performance. This drop is caused by an increasing demand for shared memory resources to store the intermediate results of the *Coding/Linking* stage, which puts a restriction on the number of CUDA threads executed concurrently on each SM.

5.1.2 Global memory data-structure layout

Applications executed on streaming architectures are strongly dependent on the data-flow organisation to and from the processing units. Memory controllers of GPUs are destined to maintain solid data streams to keep the processing resources maximally busy. As the synchronous DRAM, being oriented to burst accesses, is the main storage of CUDA platforms, an application with an

appropriate data layout in the global device memory can benefit from a maximum utilisation of DDR SDRAM channel bandwidth.

Considering CUDA as a SIMD-like architecture, the optimal layout of data in the global memory is if the data elements accessed by load and store operations within a thread block are placed continuously in the memory space, so that the memory controller can pack the data executed by thread warps in bursts (i.e. generate coalesced memory transactions). In practice this means that those arrays of compound data units being processed by concurrent threads, and whose size exceeds a memory word, are better to be sliced into plains of their constituent elements.

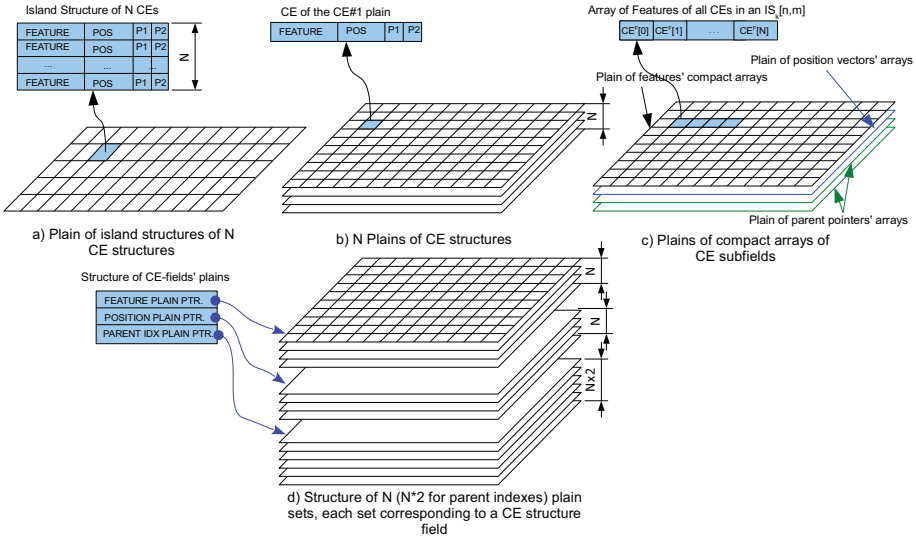


***Figure 37**: Data layout for a hierarchical island level in the GSC database*
N – number of regions in an island; POS – region position bit vector; P1,P2 – parent pointer

Figure 37 illustrates the appliance of this principle to the layout of the region data for one hierarchical level in the GSC database. The data organisation scheme in Figure 37a is the typical layout for the sequential code execution without thread parallelism. Each hierarchical island is represented as an array of *N CE* structures placed sequentially in memory. This layer organisation is referred to as *an array of structures*.

As the *CE* structure can be represented in a compressed form that fits into a 32-bit memory word, the array can be effectively regrouped in *N* plains of *CE* structures, so that all *CE*s with the same indexes in the island structures are placed serially in memory forming a plain of *CE*s (Figure 37b). This data layout is better optimised for GPU memory accesses, but places the additional requirement for packing and unpacking the *CE* structure fields on the GPU using bit-wise operations, as GPU registers are 32-bit words. Moreover this scheme (referenced to as *a stack of compact data-structures*) has two disadvantages for the memory channel utilisation.

During *the linking phase* a region of the underlying hierarchical level is updated with parental information, which does not require the modification of the whole *CE* (see Section 4.1). Meanwhile, the SDRAM memory array is only 32-bit word addressable making a partial byte modification only available with a byte masked write operation, which requires a full memory word transaction. This leads to a nonoptimal utilisation of the memory channel. The same is true for read operations while detecting overlaps during the *Overlap-list Creation.* It requires only the parental information from the lower level and the region feature and region position information from the upper level.

Moreover the parent pointers in a *CE* are updated by two different linking threads, which can reside in two different thread blocks. Thus, if both parent indexes are packed in one byte the parent index update operation will generally be a read-modify-write operation.

These problems with the partial data modification can be eliminated by regrouping island arrays of *CE* structures either in arrays of similar *CE* subfields (features, position vectors, and parent pointers) placed linearly in memory and forming several subfields' plains as shown in Figure 37c[80] or applying layout scheme shown in Figure 37d. In this later scheme a hierarchical level is represented as a set of *N* (or *N*2* for parent indexes) plains for each element of the *CE* structure with the same *CE* indexes within the island structures. The first scheme is referred to as *subfield plains* and the second referred to as *a stack of subfield plains* (or simply *a stack of plains*)

The overlap-list data structure was organised in similarity to *the GSC DB* layout with the number of plains being correspondent to the number of overlapping pair entries. The principle difference with this data organisation is that overlap-list entries can be read and written as single data units without the need for partial field modification.

The data format of the elementary data units that comprises the various data structures of the application are described in Table 7.

***Table** 7: Elementary data unit types*

Unit name	*Intrinsic data type (compact/mem.word size)*	*Data representation*
pixel feature	unsigned char (8-bit)	scalar
region feature	unsigned short (16-bit) / int (32-bit)	scalar
region position	unsigned char (8-bit) / int (32-bit)	bit-vector
parent pointer (index)	unsigned char (8-bit) / int (32-bit)	scalar
overlap-list pair feature	unsigned short (16-bit) / int (32-bit)	scalar
segment label	unsigned int (32-bit)	scalar

All elementary data units of the GSC data structures were implemented both as the most compact data formats (unsigned char / unsigned short) and the format corresponding to a CUDA memory word (unsigned integer) to test the influence on the kernels' execution performance in various optimisation schemes.

[80] Although this scheme has the same disadvantage as layout type *LoS-A* for data organisation in global memory it can be useful in some cases for overcoming bank conflicts in shared memory of GPUs.

5.1.3 General optimisation strategies

During the GPU-GSC design a number of kernel implementations were realised to measure the influence of different optimisation approaches to the application performance. These measurements were done using specialised hardware counters available within the CUDA profiler. In general the application optimisation steps can be grouped into the following categories.

Global memory optimisation (addressing scheme and data compaction)

Although the positive effect of an appropriate global memory layout to the device memory channel traffic is intuitive, it is difficult to estimate its influence to the overall kernels' performance. This is based on the fact that a different data organisation in the global memory may affect the data manipulation procedure inside a kernel (e.g. data extraction or local storage reuse).

Memory architecture peculiarity exploitation (texture and constant memory mapping)

The data that do not need to be modified by a kernel can be mapped to cacheable texture and constant memories residing on-chip close to the processing units. This can release some shared memory resources occupied by input data arrays, thus allowing a higher occupancy of multiprocessors. However, the cacheable memory accesses are generally slower than schedulable coalesced accesses to the global memory, due to the additional layer in the memory subsystem between the SDRAM and the processing units. The compromise between the higher processing load and the data loading efficiency is examined in this optimisation step.

Shared memory layout (addressing scheme and data volume optimisation)

The shared memory layout, including the addressing scheme and elementary data-type selection, has its impact on the overall performance of a kernel arising from the need for resolving possible bank conflicts. Another important characteristic of a kernel is the size of the shared memory required by a thread block, which determines the number of threads that can be executed on a single multiprocessor. This size depends on data organisation used in a kernel for the internal representation of data and the shared memory reuse capability. A careful planning of a kernel algorithm may increase the utilisation efficiency of allocated shared memory resources, yet influencing negatively the maintainability of the code. The influence of the memory layout and memory reuse approaches in the GSC kernels is studied at this optimisation stage.

Kernel control-flow optimisation (execution path and branching minimisation)

When the memory resources allocation is done, the fine tuning of the execution flow of the kernels can be performed. The optimisation progress of the control flow is examined at this point.

Operation-level optimisation (exploitation of intrinsic GPU functions)

Bit operations are intensively used in the GPU-GSC implementation. The CUDA environment offers a set of specialised API functions that are fine tuned for the ISA of the target GPU. This allows some basic operations described with the means of high-level programming languages to be

performed in hardware in an optimal way. The efficiency of the kernel optimisation at the operation level is tested in this phase.

Block size parameters selection

The block dimensions define the portion of computation resources utilised in a single multi-processor and may influence data traffic in the device memory channel. This is expressed by the occupancy, which describes the ratio between the number of warps running and the maximum number of warps that can run concurrently on a multiprocessor. On the other hand the block size is restricted by memory resources reserved by the block. In this tuning step the influence of the block configurations to the different GSC kernels is examined.

5.2 Kernels' implementation

All GSC pipeline stages are implemented in an algorithmically similar way to the FPGA hardware implementation (Section 4.4.1). Only the linking kernel algorithm has been slightly adapted to the peculiarities of the GPU hardware to minimise the divergence in the control flow of the parallel threads.

5.2.1 Region coding

The processing stages of the basic *Region Coding* kernel implementation are schematically represented in Figure 38 below. Each CUDA thread is assigned to process its own region island, having a private copy of a pixel island and a region data-structure in the shared memory. The *Region Coding* kernel realises the triplet coding algorithm suitable for SIMD-like architectures.

In the first step a thread prebuffers a number of pixels comprising a pixel island from the global memory. The addresses of the island pixels in the global memory are defined by the central pixel coordinates, derived from the CUDA thread indexes, and the relative displacements to the central pixel of the surrounding pixels stored as a lookup table in the constant memory.

In the next step the pixel pairs comprising the four basic regions are examined whether they satisfy the linking condition and the correspondent bits are set in the region position vectors provided merging condition is satisfied. The pixel pair indexes for each basic region are stored in a lookup table in the constant memory as well as the correspondent pair position bit-masks. Thus, base region population is realised as a single pass over the index pair array performing a bit-wise operation on each region position flag vector.

The regions are glued in the next step, if a common position is detected by using bit-wise operations on pairs of region position vectors. This is done in two passes over the array of region position vectors. After all base regions are populated the kernel searches for equivalent regions and sets duplicated regions to null.

When all regions are unique, each thread checks the positions set in every region and sums up the features from the correspondent cells of the pixel island array. The averages are computed by a

floating point division of the feature sum by the number of the positions in the regions followed by a normalisation of the result to be compliant to a 12-bit fixed point value representation. The region island structure is sent to the global memory afterwards. The algorithm is attractive for GPU implementation as it is almost free from divergence in the control flow for all parallel threads, except for the section in which an average is calculated.

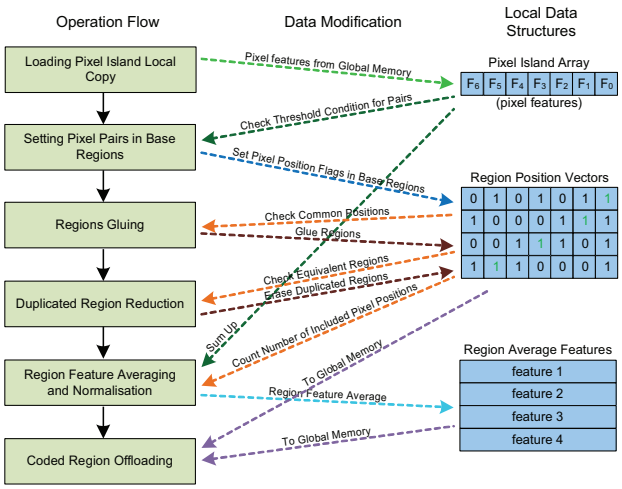


***Figure 38**: Coding kernel processing flow*

The alternative implementations differ from the basic one in the way how the threads manage the pixel island data. As the pixel islands are arranged in an overlapping structure, the same pixels stored in the shared memory can be reused by neighbouring threads. This approach may benefit form a pixel traffic reduction and lower shared memory occupation. The later can lead to a higher occupancy of a multiprocessor. To test the efficiency of this approach a correspondent *Region Coding* kernel is implemented. In the first processing stage this implementation buffers a block of image pixels to the shared memory with the rest of the *Region Coding* pipeline remaining the same. To avoid pixel reloading during pixel buffer filling each thread in a thread block should load a unique set of pixels. In the implemented kernel each thread fetches four pixels in the bottom-right segment of an island creating a local copy of a continuous segment of the pixel image. Although the advantages of this approach look obvious, the implementation has a number of disadvantages. Firstly, pixel address computation in the shared memory becomes more complex, as pixel coordinates need to be computed using the displacement LUT. Moreover a common pixel buffer can generally increase the number of bank conflicts, influencing the performance negatively. Another downside of this implementation is that the pixel islands at the upper left borders of a thread block are not complete. This means that thread blocks must overlap in one column and one row of islands, reducing the effective number of islands processed by a single block.

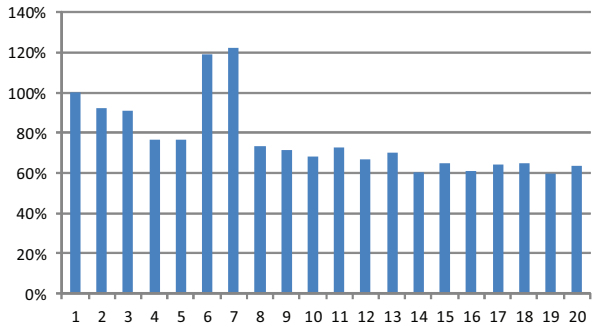


***Figure 39**: Relative performance graph of Coding kernel implementations C1-C20*

A solution free from the latter problem is in mapping the pixel image array to the texture memory without an explicit buffering operation, thus setting the shared memory free from pixel data. The required pixel data can still reside close to computing resources due to the caching mechanism. To increase the caching efficiency the pixel image is mapped to a 2D cache that is specially adopted for data with spatial locality.

The efficiency of different approach described above can be seen in Table 8 and the relative performance of the implementation approaches (identified by numbering) is shown in Figure 39 below.

The basic implementation is realised using different addressing and data compaction schemes in the global and shared memories to measure the influence of the global memory traffic organisation, the warp serialisation effect due to related bank conflicts in shared memory, and impact of the shared memory occupancy on the kernel performance.

The first group of implementations (C1-C4, marked yellow in Table 8) is characterised by the layout of the *CE* data-structure subfields, in which the structure subfields are grouped in arrays of the same subfield types laying in sequential positions in memory (row-wise organisation). At the same time *CE* subfields of different types are grouped in different plains representing a structure of plains at a higher data-abstraction level (*subfield plains* as defined in Section 5.1.2).

The next group of implementations (C5-C9, marked red in Table 8) is characterised by organising *CE* data-fields in arrays of plains of region features and position vectors grouped by *CE* indexes within the island data-structure (*stack of plains* layout) in the global memory. In this case the total arrays of correspondent *CE* data-fields are seen as a column-wise layout of islands' *CE*s keeping the *structure of arrays* organisation in a higher data-abstraction level view (refer to Figure 37d). C5-C8 have *stack of plains* organisation also in shared memory. This column-wise addressing scheme may be beneficial for the shared memory data layout as it allows private data of a CUDA thread to be placed in one memory bank, potentially reducing bank conflicts within a warp.

The implementations in both groups vary in data format for pixels, region features and position vectors both in global and shared memories. C1 to C4 indicate that the global memory traffic

volume does not seriously influence the kernel performance. The more important factor for this group is the shared memory occupancy, which increases using compact data types.

Compared to the implementations of the first group, C5 shows a positive influence of the data layout in the global memory on the global memory traffic by increasing the coalescing access rate as indicated by the global store column of the table (*gst 32b*, *gst 64b*, *gst 128b*)[81]. C5 and C8 compared to C6 and C7 indicate a positive effect of the application of bank conflict resolving schemes (see *Warp Serialised* column of the table). Meanwhile, it can be seen that a column-wise organisation of data is not universally applicable. C6 and C7 show that the utilisation of small data types can destroy the conflict-free organisation, since 8- or 16-bit data of adjacent threads can be found in the same memory words of the shared memory. This leads to a dramatic decrease in performance regardless of an increasing occupancy due to data compaction. In this case the row-wise array layout in the shared memory is more favourable as shown by C9. A compromise between shared memory occupancy and bank conflict-free layout is addressed in the next implementation group.

The third group of implementations (C10-C12, marked green in Table 8) represents different combinations of the data organisation in the shared memory mixing row-wise and column-wise layouts of the different private data structures of a thread, but keeping the layout in the global memory like in the second group. The best performance result is achieved with C12, in which only the region position vectors are organised in column-wise arrays of 32-bit elements, the other arrays having row-wise layout. This is explained by the fact that the region position vectors are accessed most intensively in a constant access pattern, while the region feature access pattern is data-dependent.

The forth group (C13-C15, marked blue in Table 8) represents a principally different approach in the organisation of the input data, in which a block of pixels becomes a common resource for a number of adjacent threads. As it has been noted before this approach suffers from a more complex address computation, which can be noticed by comparing the number of issued instruction (*Instructions* column in the profiling table) in C12 and C13, notwithstanding a significant reduction of the incoming pixel traffic (*gld 32b*).

Mapping the pixel image to the texture memory significantly reduces the bank conflict rate related to the common pixel block accesses in the shared memory as indicated by the profiling data of C14. At the same time the efficiency of the texture cache is relatively high (*tex cache hit* and *tex cache miss* columns of the table) which allows a relatively low latency of pixel data access like in the case of shared memory utilisation.

The profiling data of the kernel implementations described above are recorded for constant thread-block configuration of 8*8 threads per block. The last group of implementations (C16-C20, marked light blue in Table 8) illustrates the influence of thread block resising to the performance of the fastest kernel C14. As it can be seen block resising does not seriously influence the overall performance, resulting to about 8% fluctuation in kernel performance.

[81] Refer to CUDA Compute Visual profiler manual for detailed description of profiling counters [136].

***Table 8**: Region coding kernel implementations' profiling*

	time (us)	*Occu-pancy*	*Smem per Thread (Bytes)*	*Smem per Block (Bytes)*	*Regs per Thread*	*Diverged Branches*	*Instru-tions*	*Warp Serialised*	*gld 32b*	*gld 64b*	*gld 128b*	*gst 32b*	*gst 64b*	*gst 128b*	*tex cache hit*	*tex cache miss*	*gld req.*	*gst req.*
C1 *CE subfield plains in shared and global memory, and memory-word length (32-bit) data types in shared and global memory*	556.064	0.250	96	3892	14	570	97473	103161	7549	0	0	3464	3264	5736	0	0	492	568
C2 *Compact data types in global memory and mem.word in shared (the rest is as in 1)*	514.560	0.250	96	3892	14	570	98315	98994	7549	0	0	9596	2868	0	0	0	492	568
C3 *Pixel data type compact in shared memory (the rest is as in 2)*	505.504	0.375	75	2548	14	570	99532	104427	7549	0	0	9596	2868	0	0	0	492	568
C4 *Compact data types in shared and global memory (the rest is as in 3)*	426.688	0.500	56	1268	12	2121	124605	13614	7549	0	0	9596	2868	0	0	0	492	568
C5 *Stack of plains in shared and global memory, and mem. word data types in shared and global memory*	426.400	0.250	96	3892	22	1009	111986	0	7549	0	0	1622	5118	312	0	0	492	568
C6 *Stack of plains in shared and global memory, and compact data types in shared and global memory*	662.720	0.500	56	1268	24	2121	143886	139355	7549	0	0	6896	156	0	0	0	492	568
C7 *Stack of plains in shared and global memory, and compact data types in shared memory only*	678.656	0.500	56	1268	23	2121	143744	139058	7549	0	0	1622	5118	312	0	0	492	568
C8 *Stack of plains in shared and global memory, and compact data-types in global memory only*	406.912	0.250	96	3892	23	1009	112961	0	7549	0	0	6896	156	0	0	0	492	568
C9 *Stack of plains in global memory, subfield plains in shared memory, and compact data types in shared and in global memory*	397.632	0.500	56	1268	14	2121	124745	13004	7549	0	0	6896	156	0	0	0	492	568

***Table 8**: Region coding kernel implementations' profiling (continued)*

	time (us)	*Occu-pancy*	*Smem per Thread (Bytes)*	*Smem per Block (Bytes)*	*Regs per Thread*	*Diverged Branches*	*Instru-tions*	*Warp Serialised*	*gld 32b*	*gld 64b*	*gld 128b*	*gst 32b*	*gst 64b*	*gst 128b*	*tex cache hit*	*tex cache miss*	*gld req.*	*gst req.*
C10, Linear pixel arrays, strack of plains of region features and positions of mem.word size in shared memory	380.320	0.375	75	2548	19	570	110045	8214	7549	0	0	6896	156	0	0	0	492	568
C11, Linear pixel array, subfield plain of region positions (8-bit), and stack of plains of region features (32-bit) in shared memory	404.384	0.500	63	1780	18	2121	128881	8020	7549	0	0	6896	156	0	0	0	492	568
C12, Linear pixel array, and subfield plain of region features of compact data type (8-bit) and stack of plains of region position of mem.word type (32-bit) in shared memory	370.240	0.500	68	2036	19	1079	110154	12934	7549	0	0	6896	156	0	0	0	492	568
C13, Common pixel array (all other data organisation are as in 12)	390.816	0.500	64	1844	21	936	115134	11878	4944	0	0	7656	752	0	0	0	288	576
C14, Common pixel array in texture (all other conditions are as in 12)	336.832	0.500	60	1588	21	620	103301	5052	0	0	0	8028	1278	0	23940	3145	0	568
C15, Common pixel array in texture and subfield plains of region features and positions in shared memory (all other conditions are as in 12)	359.872	0.500	48	820	16	1276	118635	5106	0	0	0	8028	1278	0	24101	2984	0	568
*C16, Block resized to 16*8*	340.544	0.500	48	3124	21	611	104469	4468	0	0	0	4253	763	0	25854	1188	0	576
*C17, Block resized to 16*16*	356.256	0.500	48	6196	21	667	104686	4208	0	0	0	2143	1485	0	26165	1118	0	576
*C18, Block resized to 32*4*	359.776	0.500	48	3124	21	936	112110	4869	0	0	0	4192	752	0	25584	1118	0	608
*C19, Block resized to 16*4*	331.552	0.500	48	1588	21	704	105014	4477	0	0	0	4265	761	0	25724	1340	0	576
*C20, Block resized to 16*12*	351.968	0.563	48	4660	21	739	101116	4163	0	0	0	4327	717	0	25714	1124	0	552

5.2.2 Overlap-list creation

The CUDA threads of the *Overlap-list Creation* kernels are associated with the overlapping points in the lattice of a hierarchical level. Each kernel detects possible overlapping regions in its lattice node and stores the overlapping region pair information in correspondent entities of an overlap-list structure for subsequently linking the regions at the next hierarchical level. A generalised processing pipeline of an *Overlap-list Creation* kernel is show in Figure 40 below.

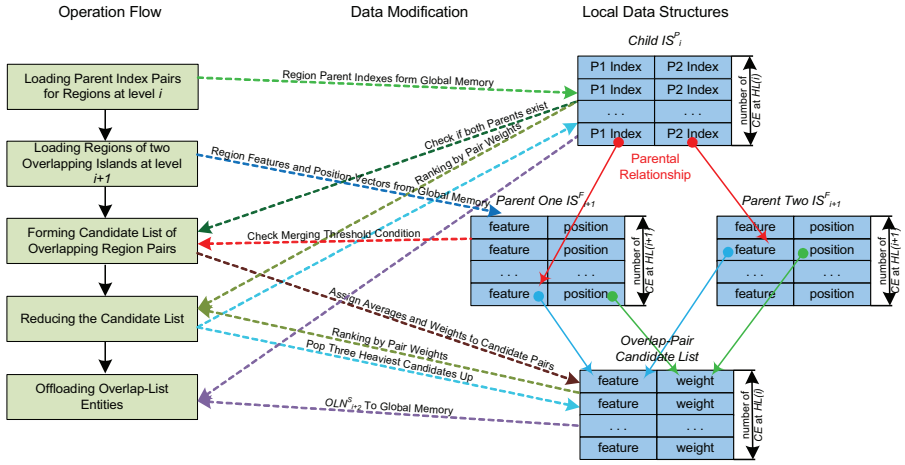


***Figure 40**: Processing flow of Overlap-list Creation kernel*

Each thread operates on its private copies of the data structures. In the initial processing phase a thread loads the parent index pairs of an overlap point and the complete region information of the two parent islands that overlap in this point into the shared memory. Provided both parent indexes are present, the two parent region features are tested for similarity. The features of two similar regions, or their average, are placed into an overlap-pair candidate table. To each candidate a weight equal to the number of positions that the parent regions occupy in their islands is assigned. The weighting is needed for selecting the most solid overlapping regions, as the number of overlap-list entities for each overlap point is limited to three. After the overlap-pair candidate list is complete, the pairs are ranked by their weight, by reordering the elements of both the overlap-pair candidate list and the correspondent index pairs. It is sufficient to only raise the three heaviest candidates, as the others will be neglected. After sorting the list, the three overlap-list entities, comprising pair feature(s) and constituent region indexes are sent to the global memory.

The *Overlap-list Creation* kernel has two particular realisations for the two levels at the bottom of the hierarchical pyramid. The *Overlap-list Creation* kernel for the pixel level lattice detects the prerequisite merging condition by analysing the region position vectors of two overlapping islands. If two regions of these islands cover the same pixel, the regions can be tested for similarity.

Dependent on the overlap-point pixel coordinates the correspondent bit-flag indexes in the position vectors for both islands are stored in a compact LUT placed in constant memory. The detection of common pixels is performed by bit-wise shift and mask operations on all regions in both islands to detect the particular presence of the flag. As a position vector is unique within an island, the kernel does not need any reduction of overlap-list pairs.

The *Overlap-list Creation* kernel of one level higher than the pixel level differs from the general processing scheme in the way the overlap-pair candidates are selected. The maximum possible number of valid regions in an overlapping point island is three, so data space for one region in island data structures is left empty by the *Region Coding* kernel. During the reduction phase the *Overlap-list Creation* kernel detects one empty slot in the candidate list and moves the last of four candidates to the first detected slot, no matter if it is a valid one or not. Thus, the kernel does not need to weight the candidates and therefore the kernel does not require region position vectors to be loaded. After rearranging the candidate list the kernel sends the first three overlap-list elements to the correspondent locations in the overlap lists in the global memory.

During the kernels profiling different implementations were exercised to figure out the influence to the kernels' performance of the different data layouts and compaction schemes both in global and shared memories. The impact of application of specialised GPU functions was also tested. The results are summarised in Table 9. The kernels are grouped by three in the profiling table corresponding to the pixel level, the secondary lowest level and the higher level overlap-list creation. Figure 41 shows the relative performance graph of the different optimisation approaches.

