4 Radio astronomy with the GPU Correlator, the GPU Imager and the FPGA Imager

John Romein, Bram Veenboer
Netherlands Institute for Radio Astronomy, ASTRON, Netherlands

romein@astron.nl

4.1 Introduction

Within DEEP-EST, parts of the imaging pipeline of a radio telescope were studied. This is a collection of applications that transforms raw telescope data into calibrated sky images. Figure 4.1 depicts this pipeline. On the left, the signals from antennas in the field are digitised and locally combined. The data are sent over Wide-Area Network links to a central location, where the correlator application filters and combines all data in real time, and writes its output to disk. After the observation has finished, bad data (due to interference) are detected and removed, and the remaining data are calibrated to create an image. During the DEEP-EST project the focus has been on the two computationally most intensive applications in this pipeline: the correlator and the imager. The correlator's main task is to combine the data from all receivers, and the imager's main task is to create sky images.

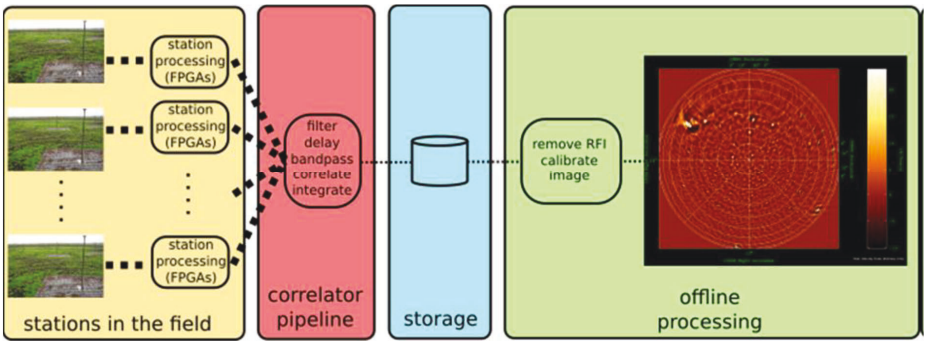

Figure 4.1: Workflow of the imaging pipeline

Both applications are highly optimised for GPUs and CPUs and run much faster on GPUs, for various reasons. The imager performs a large number of sine/cosine operations, for which there is efficient hardware support on GPUs but not on CPUs. The newly developed GPU correlator takes advantage of the *tensor cores* of the latest GPU generation. Tensor cores are special-purpose hardware, which achieve an

exceptionally high performance when executing matrix multiplications at mixed-precision, e.g. 71 TFLOP/s on correlations. Hence, both applications are tens of times faster on GPUs than on CPUs, and this affects the way we will use them on the DEEP-EST MSA.

The imager was also ported to FPGAs. FPGAs used to be programmed in Hardware Description Languages like VHDL and Verilog, which is difficult, time consuming and error prone, and not feasible for complex applications like the imager. New FPGA technologies (the OpenCL high-level programming language, hard Floating-Point Units, and tight integration with CPU cores) have changed this: they should reduce the programming effort of “simple” tasks like a correlator, and should allow complex applications like the imager. We explored Intel's OpenCL/FPGA technology to allow comparisons with GPUs (with respect to performance, energy efficiency, and programming effort), and to bring these technologies into radio astronomy.

4.2 The GPU Correlator

4.2