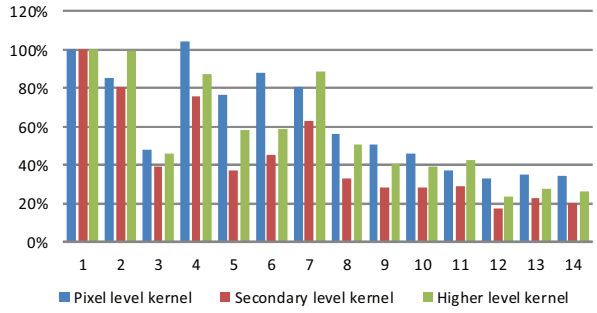


***Figure 41**: Relative performance graph of Overlap-list Creation kernel implementations OLC1-OLC14*

The approach to search for the optimal implementation by varying the data organisation in the global and the shared memories remains the same for the first two groups of kernels' implementations (OLC1 to OLC3 and OLC4 to OLC8) as in the previous section. The overall conclusion is quite similar to the conclusions made for the *Region Coding* kernel:

- the reduction of the overall global memory traffic due to smaller data representation types is favourable for a better kernel performance,

***Table 9**: Overlap-list creation kernel implementations' profiling*

	time (us)	*Occu-pancy*	*Smem per Thread (Bytes)*	*Smem Per Block (Bytes)*	*Regs Per Thread*	*Diverged Branches*	*Instruc-tions*	*Warps Serialise*	*gld 32b*	*gld 64b*	*gld 128b*	*gst 32b*	*gst 64b*	*gst 128b*	*tex cache hit*	*tex cache miss*	*gld req.*	*gst req.*
OLC1) CE subfield plains in shared and global memory & mem word length data-types	888.672	0.188	112	4160	16	2176	116991	157404	19672	19656	16328	5724	4092	9816	0	0	4352	1088
	1290.340	0.063	260	12372	12	795	47581	259572	13856	13184	26512	13084	412	18152	0	0	2840	852
	892.032	0.063	500	6996	18	420	57398	148123	21872	4112	9072	3572	716	3652	0	0	2304	580
OLC2) Compact data types in global memory and mem.word in shared (the rest is as in 1)	758.080	0.188	112	4160	16	2176	122701	136359	47492	8164	0	14724	4908	0	0	0	4352	1088
	1040.860	0.063	260	12372	12	795	50497	223489	46088	7464	0	22572	9076	0	0	0	2840	852
	885.440	0.063	500	6996	18	425	59760	155766	33032	3536	0	5942	2118	0	0	0	2304	584
OLC3) Compact data types in shared and global memory (the rest is as in 2)	424.960	0.500	72	1600	16	2176	124319	14485	47492	8164	0	14724	4908	0	0	0	4352	1088
	500.000	0.188	140	4692	12	795	49373	51640	46088	7464	0	22572	9076	0	0	0	2840	852
	410.368	0.125	260	3156	17	425	58579	27946	33032	3536	0	5942	2118	0	0	0	2304	584
OLC4) Stack of plains in shared and global memory, and mem. word data types in shared and global memory	926.432	0.188	112	4160	20	2176	131956	14515	30656	8568	4324	36614	2666	1316	0	0	4352	1088
	976.448	0.063	260	12372	17	795	67031	2271	19424	7601	3068	27790	2166	1078	0	0	2840	852
	778.080	0.063	500	6996	23	421	75845	2659	17348	2118	1054	14892	0	0	0	0	2304	584
OLC5) Stack of plains in shared and global memory, and compact data types in shared and global memory	681.568	0.500	72	1600	22	2176	140646	98272	41386	2162	0	39938	658	0	0	0	4352	1088
	479.328	0.188	140	4692	20	795	70189	53105	29053	1040	0	30495	539	0	0	0	2840	852
	517.184	0.125	260	3156	22	427	79217	36137	20365	267	0	15012	0	0	0	0	2304	588
OLC6) Stack of plains in shared and global memory, and compact data types in shared memory only	784.416	0.500	72	1600	21	2176	138464	101043	47872	4284	0	36614	2666	1316	0	0	4352	1088
	585.472	0.188	140	4692	20	795	69111	56084	36578	2092	0	27790	2166	1078	0	0	2840	852
	525.280	0.125	260	3156	22	421	78349	35496	22602	545	0	14892	0	0	0	0	2304	584
OLC7) Stack of plains in shared and global memory, and compact data-types in global memory only	717.376	0.188	112	4160	21	2176	137676	12294	41386	2162	0	39938	658	0	0	0	4352	1088
	812.000	0.063	260	12372	19	795	69804	2344	29053	1040	0	30495	539	0	0	0	2840	852
	793.376	0.063	500	6996	23	427	78124	2971	20365	267	0	15012	0	0	0	0	2304	588

***Table 9**: Overlap-list creation kernel implementations' profiling (continued)*

	time (us)	*Occu-pancy*	*Smem per Thread (Bytes)*	*Smem Per Block (Bytes)*	*Regs Per Thread*	*Diverged Branches*	*Instruc-tions*	*Warps Serialise*	*gld 32b*	*gld 64b*	*gld 128b*	*gst 32b*	*gst 64b*	*gst 128b*	*tex cache hit*	*tex cache miss*	*gld req.*	*gst req.*
OLC8) Stack of plains in global memory, subfield plains in shared memory, and compact data types in shared and in global memory	497.088	0.500	72	1600	18	2100	116401	18060	38784	1920	0	37344	480	0	0	0	4096	1024
	423.040	0.188	140	4692	15	778	52515	48481	29754	1059	0	30110	152	0	0	0	2920	876
	453.152	0.125	260	3156	18	404	63510	38164	20309	275	0	14904	0	0	0	0	2368	572
OLC9) Overlap-list entity averaged (data organisation in global and shared memories remains the same as in 8)	448.160	0.500	72	1600	18	2100	116408	17451	38784	1920	0	28128	240	0	0	0	4096	768
	364.864	0.188	132	4180	15	778	52626	36208	29754	1059	0	22622	75	0	0	0	2920	657
	360.864	0.156	236	2772	17	343	61053	25463	20309	275	0	13945	0	0	0	0	2368	528
OLC10) Operation level optimisation (data organisation in global and shared memories remains the same as in 9)	409.568	0.500	72	1600	20	2277	100632	18889	38784	1920	0	28128	240	0	0	0	4096	768
	359.620	0.188	132	4180	15	778	51302	36208	29754	1059	0	22622	75	0	0	0	2920	657
	348.800	0.156	236	2772	17	342	54216	26655	20309	275	0	13945	0	0	0	0	2368	528
OLC11) Overlap-list entity compaction (data organisation in shared memory remains the same as in 9)	330.752	0.500	68	1596	20	2314	104142	13868	42224	2288	0	9152	832	416	0	0	4416	276
	372.000	0.188	128	4176	20	801	68057	47927	29868	1027	0	7331	154	79	0	0	2840	213
	376.992	0.156	232	2768	20	347	62125	22133	20205	263	0	3717	0	0	0	0	2304	150
OLC12) Texture memory mapping with overlap-list entity compaction (the rest is as in 9)	293.440	0.500	44	60	17	2428	97117	6682	0	0	0	9152	832	416	70474	9000	0	276
	221.920	0.500	80	1104	32	947	63170	21221	8432	0	0	7270	153	76	23854	2686	568	213
	211.360	0.250	160	1616	22	345	57604	29681	7280	0	0	3726	0	0	12907	701	592	146
OLC13) Texture memory mapping without overlap-list entity compaction (data layout as in 10)	312.704	0.500	48	64	17	2309	93942	10104	0	0	0	28128	240	0	65089	8277	0	768
	289.600	0.500	84	1108	32	912	65827	24836	8449	0	0	22622	75	0	24417	2363	584	657
	247.232	0.250	164	1620	22	346	60200	30287	7280	0	0	11178	0	0	12902	706	592	441
OLC14) Block resize	303.488	0.750	48	64	17	842	95838	4234	0	0	0	15582	489	0	73575	4761	0	840
	258.112	0.500	84	4180	32	632	65165	18063	5847	0	0	11666	317	0	25357	761	584	657
	231.808	0.125	164	6228	22	194	31163	48841	4070	0	0	6595	59	0	13233	467	304	249

- subfields of compact data types should be organised in row-wise arrays with regard to the reduction of bank conflicts and
- the *strack of plains* layout in the global memory results to coalesced memory accesses and hence in a better traffic organisation.

The later conclusion is not evident from the data shown in the profiling table of this section and is worth a special discussion. The profiling data for OLC3 and OLC8 shows a small increase in the elapse time for the pixel level and the higher level *Overlap-list Creation* kernels for the implementation with the *stack of plains* data layout in the global memory. The information given in the *global load* and *global store* columns of the table (gld 32b, gld 64b and gst 32b, gst 64b) shows a solid upstream traffic quality improvement in favour of *stack of plains* data layout. Meanwhile, the quality of the downstream traffic drops more dramatically. It happens due to the fact that indexes of overlap-pair regions are stored in two separate stacks of plains (separate stacks for right-hand and left-hand parents in a pair). This results in a significant distance between the region indexes in the address space of the global memory leading to wide scatter operations for memory controller, which cannot be well coalesced. Meanwhile, this traffic overhead is much more than compensated by a significant improvement of the upstream traffic to the *Linker* kernel implementation if exploiting the *stack of plains* data-organisation scheme (see Table 10).

OLC9 addresses the problem of reducing the downstream traffic. To do this the overlap-list entity structure has been modified to the variant in which the feature information of a pair is represented by the average of two overlapping regions instead of two region features stored separately. Although this approach results in a significant traffic reduction, it requires extra computation for the average calculations. The matter is that for the average feature the quality of the linking may be influenced, as the mean values of two overlap pairs already include the feature contribution of a common subregion and thus may lead to the chain error effect. To eliminate this effect the overlap-pair average is calculated as a weighted average of the regions, in which the weights are determined as the number of bits set in their region position vectors. In this case the average is shifted to the feature of the most solid region. As indicated in the profiling table the implementation of the approach with downstream traffic reduction results in a significant performance increase of the kernel.

OLC10 demonstrates the influence of operation-level optimisations. In the pixel-level kernel the optimisation is related to the procedure for the common pixel detection in regions of two overlapping islands. The four 8-bit region position vectors of each island are packed into one single 32-bit register and the whole register is shifted once for a fixed number of bits taken from the lookup table, instead of shifting each region position vector in the shared memory. Afterwards the whole register is masked with a 32-bit vector, in which every first bit in a byte is set to one. Alternatively the register can be masked directly with a bit vector from a lookup table in the constant memory that contains the above bit masks shifted for a certain number of bit positions in advance. The later modification gives a slight performance improvement. After masking, only one

position in the vector can remain to be set. Using the *find first set* function from the CUDA API this position can be identified. The integer division of the returned number by the number of bits in a region position vector results in the required index for the region. Another CUDA API function counting the number of set bits in a register is used to determine the weight of a region and is applied to all three OLC kernel types.

The idea realised in OLC11 consists in the further traffic optimisation for the overlap-list entities and the minimisation of the scattered accesses to the overlap-list array in the global memory. For this the overlap-pair entity is stored as a compact structure, in which the pair average and the two region indexes are stored in a single memory word. Although this approach increases the complexity of the kernels due to additional operations for structure compaction and the overall downstream traffic for one byte per overlap-list entity, the approach is beneficial for the Linking kernels (see Table 10 in the next section).

The performance balance between the implementation with and without overlap-list entity compaction is even shifted to the prior solution with the exploitation of texture memory in OCL12 and OCL13. In these implementations the read-only parent region data were mapped to the texture memory. This showed a notable impact to the performance of the secondary lowest and higher level kernels due to a significant improvement of the shared memory utilisation leading to a higher multiprocessor occupancy.

OLC14 shows the performance of the kernels with optimal thread block configurations of 16*12 for the pixel level kernel, 16*16 for the secondary lowest level kernel and 8*8 for the higher level kernel instead of 8*8, 8*8 and 4*4 exercised in previous implementation.

5.2.3 Linking kernel implementation

The direct implementation of the linking approach realised in hardware (Section 4.4.1.2) showed a relatively low performance of the linking state machine due to the high divergence in the parallel threads. For this reason an alternative algorithm was elaborated for realising the linking procedure, which allowed an increase of over three times in performance, but still implemented the same functionality. This alternative approach is based on the bit-vector representation of the overlap-list connectivity information. Each bit vector (referred to as a *connectivity vector*) corresponds to an overlap-list entity and represents the relative positions of two overlapping regions in a macroisland. A connectivity bit-vector consists of seven subvectors correspondent to the seven subislands in a macroisland (referred to as *subisland bit-groups*). The number of bits in the subvectors corresponds to the number of regions in the subislands of one hierarchical level. A bit set in a connectivity vector unambiguously identifies a particular subregion within a macroisland. If two connectivity vectors have two bits set at the same position, it indicates that two overlap-list pairs have a common subregion and the subregions represented in both pairs may be linked together to form a

macroregion. The processing pipeline of the *Linking* kernel realising this approach is depicted in Figure 42.

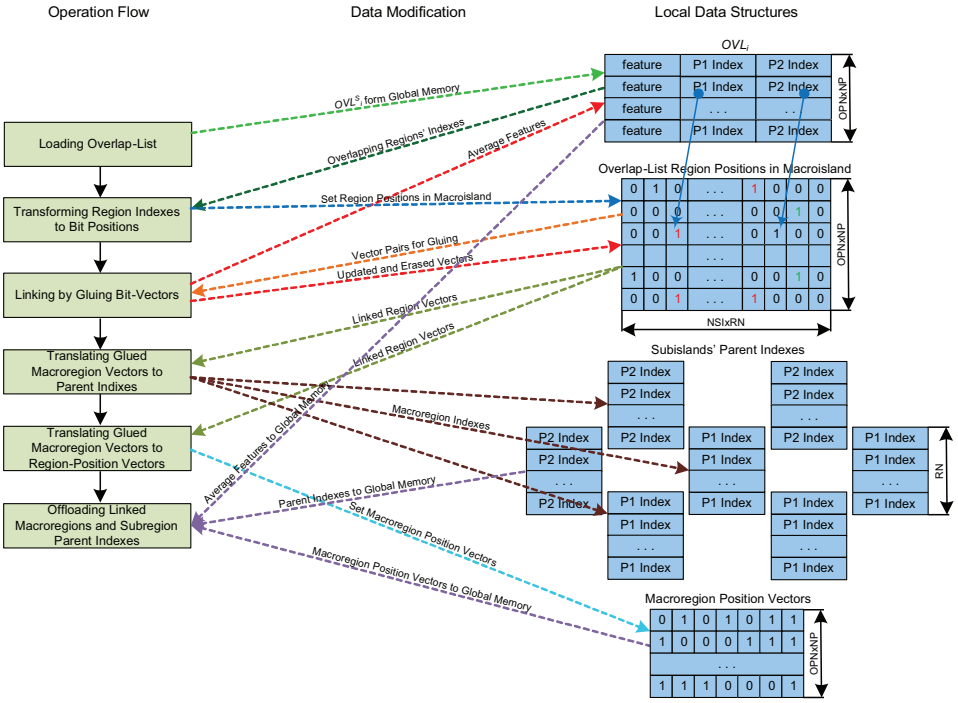


***Figure 42**: Processing flow of Linking kernel*
OPN – number of overlap points in a macro-island; NP – number of entities per overlap points in overlap-list structure; NSI – number of sub-islands in an island; NR – number of regions in a sub-island at a given hierarchical level; Pn indx – parent index of sub-regions

The CUDA threads of the *Linking* kernel are assigned to macroislands of a hierarchical level *i*. Each thread works on its own copy of data structures placed in the shared memory. At the initial stage of the processing pipeline the overlap list correspondent to the thread's macroisland is loaded from the global memory. At the next stage every index pair in the overlap list is converted to the bit-vector representation using bit-wise operations. Afterwards the connectivity vectors are glued realising the linking operation on subregions.

The gluing operation is performed by two nested loops over the connectivity vector table. In the outer loop a search for a new macroregion root is done. When a new root is detected the inner loops start a search for the next link candidate. If a common subregion is detected in both connectivity vectors they are tested for similarity. In the case of similarity one of the connectivity vectors is updated with the new bit positions and the average feature is written to a correspondent entity of the overlap-list table. The second connectivity vector is nullified and the search for the next linking candidate proceeds until the bottom of the table is reached. The decision which vector is to be updated depends on the processing of the connectivity vector table. The experiments show that the

descending vector propagation, in which the chain root is reassigned to the newly linked candidate, reduces the overall execution path and leads to a higher performance of the kernel.

After all possible connectivity vectors are glued the resulting vectors represent complete chains of subregions each corresponding to newly linked macroregions within a macroisland and respective feature fields in the overlap-list table.

In the next step the parental relationship between macroregions and the contributing subregions is established. A thread searches for a nonempty connectivity vector in the connectivity vector table and analyses the bits set in it. The position of a set bit gives the index of the subisland and the index of the contributing subregion within the subisland. Using these indexes, the order number of the nonempty connectivity vectors is written to a special data structure representing the parent index fields of the sub*CE*s.

Afterwards, the connectivity vector table is scanned again to generate the region position vectors of the newly linked macroregions. This is done by analysing the complete subisland bit-groups of nonempty connectivity vectors.

In the last stage the results are offloaded to the GSC database. Sending the macroregions to the global memory in the same order as they were processed during the parent index assignment guaranties the consistency of the parental information in the *CE*s of a hierarchical level *i-1*.

The profiling results of the *Linking* kernel implementations with different data organisation and the impact of the application of specialised GPU API functions are shown in Table 10. The kernels are grouped by two in the profiling table corresponding to the *Initial Linking* and the *General Linking*. The relative performance of the different optimisation approaches can be seen in Figure 43.

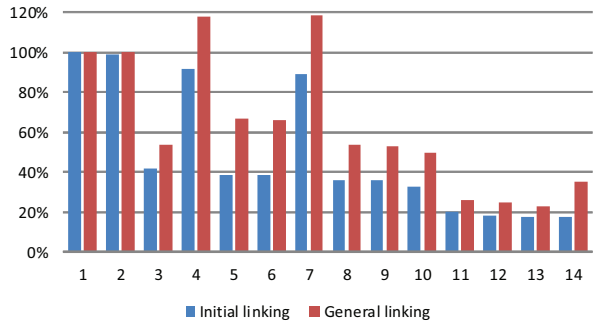


***Figure 43**: Relative performance graph of Linking kernel implementations L1-L14*

The experiments with different data layout in the shared and the global memories in L1-L3 and L4-L8 lead to the same conclusions as for the *Region Coding* and *Overlap-list Creation* kernels described in the two sections above.

***Table 10**: Linking kernel implementations' profiling**

	time (us)	*Occu-pancy*	*Smem per Thread (Bytes)*	*Smem Per Block (Bytes)*	*Regs Per Thrd.*	*Diverged Branches*	*Instrc-tions*	*Warp Serialised*	*gld 32b*	*gld 64b*	*gld 128b*	*gst 32b*	*gst 64b*	*gst 128b*	*gld req.*	*gst req.*
L1) CE subfield plains in shared and global memory & mem word length data-types	3864.260	0.063	420	5716	17	1374	323612	656410	50304	0	14976	15758	3128	26074	1776	2132
	13953.300	0.063	1556	6036	16	1144	1345627	1557830	61056	0	0	31792	766	1786	5328	3484
L2) Compact data types in global memory and mem.word in shared (the rest is as in 1)	3829.980	0.063	420	5716	17	1374	324466	647678	57792	7488	0	42629	2331	0	1776	2132
	13942.400	0.063	1556	6036	16	1144	1345227	1534858	61056	0	0	33451	893	0	5328	3484
L3) Compact data types in shared and global memory (the rest is as in 2)	1621.440	0.188	204	2260	16	1374	330509	186269	57792	7488	0	42629	2331	0	1776	2132
	7484.160	0.125	908	3444	16	1144	1344117	862389	61056	0	0	33451	893	0	5328	3484
L4) Stack of plains in shared and global memory, and mem. word data types in shared and global memory	3551.780	0.063	420	5716	26	1374	376587	6192	16794	3738	1848	23171	2739	4573	1776	2132
	16422.900	0.063	1556	6036	32	1144	1932715	0	29074	1886	938	19506	1787	907	5328	3484
L5) Stack of plains in shared and global memory, and compact data types in shared and global memory	1477.540	0.188	204	2260	29	1374	386759	57043	21456	924	0	30247	236	0	1776	2132
	9269.220	0.125	908	3444	32	1144	1943428	130993	31429	469	0	22168	32	0	5328	3484
L6) Stack of plains in shared and global memory, and compact data types in shared memory only	1490.080	0.188	204	2260	27	1374	387111	61346	22380	1869	924	23171	2739	4573	1776	2132
	9194.530	0.125	908	3444	32	1144	1942630	129565	31898	943	469	19506	1787	907	5328	3484
L7) Stack of plains in shared and global memory, and compact data-types in global memory only	3441.950	0.063	420	5716	28	1374	377125	5623	21456	924	0	30247	236	0	1776	2132
	16585.700	0.063	1556	6036	32	1144	1936314	0	31429	469	0	22168	32	0	5328	3484
L8) Stack of plains in global memory, subfield plains in shared memory, and compact data types in shared and in global memory	1374.080	0.188	204	2260	21	1374	359497	13876	21456	924	0	30247	236	0	1776	2132
a) with column-wise organisation of connectivity vector table	8372.160	0.125	908	3444	30	1144	1824026	0	31429	469	0	22168	32	0	5328	3484
b) with row-wise connectivity vector table layout	7465.630	0.125	908	3444	27	1144	1354812	859055	31429	469	0	22168	32	0	5328	3484

**the fist rows of implementation groups correspond to the initial linking kernel, the second rows correspond to the general linking kernel*

***Table 10**: Linking kernel implementations' profiling(continued)*

	time (us)	*Occu-pancy*	*Smem per Thread (Bytes)*	*Smem Per Block (Bytes)*	*Regs Per Thrd.*	*Diverged Branches*	*Instruc-tions*	*Warp Serialised*	*gld 32b*	*gld 64b*	*gld 128b*	*gst 32b*	*gst 64b*	*gst 128b*	*gld req.*	*gst req.*
I9 *Overlap-list entity averaged (data organisation in global and shared memories remains the same as in 8)*	1371.140	0.188	204	2260	20	1374	356680	14081	16326	540	0	30247	236	0	1332	2132
	7408.480	0.125	908	3444	27	1144	1344918	857387	23692	233	0	22168	32	0	3996	3484
I10 *Gluing Loop optimisation (data organisation in global and shared memories remains the same as in 9)*	1251.940	0.188	204	2260	22	1838	346553	16627	16326	540	0	34392	163	0	1332	2336
	6891.710	0.125	836	3444	27	1052	1291679	799367	23692	233	0	22120	30	0	3996	3452
I11 *Operation level optimisation (data organisation in global and shared memories remains the same as in 9)*	774.848	0.188	204	2260	21	1909	198473	17667	16326	540	0	30411	224	0	1332	2102
	3587.870	0.125	836	3444	27	1609	657334	508960	23692	233	0	18964	5	0	3996	3088
I12 *Overal-list entity compaction (data organisation in shared memory remains the same as in 9)*	707.904	0.188	188	2064	22	2186	197974	13400	4005	1089	561	30817	237	0	432	2120
	3413.540	0.125	868	3296	22	1223	608124	526773	7269	471	235	18928	1	0	1332	3033
I13 *Shared memory reuse with no overlap-list entity compaction (data organisation in global and shared memories remains the same as in 9)*	664.736	0.250	168	1684	22	2427	212996	10140	16293	390	0	27793	264	0	1296	2112
	3133.150	0.156	800	3012	26	1553	644465	474797	23578	232	0	18967	1	0	3888	3008
I14 *Block resize*	673.632	0.125	168	3284	22	1692	122461	11420	8987	571	0	18977	386	0	684	1158
	4931.650	0.063	800	5940	26	1417	381953	1122317	12561	351	0	11902	9	0	2052	1661

Interesting results are observed in the profiling data of L8 for the *General Linking* kernel. The implementation is configured with two different options for the layout of the connectivity vector table in the shared memory. Connectivity vectors of the *General Linking* kernel are coded in four full memory words. Therefore, the column-wise organisation of the connectivity vector table may benefit from bank conflict-free accesses to its entities as indicated by the *Warp Serialised* column of the profiling table. Meanwhile, the computation of the addresses for the column-wise organised array become more complex than the address calculation row-wise layout, which is indicated by a significant difference of instructions (*Instruction* column of the profiling table). It should be pointed out that the overall performance shifts the solution towards a tangible bank conflict rate.

L9 shows the influence of a change in the overlap-list entity structure, in which overlap lists contain averages of an overlapping region pair. The small increase in the kernels' performances indicates the low impact of the upstream traffic to the overall kernels' elapse time, emphasising the computation intensity of the linking.

L10 targets an optimisation of the kernel control flows. The connectivity vectors' gluing loop is modified to reduce the number of passes over the connectivity vector table in the way that roots of newly linked region chain are moved down towards the bottom of the table instead of keeping the initial positions of the root regions as it was realised in the implementations before. Additionally the procedure for updating region position vectors of newly linked macroregions was extracted from the gluing loop to a separate pass over the connectivity vector table after the table is glued. This resulted in shorter branch traces beneficial for the execution flow of a multiprocessor in case of execution flow divergence in thread warps.

L11 applies CUDA API functions to the procedure of parent index assignment to subregions within the macroislands. In the former procedure positions set in a connectivity vector were detected by shifting the vector bit-by-bit over the whole vector width. The application of the *set-bit counting* function (*__popc*) and the *find first set* function (*__ffs*) allows the reduction of the amount of iterations over the connectivity vector to the number of current positions set, thus leading to significant performance improvement.

L12 shows the positive impact of the overlap-list entity compaction on the overall performance of the kernels, which shows the best result among the implementations described above. Meanwhile, the optimisation approach realised in the next implementation excels it in this parameter due to the higher occupancy of multiprocessors caused by reducing the shared memory used per thread. The main idea of this approach is that the indexes of subregions stored in overlap-list entities may only be used once for the initial filling of the connectivity vector table and that the subisland and subregion indexes for the establishing of the parental relationship between newly linked macroregions and their contributing subregions are derived from the connectivity vectors after gluing. Therefore, the shared memory allocated for subregion indexes in the overlap-list table may be reused after the initial connectivity vector table is built.

In the implementation realising the shared memory reuse approach, the *Linking* kernels use the same overlap-pair feature array for storing both subregion indexes and the average features in the different kernel execution phases. The average feature loading is postponed until the connectivity vector table is built using the subregion indexes loaded before. Afterwards the indexes are overwritten by the overlap-list entity features. This means that the approach with the shared memory reuse is not compatible with the overlap-list entity compaction approach because in this case the same data will be loaded twice by the global memory controller. That is why the kernel applies the *stack of plains* layout for data organisation in the global memory.

Unfortunately the texture memory utilisation, which showed a positive impact on the performance of the *Region Coding* and *Overlap-list Creation* kernels, is not applicable here. The reason is that the data manipulated by the linking procedure is constantly modified in the lifetime of the kernel, which would make local copies of the data structures in the read-only texture cache incoherent.

The high amount of the shared memory used by a thread does not give much space for block size optimisation. The block dimension configurations – 4*4 for the *Initial Linking* kernel and 2*2 for the *General Linking* kernel – used for the profiling implementations described above appeared to be the optimal. L14 illustrates an attempt to increase the block sizes to 8*4 and 4*2 configurations, respectively. As it can be seen a change in the blocks' dimensions leads to an overall performance drop.

5.2.4 Downpropagation and result generation

The CUDA threads of the *Downpropagation* kernels for the result generation are bound to the islands of a level *i*. The regions in these islands inherit the segment information (feature and/or label) from the parents one level above (*i+1*). The threads of the initial kernel implementation work with private copies of their islands and the two parent islands all placed in the shared memory. The generalised processing pipeline of the kernel is shown in Figure 44.

At the first stage a thread loads all *CE*s of an island (feature, position and parental information) and their two parents (segment features and labels). At the next stage going over all regions the segment information from the upper level is downpropagated if parents exist or, otherwise, a new segment root is formed. A new segment key is generated as a combination of the hierarchical level number, the CUDA thread index, and the index of a region in an island. This mechanism guaranties the uniqueness of a segment label. The validity of a region is checked by the region position vector[82]. In the final step the segment features and labels assigned to regions are offloaded to the global memory for the next downpropagation iteration.

At the bottom of the GSC pyramid, the regions are not updated in the global memory. Instead of that, the segment labels and features are directly assigned to the pixels of the label and feature images. This image generation is done using the information stored in the region position vectors.

[82] Region feature can be used as alternative for region validity detection, as a feature is stored in an intrinsic data type having more bits than it is needed for feature presentation.

The downpropagation to the bottom level is done by a specialised downpropagation kernel called *Image Generation* kernel.

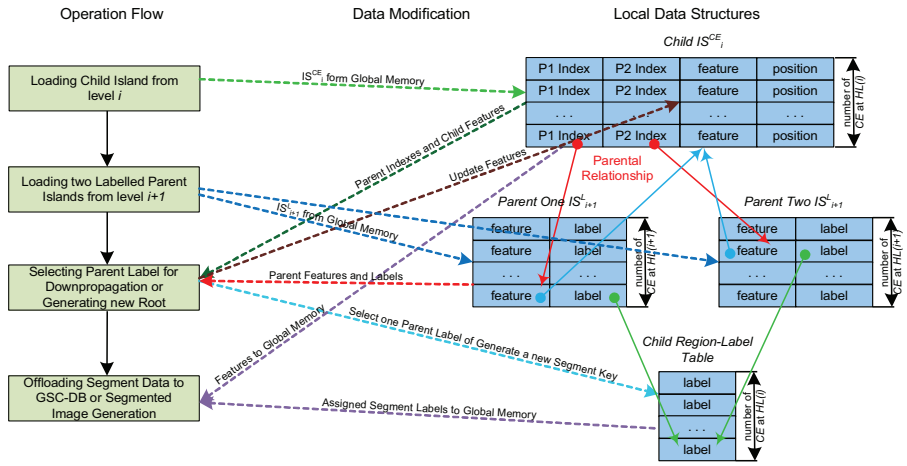


***Figure 44**: Processing flow of Downpropagation kernel*

As an alternative to the initial implementation different versions of the kernel exploiting the texture memory instead of shared memory and using an explicit buffering of read-only data are realised and analysed for efficiency. The joint results of the kernel implementation profiling can be found in Table 11. The kernels are grouped by two in the profiling table corresponding to the *Downpropagation* and the *Image Generation* kernel. The relative performance of different optimisation implementation approaches is shown in Figure 45.

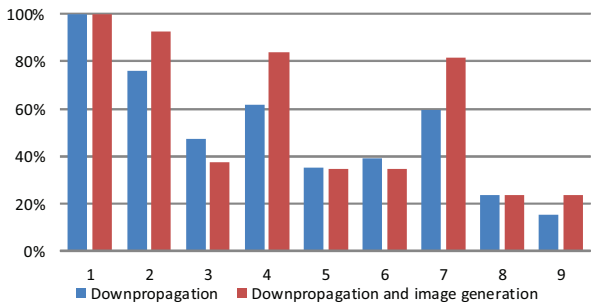


***Figure 45**: Relative performance graph of Downpropagation kernel implementations DP1-DP9*

The results for the different data layouts in the global and the shared memory indicated that the use of compact data types is preferable for the *Downpropagation* and the *Image Generation* kernels. The data organisation scheme in the global memory does not affect the performance of the kernels seriously as can be seen from comparison of profiling data for DP3, DP5 and DP6. It should be

***Table 11**: Downpropagation kernel implementations' profiling*

	time (us)	*Occu-pancy*	*Smem per Thread (Bytes)*	*Smem Per Block (Bytes)*	*Regs Per Thrd.*	*Diverged Branches*	*Instrc-tions*	*Warp Seria-lised*	*gld 32b*	*gld 64b*	*gld 128b*	*gst 32b*	*gst 64b*	*gst 128b*	*tex cache hit*	*tex cache miss*	*gld req.*	*gst req.*
DP1) CE subfield plains in shared and global memory & mem word length data-types	551.008	0.094	340	4676	14	321	27312	123867	14668	5348	18140	1760	1632	2472	0	0	2016	288
	1131.620	0.063	216	10060	12	626	71374	135075	9760	14008	24840	9348	1648	2896	0	0	2336	798
DP2) Compact data types in global memory and mem.word in shared (the rest is as in 1)	420.352	0.094	340	4676	14	288	28420	90702	32472	5844	0	3392	2496	0	0	0	2016	288
	1048.480	0.063	216	10060	12	621	72059	114985	38248	10360	0	10994	2896	0	0	0	2272	770
DP3)Compact data types in shared and global memory (the rest is as in 2)	262.176	0.250	160	1796	14	288	28369	23405	32472	5844	0	3392	2496	0	0	0	2016	288
	424.704	0.250	120	3852	12	623	76282	26838	38248	10360	0	10994	2896	0	0	0	2272	770
DP4) Stack of plains in shared and global memory, and mem. word data types in shared and global memory	339.552	0.094	340	4676	23	321	32690	0	16979	3083	1564	2789	615	315	0	0	2016	288
	946.144	0.063	216	10060	25	626	83499	0	18830	4953	5513	10677	309	1303	0	0	2336	798
DP5) Stack of plains in shared and global memory, and compact data types in shared and global memory	193.952	0.250	160	1796	25	288	33795	16289	21016	620	0	3407	312	0	0	0	2016	288
	393.088	0.250	120	3852	27	927	87555	37319	26828	2417	0	10978	1306	0	0	0	2272	770
DP6) Stack of plains in shared and global memory, and compact data types in shared memory only	216.960	0.250	160	1796	24	321	34206	18598	23803	1235	0	2789	615	315	0	0	2016	288
	393.536	0.250	120	3852	26	914	88953	42505	34800	2481	0	10677	309	1303	0	0	2336	798
DP7) Stack of plains in shared and global memory, and compact data-types in global memory only	329.504	0.094	340	4676	24	288	33842	0	21016	620	0	3407	312	0	0	0	2016	288
	923.008	0.063	216	10060	26	617	83973	0	26828	2417	0	10978	1306	0	0	0	2272	770
DP8) Texture memory mapping (data layout as in 5)	129.344	0.250	83	836	30	276	23145	8410	10881	303	0	3425	303	0	3113	2114	888	296
	267.648	0.500	100	1548	32	906	72050	22010	11226	150	0	10932	150	0	16501	3943	852	770
DP9) Block resize	84.896	0.250	83	1604	30	157	11792	5729	5872	382	0	1705	382	0	4064	1182	432	144
	265.952	0.500	100	1548	32	945	72854	23549	11226	150	0	10932	150	0	16523	3921	864	782

determined by the requirements of *the* more time consuming *linking phase*, which benefit more from the *stack of plains* layout in the global memory.

The data organisation scheme in the shared memory does not influence the performance much as well due to the fact that the shared memory is not intensively used be the kernels. Most memory operations are read-only accesses to the parent indexes of regions and the parent regions' data. The accesses of the second kind are sporadic (parental index dependent) and thus having no regular access pattern over the kernels' execution flow, which makes the bank conflict prediction not realisable.

Taking into account the peculiarities of data manipulated in the kernels, most of the data structures are moved to the texture memory and only modifiable data structures are kept in the shared memory (segment features and labels of regions, local copies of a pixel island in the *Image Generation* kernel). The results of this modification can be seen in the kernel profiling table for DP8. The influence of the thread block resizing is shown with the profiling data for DP9. It has the optimal block sizes of 8*4 for the *Downpropagation* kernel and 8*8 for the *Image Generation* kernel instead of 4*4 and 16*4 in the initial configuration.

6 Results comparison

6.1 Application performance and dataset scalability

The main exploitation indicators for HPC that are paid attention to firstly are computation performance, hardware and setup costs, and power consumption. The cost and power parameters of both solutions lay in the same range of a few thousands Euro and some hundred Watts per board and are not specially addressed in this work as well as the systems' operating characteristics as fault tolerance, computing reliability, etc. The present work addresses only the question of application performance.

Figure 46 illustrates the performance comparison of two GSC solutions – the maximally optimised GPU implementation on a top class nVidia Tesla C1060 at 1.3GHz of graphical core clocking and the FPGA implementation in the 3x3x3 configuration[83] maximal supported by Xilinx XC2VP100 at 100MHz. Figure 46a gives the absolute processing time for a number of image resolutions. It is significant that in the rage of small resolutions the GPU elapse time stays approximately the same, although the amount of raw data increases four times in each resolution step. Additionally the GPU solution yields to the FPGA implementation notably up to 512^2 image sizes (Figure 46b). For higher resolutions the performances become almost even[84].

Figure 46c shows how the processing time scales with the change of the image resolution for each solution. The processing time of the FPGA solution scales almost proportionally to change of data volume, although there is a small tendency for the ratio to grow with an increase in image size. This is explained by the relative increase of the auxiliary data traffic related to overlapping rows of islands in the hierarchical level processing organisation (refer to Section 4.2.3) for fixed processor arrays' sizes in the GSC processing units. The processing time of the GPU solution does not scale proportionally to the data volume for smaller resolutions. Together with the almost equal processing time this indicates that for the GPU solution some fixed constituent related to the computation

[83] 3x3x3 indicates the number of island rows in the island row blocks processed by initial linking processing, general linking and down-propagation processing units, respectively.

[84] For the higher resolutions the performance behaviour does not follow the trend, because both implementations suffer from problems with data placement in memory caused by limitations of data size allocation in specialised memories of the current GPU architecture and sizes of memory banks in case of FPGA solution. Therefore, the both solutions require different implementation modifications.

organisation on the GPU exists, which becomes significant for smaller data volumes processing. No optimal computing payload distribution for small data sets is the other reason for this behaviour.

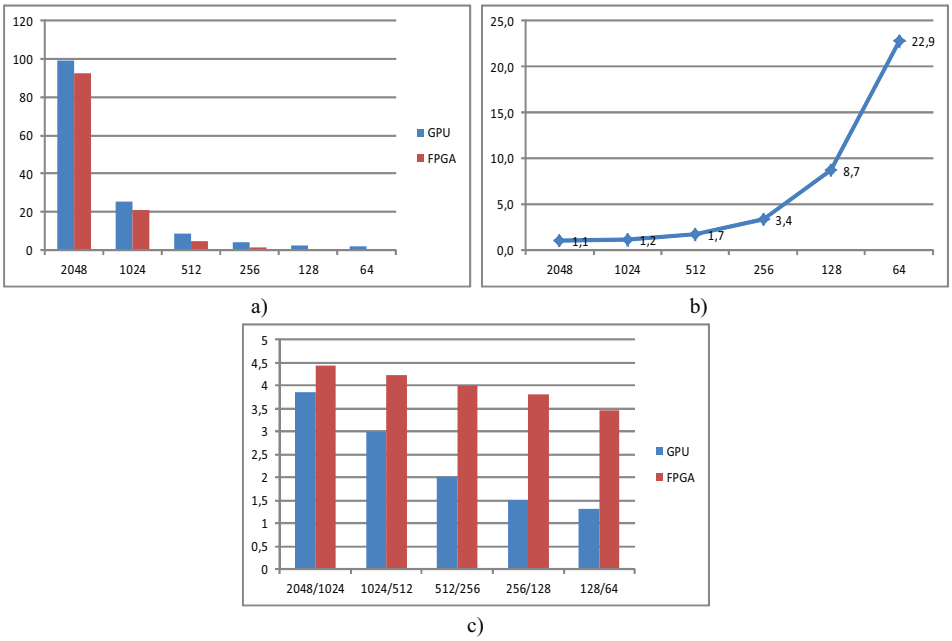


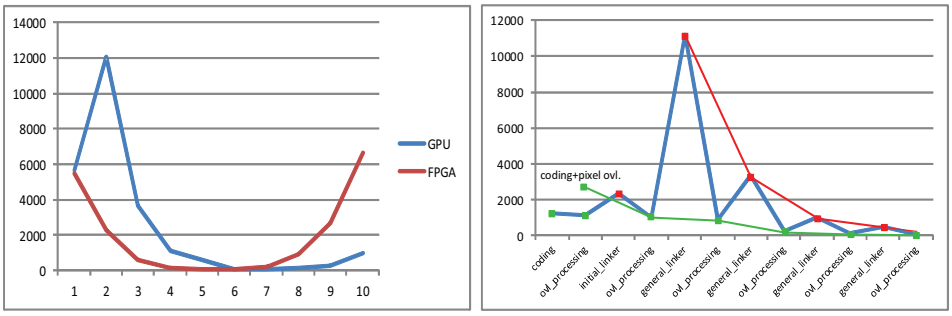
***Figure 46**: Processing times and dataset size scaling ratios*
a) Processing time for different resolutions (ms); b) GPU to FPGA processing time ratios
c) Processing time scaling ratios (next to previous resolutions)

In general the systems built around shared memory resources do not have such elastic scalability over the number of processing elements as distributed systems have. The ultimate limitation of the performance for an application on such systems is the bandwidth of their data channels to shared storage. On both platforms the GSC application is implemented in a way that the shared external storages accommodate the main volume of intermediate computation data, although serious efforts have been applied to minimise the external data traffic and to exploit the local memory resources. Considering the data models of the GSC implementations as they are described in Chapter 4 and Chapter 5, it can be seen that the FPGA implementation has a solid potential for performance improvement by increasing the number of specialised processors (currently limited by resource budget of the die) in the GSC processing units until the memory bandwidth is saturated. Building these characteristics for GPU is not feasible. However, it can be assumed that the top model GPU card is designed to exploit the bandwidth of its DRAM maximally, thus further system scaling is not hypothetically effective.

The peculiarities of the computing architectures are well illustrated by the processing time distribution among the different hierarchical levels. Typical level processing time profiles for the

FPGA and GPU implementations are shown at Figure 47a. The graph shows the processing time for the first five hierarchical levels of linking (1-5) and the downpropagation (6-10). In order to ease the comparison, the stages of the GPU-GSC pipeline are grouped together to match the FPGA-GSC pipeline stages, e.g. the processing time for the coding, the initial overlap-list creation and the initial linking GPU kernels are summed up together to be comparable with the elapse time of the initial linking stage of the FPGA solution. It can be seen that the processing time curve for FPGA solution precisely follows the curve of data traffic to external memory, which highlight the data-flow nature of the FPGA solution. The GPU elapse time profile shows that only the downpropagation tail (6-10) of the curve achieves a proportionality of data volumes. The linking curve of the GPU solution seems to be much more disproportional. This curve shape accentuates the difference in the way the streaming architecture of the GPU handles different type of algorithms.

The downpropagation kernels are characterised by short, predominantly linear control flow paths. The kernels are light and are not burdened with large and complex data structures placed in the local memory of processing cores. The linking is a difficult task for an SPMD architecture, which can be seen from the significantly steeper incline of the left half of the GPU curve at Figure 47a. The control flow of the linking kernels is intricate; the kernels have to handle a large amount of data in shared memory of the Streaming Multiprocessors.

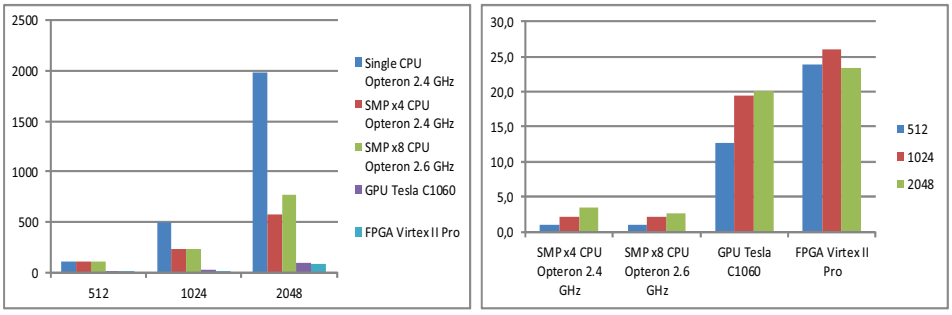


a) b)

***Figure 47**: Typical level processing time profiles for FPGA and GPU implementations (sampled for 1024^2 image size, y-axis is time in micro sec.) a) FPGA and GPU combined profile; b) GSC GPU stage detailed profile*

Figure 47b shows the impact of the linking kernels to the processing time of *the linking phase* in more detail. While the coding and overlap-list creation kernels obey the general tendency of the left half of the complete GPU curve, the linking kernel processing times stand out against the general trend as local seriously distinguished maximums, although the data flows into and out of the adjacent processing stages are equal. Notable is the big difference between the execution times of the initial linker kernel and the first general linker kernel. This is explained by the significant higher complexity of the linking algorithm at higher levels. Considering only the data traffic associated trends it can be assumed that the GPU could seriously outperform the FPGA solution, if it could

overcome the limitations of the current SPMD architecture. In the simplest case this can be achieved by just an increase of local memory resources (refer to Section 5.2.3).



a) b)

***Figure 48**: GSC application performance in comparison to CPU solutions*
a) Elapse time in microseconds; b) Performance acceleration ratios

At Figure 48 both solutions are compared to the CPU implementations of the GSC. It can be seen that both the FPGA and the GPU solutions show a drastic acceleration of the application compared to the single CPU approach and a significant speed-up compared to the OpenMP parallelisation approach. It is notable that the SMP solution is not advantageous from the point of view of system scaling. So, for example, doubling of the number of processors does not have any noticeable effect. This is related to the fact that the granularity of the parallelism is very high for the GSC implementation. Several parts of an image are processed in parallel by different CPUs, but should not be made smaller due to increasing overheads for synchronisation.

6.2 Comparison of application development factors

The other group of metrics being significantly important for HPC are characteristics related to the application development process. These characteristics are difficult to compare quantitatively, thus they are treated in a form of an expert assessment. Among the major factors for comparison are the implementation efforts, modifiability, maintainability and portability.

6.2.1 Implementation efforts

Estimations of implementation efforts can normally be supported with numbers. Good candidates to be a measure are the market prices for similar solutions, manpower required for the implementation of each solution, and number of code lines written for each implementation. Each of these metrics has its benefits and limitations in use.

The most objective measure is the price comparison as the market levels all nuances of the development process such as qualification of workers, methodological or instrumental difference in the design approaches, cost of equipment and intellectual property purchase, etc. A big advantage of

this method is that, provided the exploitation characteristics of two solutions are equal, it can be used to compare solutions based on principally different technologies such as HW design and programming. However, this indirect economic assessment approach has a serious limitation – the comparable solution should be available at the market or there need to be a methodology in order to make the available solutions comparable.

The comparison of the solutions based on the needed manpower can be attractive as the approach directly operates with the amount of time required for each solution development. The approach allows the analysis of the time structure of a work package in details, i.e. the time for designing, coding, debugging, testing, etc. However, the time-based method has a notable expense; it requires a well-developed project-management system and extra administrative efforts. For the comparison of solutions this approach poses a serious restriction to the implementations. It is necessary that the solutions are implemented by executors with comparable skills in each technology. In case different solutions are implemented by the same person, it is necessary to take into account an education curve and training time if the developer is not of the same experience level in different technological fields.

The effectiveness of the development efforts' estimation using the third approach is the least evident. The approach has a number of restrictions, which make the method limited in applications:

- the implementation effort indicators can only be derived indirectly with some propositions in mind; in particular, the development efforts for making a code functional, e.g. debugging and testing, which normally take a serious bulk of the development time, are completely invisible;
- the code style directly influences the number of lines in a code;
- the syntax specifics of different implementation languages allow a different effectiveness of the coding.

However, in the present work this method can be effectively applied for the comparison of the FPGA and GPU solutions. Taking in account that SystemVerilog has a C-like syntax and the implementations are realised by the same person the two last limitations mentioned above can be neglected as the code structure of both solutions can be considered being equivalent. The advantage of this approach is that it eliminates the education curve by comparing only the mature resulted code of both solutions. The efforts for the code debugging are supposed to be proportional to the code amount and do not need to be illustrated in absolute values for the comparison analysis.

Figure 49 compares the code amount of the two implementations in their final versions and illustrates the structure of the code for the FPGA and GPU solutions. Alternative realisations of the FPGA modules and GPU kernels are not counted, so as if the projects have been implemented in accordance to precise specifications of computing systems without an experimental design search. The code for the preimplementation modelling is not included as well, as it is assumed to be used for both the FPGA and GPU solution.

a) b)

***Figure 49**: Number of code line comparison for GSC Implementations*
a) FPGA and GPU solution comparison; b) FPGA code structure

The left diagram shows the relationship of the FPGA and GPU codes. It compares both the GSC specific and the auxiliary codes for the computation organisation. In case of the FPGA solution the auxiliary code implies the code being related to the infrastructural components of the SoC[85]. The GPU auxiliary code includes the procedures needed for computations' organisation, memory management, kernels' configuration, procedures needed for debugging purposes, etc., while the GSC specific code generally covers the code of the GPU kernels. As it can be seen from the diagram the amount of the overall FPGA code is 8.5 times higher than the total GPU code, while the GSC specific code ratio is about 7.7. These figures emphasise the complexity of a hardware design compared to a GPU software development.

The right diagram offers a more detailed view to the FPGA code. It can be seen that the infrastructure development took a solid bulk of the logic design. The verification code of the infrastructural modules is approximately equal to the RTL code of the modules. Taking into consideration that the verification code is written in a higher level of abstraction and that it is much more compact for describing the same functionality, the figures show how much efforts are normally made for the verification of reusable universal components of a system. Most of efforts in this case are initial investments for the development of the verification environment and reusable verification components. At the other hand the proportion of the verification code for the GSC specific components is much lower due to the fact that the computation modules are built using universal reusable components already constrained with intrinsic verification checkers and that the subject of verification in this case becomes a pure functionality of the computation units without signal or temporal-logic level details.

6.2.2 Functional modifiability

For the comparison of the two solutions also the term functional modifiability is used in this work. The term is notably distinguished from the ease of code modification[86], which is per se associated

[85] FPGA SoC line count does not include the DDR SDRAM controller core, which has been delivered as a netlist IP.
[86] In software engineering this characteristic of a code is defined as Perfective Maintenance as stated in ISO/IEC 14764.

with the implementation effort parameter. The functional modifiability implies more the characteristics of underlying computing architecture flexibility and is closely associated with the performance of an implementation after introducing algorithmic variations.

As it can be seen from the previous section, the code amount for describing a similar functionality for the FPGA and GPU significantly differs in favour of the latter technology. This is natural as the description of the functionality is done at different levels of abstraction – the logic design requires some hundred thousand signals to be described with their logical connections and synchronisation, whereas the software development only needs the description of the data manipulation sequence on a well-established set of hardware. Therefore, it is evident that bringing in changes to a code and ensuring its functional correctness is significantly easier for the software approach.

However, the extent of changes for upgrading a code to an enhanced functionality, i.e. the proportion of newly introduced code to the stable code, and the influence of these changes to the application performance is ambiguous. Different solutions will react to the introduction of minor qualitatively changes to a target algorithm with a variable response for the application performance. The functional modifiability aims to reflect the flexibility of an implementation on a target computing architecture to enhance the algorithm for performance. This characteristic is definitely not a universal characteristic of a system but should be viewed in combination with a particular algorithm or computing method[87].

For the GSC method this metric is best applied to *the linking phase* which offers several variations of sequential linking approaches each influencing the quality of the segmentation results (refer to Section 4.1.1.2). Both the FPGA and GPU solutions realise the Centroid Linking Method, which can be enhanced by two modifications. One realises an advanced procedure for the next candidate selection by using the minimal feature distance criteria. The other consists in an enhanced feature computation by weighting the subregions depending on their sizes (number of CE_{i-2} or number of positions taken in a respective island I_{i-1}).

For the realisation of the enhanced candidate selection in FPGA a reduced variant of the bitonic sorting algorithm can be effectively used as a parallel solution to find the minimal feature distance for the candidates. For six potential candidates the utilisation of this algorithm increases the number of clocks per linking step to four clocks in case of forward tree traversing. This modification increases the linking time by about 1.6 times as modelling indicated. The modification of the linking processor requires adding two extra stages in the linker SM and some combinatorial logic for the realisation of the bitonic algorithm. The implementation of this upgrade changed the code length of the general linking processor by about 8% counted in physical SLOCs (added, removed, or modified against the original number of code lines).

The realisation of this linking enhancement for the GPU leads to more dramatic changes in the code. In fact the optimised approach of the bit-vector gluing does not work for the selective linking

[87] When treated generally this characteristic should be understood as flexibility of a computing system or similar. Such analysis is too general and is beyond the scope of the work.

as it cannot use the bit-position analysis in the connectivity table. The modification of the linking method requires primarily the modification of the gluing loop (refer to Section 5.2.3), which increases dramatically the number of connectivity vector traverse iterations and makes a compact overlap-list representation impossible, thus leading to a higher shared resource demand. The modification of the linking algorithm results in a 36% enlarged code and increases the elapse time of general linker kernel by 2.7 times as the profiling showed.

The expenses for the weighted feature modification are much higher in both cases. Due to the significant modifications in the designs requiring a solid reengineering these expenses are described roughly without precise figures.

The weighted feature modification requires cardinal changes in those GPU kernels that are related to the global modification of data structures. To implement the weighting a kernel needs to have the information about the size of a region. This size is computed during the linking of lower hierarchical levels and needs to be forwarded to an upper linking procedure via an overlap-list creation process using *the GSC DB* (joining the linking and overlap-list creation kernels showed to be ineffective for application performance), which can significantly influence the traffic to and from the external memory. In this case the code of the kernels requires not a lot of modifications and only relatively inexpensive arithmetic add-ons, which can be estimated in a dozen of lines per kernel.

In contrast, the implementation of this functionality in the FPGA solution faces a problem being related to the realisation of arithmetic operations. At the same time the data structures in the external memory are not affected significantly. The region weight information can be integrated in the existing overlap-list structure without any changes in its size. This means that the overall external traffic of the system does not change, which is significant for the performance of a data-driven system. At the same time the FPGA solution suffers from the lack of the implementation of a fast division operation. The division module being implemented in the studies and realising a slow division algorithm requires $n+2$ clocks for one division operation, where n is the bit width of a divisor. Although the division operation is executed only once per linked region, this additional time can be quite sensitive for the linker's performance provided that the number of linking processors cannot be increased for pipeline rebalancing. The implementation of fast division methods, e.g. multiplication by the divisors' reciprocal is the subject of a serious redesigns of the data path of the linker processor. The code changes in this case can be comparable to the size of the original module.

7 Summary

The progress in computer science and technology, achieved in the beginning of the XXI century, has opened the era of high performance computing in the segment of mini- and microcomputers. This made the power of supercomputers and the high accuracy and realism of advanced computing algorithms available to a vast range of end users. This technological progress became possible with the adaptation of parallel computing to small- and midrange machines. However, the price for this advancement is the necessity for end users to abandon the old stereotypes of application development and to conform their mind with parallel computation thinking.

After a transient period of compact HPC maturing two technologies stood out of many parallel computing initiative: FPGA- and GPU-based solutions. The FPGA technology took its respective place due to its relatively long and successful history in the field of embedded solutions and prototyping. There it showed a distinctly high flexibility to realise application specific designs and a relative ease of application design which is typical for solutions based on standard devices. The maturity of the development chain and the high integration level of ICs, achieved around the beginning of the 2000s, played an important role for its popularity in the incipient field of compact HPC.

The success of general purpose computing on GPU is explained by the market competition of the PC entertainment manufacturers. The increasing demands for virtual reality in 3D games have forced the graphical card vendors to endeavour in refining the architectures of graphical processors. At a certain point of the evolution when GPU architectures become flexible enough a number of initiatives appeared to adapt the architecture for general purpose computing in the academic society. Eventually this initiative was enthusiastically taken up by the GPU vendors when promising prospects of this trend became obvious. This strong support of GPGPU from the side of the manufacturers brought new life to GPU technologies. Computations on GPU became a separate trend in high performance computing steering aside from the initial graphical background. In the present work both technologies were addressed for the implementation of a complex image processing algorithm, which allowed the comparison of their peculiarities, their relative merits, and downsides.

Although new parallel computing technologies open broad perspectives for high performance computing, not all existent algorithms can be adopted for parallel processing. That is why a scrupulous analysis of computation methods in order to discover their parallelisation potentials is of primary importance for the adoption of HPC.

The application field in this work addresses image segmentation, which takes a central place in the image processing chain for automated image analysis. Segmentation methods being fast, having high segmentation quality and a high level of automation, are of high interest in applications. The Grey Value Structure Code (GSC) studied in this work belongs to those methods.

In the present work the GSC method has been analysed for its parallelisation potential at different levels of granularity and implemented on a Xilinx Virtex II Pro FPGA card specially designed for memory intensive applications and on the high performance nVidia G80 and GT200 GPU cards designed for general purpose computing. Thus, the work covers two aspects of computer science: algorithmic and technological.

As an immersion into the technological field the general tendencies of modern computing technologies were brought up, followed by a detailed discussion of the FPGA and GPU technologies' fundamentals. As particular modern examples of the two computing platforms the Xilinx Virtex II and the nVidia G80 architectures were described in brief. Although both platforms are competitors in compact HPC they are two principally different classes of computing architectures. This means that the application implementation approaches for the two computing platforms are principally different. This difference has been described in this work in the form of a survey on methodologies for the application development on FPGA and GPU platforms after an initial introduction to the underlying hardware.

The importance of the methodology rises with the increasing complexity of a design. The absence of a methodological guideline endangers both the quality of the design and the realisability of a project in an appropriate time. In this work a methodical top-down approach to the design process has been worked out as the only way to handle a complex system development.

This approach is especially important for the development of the FPGA-based applications, as a designer needs to bridge a large gap between the initial highly abstract specification and very detailed description of a system at the logic design level. The top-down design approach was described as a gradual refinement process of the initial specification through different levels of abstraction and represented as a spiral of specification-implementation-verification-modelling development cycles stretched over the description abstraction axis. A special attention was paid to the verification of RTL models. The significance of an exhaustive verification drastically increases with the increase of a design complexity. An application development cannot be treated as being complete until the functional correctness is approved by thorough verification. Moreover errors that inevitably appear during a development process can hardly ever been detected without a carefully

planned verification policy and robust verification infrastructure. The questions related to the creation of an effective verification environment were dealt with in this work.

The methodology for an efficient application partitioning for the GPU platform was based on the peculiarities of the stream processing architecture and has been discussed in this work. Consequently the GSC method was analysed and implemented in compliance with these methodologies.

In the application development phase a number of application models were implemented at different levels of abstraction for the analysis of the GSC method in many implementation aspects. The *functional data-flow model* allowed figuring out the maximal potential for the parallelisation of the method at different granularity levels. The analysis of the *data-flow model* showed that the GSC method has a strong parallelisation potential which can be exploited using different massive parallel architectures of *symmetric memory access* class. The analysis helped to tailor the method for maximal exploitation of the algorithm potential on the parallel platforms. Using this model the data dependency graphs of the method together with the data-flow intensities were analysed to get a clear picture of the optimal partitioning of computation resources and the efficient organisation of data structures of the application.

Using the *functional data-flow model* it was particularly shown that the hierarchical islands can be processed independently within a hierarchical level and the data dependency exists only between the superimposed hierarchical layers, which makes the GSC layer processing attractive for massive parallel implementations. The *regular array data organisation* has been shown the most effective for the target implementation platforms with symmetric memory access. An optimal data exchange scheme between the hierarchical layers during *the linking phase* was identified. This scheme based on the special region connectivity structures called *Overlap Lists* allows significant memory access reduction critical for parallel computing machines. It was figured out that for massive parallel architecture the most efficient approach to result generation for the GSC method is the *layer-wise downpropagation* of the segment data.

The executable functional model built at this stage was used as the basic model for the further hardware system refinement and the reference for the verification of subsequent detailed system models for FPGA- and GPU-based designs. It was also used for statistical information gathering by profiling the model with the realistic data sets for detailed specification of precise data formats effective for further hardware and software implementations.

Using the knowledge obtained from the functional models' analysis, after the precise specification of the FPGA computing platform and the elaboration of an appropriate data-structure layout, the GSC *architectural models* were built to steer the design process for the precise hardware platform implementation. Analysis of the *data-stream model* was used for optimisation of the external data traffic of the system and defining effective GSC data-to-memory space mapping configurations in the FPGA-board architecture.

In particular, it was figured out that *joining* the coding and the linking *processing of the first and the second hierarchical layers* during *the linking phase* can drastically decrease the data traffic to GSC data structures stored in the external memory. Additionally, a special island processing scheme for *the linking* and *the result generation phases* was developed for external traffic reduction and for improvement of memory assess scheduling efficiency. This scheme implies processing islands using *sliding window* moving over the *overlapping island row* blocks.

After specifying the precise hardware data model, the *static data-stream model* (*Traffic Model*) enabled the first precise estimation of the data traffic circulating in the system, which made initial measurements of the potential application performance available. These early performance estimations served as a criterion for the efficiency of a particular HW platform and the efficiency of a particular application implementation on an FPGA.

The *partially-timed executable architectural model* operated on the data-token level made this data-flows circulation available for dynamic observation in the complete communication system of the FPGA platform. The *cycle accurate architectural model* defined precise behaviour of the GSC processing pipelines in the clock domain. It was used for careful balancing of different pipeline stages to find a performance/resource compromise for implementation of the functional units in the given hardware. During elaboration of the precise GSC algorithms for time domain partitioning a special attention was paid to finding an optimal coding scheme for initial region forming at the pixel level. It was shown that an effective scheme for structural coding algorithm implementation can be realised as a pipeline using four basic subgraphs in the hexagonal pixel island graph for optimal resource utilisation. For acceleration of the sequential linking method a special topology aware approach to connectivity graph traversing was proposed. A weight-ranking mechanism was also proposed to increase the quality of the region connectivity data in the fixed-sized overlap-list structures.

The early *resource model* was developed to give an approximate estimation on the realisability of the proposed implementation in the given FPGA before implementing the GSC at the RTL level. The approach is based on the calculation of internal data-aggregate sizes requiring clocked on-chip memory (flip-flops) of the FPGA and showed relatively high correlation with the results of RTL synthesis.

The *cycle accurate interface model* of the communication system was implemented to gather statistical data on the delays and latencies introduced by the communication infrastructure of the system. Implementation of this model made the performance estimation totally realistic. The complete cycle accurate model of the system allowed the optimal configurations for the GSC application pipeline for achieving the maximal application performance on the given platform to be determined.

After the implementation at RTL, the synthesisable application was exhaustively verified using the previously developed functional and temporal logic models to assure the functional correctness of

the design. The postsynthesis place-and-route procedure produced the exact timing and resource utilisation parameters of the final physical implementation.

The most valuable information of the modelling, the analysis, and the implementation of the GSC application in the FPGA was reported in the present work. The studies clearly have showed that performance of the GSC application on application specific design platforms is definitely restricted by the throughput of the external memory. The traffic pattern defined by the application's buffering scheme has a certain influence to the application throughput. The most serious constraint for the application performance on the target Virtex II Pro platform is the lack of hardware resources due to the high complexity of the application. This resource shortage does not allow a well-balanced GSC pipeline configuration to achieve the maximal throughput of the memory subsystem.

Using the information attained from analysis of the high level GSC models, the executable functional model was partitioned to the GPU stream processing architecture and implemented using the nVidia CUDA C language extensions. Although there is a general guideline for the application implementation in the field of stream processing, which is based on architectural assumptions, strict coding rules for achieving the optimal performance do not exist. This is why the optimisation work for a GPGPU application consists in scrupulous profiling of an application. Many implementations of the GSC kernels have been created for determining the optimal configurations of the kernels. The GPU-GSC application was ran on two nVidia GPU architectures: G80, the first architecture with CUDA support, and GT200, the second GPGPU architecture. The experiments with the G80 architecture showed unacceptable results in the performance, which was even much behind the reference CPU implementation. A top GT200 card performed significantly better and could compete with the FPGA solution. Therefore, only the results related to GT200 architecture have been reported in this work.

The GPU experiments were primarily focused on the external and internal data structures' organisation, the influence of data traffic manipulations, the optimisation of the code flow, the application of GPU hardware-accelerated functions, the influence of specialised memory resources, and the block size configuration. The most distinguished results have been reported in this work.

During the studies it was found that GPUs are notably sensitive to the complexity of a kernel's code flow and shared resource demands, as it can be particularly seen for the linking kernel. Using special memory resources, such as texture and constant memory, can significantly ease the problem with the shared memory resource deficit, if no intensive interaction over the data structure is needed. In some cases the utilisation of a GPU-accelerated function can significantly boost up computations. A careful control-flow analysis and optimisation can help in many cases, although in some cases the results are contrary to the expectations, especially if memory operations are involved. The search for an optimal thread block configuration is more a heuristic than a logical procedure and in many cases it was carried out by trying.

The implementation results of both solutions were compared with each other and with the reference single thread and SMP CPU implementations. The performance comparison showed a big advantage of parallel implementations due to their extensive exploitation of finer parallelisation granularity. The comparison of the FPGA and GPU solutions indicated that they became rivals at higher image resolutions, though with smaller data sets the FPGA-GSC left the GPU-GSC behind.

Except the performance parameters the solutions were compared by the implementations' efforts using a physical code line count metrics and by the flexibility of the computing platforms to adapt the solutions to functional enhancements in the GSC method. As the comparison showed the GPU implementation requires much lower development efforts due to the significantly higher level of abstraction required for the GSC description. At the same time the ease of the functional upgrade is more intricate. In some cases the constrains induced by the SIMD architecture peculiarities make functional modifications complicated leading to significant changes in the code or to a complete redesign of a kernel. In these cases the flexibility of the FPGA architecture gives a big advantage to a designer requiring only minimal changes to a code. At the same time the lack of primitive operation units in the FPGA architecture can lead to a cardinal reengineering of complete modules, whereas the standard ISA of a GPU allows those algorithmic enhancements without any costs.

8 Conclusions and outlook

8.1 Conclusions

Generalising the knowledge acquired from the work, it can be stated that data-parallel segmentation applications using low precision fixed point values, such as the GSC, can be successfully implemented on both FPGA- and GPU-based platforms gaining a significant speed-up factor compared to sequential CPU processing. The achieved acceleration endorses the use of the parallel GSC in real-time and interactive applications. The studies showed that a huge bulk of the work is dedicated to the analytic aspects of the application implementation on a parallel platform, i.e. the exploration, identification and development of the parallelisation potential of an algorithm. In general, an FPGA implementation designer has more freedom for the organisation of the computation. In the case of the GPU, the application design requires an additional adaptation of the algorithm to the fixed target architecture having a lower adaptation degree for the parallelisation granularity. In particular, a designer needs to keep in mind the SIMD-like organisation of the computations and the strongly restricted memory resources of the multiprocessors.

From the practical point of view, the studies showed that the development cycle for an FPGA application is significantly longer and more complex than for a GPU application. In particular, the verification of a project is a specific challenge for a hardware design.

The implementation of this particular application showed that GPU platforms are most effective for a huge amount of data, which enables the scheduler to exploit the full potential of the memory channels and to optimally distribute the workload. At the same time, the flexibility of an FPGA design allows building fully customised data management schemes elastic to data volumes.

Looking at computing applications at the platform (macro architecture) level, the performance of a data parallel application is limited by the throughput of the memory subsystem of the computing platform it is realised on. Therefore, one of the primary targets for an application design is to optimally use the bandwidth of the platform's memory channels. Traffic planning plays an important role in an FPGA design. Hence it is essential to have preimplementation models for the strategical planning of the design. In particular, the planning of the data-structure organisation and traffic models can give a lot of information for an accurate platform architecture design to obtain

the maximum performance gain. The studies showed that, although, having a much lower bandwidth in the external memory channels and operating at lower frequencies, the semi-custom solutions can be competitive or even outperform the standard processing systems. The fundamental reason for this is that the FPGA design can use the bandwidth of the available channels more efficiently.

First of all, this can be explained by the fact that the utilisation of more compact data formats reduces the overall volume of the traffic, which is in its turn enabled by the bit-level and operation-level parallelism available in FPGAs. In contrast, the regular ISA of standard architectures (GPU, CPU) do not allow such flexible operations on bit fields. This forces a designer to choose between a higher number of processor cycles for the data preparation (data extraction and packing), or a higher volume of unpacked data which consumes a solid portion of the bandwidth. Meanwhile, if the data size is critical, as occurred in the GPU-GSC application when placing data into the shared memory, the special bit functions can be extremely helpful for reducing the processor cycles.

The second reason for a more effective memory bandwidth utilisation of FPGAs is their ability to organise a more dense traffic for the same data volume, which is achieved by application-specific buffering schemes and traffic scheduling. In a GPU engine the task of the traffic organisation is entrusted to a universal scheduler, which cannot always provide an optimal data flow. Meanwhile, the GPU schedulers are known to be specially designed for SIMD-like execution which can benefit from burst-oriented linear accesses to the dynamic memory. Thus, the efficiency of the bandwidth utilisation can be significantly increased by carefully organising the data layout as shown in this work. In simple terms, the basic principle of data layout is a transition from arrays of structures to structures of arrays.

Looking at the application performance from the point of view of the computing microarchitecture it is necessary to note that the ability to create custom data-paths is a significant advantage of the FPGA architecture. The data storage and processing elements in those paths are placed maximal-logical compact enabling an exclusive memory space for the processing elements. In contrast, in GPUs the data resides in common data pools (shared memory, register files), which may cause memory resource conflicts among the parallel processes, increasing the number of processor cycles. Therefore, similar to the external memory, the data layout in the on-chip memory of a GPU may influence the performance of an application significantly. Similarly, the problem of representing the data in the shared memory in a compact form can be critical for the performance. In contrast to FPGAs having a large volume of fine-grained memory elements, which enable a precise allocation of the memory for the data structures being specific for customised data paths, the memory inside the multiprocessors of a GPU is distributed evenly among all processes in a block of threads, while the functional units are designed for processing fixed size memory words. Both features can significantly increase the volume of data structures for a single process and thus decrease the effectiveness of the workload of a multiprocessor. As the work showed, the resources of the shared memory are one of the most critical problems for the performance of memory resource demanding

computation tasks. Partially, this problem can be solved by using specialised on-chip memory (constant and texture) with some limitation to the data access types and by exploiting the benefits of full-fledged caching mechanisms being principally new in GPU technology.

The studies showed that the performance of the application implemented on GPUs is dependent on the complexity of the kernel's control flow as a result of the SIMD nature of the GPU architecture. Thus, the control flow optimisation of GPU kernels can be an effective mean for evening-out the peculiarities of SIMD-like architectures for minimising performance losses. In contrast, the FPGA implementation, in addition to its data parallel computation capability, can benefit from the true parallelism of the execution flow being enabled by using independent customised state machines.

At the same time, the realisation of arithmetic intensive computations, with a large variety of non-trivial arithmetic operations, can be rather problematic for the implementation on FPGA-based computing platforms. For gaining application versatility, the FPGA architecture is built primarily on fine-grained logic architectural primitives supplemented by relatively simple signal-processing oriented functional blocks. Although the complexity of these blocks gradually increases, shifting towards primitive ALUs [137], complex arithmetic functions are not implemented as hardcore blocks and therefore require a multicycle implementation. In contrast, GPU multiprocessors contain arrays of hardcore functional units implementing a variety of arithmetic functions. Although the complex arithmetic functions typically require a number of clock cycles to be completed, the hardcore implementation allows them to be executed faster thus reducing the operation delays. This makes the GPU attractive for parallel applications with a predominant arithmetic computation burden.

8.2 Outlook

8.2.1 Performance on prospective FPGA and GPU platforms

Considering the implementation of the 2D-GSC on the latest and prospective FPGA and GPU platforms, the throughput of the memory subsystem should be primarily taken into account as a limiting factor for the performance progress. An increase in the throughput of the memory subsystem of the FPGA platform can be achieved by the implementation of the newer DDR3 [138] and the perspective DDR4 [139] SDRAM technologies. However, the achievable bandwidth enlargement is generally restricted by the technological aspects of FPGA ICs' manufacturing and, in particular, the capability of the device's I/O circuitry. For the Xilinx 7 FPGA device family the I/O capacity is limited by DDR3 operating at 933 MHz [140].

The other parameter, determining a possible increase in the application performance, is the capability of the FPGA-GSC design to satisfy the traffic throughput generated in the external data channels. Figure 50 shows the increase in the resources for the logic and memory elements for Xilinx FPGAs [141-145]. It can be seen that the fine-grained logic resources of the top models of

the 7th Xilinx FPGA family are increased more than 20 times in flip-flops and more than 10 times in LUT function generators[88] compared to the Virtex II Pro device used for the GSC implementation.

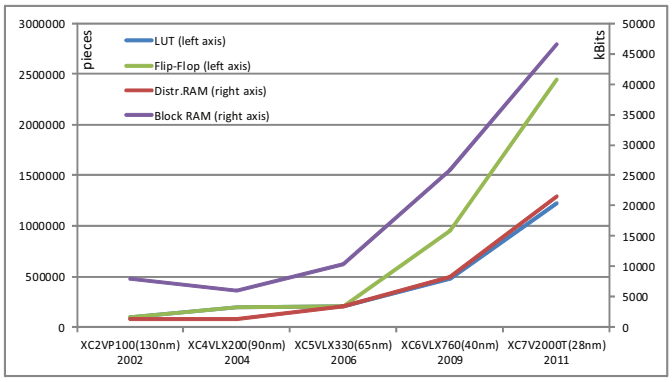


***Figure 50**: FPGA technological trends (Virtex FPGA family evolution)*

Provided the FPGA-board architecture persists the same and taking conservative clocking assumption for the external memory channels equal to 533 MHz of a single clock rate[89], the data throughput of the system increases 5.3 times. Additionally, assuming only a clock frequency of 266MHz[90] for the GSC core to ease the timing inside the chip, the throughput can be satisfied by GSC processing units containing twice as many processors in the processor arrays. Moreover, the remaining chip resources can be used to achieve the saturation of the memory channel bandwidth by a further increase of the number of GSC linking and labelling processors, as discussed in Section 4.3.4, gaining another 60% decrease in processing time. All in all, implementing the FPGA-GSC on a modern Virtex device might speed up the application 10 times at least.

The latest generation of the Xilinx Virtex 7 family is manufactured in cutting-edge 28 nm technology [146]. According to the plans of the IC contract manufacturers, who provide the production lines for fabless semiconductor companies [147-148] as Xilinx and nVidia, the massive transition to a finer manufacturing technology is expected not earlier than 2013. This indicates that the next generation of Xilinx FPGA devices can likely not be expected in the next two years.

To estimate the performance potential of the GPU-GSC implementation on next generation GPU platforms, the following technical trends are considered. The first determinative parameter, influencing the peak performance of a GPU architecture, is the bandwidth of its external memory channel. The second significant characteristic of the GPU architecture is related to the memory subsystem architecture and can be quantitatively valued by the amount of on-chip memory being

[88] From Virtex 5 FPGA generation on the Xilinx FPGA architecture includes 6-input and 2-output LUT elements capable to be configured in a two 5-to-1 LUT mode with common inputs or a combination of two 3- or 2-input LUT mode, which offer more powerful and configuration-flexible function generation resources compared to the 4-to-1 LUTs of Virtex-II architecture.

[89] Clocking of the memory control interface.

[90] The half frequency of the external RAM memory interfaces for core clocking is selected to avoid timing constraint problems, which can arise due to an increase in the number of the processing elements in the GSC processing units. Moreover the doubled frequency of the memory channels can solve the problem of buffer latencies while reading data.

available for the application[91]. Although the GPU generations can differ in the architecture of the processing units and in the multiprocessor organisation, the influence of these qualitative changes to the performance of an arbitrary application is difficult to predict, therefore, they are not treated as prognostic features here.

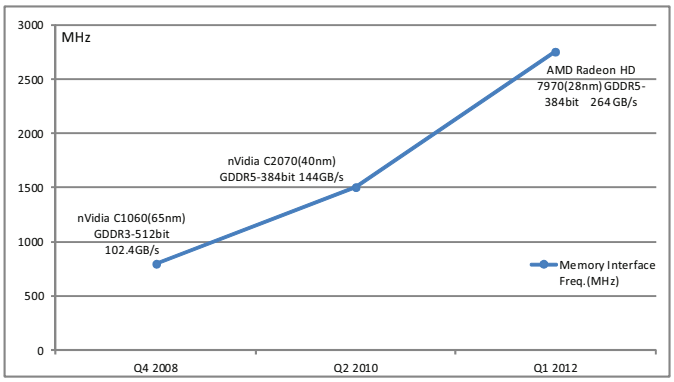


***Figure 51**: Bandwidth trend of GPU-board memory interfaces*

The tendency of the improvement in GPU technology can be seen by comparing the nVidia C1060 (GT200 architecture) [149], the GPU-GSC was implemented on, with the latest GPGPU nVidia C2070 card [150] which is based on the Fermi (GF100) architecture [151]. The bandwidth increase of the graphical memory of those cards is 1.4 caused by migration from the GDDR3 to the GDDR5 memory type (102.4 GB/s with an effective data rate of 1.6 Gbps[92] against 144 GB/s with an effective data rate of 3 Gbps). A significantly higher increase appears for the on-chip nonspecialised memory per SM: 16 KB of shared memory for the GT200 architecture against 64 KB of configurable on-chip memory (shared and L1 cache memory with 48 KB maximally available as shared memory) for the GF100 architecture. Directly porting the GPU-GSC to the C2070 platform results in a performance increase of about 2.5. This difference is significant due to the fact that the performance increase cannot be solely explained by broadening the graphical memory bandwidth. Thus, the qualitative increase can be ascribed to changes in the architecture, most likely to the changes in the shared memory having been shown to be of primary importance of for the application performance. Nonetheless, the impact of qualitative architecture changes cannot be neglected for further possible improvements of the GPU-GSC. Most notable among those changes are those being introduced as a result of the higher adaptation of the GPU architectures to general purpose computing. In particular, the hierarchy of the memory subsystem has adapted characteristic features of CPUs. The emphasis has shifted towards the reduction of the data access latency by implementing a two-level cache mechanism. This may ease the problem with the shared memory shortage in the GPU-GSC application.

[91] For example, the amount of shared memory per SM is critical for *the linking phase* which is the most time consuming stage of the segmentation pipeline.
[92] Gigabit per second is an effective data rate at one data lane of a memory interface.

In general, the change in the memory subsystem can be quite significant for the performance of an GPU application, but difficult to estimate numerically. The appearance of the full-fledged caching mechanism critically shifts the application of the shared memory from means of fast access storage, placed maximally close to the processing units, to a pure mechanism of process communication. This shift in the paradigm releases the multiprocessors from the problem of insufficiently loading the work for kernels that require high-volume, versatile, and intensive random memory-access operations and can increase the efficiency of the traffic scheduling and thus the general application performance.

The latest tendency in the GPU technology can be seen with the recent transition of AMD GPUs to the 28 nm technology[93]. The top card, issued at the beginning of 2012, has a graphic memory interface capable for 264 GB/s [152] which is about 1.8 faster than the memory interface of the nVidia C2070 (Figure 51). The increase in the amount of on-chip memory for the next generation of nVidia products (28 nm technology) can be presumably expected to be equal to the existing ratio of 1.7 between the on-chip resources of the AMD Northern Island (Barts) architecture (40 nm technology) [153] and the AMD Southern Island (Tahiti) architecture (28 nm technology) [152][94]. Assuming that the scaling factor of the application performance is not less than the increase of the memory bandwidth and that at the same time the extra gain from on-chip architectural improvement does not exceed the ratio of the on-chip memory resources increase, the performance of the GPU-GSC implementation can be expected to be approximately 5 times higher in a planning horizon of minimum two years as a rough extrapolation. Thus, the GPU prospective shows less potential in performance in comparison with FPGA platforms.

8.2.2 Future work

The present studies are primarily concentrated on the performance gain of the application exploiting the two popular massive parallel architectures. Meanwhile, the question of feasibility of the parallel GSC in different application fields is not highlighted in this work. Different application areas can be more or less sensitive to the quality of the segmentation. The parallel implementations are designed to gain the most performance out of the inspected parallel architectures, although keeping the quality of segmentation as high as possible in the given conditions. Meanwhile, it is interesting how critical the quality improvements, introduced in [3] for different application areas (singularity correction, pixel splitting, improved linking approaches), can be and how critical they are for the application performance on a particular platform. Of special interest is the question of the influence to the application performance when integrating the quality improvement methods into the parallel implementation or performing them as preprocessing/postprocessing stages to the parallel application. In particular, using local threshold methods, in which, for example, the thresholds are defined as an image map derived from the local image dynamics, can be of an interesting option for

[93] The first nVidia GPU card done with 28 nm technology are not officially announced as for January 2012.
[94] Compared are Barts' shared memory (48 KB) and Tahiti's shared memory together with the level one cache (80KB) per multiprocessor. Barts architecture does not have general purpose caches.

segmentation quality improvement in comparison to advanced and potentially performance critical linking methods.

Relying solely on the same principles for the regions generation and labels propagation as in the 2D case, the 3D version of the GSC suffers from a dramatical increase in the sizes of the data structures, which make them unattractive for those computation platforms the parallel 2D-GSC was implemented on. For example, the overlap list should be increased from 12 overlap positions of 3 overlapping pairs (144 bytes in the most compact representation) in 2D case to 24 by 8 overlapping pairs (816 bytes minimum) in the 3D version[95]. With the growth of the resources in modern FPGAs and GPUs it can become possible to extend the parallel GSC to the 3rd dimension and to open new application areas for this segmentation approach.

[95] The number of CE position bit-flags in the connectivity vector table in GPU- GSC will increase in about 36 times.

A Appendix

A.1 Virtex II Pro architecture elements

A.1.1 Input/Output blocks

Input/output block buffers contain a variety of resources to guarantee the electrical protection of die circuits, signal integrity and data consistency. These resources include pull-up and pull-down resistors, bus weak-keeper circuits, digitally control impedances for parallel and series line termination, clamping diodes, drive-strength, slew-rate control circuits, and programmable input delays.

The I/O blocks of the FPGA contain additional logic to provide fully synchronous and constraint data paths inside or outside of the chip. Each I/O block contains six storage elements which can be configured as either edge-triggered D-type flip-flops or as level-sensitive latches with asynchronous or synchronous set and reset. These registers are used for input and output paths and tri-state output buffer control. The number of registers is doubled for double-data-rate operating. In case of DDR mode each coupled register is to be clocked 180 degrees out of phase.

The I/O blocks are organised in groups of two or four to be connected to a single switch matrix. The adjacent blocks can be used as a differential pair.

A.1.2 Logic resources

The Virtex II Pro has three types of logic resource blocks which are seen as architectural units in the global interconnection system of a die. They are Configurable Logic Blocks, 18Kb Block SelectRAM modules and 18x18 two's complement signed multiplier blocks.

Configurable Logic Block (CLB)

A CLB element comprises four similar slices connected by a fast local network for signal feedback within the CLB. Each CLB element is tied to a switch matrix to access the general routing network and is equipped with a number of dedicated resources for fast signal exchange with neighboring

CLBs. The four slices are organised in two columns of two slices with two independent carry logic chains and one common shift chain.

Each slice contains the similar set of logic resources, which are:

- two LUT-based function generators each capable of implementing any arbitrarily defined Boolean function of four inputs, 16-bit shift registers, or 16-bit distributed RAM memory (deeper and wider RAM configurations are possible if combined and cascaded within a CLB);
- two storage elements that can be configured as either edge-triggered D-type flip-flops or level-sensitive latches with common synchronous or asynchronous set and reset signals;
- arithmetic logic gates including an XOR gate that allows a 2-bit full adder to be implemented within a slice and a dedicated AND gate that improves the efficiency of multiplier implementation;
- large multiplexers capable to combine function generators to provide any function of five to eight inputs and selected functions of nine inputs within a slice;
- shift register dedicated chains that provide connection between shift registers without using the ordinary LUT outputs. The shift register chaining together with large multiplexers allow an up to 128-bit shift register with addressable access to be implemented in one CLB;
- a dedicated carry logic that provides fast arithmetic addition and subtraction signal propagation between slices and CLBs;
- a dedicated carry multiplexer together with a carry path can be used to cascade function generators for implementing wide logic functions;
- a horizontal cascade chain together with a dedicated OR gate are designed for implementing large, flexible Sum of Product planes or other combinatorial logic functions.

18 Kbit Block SelectRAM resources

The on-chip storage, used for buffering, caching or queuing mechanisms, play an important role for complex SoC designs. The Virtex II Pro provides a large number of hard-core RAM modules with flexible configuration capabilities for the resource/performance optimisation of a design. Block RAM modules save Configurable Logic Block resources in memory intensive applications.

The block SelectRAM modules are 18 Kb dual-port single cycle latency synchronous static RAMs. The RAM modules are configurable from 16Kx1-bit to 512x36-bits in various depth and width configurations. Each port is totally uncoupled in control and clocking. Both ports of a module can be configured with independent data/address aspect ratios. Memory modules are placed in interleaving columns that simplify the cascading of the modules to implement large embedded storage blocks.

Multiplier blocks

Another kind of dedicated hard-core blocks is the 18x18 two's complement signed multiplier. The columns of the multiplier blocks are placed next to the block RAM columns. Each multiplier can be used independently or in association with the adjacent RAM block. A multiplier is optimised for operations on the content of the adjacent memory block, which can be helpful in DSP applications, for example.

A.1.3 Clocking resources

It is normal for modern complex systems to have multiple clock domains on a single die. The reason for that could be the need for interfacing buses with clocking constraints, which are different to a chosen reference frequency of the on-chip design or a particular decision for internal computation components clocking disparity. The Virtex II Pro conforms to modern imperatives. It contains a variety of resources for multidomain designs, which includes:

- 16 clock input pins that can also be used as regular user I/Os;
- 16 global clock buffers that can either be driven by the clock pad to distribute a clock directly to the device, or driven by a Digital Clock Manager, or by local interconnects. These buffers can be configured to gate the clock or to multiplex between two independent clock inputs;
- 4 to 12 Digital Clock Managers with fully digital delay lines allowing a high-precision control of the clock phase and frequency for clock deskewing, frequency synthesis and phase shifting;
- 8 dedicated low-skew global clock trees per quadrant of a device.

A.1.4 Interconnection system

The interconnection network of an FPGA can be divided into global, local and dedicated signal routing subsystems. The global network provides general purpose resources for the connectivity of programmable logic elements such as CLBs, IOBs, RAM blocks, and multipliers of the device. Local routing resources are capable for a fast connection within the CLBs without using the global interconnection resources. The dedicated routing subsystem is a collection of specialised signal resources, which are typically used for particular functionality.

The global interconnection system consists of programmable routing switch matrices and connection lines of different types. The programmable logic blocks are all connected to identical switch matrices for the access to global routing resources. The programmable switch matrices are placed at the intersections of vertical and horizontal routing channels to configure a network of signal lines. There are four types of signal lines that carry signals for different distance:

- the long lines span the full height and width of the device;
- the hex lines route signals to every third or sixth block, but can be driven only at their end points and not at the middle;
- the double lines are similar to hex lines, but three times shorter and route signals to every first or second block;
- the direct connect lines route signals to all eight neighboring blocks.

The interconnection network is characterised by the density of connection lines. Each routing channel of a device consists of 24 long lines, 120 hex lines, 40 double lines. Each CLB has a total of 16 direct connections.

The subsystem of local interconnections provides a fast connectivity between slices inside CLBs. A local network consists of eight of such lines per logic block.

Additionally the Virtex II Pro has dedicated signal resources. They include shift-register chain signals (one per CLB), a sum of product logic expander (2 per CLB row), fast arithmetic carry chains (2 per CLB column), and segmentable 3-state bus on-chip lines (4 per CLB row).

A.2 Implementation calculations

A.2.1 Overlap-list rationales

The advantages of the proposed overlap-list approach comparing to the overlap detection scheme via up-and-down hierarchical search can be shown with a simple estimation of the number of memory accesses to external storage required for deriving a connectivity map of a macroisland. During linking in a macroisland I_{i+1} all regions R_i in all seven islands I_i covered by the macroisland have to be checked for existence of a child R_{i-1}. This implies accesses to all 84 regions R_i in a macroisland I_{i+1} (the number of regions in an island is limited to 12 due to statistical assessment). This number is the minimal number of accesses, provided that all inspected regions R_i have no inheritance. Each downward relationship detection leads to an access to a subregion R_{i-1} two layers below the current level. Assuming that a data prefetching mechanism is used, which is favourable for the systems with a plain memory access scheduling, the implementation has to prebuffer all the subregions R_{i-1} available in the 12 overlapping nodes (I_{i-1}) of a macroisland I_{i+1}, as the subregions R_{i-1} common for the regions R_i from different I_is can be found only there. Thus, the number of data pieces needed to be retrieved from external storage increases by 144.

Provided that the connectivity information is specially generated in the previous level pass and stored separately in specialised overlap-list data structures, the approach implies only access to 72 regions per macroisland, which are organised in 36 region pairs per overlap-list (the number of pairs is determined by the statistical assessments).

A.2.2 Average region feature calculation error for sequential linking

The arithmetical average of region feature is

$$F_{arith} = \sum_{i=1}^{n} f_i / n ,$$

where n is the number of subregions in the chain, f_i is the feature of a subregion.

The computed feature for sequential linking is

$$F_{seq} = f_1 / 2^n + \sum_{i=1}^{n} \left(f_i / 2^{(n+1-i)} \right) .$$

$$Assumption: \forall f_i \approx f(similar\ features) \Rightarrow \Delta = F_{arith} - F_{seq} = n * f / n - f / 2^n - \sum_{i=1}^{n} f / 2^{(n+1-i)}$$

$$\Rightarrow \Delta = f\left(1 - 2^{-(n+1)} * \left(2 + \sum_{i=1}^{n} 2^i\right)\right) = f\left(1 - 2^{-(n+1)} * \left(1 + \sum_{i=0}^{n} 2^i\right)\right);\ \sum_{i=0}^{n} x^i = \left(1 - x^{(n+1)}\right) / (1 - x)$$

$$\Rightarrow \Delta = f * \left(1 - 2^{-(n+1)} * \left(1 + \left(1 - 2^{(n+1)}\right) / (1 - 2)\right)\right) \quad = f * \left(1 - 2^{-(n+1)} * \left(1 - 1 + 2^{(n+1)}\right)\right) = f * (1 - 1) = 0$$

A.3 Traffic model

The Traffic Model is based on the number of memory words transferred to the GSC data-structures in the external memory. Depending on channel configurations of the Processing Units each data-stream traffic may contribute to the GSC processing time or neglected as a parallel data stream. The Traffic Model is represented in Table A.1-Table A.2.

***Table A.1**: Linking Phase memory words generated*

HL(i) #	Data Stream Description	4096^2	2048^2	1024^2	512^2	256^2
	Image Load $n=image\_regolution^2/16$	1048576	262144	65536	16384	4096
1	*Pixel Island Store (without parental pointers)* $n=(level\_width(HL_i)^2$	4198401	1050625	263169	66049	16641
	*Ovl.List for Secondary Lowest Level Store and Load** $n=level\_width(HL_{i+1})^2*12*2$	25215000	6316056	1585176	399384	101400
2	*Pixel Island Load, Modify and Store (parental pointers added)** $n=level\_width(HL_{i-1})^2*2$	8396802	2101250	526338	132098	33282
	Secondary Lowest Level Island Store (without parental pointers) $n=level\_width(HL_i)^2*2$	1050625	263169	66049	16641	4225
	Ovl.List Store and Load $n=level\_width(HL_{i+1})^2*12*2$	6316056	1585176	399384	101400	26136
3	*Secondary Lowest Level Island Parental Info Store* $n=level\_width(HL_{i-1})^2*2$	1050625	263169	66049	16641	4225
	Island Regions Store $n=level\_width(HL_i)^2*2$	526338	132098	33282	8450	2178
	Ovl.List Store and Load $n=level\_width(HL_{i+1})^2*12*2$	1585176	399384	101400	26136	6936
4	*Island Parents Store* $n=level\_width(HL_{i-1})^2$	263169	66049	16641	4225	1089
	Island Regions Store $n=level\_width(HL_i)^2*2$	132098	33282	8450	2178	578
	Ovl.List Store and Load	399384	101400	26136	6936	1944
5	*Island Parents Store*	66049	16641	4225	1089	289
	Island Regions Store	33282	8450	2178	578	162
	Ovl.List Store and Load	101400	26136	6936	1944	600
6	*Island Parents Store*	16641	4225	1089	289	81
	Island Regions Store	8450	2178	578	162	50
	Ovl.List Store and Load	26136	6936	1944	600	216
7	*Island Parents Store*	4225	1089	289	81	25
	Island Regions Store	2178	578	162	50	18
	Ovl.List Store and Load	6936	1944	600	216	96
8	*Island Parents Store*	1089	289	81	25	9
	Island Regions Store	578	162	50	18	8
	Ovl.List Store and Load	1944	600	216	96	24

HL(i) #	*Data Stream Description*	*4096^2*	*2048^2*	*1024^2*	*512^2*	*256^2*
9	*Island Parents Store*	289	81	25	9	4
	Island Regions Store	162	50	18	8	2
	Ovl.List Store and Load	600	216	96	24	
10	*Island Parents Store*	81	25	9	4	
	Island Regions Store	50	18	8	2	
	Ovl.List Store and Load	216	96	24		
11	*Island Parents Store*	25	9	4		
	Island Regions Store	18	8	2		
	Ovl.List Store and Load	96	24			
12	*Island Parents Store*	9	4			
	Island Regions Store	8	2			
	Ovl.List Store and Load	24				
13	*Island Parents Store*	4				
	Island Regions Store	2				
	** executed in Initial Linker approach*					
	Total Number of Mem.Wrd. Transfers					
	Coders as a stand-alone modules	50 452 742	12 643 563	3 176 144	801 717	204 314
	Initial Linker approach	16 840 940	4 226 257	1 064 630	270 235	69 632
	Times of Traffic Reduction	3	2,99	2,98	2,97	2,93
	Overhead	33 611 802	8 417 306	2 111 514	531 482	134 682

***Table A.2**: Result Generation Phase memory words generated*

HL(i) #	*Data Stream Description*	*4096^2*	*2048^2*	*1024^2*	*512^2*	*256^2*
1	*Label Image Store* $n=image\_resolution^2/4$	4194304	1048576	262144	65536	16384
	Pixel Regon Island Load $n=level\_width(HL_i)^2$	4198401	1050625	263169	66049	16641
	Secondary Lowest Level Label Island Load $n=level\_width(HL_{i+1})^2*2$	2101250	526338	132098	33282	8450
2	*Secondary Lowest Level Region/Label Island Load-Modify-Store* $n=level\_width(HL_i)^2*2*2$	4202500	1052676	264196	66564	16900
	Label Island Load $n=level\_width(HL_{i+1})^2*3$	789507	198147	49923	12675	3267
3	*Region/Label Island Load-Modify-Store* $n=level\_width(HL_i)^2*3*2$	1579014	396294	99846	25350	6534
	Label Island Load $n=level\_width(HL_{i+1})^2*3$	198147	49923	12675	3267	867
4	*Region/Label Island Load-Modify-Store*	396294	99846	25350	6534	1734
	Label Island Load	49923	12675	3267	867	243
5	*Region/Label Island Load-Modify-Store*	99846	25350	6534	1734	486
	Label Island Load	12675	3267	867	243	75
6	*Region/Label Island Load-Modify-Store*	25350	6534	1734	486	150
	Label Island Load	3267	867	243	75	27
7	*Region/Label Island Load-Modify-Store*	6534	1734	486	150	54
	Label Island Load	867	243	75	27	12
8	*Region/Label Island Load-Modify-Store*	1734	486	150	54	24
	Label Island Load	243	75	27	12	3
9	*Region/Label Island Load-Modify-Store*	486	150	54	24	6
	Label Island Load	75	27	12	3	0
10	*Region/Label Island Load-Modify-Store*	150	54	24	6	
	Label Island Load	27	12	3	0	
11	*Region/Label Island Load-Modify-Store*	54	24	6		
	Label Island Load	12	3	0		
12	*Region/Label Island Load-Modify-Store*	24	6			
	Label Island Load	3	0			
13	*Region/Label Island Load-Modify-Store*	6				
	Label Island Load	0				
	Number of Mem.Wrd. Transfers					
		17860693	4473932	1122883	282938	71857

A.4 HW-GSC profiling statistics

a) b)

c) d)

***Figure A.1**: Sample profiling images*
a) Sample A "Lena"; b) Sample B "Erika"; c) Sample C "Venedig"; d) Sample D "Bagdad"

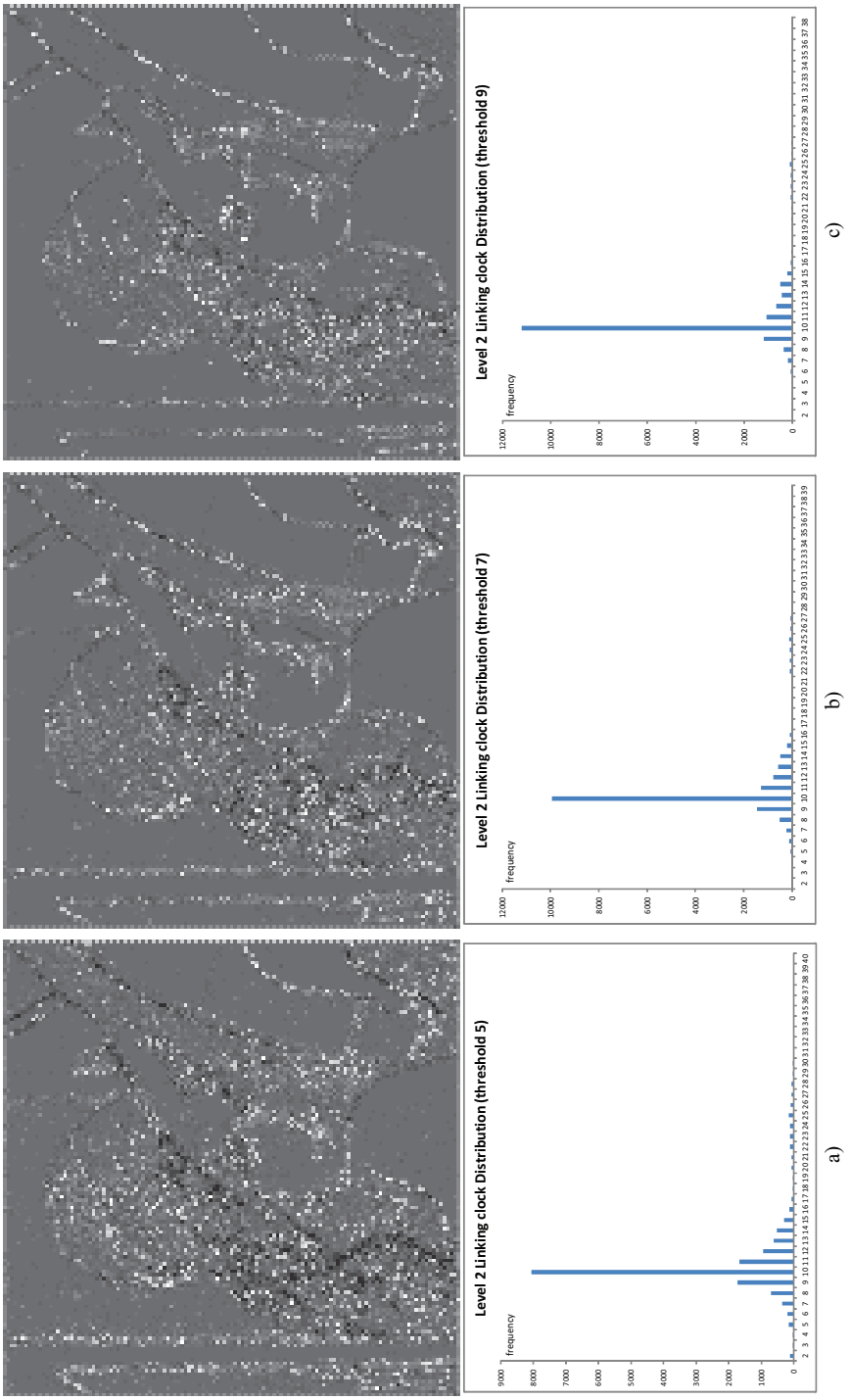
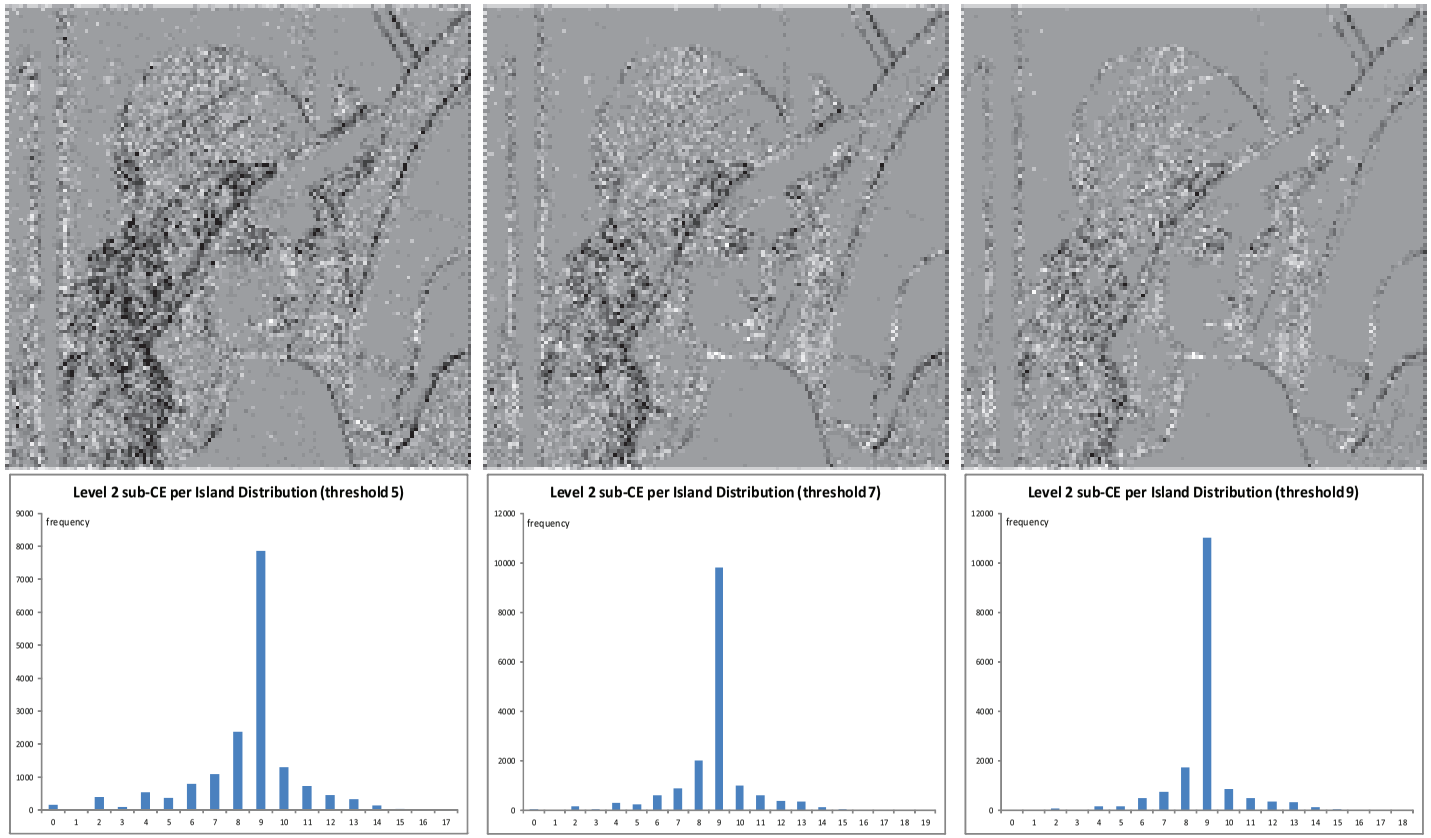
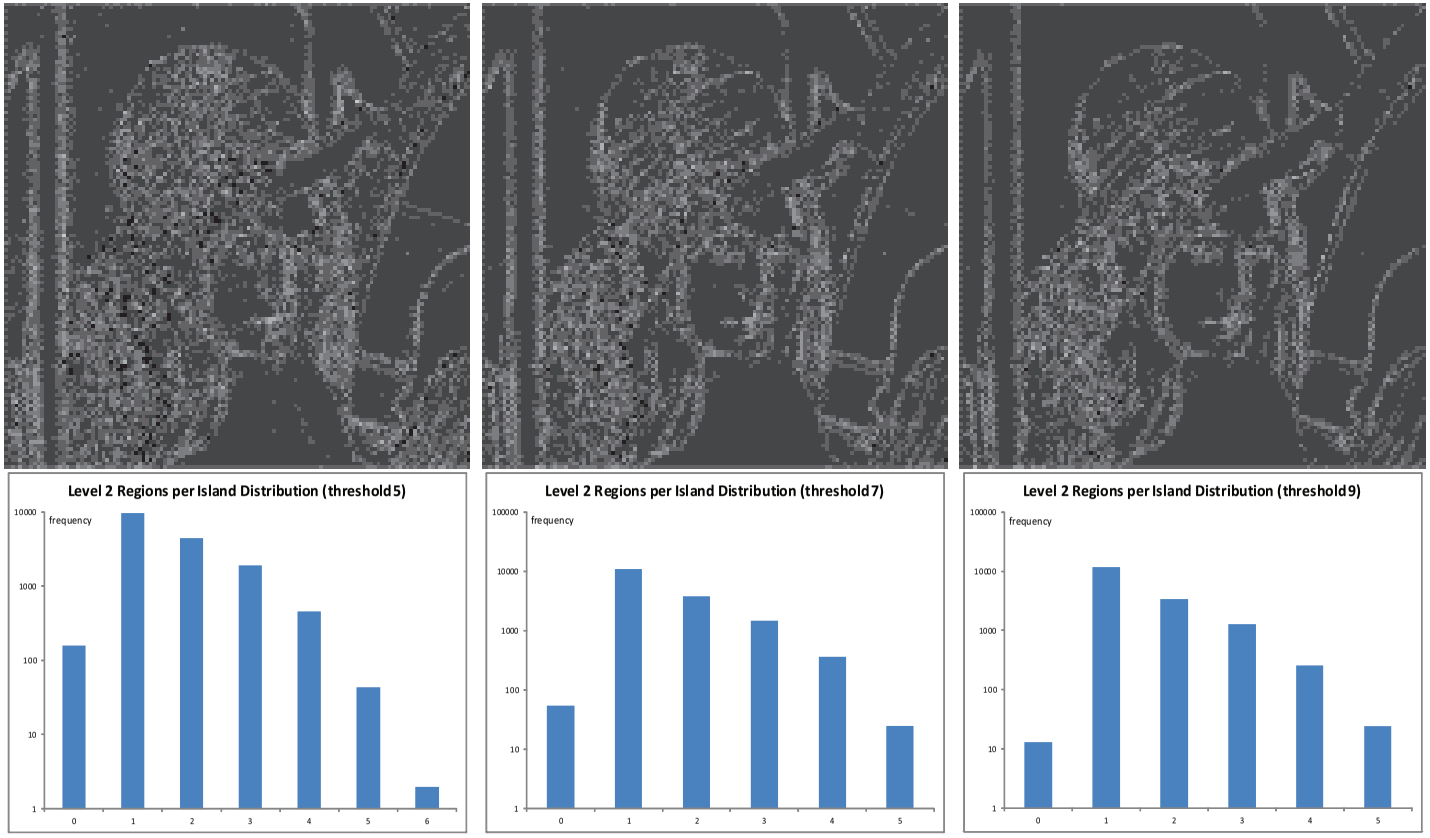


***Figure A.2**: Linking clocks per island distribution for different threshold values (Sample A, HL=2)*
(brighter areas of images correspond to higher values of measured parameters)



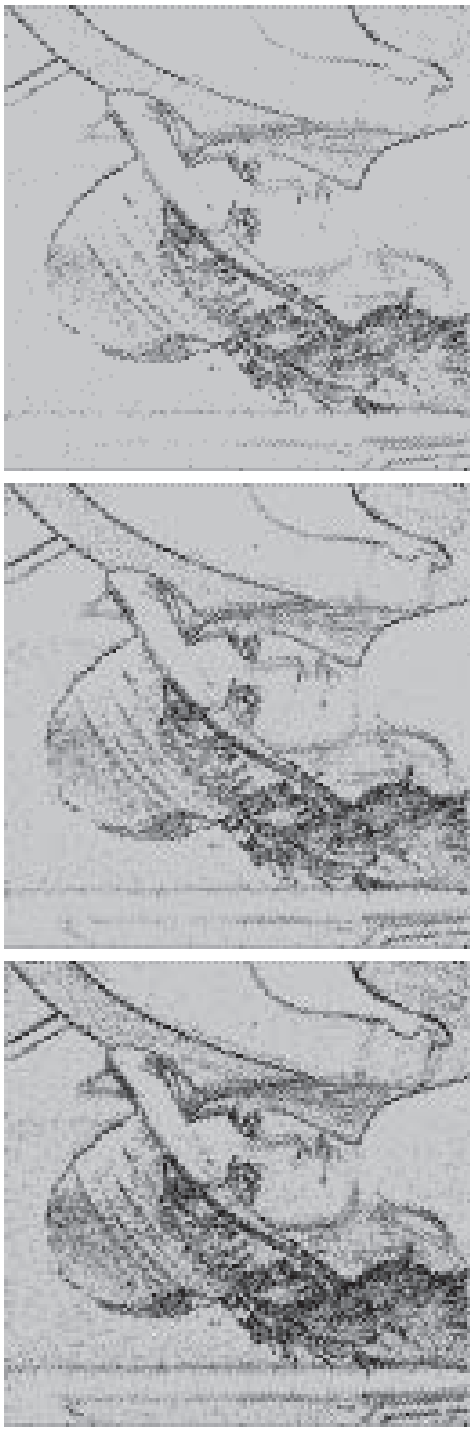
a) b) c)

***Figure A.3**: SubCEs per island distribution for different threshold values (Sample A, HL=2)*
(brighter areas of images correspond to higher values of measured parameters)



a) b) c)

Figure A.4*: Regions per island distribution for different threshold values (Sample A, HL=2)*
(brighter areas of images correspond to higher values of measured parameters)



a) b) c)

***Figure A.5**: Overlaps per island distribution for different threshold values (Sample A, HL=2, threshold 5, 7, 9 from left to right)*
(brighter areas of images correspond to higher values of measured parameters)

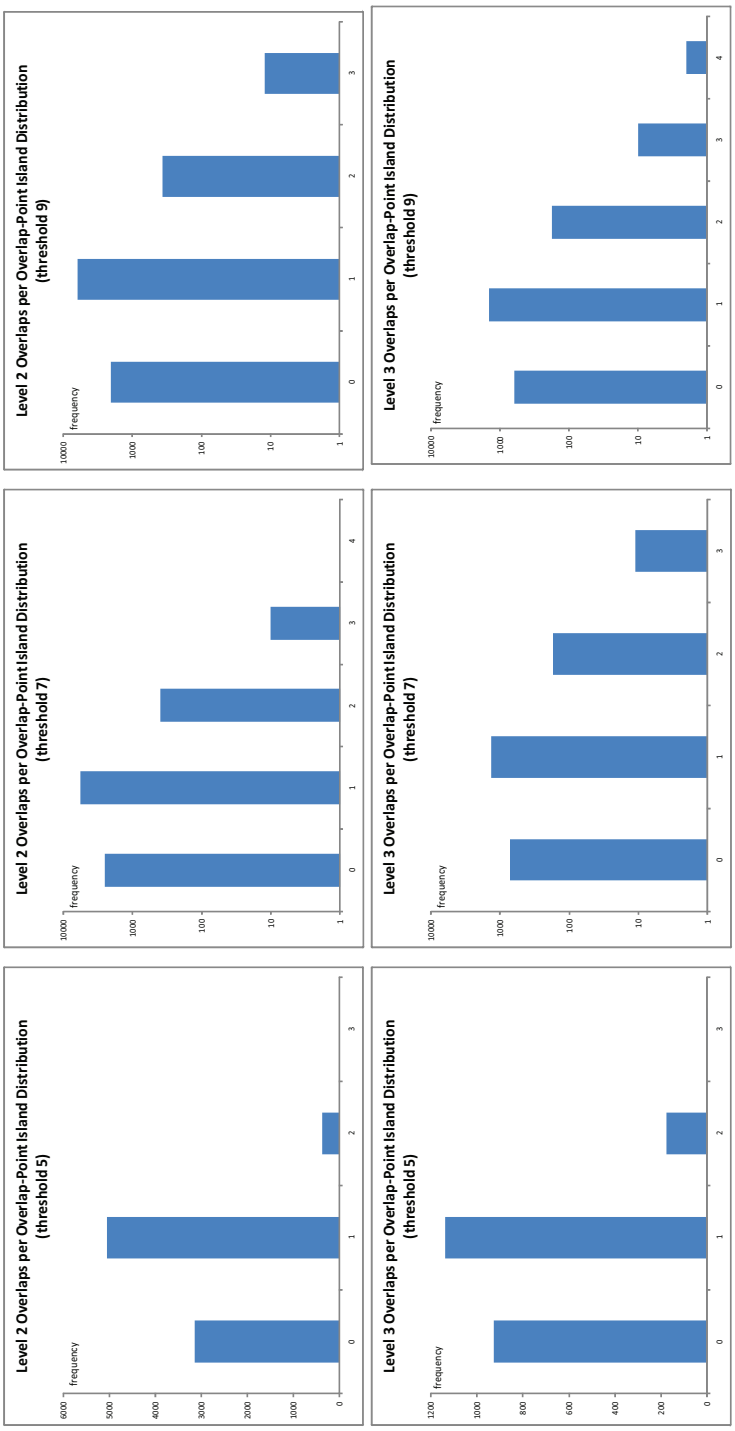
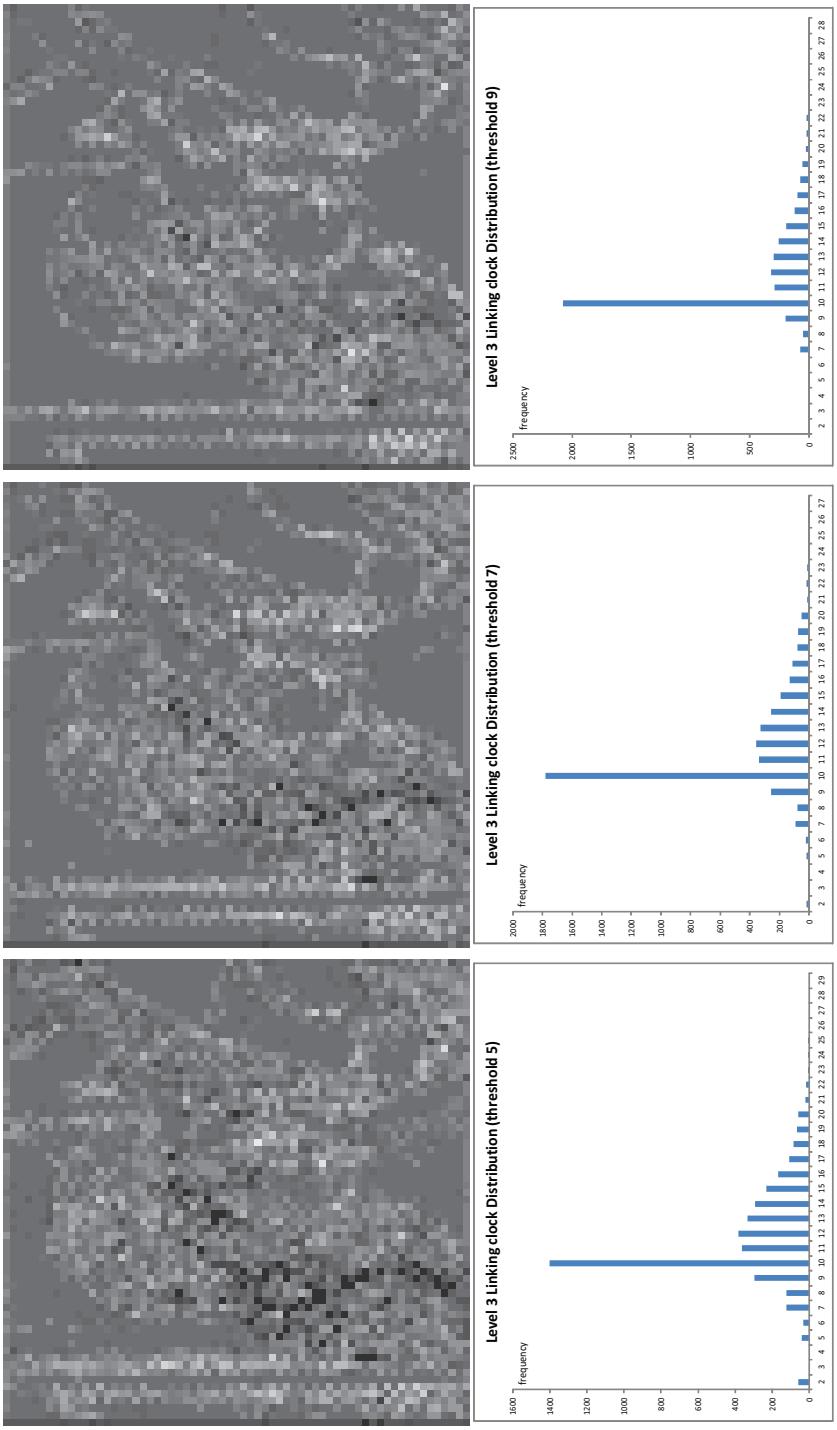
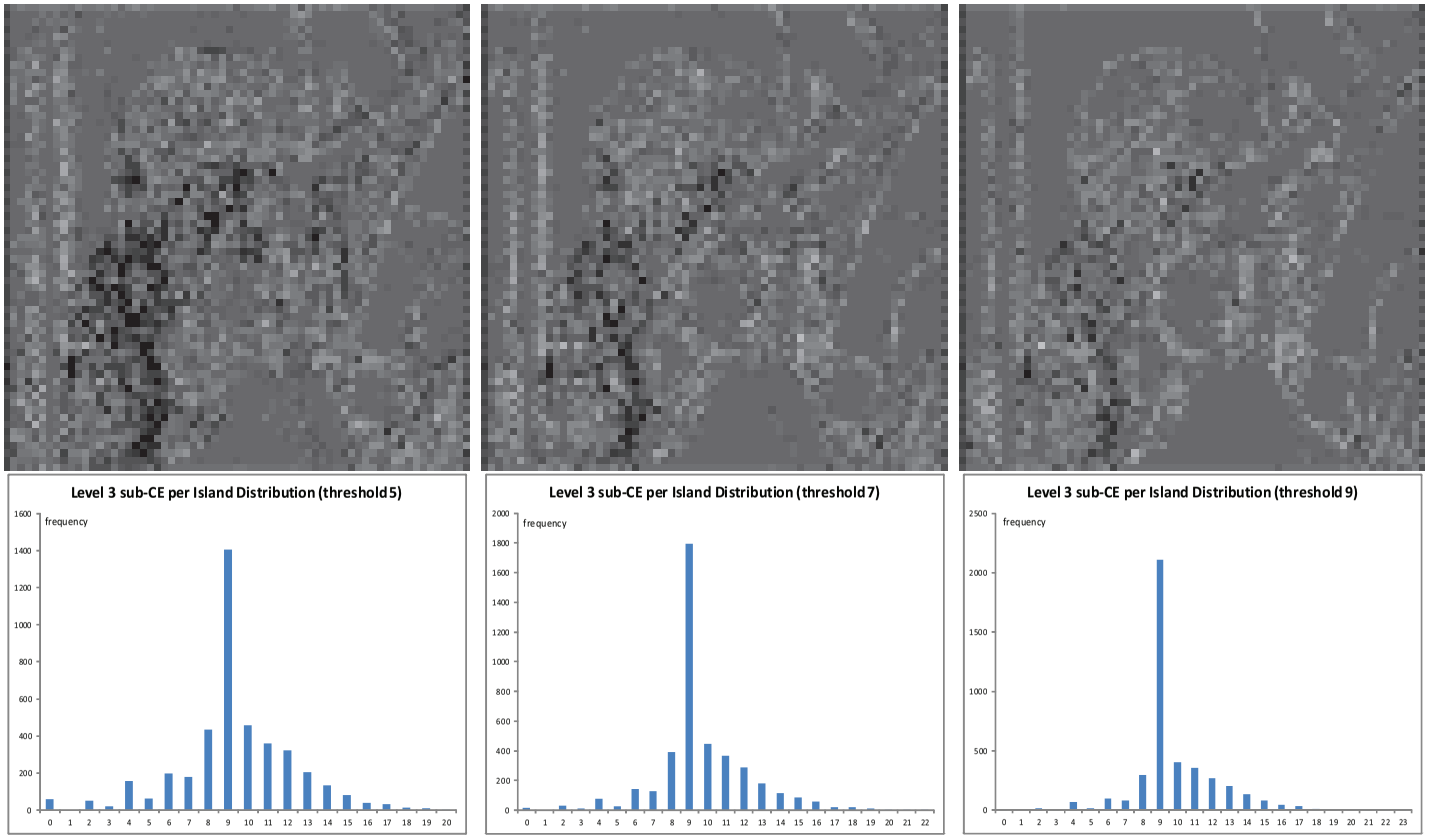


***Figure A.6**: Overlaps per overlap-point island histogram for different threshold values (Sample A, HL=2 and HL=3, threshold 5, 7, 9 from left to right)*



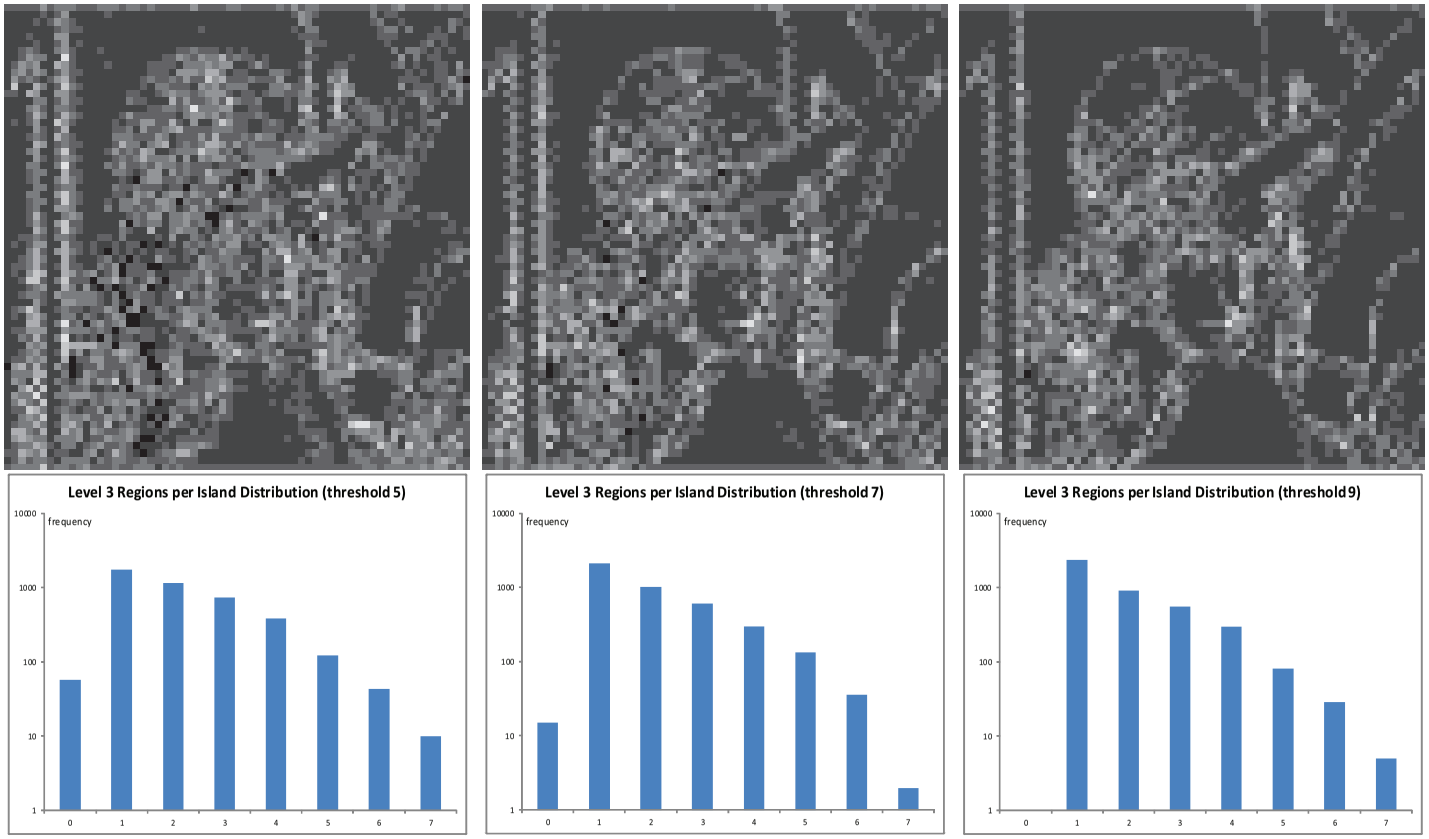
a) b) c)

***Figure A.7**: Linking clocks per island distribution for different threshold values (Sample A, HL = 3, threshold 5, 7, 9 from left to right)*
(brighter areas of images correspond to higher values of measured parameters)



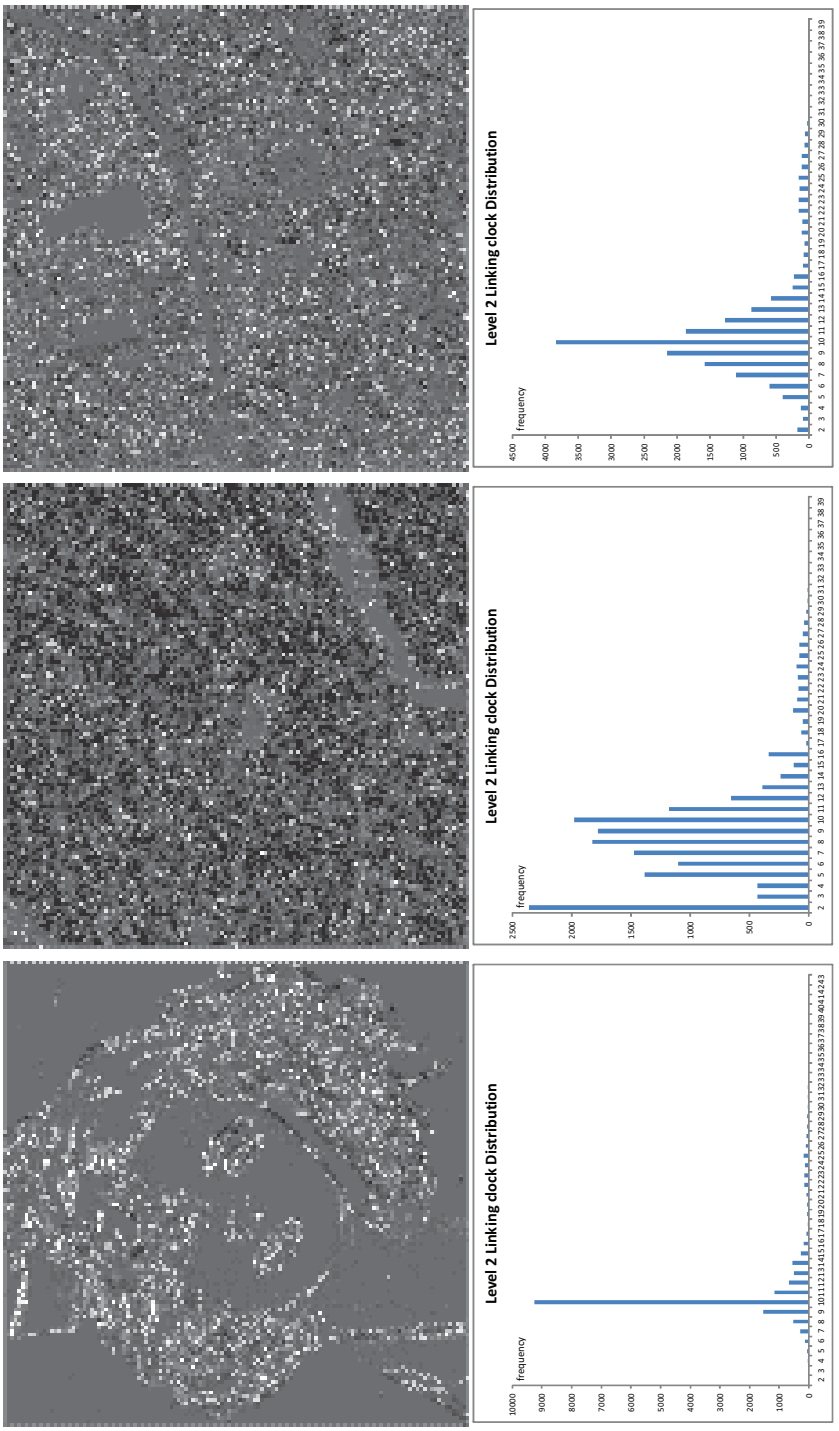
a) b) c)

Figure A.8*: SubCEs per island distribution for different threshold values (Sample A, HL=3, threshold 5, 7, 9 from left to right)*
(brighter areas of images correspond to higher values of measured parameters)



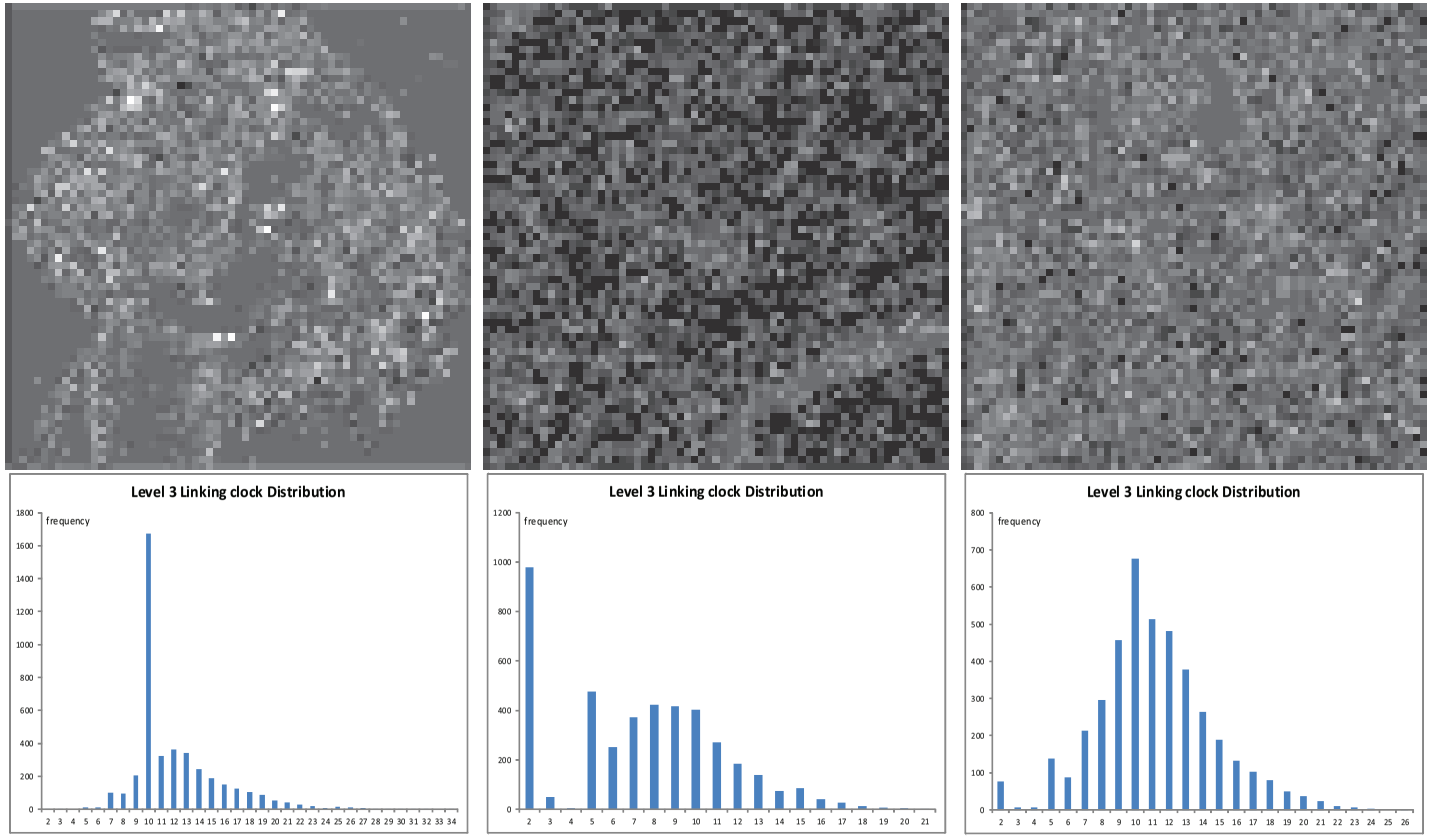
a) b) c)

Figure A.9*: Regions per island distribution for different threshold values (Sample A, HL=3, threshold 5, 7, 9 from left to right) (brighter areas of images correspond to higher values of measured parameters)*



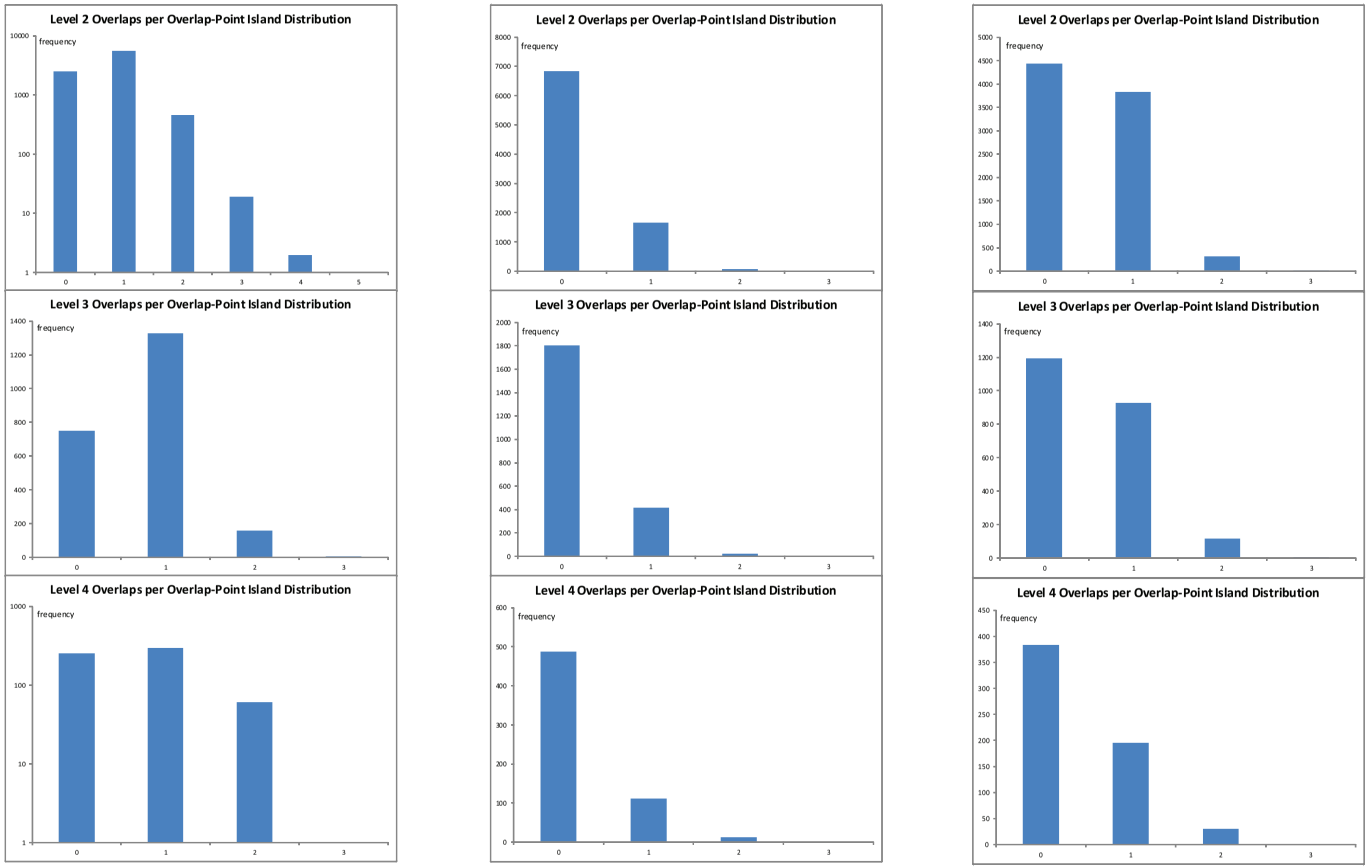
a) b) c)

***Figure A.10**: Linking clocks per island for different sample images (threshold 7, HL=2, Sample B – column a, Sample C – column b, Sample D – column c) (brighter areas of images correspond to higher values of measured parameters)*



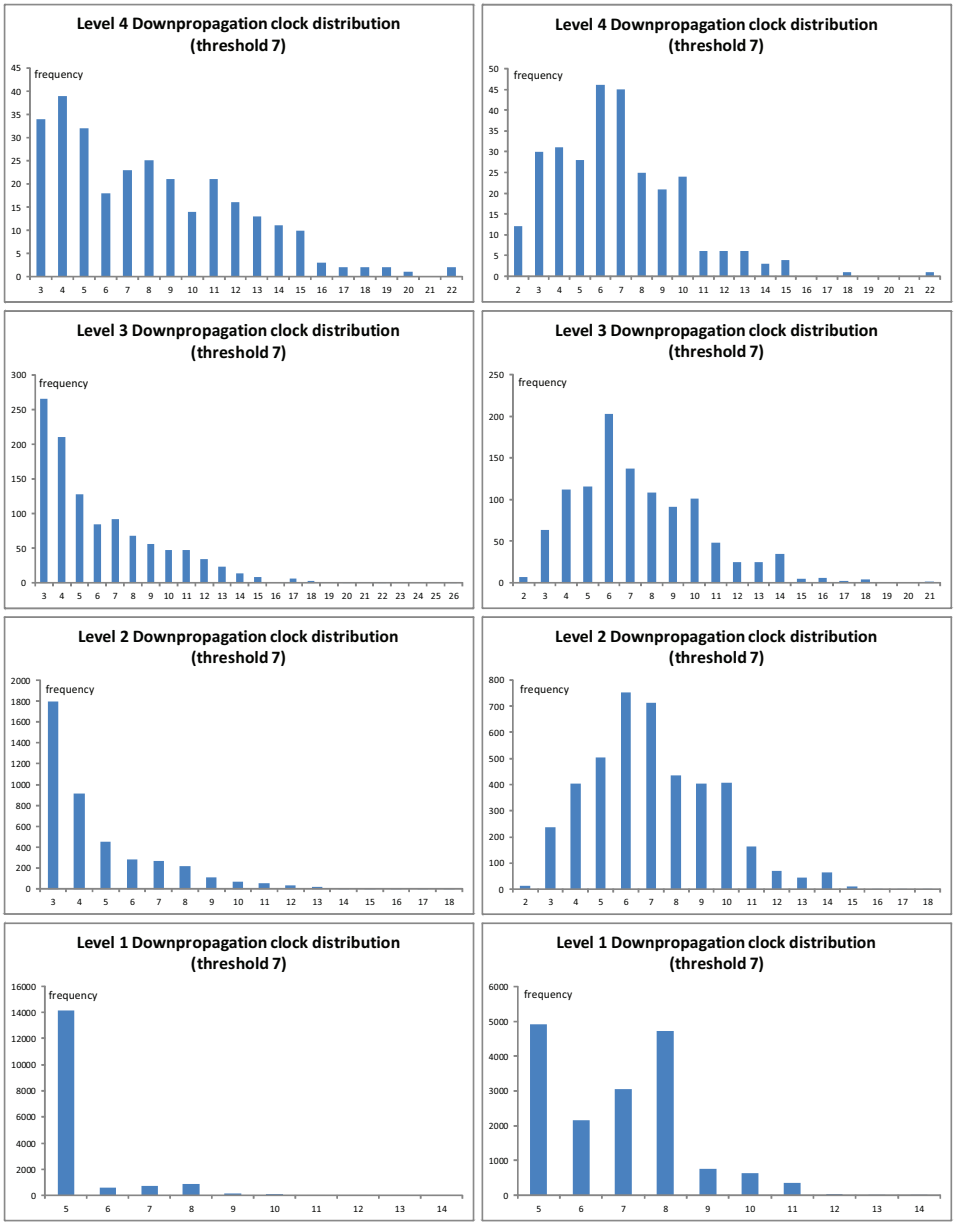
a) b) c)

***Figure A.11**: Linking clocks per island for different sample images (threshold 7, HL=3,Sample B – column a, Sample C – column b, Sample D – column c) (brighter areas of images correspond to higher values of measured parameters)*



a) b) c)

***Figure A.12**: Overlaps per overlap-point island histogram for different sample images (Sample B – column a, Sample C – column b, Sample D – column c)*



a) b)

***Figure A.13**: Downpropagation processing clocks per island distribution for different sample images (Sample A – column a, Sample C – column b)*

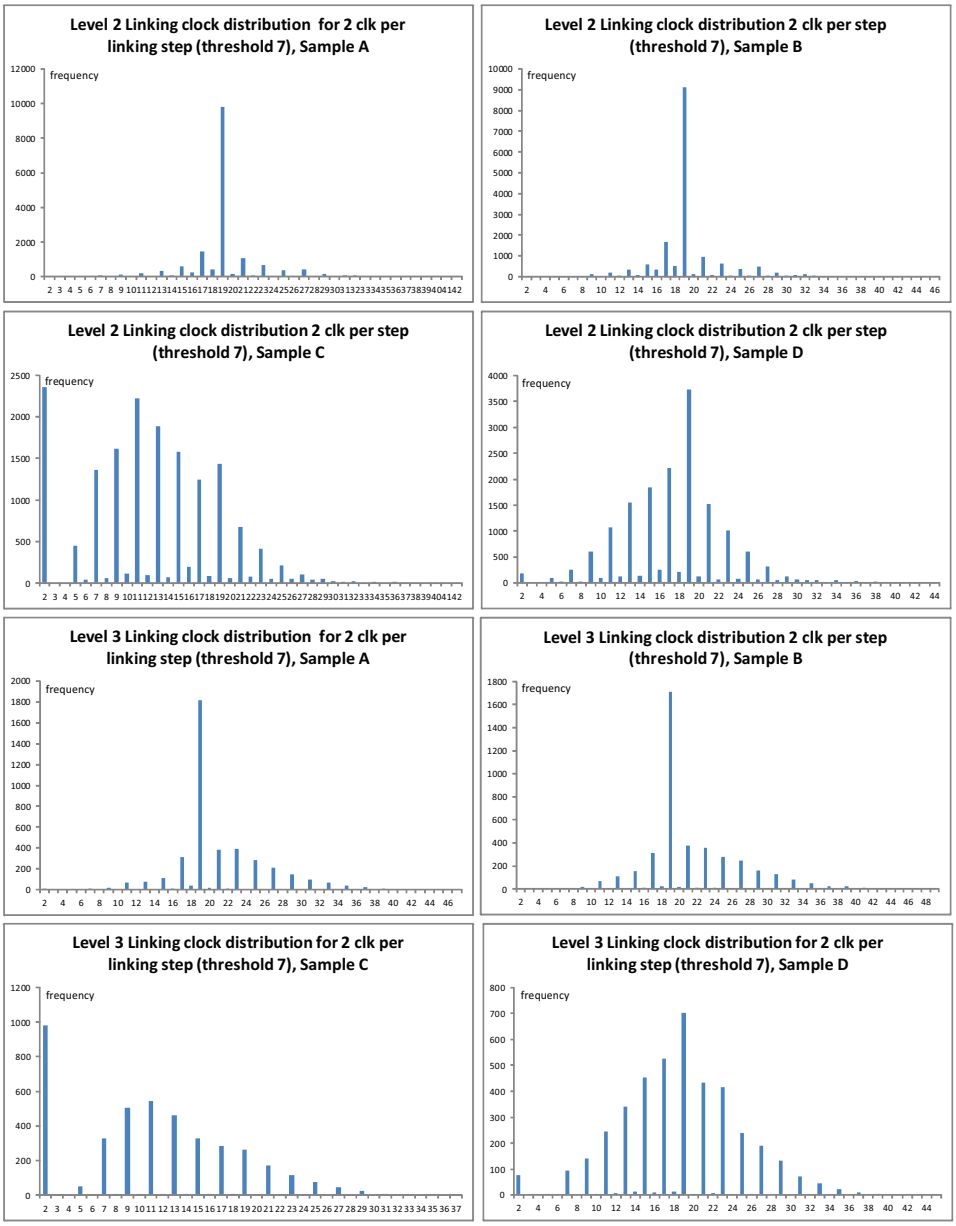


***Figure A.14**: Linking clocks per island distribution for 2clk per linking step implementation*

A.5 Hardware resource model

The *Hardware Resource Model* is based on counting of clocked memory (in bits) required for allocation of local data aggregates of the FPGA-GSC. Although it is not intended to give a precise flip-flop budget for each processing unit of the FPGA implementation, it can predict the order of complexity of each design module with high degree of certainty. Table A.3- Table A.12 show results of the Resource Model computed with the help of electronic tables for the FPGA-GSC for different numbers of rows in the island row blocks. Table A.13 offers the summary of synthesis for some configurations. As it can be seen the correlation between the model and synthesis results is high. Meanwhile, the Resource Model can hardly obtain the absolute accuracy in prediction as the results of synthesis depends on the optimisation algorithms, e.g. logic replication, typically exploited by modern synthesis tools for improving timing characteristics of synthesised designs.

***Table A.3**: FPGA GSC data-structure constants*

Constants	*Number of bits*
MEMORY_WORD_WIDTH	128
BIT_PER_FEATURE_PIXEL	8
BIT_PER_REGION_FEATURE	12
BIT_PER_LABEL_PIXEL	32
REGION_POSITION_VECTOR_WIDTH	7
CORE_REGION_FEATURE	10
PIXEL_PER_MEMORY_WORD	16
LOWEST_LEVEL_MAX_CE_NUMBER_PER_ISLAND	4
PRELOWEST_LEVEL_MAX_CE_NUMBER_PER_ISLAND	8
HIGHER_LEVEL_MAX_CE_NUMBER_PER_ISLAND	12
PARENT_POINTER_WIDTH	4
PARENT_POINTER_PAIR_WIDTH	8
LOWEST_LEVEL_CE_WIDTH	27
HIGHER_LEVEL_CE_WIDTH	20
LOWEST_LEVEL_REGION_WIDTH (CEF)	19
HIGHER_LEVEL_REGION_WIDTH (CEF)	12
LINKING_DIGIT_TRUNCATION	4
LOWEST_LEVEL_ISLAND_MEM_WORD_NUMBER	1
PRELOWEST_LEVEL_ISLAND_MEM_WORD_NUMBER	2
HIGHER_LEVEL_ISLAND_MEM_WORD_NUMBER	3
NUMBER_OF_ISLANDS_IN_MACROISLAND	7
NUMBER_OF_OVERLAP_POINTS_IN_ISLAND	6
NUMBER_OF_OVERLAP_POINTS_IN_MACROISLAND	12

Constants	*Number of bits*
ISLAND_INDEX_WIDTH	3
CE_INDEX_WIDTH	4
LOWEST_LEVEL_REGION_INDEX_WIDTH	2
OVERLAP_LIST_REGION_FEATURE_WIDTH	12
REGION_WEIGHT_FIELD_WIDTH	0
OVERLAP_LIST_REGION_WIDTH	16
OVERLAP_LIST_PAIR_WIDTH	32
OVERLAP_LIST_PAIR_NUMBER_IN_NODE	3
MAX_OVERLAP_LIST_PAIR_NUMBER_IN_MEM_WORD	4
OVERLAP_LIST_LENGTH	36
LOWEST_LEVEL_OVERLAP_LIST_LENGTH	12
LABEL_POINTER_WIDTH	24
HIGHER_LEVEL_LABEL_POINTER_WIDTH	20
LABELED_CE_WIDTH	36

***Table A.4**: FPGA GSC data-aggregate sizes*

Data structures	*Number of bits*
LOWEST_LEVEL_CE_ISLAND	108
PRELOWEST_LEVEL_CE_ISLAND	160
HIGHER_LEVEL_CE_ISLAND	240
LOWEST_LEVEL_REGION_ISLAND	76
PRELOWEST_LEVEL_REGION_ISLAND	96
HIGHER_LEVEL_REGION_ISLAND	144
LOWEST_LEVEL_PARENT_POINTER_ISLAND	32
PRELOWEST_LEVEL_PARENT_POINTER_ISLAND	64
HIGHER_LEVEL_PARENT_POINTER_ISLAND	96
LOWEST_LEVEL_OVERLAP_LIST_ISLAND	384
OVERLAP_LIST_ISLAND	1152
OVERLAP_LIST_NODE	96
LOWEST_LEVEL_LABEL_ISLAND	144
PRELOWEST_LEVEL_LABEL_ISLAND	288
HIGHER_LEVEL_LABEL_ISLAND	432
ISLAND_OF_PIXELS	56
BASIC_REGION_STRUCTURE	13
BASIC_REGION_ARRAY	52
EXPORT TABLE	66
MACRO_ISLAND_OVERLAP_STRUCTURE	1728
REGION_TRANSITION_TABLE	432
OVERLAP_POSITION_ENTITY	72
OVERLAP_LIST_ENTITY_IN_MIOS	24
OVERLAP_ISLAND	144
MACRO_ISLAND_OVERLAP_STRUCTURE_HL1	264
IMPORT_TABLE	66
REGION_DESCRIPTOR	11
PRELOWEST_LEVEL_REGION_ISLAND	160
LABELLED_ISLAND	384

***Table A.5**: Coding Unit flip-flop memory budget*

CODING UNIT			#	*bits*
Front End				
	Pixel-Prefetch Memory Word Buffer			
		words per pixel island	2	256
	Pixel-Island Extraction Memory Word Area			
		words per pixel island	2	256
Number of Islands			7	3584
		overhead for buffers		256
Procesing Core				
	Processor:			
	Pixel Island Fetch		1	56
	Basic Regions		4	52
	Basic Region Linkage Vector		4	16
	Core Region Partial Sums		16	192
	Core Region Position Vector		4	28
	Lowest Level Region Island		1	76
	Export Table		1	66
	Miscellaneous		1	30
	Shared Storage:			
	Coded Island Array			
		region islands per pixel island	1	76
	Parent Index Pair Array			
		parent pointer islands per pixel island	6	24
Number of Islands			7	4312
*Back End**				
	Region Island Output Buffer			
		region islands per pixel island	1	76
	Overlap-list Node Output Buffer			
		overlap-list nodes per pixel island	3	96
Number of Islands			6	1032
**-not included if coding unit is used as an add-on*				
Number of coding processors	7			
Total				8152

***Table A.6**: General Linking Unit flip-flop memory budget*

GENERAL LINKING			#	*bits*
Front-End				
	Overlap-list Island Prefetch Buffer			
		islands per linking island	1	1152
Number of Islands			3	3456
Processing Core				
	Processor:			
	Macroisland Overlap Structure		1	1728
	Current Overlap Island		1	144
	Transition Region Index Candidates		18	72
	Transition Region Feature Candidates		18	216
	New root fegures		18	216
	Linked region macroisland		1	144
	Miscellaneous			
	Transition Flags		72	72
	Current Island Transition Table		6	18
	threshold register		2	26
	FSM related		1	51
	Shared Storage:			
	Linked Island Array			
		region islands per linking island	1	144
	Parent-Pointer Island Array			
		parent-pointer islands per linking island	7	672
Number of Islands			3	10509
Back-End				
	Region Island Output Buffer			
		region islands per linking island	2	288
	Parent-pointer Island Output Buffer			
		parent-pointer islands per linking island	4	384
	Overlap-list Node Output Buffer			
		overlap-list node per linking islad	3	288
Number of Islands			2	1920
Number of island rows	3			
Total				15885

***Table A.7**: Initial Linking Unit flip-flop memory budget*

INITIAL LINKING			#	*bits*
Front End				
	Overlap-list Island Prefetch Buffer			
		islands per linking island	1	384
Number of Islands			3	1152
Processing Core				
	Processor:			
	Macroisland Overlap Structure HL(1)		1	264
	Region Vectors of Coded Islands in Macroisland		7	168
	Linked Region Macroisland		1	160
	Current Overlap Island		1	66
	Current Region Vectors of Coded Island		1	28
	Miscellaneous:			
	thershold register		2	26
	FSM related		1	64
	Shared Storage:			
	Linked Island Array			
		region islands per linking island	1	76
	Parent-Pointer Island Array			
		parent-pointer islands per linking island	7	224
Number of Islands			3	3228
Back End				
	Region Island Output Buffer			
		region islands per linking island	2	192
	Parent-pointer Island Output Buffer			
		parent-pointer islands per linking island	4	128
	Overlap-list Node Output Buffer			
		overlap-list node per linking island	3	288
Number of Islands			2	1216
Number of island rows	3			
Total				5596

***Table A.8**: Downpropagation Unit flip-flop memory budget*

DOWNPROPAGATION			#	*bits*
Front End				
	Label Island Prefetch Buffer			
		islands per labeled island	1	432
	Unlabelled subisland Prefetch Buffer			
		islands per labeled island	4	960
Number of Islands			2	2784
		overhead for label island buffer		432
Processing Core				
	Processor:			
	Child CE Fetch		12	324
	Parental Features and Labels		8	288
	Child Labelled Island		4	1536
	Miscellaneous:			
	Pipeline Auxiliary		4	164
	Shared Storage:			
	Parent Island Source Zone			
		labelled islands per label island	3	1296
	Subisland Labelling Zone			
		labelled subsland per labelled island	6	2592
	Region Sub-island Source Zone			
		region subislands per labelled island	6	1440
	Label Image Mapping Zone			
		memory words per labelled island	8	1024
Number of Islands			2	17328
		overhead for parent source buffer		1296
Back End				
	Labelled Subisland Output Buffer			
		labelled subisland per labelled island	4	1728
	Labelled Image Output Buffer			
		memory words per labelled island	4	512
Number of Islands			2	4480
Number of island rows	3			
Total				26320

***Table A.9**: Coder to Linker integration resources*

CODER TO LINKER INTEGRATION MODULE			#	*bits*
Intermediate Buffer				
	Parent Index Piar Bridge Buffer			
		parent pair per linking processor	12	72
	Coded Island Bridge Buffer			
		coded island per linking processor	6	456
	Export Tables		6	330
Number of island rows			7	3696
Linking Unit Intrusion				
	Coded Island Temporal Sorage			
		coded islands per linking processor	8	608
	Coded Island Output Buffer			
		coded region islands per linking processor	4	304
Number of island rows			4	3648
Number of island rows	7			
Total				7344

***Table A.10**: Overlap Node Processor flip-flop memory budget*

OVERLAP NODE PROCESSOR			#	*bits*
Processing Core				
	Processor:	computed for three overlap points per island row		
	Parental Indexes		6	24
	Parental Features		6	72
	Parental Position Vectors		6	42
	Local thresholds		6	78
	Features Distanses		6	72
	Region Weights		9	36
	Child Island Local		0	0
	Parent Islands Local		0	0
Number of island rows			2	648
*Back End***				
	**included in correspondent linking module			
Number of Islands			2	0
Number of island rows	2			
Total				648

***Table A.11**: Resource Model summary (in bits and as a percentage of XCV2P100 flip-flop memory)*

rows in block	*Coding Unit*		*Lowest Level Linking Unit*		*General Linking Unit*		*Overlap Node Processor*	
2	2684	2.94%	3228	3.54%	9778	10.71%	324	0.35%
3	3984	4.36%	5296	5.80%	15717	17.21%	648	0.71%
4	5284	5.79%	7364	8.06%	21656	23.72%	972	1.06%
5	6584	7.21%	9432	10.33%	27595	30.22%	1296	1.42%

***Table A.12**: Resource Model summary (continued)*

rows in block	*Coded Islands*	*Coder Add-on*	*Extended Integration*	*Initial Integration*	*Initial Linking Unit*		*Extended Linking Unit*		*Down-propagation Unit*	
2	5	5396	4464	2562	11510	12.61%	19962	21.86%	14024	15.36%
3	7	7452	7344	4134	17530	19.20%	31161	34.13%	26320	28.82%
4	9	9508	10224	5706	23550	25.79%	42360	46.39%	38616	42.29%
5	11	11564	13104	7278	29570	32.38%	53559	58.65%	50912	55.76%

***Table A.13**: Synthesis* results for XCV2P100*

rows in block	*General Linking Unit DFFs*		*Initial Linking Unit DFFs*		*Downpropagation Unit ` DFFs*	
2	14531	15.91%	15888	17.40%	17146	18.78%
3	21360	23.39%	19031	20.84%	28880	31.63%
4	30813	33.74%	26737	29.28%	44323	48.54%
5	35933	39.35%	30852	33.79%	54198	59.35%

* MentorGraphics Precision 2010, Place and Route with Xilinx ISE 10.1

A.6 Labelling Processor cache efficiency

The cache scheme of the Labelling Processor in the separate segment-feature storing scheme (the higher resolution image processing) has been profiled to estimate the efficiency of the caching mechanism for different cache sizes. Table A.14-Table A.18 below show the profiling information gathered for the last three hierarchical levels for the four sample images.

It is notable that due to spatial locality of the data the cache scheme achieve relatively high effectiveness with rather small cache sizes and shows a negligible improvement with increase of the number of data positions in the caches after a certain level. It is also important to note that the L2 cache plays an important role in the cache scheme and is especially important for highly segmented or fractal images.

***Table A.14**: Cache efficiency for 24-position L1 and 64-position L2 scheme*

L1(24),L2(64)	*Sample A*	*Sample B*	*Sample C*	*Sample D*	*Total*
Segment key read attempt	50184	51150	23637	44860	169831
Cache Hit	35861	35518	11854	25380	108613
Cache Efficiency	0.71	0.69	0.50	0.57	0,64

***Table A.15**: Cache efficiency for 24-position L1 and without L2 scheme*

L1(24),L2(0)	*Sample A*	*Sample B*	*Sample C*	*Sample D*	*Total*
Segment key read attempt	50184	51150	23637	44860	169831
Cache Hit	27215	27142	6720	15886	76963
Cache Efficiency	0.54	0.53	0.28	0.35	0.45

***Table A.16**: Cache efficiency for 24-position L1 and 128-position L2 scheme*

L1(24),L2(128)	*Sample A*	*Sample B*	*Sample C*	*Sample D*	*Total*
Segment key read attempt	50184	51150	23637	44860	169831
Cache Hit	37197	36196	12145	26264	111802
Cache Efficiency	0.74	0.71	0.51	0.59	0.66

***Table A.17**: Cache efficiency for 48-position L1 and 64-position L2 scheme*

L1(48),L2(64)	*Sample A*	*Sample B*	*Sample C*	*Sample D*	*Total*
Segment key read attempt	50184	51150	23637	44860	169831
Cache Hit	36349	35793	11993	25668	109376
Cache Efficiency	0.72	0.70	0.51	0.57	0.64

***Table A.18**: Relative efficiency of different caching scheme implementations*

Efficiency Gain	*Sample A*	*Sample B*	*Sample C*	*Sample D*	*Total*
L2(0)vs.L2(64)	31.8%	30.9%	76.4%	59.8%	41.1%
L2(64)vs.L2(128)	3.7%	1.9%	2.5%	3.5%	2.9%
L1(24)vs.L1(48)	1.4%	0.8%	1.2%	1.1%	1.1%

A.7 Interface descriptions

A.7.1 ZBT controller core interface

Memory transaction is initialised under the following conditions: at the rising edge of the *clock* signal (*clk*) *vld_n* is asserted LOW, a valid address of the memory transaction is sampled at the address bus *addr*, *rw* signal is determining the direction for the memory transaction (*rw* = HIGH : read, *rw*=LOW : write), *brst_n* signal specifies burst mode memory transaction.

The transaction initialisation is ignored under any of the following conditions: reset signal *reset_n* set LOW, *stand_by* asserted HIGH, or *vld_n* asserted HIGH, or user busy input *ubusy* asserted HIGH.

Memory transaction is completed if the following is true: sampled at the rising edge of *clk*, data out ready signal *dout_rdy_n* is set low (for a read operation) one clock before the data bus *data_out* contains valid data, and the transaction is not invalidated at the next clock edge. If data in ready (*din_rdy_n*) signal is set low one clock before valid data appear at the data bus *data_in*, and the transaction is not invalidated at the next clock edge.

Data transference phase is considered being invalid if the following is true: reset signal *reset_n* set LOW, *stand_by* asserted HIGH, or user busy input *ubusy* asserted HIGH.

User busy input *ubusy* is used to suspend a data transference phase (e.g. when asserted HIGH at a read data transference phase the data at *data_out* bus are frozen till *ubusy* is deasserted).

Burst mode is evoked at a valid transaction initialisation phase if *brst_n* is asserted LOW. At each valid subsequent transaction initialisation phase transaction address on the bus *addr* is ignored and the memory operation is initialised for the subsequent memory location. SRAM controller exits a burst mode when *brst_n* is deselected at a valid transaction initialisation phase.

The signal interface is summarised in Table A.19.

***Table A.19**: ZBT SRAM Controller interface*

Signal	*Direc -tion*	*Act- ive*	*Description*
SRAM interface			
A [addr_bus_width-1:0]	O	-	Address used to select one of the addressable locations. Sampled at the rising edge of the clock.
WE_n	O	low	Write Enable. Sampled at the rising edge of clock if CEN_n is active. This signal must be asserted low to initiate a write sequence.
ADV_LD	O	-	Advance/Load used to advance the internal SRAM-chip address counter or load a new address. As the burst is restricted to the length of four memory position, this signal is always set low and the controller core address counter is used in the burst mode.

Signal	*Direc -tion*	*Act- ive*	*Description*
CE_n	O	low	Chip Enable. Sampled on the rising edge of clock. Used to select/deselect the SRAM module. Set high during initialisation period of at least 1 ms.
OE_n	O	low	Output Enable. Combined with the synchronous logic block inside the device to control the direction of the I/O pins. When low, the I/O pins of RAM module are allowed to behave as outputs. When deasserted high, I/O pins are three-stated, and act as input data pins. OE_n is masked during the data portion of a write sequence, during the first clock when emerging from a deselected state and when the SRAM chip has been deselected. Presently is always asserted active. If the board-level time model shows data-bus contention, the control of this pin should be reviewed.
CEN_n	O	low	Clock Enable. When asserted low the clock signal is recognised by the SRAM. When deasserted high the clock signal is masked.
DQ[word_width-1:0]	B	-	Bidirectional Data I/O lines. As inputs to ZBT SRAM, they feed into an on-chip data register, which is triggered by the rising edge of clock. As outputs, they deliver the data contained in the memory location specified. The direction of the pins is controlled by OE.
MODE	O	-	Mode. Selects the burst order of the device. Tied HIGH selects the interleaved burst order. Pulled LOW selects the linear burst order. This signal is always asserted low to select linear burst mode, for the interleaved bursts are architecture specific option.
ZZ	O	high	ZZ “sleep” signal. This signal places the device in a nontime critical “sleep” condition with data integrity preserved. Sleep mode is currently configured to get in after 4000 idle bus cycles.
			User interface
clk	I	pos. edge	Clock input of the core. All signals are sampled at the rising edge of clk.
reset_n	I	low	Reset. This signal resets the internal logic of the core. This signal is asynchronous to clk.
brst_n	I	low	Burst mode.
rw	I	-	Read/write. This signal selects the direction of the memory transaction when vld_n is active. Asserted high means reading data from a memory location specified by addr bus. Setting low – writing to a memory location indexed by address bus.
vld_n	I	low	Transaction valid. Sampled low at the rising edge of clk specifies a valid memory transaction with the direction specified by rw signal to/from the memory location specified by addr bus or by internal burst address counter if in a burst mode.
addr [addr_bus_width-1:0]	I	-	When sampled at the rising edge of clk with vld_n active specifies the memory cell location of the memory transaction. Ignored if the core is in a burst mode.
data_in [word_width-1:0]	I	-	Data input is sampled at the rising edge of clk if din_rdy_n is active for write transaction.
data_out [word_width-1:0]	O	-	Data output contains valid data of a read transaction at the rising edge of clk if din_rdy_n is active.

Signal	*Direc -tion*	*Act- ive*	*Description*
din_rdy_n	O	low	Data In Ready. When asserted low indicates that the data of the current write transaction are accepted from data_in bus at the next rising clock edge.
dout_rdy_n	O	low	Data In Ready. When asserted low indicates that the data of the current read transaction are valid at data_out bus at the next rising clock edge.
ubusy	I	high	User busy. Asserted high shows that the user is busy and can not perform bus activity. Thus all bus operations are ignored by the SRAM controller core. The data busses are frozen and maintain the valid data of current transaction till ubusy is deselected.
perr	O	high	Parity error. When set high shows that there was a data corruption and the parity of data transferred three clocks before was not fulfilled. Parity check is performed as a complementary "one" to odd number of "ones" for each eight bits of data bus. (DQ[8], DQ[17], DQ[26], DQ[35], DQ[44], DQ[53], DQ[62], DQ[71], DQ[80], DQ[89], DQ[98], DQ[107], DQ[116], DQ[125], DQ[134], DQ[143])
stand_by	O	high	Stand by mode. This signal when set high is showing that the SRAM controller is not active and all bus activity are ignored.

A.7.2 Switch Matrix interface

Table A.20*: Switch Matrix active port interface*

Signal name	*Direc -tion*	*Description*
ADDRESS BUNDLE		
ADDRESS[bus_width:0]	IN	Address of a memory access operation
BURST_NUMBER[2:0]	IN	Number of words in the burst. Specially coded: 100 – 3 words; 110 – 2 words; 111 – 1 word. Other combinations will lead to unpredictable results and need for restart (sampled together with ADDRESS)
READ_WRITE	IN	Direction of the memory access operation: HIGH(1) – read; LOW(0) - write (sampled together with ADDRESS)
Address bundle control		
ADDR_write_op_req	IN	Request signal for operation. Active HIGH(1). Together with active HIGH ADDR_write_op_gnt indicates that operation was admitted for execution.
ADDR_write_op_gnt	OUT	Grant signal for operation. Active HIGH(1). Together with active HIGH ADDR_write_op_req indicates that operation was admitted for execution.
DATA TO MEMORY CHIP (outward FPGA)		
DATA_OUT_data_write [data_bus_width:0]	IN	Data for writing to the memory at corresponding position indicated by valid ADDRESS
DATA_OUT_write_op_req	IN	Request signal for data write. Active HIGH(1). Together with active HIGH DATA_OUT_write_op_gnt indicates that data were taken from DATA_OUT_data_write.
DATA_OUT_write_op_gnt	OUT	Grant signal for data write. Active HIGH(1). Together with active HIGH DATA_OUT_write_op_req indicates that data were taken from DATA_OUT_data_write.

Signal name	*Direc -tion*	*Description*
DATA FROM MEMORY CHIP (inward FPGA)		
DATA_IN_data_read [data_bus_width:0]	OUT	Data for reading to the memory at corresponding position indicated by valid ADDRESS
DATA_IN_read_op_req	IN	Request signal for data read. Active HIGH(1). Together with active HIGH DATA_IN_read_op_gnt indicates that data were taken from DATA_IN_data_read.
DATA_IN_read_op_gnt	OUT	Grant signal for data read. Active HIGH(1). Together with active HIGH DATA_IN_read_op_req indicates that data were taken from DATA_IN_data_read.

Bibliography

[1] H. Sutter, "The Free Lunch Is Over. A Fundamental Turn Toward Concurrency in Software", *Dr. Dobb's Journal, vol. 30, no. 3*, pp. 202-210, 2005.
Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[2] J.-F. Vogelbruch, "Segmentation of Volume Data Sets Using Three-Dimensional Hierarchical Island Structures," Ph.D. dissertation, Dept. Electron. and Inform. Techn., RWTH Aachen, *Schriften des Forschungszentrum Jülich, Reihe Informationstechnik, Band 3*, Jülich, 2002, in German. ISBN 3893363092

[3] J.-F. Vogelbruch, J. Leinen, P.H. Dillinger et al., "BMBF-project 3D-RETISEG. Optimized 2D/3D real-time segmentation and classification using hierarchical island structures," *Technische Informationsbibliothek (TIB) Hannover*, Hannover, Final Report, 2007, in German.
Available: http://edok01.tib.uni-hannover.de/edoks/e01fb08/558081827.pdf

[4] K.S. Fu and K.K. Mui, "A survey on image segmentation", *Pattern Recognition, vol. 13, no. 1*, pp. 3-16, 1981.

[5] R.M. Haralick and L.G. Shapiro, "Image segmentation techniques", *Computer Vision, Graphics, and Image Processing, vol. 29, no. 1*, pp. 100-132, 1985.

[6] N.R. Pal and S.K. Pal, "A review on image segmentation techniques", *Pattern Recognition, vol. 26, no. 9*, pp. 1277-1294, 1993.

[7] L.P. Clarke, R.P Velthuizen, M.A. Camacho et al., "MRI segmentation: methods and applications", *Magnetic Resonance Imaging, vol. 13, no. 3*, pp. 343-368, 1995.

[8] R. Jain, R. Kasturi, and B.G. Schunck, *Machine Vision*, 1st ed., McGraw Hill International Editions, 1995. ISBN 978-0070320185

[9] T. McInerney and D. Terzopoulos, "Deformable models in medical images analysis: a survey", *Medical Image Analysis, vol. 1, no. 2*, pp. 91-108, 1996.

[10] A. Sing, D. Goldgof, and D. Terzopoulos, Eds., *Deformable Models in Medical Image Analysis*, IEEE Computer Society Press, Los Alamitos, CA, 1998.

[11] S.U. Lee, S.Y. Chung, and R.-H. Park, "A comparative performance study of several global thresholding techniques for egmentation", *Computer Vision, Graphics and Image Processing, vol. 52, no. 2*, pp. 171-190, 1990.

[12] P.K. Sahoo, S. Soltani, and K.C. Wong, "A survey of thresholding techniques", *Computer Vision, Graphics and Image Processing, vol. 41, no. 2*, pp. 233-360, 1988.

[13] L.S. Davis, "A survey of edge detection techniques", *Computer Graphics and Image Processing, vol. 4,* pp. 248-270, 1975.

[14] D. Ziou and S. Tabbone, "Edge detection techniques – An overview", Universite de Sherbrooke, Sherbrooke, Technical Report No. 195, 1997.

[15] M. Kelly, "Edge detection by computer using planning", *Machine Intelligence, Edinburgh University Press, vol. 6,* pp. 397-409, 1971.

[16] Y.P. Chien and K.S. Fu, "Processing and feature extraction of picture pattern", Purdue University, West Lafayette, Indiana, Technical Report, EE 74-20, 1974.

[17] D. Marr and E. Hildreth, "Theory of edge detection", in *Proc. of Royal Society of London. Series B.*, pp. 187-217, London, 1980.

[18] Y. Sato, S. Nakajima, N. Shiraga et al., "Three-dimensional multi-scale line filter for segmentation and visualisation of curvilinear structures in medical images", *Medical Image Analysis, vol. 2, no. 2*, pp. 143-168, 1998.
Available: http://perso.telecom-paristech.fr/~bloch/P6/PRREC/linefilter.pdf

[19] B.W. Reutter, G.J. Klein, and R.H. Huesman, "Automated 3-D segmentation of respiratory-gated PET transmission images", *IEEE Trans. on Nuclear Science, vol. 44, no. 6*, pp. 2473-2476, 1997.

[20] R. Hayasaka, J. Zhao, and Y. Matsushita, "Outstanding Object-Oriented Color Image Segmentation Using Fuzzy Logic", in *Proc. SPIE'97 Multimedia Storage and Archiving Systems II*, pp. 303-314, Dallas, TX, 1997.

[21] A. Moghaddamzadeh and N. Bourbakis, "A Fuzzy Region Growing Approach for Segmentation of Color Images", *Pattern Recognition, vol. 30, no. 6*, pp. 867-881, 1997.

[22] W. Skarbek and A. Koschan, "Color Image Segmentation – A Survey", Technische Universität Berlin, Berlin, Technical Report, TR 94-32, 1994.

[23] L. Vincent and P. Soille, "Watersheds in digital spaces – an efficient algorithm based on immersion simulations", *IEEE Trans. Pattern Anal. Mach. Intell., vol. 13, no. 6*, pp. 583-597, 1991.

[24] J.-P. Thiran, V. Warscotte, and B. Macq, "A queue-based region growing algorithm for accurate segmentation of multi-dimensional digital images", *Signal Processing, vol. 60, no. 1*, pp. 1-10, 1997.

[25] Y.-L. Chang and X. Li, "Adaptive Image Region-Growing", *IEEE Trans. on Image Processing, vol. 3, no. 6*, pp. 868-872, 1994.

[26] J. Sijbers, M. Verhoye, P. Scheunders et al., "Watershed-based segmentation of 3D MR data for volume quantization", *Magnetic Resonance Imaging, vol. 15, no. 6*, pp. 679-688, 1997.
Available: http://webhost.ua.ac.be/visielab/papers/sijbers/mri97.pdf

[27] R. Adams and L. Bischof, "Seeded Region Growing", *IEEE Trans. on Image Processing, vol. 16, no. 6*, pp. 641-647, 1994.

[28] S.A. Hojjatoleslami and J. Kittler, "Region growing: a new approach", *IEEE Trans. on Image Processing, vol. 7, no. 7*, pp. 1079-1084, 1998.

[29] A. Siebert, "Dynamic Region Growing", *Vision Interface,* 1997.
Available: http://www.cipprs.org/papers/VI/VI1997/pp001-007-Siebert-1997.pdf

[30] S.L. Horowitz and T. Pavlidis, "Picture segmentation by a tree traversal algorithm", *Journal of the ACM, vol. 23, no. 2*, pp. 368-388, 1976.

[31] K.C. Strasters and J.J. Gerbrands, "Three-dimensional image segmentation using a split, merge and group approach", *Pattern Recognition Letters, vol. 12, no. 5*, pp. 307-325, 1991.

[32] R. Marfil, L. Molina-Tanco, A. Bandera et al., "Pyramid segmentation algorithms revisited", *Pattern Recognition, vol. 39, no. 8*, pp. 1430-1451, 2006.
Available: http://nichol.as/papers/Marfil/Pyramid%20segmentation%20algorithms%20revisited.pdf

[33] P.J. Burt, "The pyramid as a structure for efficient computation", *Multiresolution Image Processing and Analysis,* pp. 6-35, Springer-Verlag, 1984.

[34] C. Xu, D.L. Pham, and J.L. Prince, "Medical image segmentation using deformable models", *Handbook on Medical Imaging — Volume 2: Medical Image Analysis,* pp. 129-174, SPIE Press, 2000.

[35] M. Kaas, A. Witkin, and D. Terzopoulos, "Snakes: Active Contour Models", *International Journal of Computer Vision, vol. 1,* pp. 321-331, 1987.
Available: http://www.cs.ucla.edu/~dt/papers/ijcv88/ijcv88.pdf

[36] D. Terzopoulos, A. Witkin, and M. Kass, "Constraints on deformable models: recovering 3D shape and nonrigid motion", *Artificial Intelligence, vol. 36, no. 1*, pp. 91-123, 1988.

[37] Z. Zhang, M. Braun, and P. Abbott, "A new deformable model for 3D image segmentation", in *Proc. of the 9th International Conference on Image Analysis and Processing*, pp. 239-246, Florence, 1997.

[38] R. Malladi, R. Kimmel, D. Adalsteinsson et al., "A geometric approach to segmentation and analysis of 3D medical images", in *Proc. of IEEE/SIAM workshop on Biomedical Image Analysis*, pp. 244-252, San Francisco, CA, 1996.

[39] H. Tek and B.B. Kimia, "Volumetric segmentation of medical images by three-dimensional bubbles", *Computer Vision and Image Understanding, vol. 65, no. 2*, pp. 246-258, 1996.

[40] J.M. Gauch, H. Pien, and J. Shah, "Hybrid deformable models for three-dimensional biomedical image segmentation", in *IEEE Nuclear Science Symposium and Medical Imaging Conference*, pp. 1935-1939, Norfolk, VA, 1994.

[41] J. Montagnat, H. Delingette, N. Scapel et al., "Surface simplex meshes for 3D medical image segmentation", in *Proc. of the IEEE International Conference On Robotics & Automation*, pp. 864-870, 2000.

[42] L.D. Cohen and I. Cohen, "Finite-element methods for active contour models and balloons for 2-D and 3-D images", *IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. 15, no. 11*, pp. 1131-1147, 1993.

[43] S. Osher and J. Sethian, "Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi Equations", *Journal of Computer Physics, vol. 79, no. 1*, pp. 12-49, 1988.

[44] Y. Shimshoni, "Introduction to Classification Methods", School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv, 1995.

[45] B.J.A. Kröse and P.P. van der Smagt, *An introduction to neural networks*, 8th ed., The University of Amsterdam, Amsterdam, 1996.

[46] P. Besl and R. Jain, "Segmentation through variable-order surface fitting", *IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. 10,* pp. 167-192, 1988.

[47] T. Pavlidis and Y. Liow, "Integrating region growing and edge detection", *IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. 12,* pp. 225-233, 1990.

[48] L.D. Griffin, A.C.F. Colchester, and G.P. Robinson, "Scale and segmentation of grey-level images using maximum gradient paths", *Image and Vision Computing, vol. 10, no. 6*, pp. 389-402, 1992.

[49] K. Haris, S.N. Efstratiadis, N. Maglaveras et al., "Hybrid Image Segmentation Using Watersheds and Fast Region Merging", *IEEE Trans. on Image Processing, vol. 7,no. 12,* pp. 1684-1699, 1998.
Available: http://ivpl.eecs.northwestern.edu/files/IEEETransIP98c.pdf

[50] V. Rehrmann, "Robust real-time color image processing", Koblenzer Schriften zur Informatik, Koblenz, 1994. ISBN 3923532504

[51] V. Rehrmann and L. Priese, "Fast and Robust Segmentation of Natural Color Scenes", in *Proc. 3rd Asian Conference on Computer Vision,* pp. 598-606, Springer Verlag, Hongkong, 1997.
Available: http://www.uni-koblenz.de/~lb/publications/Rehrmann1998FAR.pdf

[52] R.H. Leckie, "Equipment and Materials: Funding the Future", *INFRASTUCTURE Advisor,* 2005.
Available: http://www.semi.org/cms/groups/public/documents/web_content/p036611.pdf

[53] M. LaPedus, "Industry agrees on first 450-mm wafer standard", *EE Times*[Online]. 2008.
Available: http://www.eetimes.com/showArticle.jhtml?articleID=211300360

[54] K. Flamm, "Economic Impacts of International R&D Coordination: SEMATECH and the International Technology Roadmap", in *Century Innovation Systems for Japan and the United States: Lessons from a Decade of Change: Report of a Symposium, National Research Council*, pp.108-125, 2009.
Available: http://www.nap.edu/catalog.php?record_id=12194

[55] *AltiVec Technology Programming Interface Manual*, Freescale Semiconductor Technical Information Centre, 1999.
Available: http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

[56] *Intel SSE4 Programming Reference*, Intel Corp., 2007.
Available: http://software.intel.com/file/17971/

[57] *Cortex-A9 NEON Media Processing Engine*, ARM Inc., Technical Reference Manual, 2009.
Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0409e/DDI0409E_cortex_a9_neon_mpe_r2p0_trm.pdf

[58] *Intel's Advanced Encryption Standard (AES) Instructions Set*, Intel Mobility Group, Israel Development Center, White paper, 2009.
Available: http://software.intel.com/file/20457

[59] *TMS320C6000 CPU and Instruction Set Reference Guide*, Texas Instruments Inc., 2000.
Available: http://focus.ti.com/general/docs/litabsmultiplefilelist.tsp?literatureNumber=spru189g

[60] M.S. Schlansker and B.R. Rau, "EPIC: An Architecture for Instruction-Level Parallel Processors", Compiler and Architecture Research HP Laboratories, 2000.
Available: www.hpl.hp.com/techreports/1999/HPL-1999-111.pdf

[61] *TilePro64 Processor*, Tilera Corp., Product Brief, 2009.
Available: http://www.tilera.com/pdf/PB019_TILEPro64_Processor_A_v3.pdf

[62] T.R. Halfhill, "Ambric's New Parallel Processor. Globally Asynchronous Architecture Eases Parallel Programming", *Reed Electronics Group*[Online]. Oct., 2006.
Available: http://www.ambric.info/pdf/MPR_Ambric_Article_10-06_204101.pdf

[63] *OMAP5910 Dual-Core Processor*, Texas Instruments Inc., Data manual, 2004.
Available: http://focus.ti.com/general/docs/litabsmultiplefilelist.tsp?literatureNumber=sprs197d

[64] D.T. Wang, "ISSCC 2005: The CELL Microprocessor", *Real World Technologies*[Online]. 2005.
Available: http://www.realworldtech.com/page.cfm?ArticleID=RWT021005084318&p=1

[65] *NVIDIA GeForce 8800 GPU Architecture Overview*, nVidia Corp., Technical brief, 2006.
Available: http://www.nvidia.com/attach/941771?type=support&primitive=0

[66] *AMD R-600 Instruction Set Architecture*, Advanced Micro Devices Inc., 2008.
Available: http://developer.amd.com/gpu_assets/r600isa.pdf

[67] A. Aizcorbe, S.D. Oliner, and D.E. Sichel, "Shifting Trends in Semiconductor Prices and the Paceof Technological Progress", *Finance and Economics Discussion SeriesDivisions of Research & Statistics and Monetary Affairs Federal Reserve Board*[Online]. 2006.
Available: http://www.federalreserve.gov/pubs/feds/2006/200644/200644pap.pdf

[68] "ASIC Cost Effectiveness," in *ASIC Outlook 1998. An Application Specific IC Report and Directory*, Integrated Circuit Engineering Corporation, 1998, sec. 5. ISBN 1877750638
Available: http://smithsonianchips.si.edu/ice/cd/ASIC98/TITLE.PDF

[69] E. Ugrjumov, *Digital Circuitry Design*, BVH, St.Petersburg, 2004, in Russian. ISBN 5820601009

[70] *XC9500 In-System ProgrammableCPLD Family*, Xilinx Inc., Product Specification, 2007.
Available: http://www.xilinx.com/support/documentation/data_sheets/DS063.pdf

[71] C. Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*, Newnes, 2004. ISBN 978-0750676045

[72] M.J.S. Smith, *Application-Specific Integrated Circuits*, Addison Wesley Professional, 1997. ISBN 978-0201500226

[73] *QuickLogic PolarPro II Device Data Sheet*, QuickLogic Corp., 2010.
Available: http://www.quicklogic.com/assets/pdf/QLPOLARPROIIDEVICEDSREVC.pdf

[74] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays", *Proc. IEEE, vol. 81, no. 7*, pp.1013-1029, 1993.
Available: http://www.eecg.toronto.edu/~jayar/pubs/rose/PIEEE93a.pdf

[75] S.D. Brown, R. Francis, Z. Vranesic et al., *Field-Programmable Gate Arrays*, 1st ed., Kluwer Academic Publishers, Dordrecht, 1992. ISBN 978-0792392484

[76] *FLEX 10K Embedded Programmable Logic Family Data Sheet*, Altera Corp., 2003.
Available: http://www.altera.com/literature/ds/dsf10k.pdf

[77] *MAX II Device Family Data Sheet*, Altera Corp., 2008.
Available: http://www.altera.com/literature/hb/max2/max2_mii5v1_01.pdf

[78] A. Medvedev, "The Present of Hardware Acceleration of Graphics", *iXBT.com*[Online]. 2004, in Russian.
Available: http://www.ixbt.com/video2/dx-current.shtml

[79] K. Fatahalian and M. Houston, "A Closer Look at GPUs", *Communications of AMC, vol. 51, no. 10*, pp. 50-57, 2008.
Available: http://graphics.stanford.edu/~kayvonf/papers/fatahalianCACM.pdf

[80] J.D. Owens, M. Houston, D. Luebke et al., "GPU Computing", *Proc. IEEE, vol. 96, no. 5*, pp. 879-899, 2008.
Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=04490127

[81] A. Medvedev and A. Berillo, "Supplemental information about the family of graphics cards RADEON R[V]4XX", *iXBT.com*[Online]. 2007, in Russian.
Available: http://www.ixbt.com/video2/spravka-r4xx.shtml

[82] A. Medvedev and A. Berillo, "Supplemental information about the family of graphics cards NV4X", *iXBT.com*[Online]. 2007, in Russian.
Available: http://www.ixbt.com/video2/spravka-nv4x.shtml

[83] A. Medvedev and A. Berillo, "Supplemental information about the family of graphics cards G7X", *iXBT.com*[Online]. 2007, in Russian.
Available: http://www.ixbt.com/video2/spravka-g7x.shtml

[84] A. Medvedev and A. Berillo, "Supplemental information about the family of graphics cards RADEON R[V]5XX", *iXBT.com*[Online]. 2007, in Russian.
Available: http://www.ixbt.com/video2/spravka-r5xx.shtml

[85] A. Berillo, "Supplemental information about the family of graphics cards RADEON R[V]6XX", *iXBT.com*[Online]. 2008, in Russian.
Available: http://www.ixbt.com/video2/spravka-r6xx.shtml

[86] A. Berillo, "Supplemental information about the family of graphics cards RADEON R[V]7XX", *iXBT.com*[Online]. 2009, in Russian.
Available: http://www.ixbt.com/video2/spravka-r7xx.shtml

[87] A. Berillo, "Supplemental information about the family of graphics cards RADEON R[V]8XX", *iXBT.com*[Online]. 2010, in Russian.
Available: http://www.ixbt.com/video2/spravka-r8xx.shtml

[88] A. Medvedev and A. Berillo, "Supplemental information about the family of graphics cards G8X/G9X", *iXBT.com*[Online]. 2009, in Russian.
Available: http://www.ixbt.com/video2/spravka-g8x.shtml

[89] A. Berillo, "Supplemental information about the family of graphics cards GT2XX", *iXBT.com*[Online]. 2009, in Russian.
Available: http://www.ixbt.com/video2/spravka-gt2xx.shtml

[90] A. Berillo, "Supplemental information about the family of graphics cards FG1XX", *iXBT.com*[Online]. 2010, in Russian.
Available: http://www.ixbt.com/video2/spravka-gf1xx.shtml

[91] D. Kanter, "NVIDIA's GT200: Inside a Parallel Processor", *Real World Technologies*[Online]. 2008.
Available: http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=1

[92] D. Kanter, "Inside Fermi: Nvidia's HPC Push", *Real World Technologies*[Online]. 2009. Available: http://www.realworldtech.com/page.cfm?ArticleID=RWT093009110932&p=4

[93] L. Dymchenko, "Parallel Computation Processors NVIDIA: the Present and the Future", *World of NVIDIA*[Online]. 2009, in Russian. Available: http://nvworld.ru/articles/cuda_parallel/

[94] L. Dymchenko, "Pentium4 from AMD", *World of NVIDIA*[Online]. 2010, in Russian. Available: http://nvworld.ru/articles/amd-radeon-evergreen/

[95] *AMD: Unleashing the Power of Parallel Compute With Commodity ATI Radeon(TM) 5800 Series GPU!* presented at SIGGRAPH Asia 2009, Advanced Micro Devices Inc.[Online]. 2009. Available: http://sa09.idav.ucdavis.edu/docs/SA09_AMD_IHV.pdf

[96] J.D. Owens, D. Luebke, N. Govindaraju et al., "A Survey of General-Purpose Computationon Graphics Hardware", *Computer Graphics Forum, vol. 26, no. 1*, pp. 80-113, 2007.

[97] *The OpenCL Specification*, Khronos OpenCL Working Group, v 1.1 rev. 44, June, 2011. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[98] *nVidia CUDA Programming Guide*, nVidia Corp., User Guide v3.2, 2010. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[99] U.J. Kapasi, S. Rixner, W.J. Dally et al., "Programmable Stream Processors", *IEEE Computer Society, vol. 36, no. 8*, pp. 54-62, 2003. Available: http://cva.stanford.edu/publications/2003/ieeecomputer_stream.pdf

[100] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-On-A-Chip Designs*, 3rd ed., Springer, 2002. ISBN 978-0387740980

[101] M. Keating, D. Flynn, R. Aitken, Alan Gibbons et al., *Low Power Methodology Manual: For System-on-Chip Design*, 2nd ed., Springer, 2007. ISBN 978-0387718187

[102] L. Scheffer, L. Lavagno, and G. Martin, Eds., *EDA for IC System Design, Verification, and Testing (Electronic Design Automation for Integrated Circuits Handbook)*, CRC Press, 2006. ISBN 978-0849379239

[103] S. Pasricha and N. Dutt, *On-Chip Communication Architectures: System on Chip Interconnect (Systems on Silicon)*, Morgan Kaufmann, 2008. ISBN 978-0123738929

[104] V. Nemudrov and G. Martin, *System-on-Chip. Design and Development*, Technosphera, 2004. ISBN 5948360296, in Russian

[105] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, 2nd ed., Kluwer Academic Publishers, 2003. ISBN 1402074018

[106] *VSIA System Level Design Model Taxonomy*, VSI Reference Document Version 2, 2001.

[107] B. Bailey, G. Martin, and T. Anderson, Eds., *Taxonomies for the Development and Verification of Digital Systems*, 1st ed., Springer, 2005. ISBN 978-0387240190

[108] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*, Morgan Kaufmann, 2007. ISBN 978-0123735515

[109] A. Piziali, *Functional Verification Coverage Measurement and Analysis*, Springer, 2004. ISBN 978-1402080258

[110] J. Bhadra, M.S. Abadir, L.-C. Wang et al., "A Survey of Functional Verification through Hybrid Techniques", *IEEE Des. Test Comput., vol. 24, no. 2*, pp. 112-122, 2007.

[111] R. Drechsler, Ed., *Advanced Formal Verification*, Kluwer Academic Publishers, 2004, ISBN 1402077211

[112] M. Ganai and A. Gupta, *SAT-Based Scalable Formal Verification Solutions*, 1st ed., Springer, 2007. ISBN 978-0387691664

[113] W.K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches*, Prentice Hall, 2005. ISBN 978-0131433472

[114] T. Kropf, *Introduction to Formal Hardware Verification*, Springer, 1999. ISBN 978-3540654452

[115] F. Wang, "Formal Verification of Timed Systems: A Survey and Perspective", *Proc. IEEE, vol. 92, no. 8*, pp. 1283-1305, 2004.

[116] *OVM User Guide*, Cadence Design Systems and Mentor Graphics Corp., 2009.

[117] *VMM Golden Reference Guide*, Doulos Ldt., 2010. ISBN 0954734572

[118] *Universal Verification Methodology(UVM) 1.0 EA User's Guide*, Accellera System Initiative, 2010.

[119] K. Golshan, *Physical Design Essentials: An ASIC Design Implementation Perspective*, 1st ed., Springer, 2007. ISBN 978-0387366425

[120] T. Tang and X. Zhou, "A Dynamic Timing Delay For Accurate Gate-Level Circuit Simulation", in *Proc. of the 39th Midwest Symposium on Circuits and Systems*, pp. 325-327, 1996.
Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.25.8071&rep=rep1&type=pdf

[121] *Standard Delay Format Specification in PDF format, version 3.0*, Open Verilog International, 1995.
Available: http://www.eda.org/sdf/sdf_3.0.pdf

[122] C. Xavier and S.S. Iyengar, *Introduction to parallel algorithms*, Wiley-Interscience, 1998. ISBN 978-0471251828

[123] R. Paige, J.H. Reif, and R. Wachter, Eds., *Parallel Algorithm Derivation and Program Transformation*, 1st ed., Springer, 1993. ISBN 978-0792393627

[124] V.P. Gergel and R.G. Strongin, *Fundamentals of Parallel Computations for Multiprocessor Systems*, Nizhegorodskiy State University, N. Novgorod, 2003. ISBN 5857466024, in Russian

[125] S.H. Roosta, *Parallel Processing and Parallel Algorithms: Theory and Computation*, Springer, 1999. ISBN 978-0387987163

[126] *NVIDIA CUDA Best Practice Guide*, nVidia Corp., Design Guide v.3.2, 2010.
Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf

[127] *Radeon X1x00 Programming Guide*, ATI Technologies Inc., 2006.
Available: http://developer.amd.com/media/gpu_assets/Radeon_X1x00_Programming_Guide.pdf

[128] *ATI Radeon HD 2000 programming guide*, Advanced Micro Devices Inc., 2007.
Available:
http://developer.amd.com/media/gpu_assets/ATI_Radeon_HD_2000_programming_guide.pdf

[129] J. Gummaraju and M. Rosenblum, "Stream Programming on General-Purpose Processors", in *Proc. of the 38th Annual International Symposium on Microarchitecture (MICRO-38)*, 2005.
Available: http://merrimac.stanford.edu/publications/micro38_streamingGPP.pdf

[130] J. Gummaraju, M. Erez, J. Coburn et al., "Architectural Support for the Stream Execution Model on General-Purpose Processors", in *16th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.
Available: http://merrimac.stanford.edu/publications/streamGPPExtsPACT07Final.pdf

[131] S. Suslov, "Synchronous Dynamic RAM controller on Programmable Logic Device", Dipl. Ing. Qualif. work, Dept. Comp. Techn., St. Petersburg National Research Univ. Inform. Techn., Mechan., Optics, St. Petersburg, 2003.

[132] *PCI 9656BA Data Book*, PLX Technology Inc., Data Sheet v1.1, Oct., 2003.
Available: http://www.plxtech.com

[133] *MICRON DOUBLE DATA RATE(DDR) SDRAM 256Mb*, Micron Technology Inc., Data Sheet 256MBDDRx4x8x16_2.fm - Rev. H 12/03 EN, Dec., 2003.
Available: http://www.micron.com/datasheets

[134] *CYPRESS 512K x 36/1M x 18 Pipelined SRAM with NoBL Architecture, CY7C1370C, CY7C1372C*, Cypress Semiconductor Corp., Data Sheet 38-05233 Rev. *C, Apr., 2004.
Available: http://www.cypress.com/?rID=14025

[135] *DDR SDRAM Controller XS Data Sheet*, Array Electronics, Data Sheet, 2007.
Available: http://www.array-electronics.de/doc/cores/array_ddr_xs_xilinx.pdf

[136] *Compute Visual Profiler User Guide*, nVidia Corp., User Guide DU-05162-001_v04, May, 2011.
Available:
http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Compute_Visual_Profiler_User_Guide.pdf

[137] *7 Series DSP48E1 Slice*, Xilinx Inc., User Guide, UG479, Jan. 30, 2012.
Available:
http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[138] *DDR3 memory technology*, Hewlett-Packard Development Company, Technology brief, 2010.
Available:
http://h20000.www2.hp.com/bc/docs/support/SupportManual/c02126499/c02126499.pdf

[139] R. Swinburne, "DDR4: What we can Expect", *bit-tech.net*[Online]. Aug., 2010.

Available: http://www.bit-tech.net/hardware/memory/2010/08/26/ddr4-what-we-can-expect/1

[140] *Achieving High Performance DDR3 Data Rates in Virtex-7 and Kintex-7 FPGAs*, Xilinx Inc., WP383 Wite Paper, Mar., 2011.
Available:
http://www.xilinx.com/support/documentation/white_papers/wp383_Achieving_High_Performance_DDR3.pdf

[141] *Virtex-II Pro Platform FPGAs: Complete Data Sheet*, Xilinx Inc., Data Sheet DS083, Apr. 22, 2004.
Available: http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf

[142] *Virtex-4 Family Overview*, Xilinx Inc., Data Sheet, DS112, Aug. 30, 2010.
Available: http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

[143] *Virtex-5 Family Overview*, Xilinx Inc., Data Sheet, DS100, Feb. 6, 2009.
Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[144] *Virtex-6 Family Overview*, Xilinx Inc., Data Sheet, DS150, Jan. 19, 2012.
Available: http://www.xilinx.com/support/documentation/data_sheets/ds150 .pdf

[145] *Virtex-7 Family Overview*, Xilinx Inc., Data Sheet, DS180, Jan. 15, 2012.
Available: http://www.xilinx.com/support/documentation/data_sheets/ds180.pdf

[146] C. Maxfield, "New Xilinx Virtex-7 2000T FPGA provides equivalent of 20 million ASIC gates", *EETimes*[Online]. Oct. 25, 2011.
Available: http://www.eetimes.com/electronics-products/electronic-product-reviews/fpga-pld-products/4230049/New-Xilinx-Virtex-7-2000T-FPGA-provides-equivalent-of-20-million-ASIC-gates

[147] A. Shilov, "TSMC Vows to Ramp Up 28nm Production in 2012, Start 20nm Manufacturing in 2013", *xbitlabs.com*[Online]. Aug., 2011.
Available:
http://www.xbitlabs.com/news/other/display/20111208191531_TSMC_Vows_to_Ramp_Up_28nm_Production_in_2012_Start_20nm_Manufacturing_in_2013.html

[148] C. Kowaliski, "Global Foundries lays out roadmap for 28 nm—and beyond", *The Tech Report*[Online]. Aug. 31, 2011.
Available: http://techreport.com/discussions.x/21568

[149] *Tesla C1060 Datasheet*, nVidia Corp., Jan., 2010.
Available:
http://www.nvidia.co.uk/docs/IO/43395/NV_DS_Tesla_C1060_US_Jan10_lores_r1.pdf

[150] *Tesla C2050 and Tesla C2070 Computing Processor Board*, nVidia Corp., Board Specification, BD-04983-001_v05, Sept., 2011.
Available: http://www.nvidia.co.uk/docs/IO/43395/BD-04983-001_v05.pdf

[151] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, nVidia Corp., Whitepaper, 2009.
Available:
http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[152] M. Chiappetta, "AMD Radeon HD 7970: 28nm Tahiti GPU Review", *hothardware.com*[Online]. Dec., 2011.

Available: http://hothardware.com/Reviews/AMD-Radeon-HD-7970-28nm-Tahiti-GPU-Review/

[153] R. Smith, "AMD's Radeon HD 6970 & Radeon HD 6950: Paving The Future For AMD", *anandtech.com*[Online]. Dec., 2010.
Available: http://www.anandtech.com/show/4061/amds-radeon-hd-6970-radeon-hd-6950

Acknowledgements

The present work was conducted in the Central Institute of Electronics of the Research Centre Jülich GmbH (FZJ) supported by the post-graduate scholarship programme of the Research Centre and partly by the German Ministry of Education and Research (BMBF) under Grant 01IRC01A (3D-RETISEG). I would like to express my sincere gratitude towards all the people who made the successful completion of this thesis possible. Amongst the others my especial thankfulness goes to:

My doctoral advisor, a wonderful person and an outstanding scientist, Professor Dr. Reinhard Männer, the Director of the Institute of Computer Engineering of the University of Mannheim (University of Heidelberg since 2008), the Chair holder at the Department for Application Specific Computing, for his interest in my work and his technical and scientific steering.

My supervisor in the Central Institute of Electronics Dr. Jan-Friedrich Vogelbruch, for exhaustive conceptual discussions on the main subjects of the thesis, for being my advisor in expressing ideas in a compact and laconic way and in reconciling the work with high scientific standards.

Dr. Karl Ziemons, Dr. Richard Patzak, Dr. Peter H. Dillinger, Dr. Stefan van Waasen, and Dr. Horst Halling from the Research Centre Jülich for their support of the work.

Professor Dr. Ulrich Brüning (University of Heidelberg) for co-referencing my work, to Professor Dr. Heinz Jürgen Müller (University of Mannheim) and Professor Dr. Holger Fröning (University of Heidelberg) as the head and a co-member of the examination board respectively.

Band / Volume 16
Substituted Coronenes for Molecular Electronics: From Supramolecular Structures to Single Molecules
P. Kowalzik (2010), ix, 149 pp
ISBN: 978-3-89336-679-8

Band / Volume 17
Resistive switching in TiO2 thin films
L. Yang (2011), VII, 117 pp
ISBN: 978-3-89336-707-8

Band / Volume 18
Crystal- and Defect-Chemistry of Fine Grained Thermistor Ceramics on $BaTiO_3$ Basis with BaO-Excess
H. Katsu (2011), xxvii, 163 pp
ISBN: 978-3-89336-741-2

Band / Volume 19
Flächenkontakte zu molekularen Schichten in der Bioelektronik
N. Sanetra (2012), XIII, 129 pp
ISBN: 978-3-89336-776-4

Band / Volume 20
Stacked device structures for resistive memory and logic
R. D. Rosezin (2012), 137 pp
ISBN: 978-3-89336-777-1

Band / Volume 21
Optical and electrical addressing in molecule-based logic circuits
M. Manheller (2012), XIV, 183 pp
ISBN: 978-3-89336-810-5

Band / Volume 22
Fabrication of Nanogaps and Investigation of Molecular Junctions by Electrochemical Methods
Z. Yi (2012), 132 pp
ISBN: 978-3-89336-812-9

Band / Volume 23
Thermal Diffusion in binary Surfactant Systems and Microemulsions
B. Arlt (2012), 159, xlvi pp
ISBN: 978-3-89336-819-8

Band / Volume 24
Ultrathin Gold Nanowires - Chemistry, Electrical Characterization and Application to Sense Cellular Biology
A. Kisner (2012), 176 pp
ISBN: 978-3-89336-824-2

Band / Volume 25
Interaction between Redox-Based Resistive Switching Mechanisms
C. R. Hermes (2012), iii, 134 pp
ISBN: 978-3-89336-838-9

Band / Volume 26
Supported lipid bilayer as a biomimetic platform for neuronal cell culture
D. Afanasenkau (2013), xiv, 132 pp
ISBN: 978-3-89336-863-1

Band / Volume 27
15th European Workshop on Metalorganic Vapour Phase Epitaxy (EWMOVPE XV) June 2-5, 2013, Aachen, Germany
A. Winden (Chair) (2013)
ISBN: 978-3-89336-870-9

Band / Volume 28
Characterization, integration and reliability of HfO_2 and $LaLuO_3$ high-κ/metal gate stacks for CMOS applications
A. Nichau (2013), xi, 177 pp
ISBN: 978-3-89336-898-3

Band / Volume 29
The role of defects at functional interfaces between polar and non-polar perovskite oxides
F. Gunkel (2013), X, 162 pp
ISBN: 978-3-89336-902-7

Band / Volume 30
Parallelisation potential of image segmentation in hierarchical island structures on hardware-accelerated platforms in real-time applications
S. Suslov (2013), xiv, 211 pp
ISBN: 978-3-89336-914-0

Information / Information
Band / Volume 30
ISBN 978-3-89336-914-0

JÜLICH
FORSCHUNGSZENTRUM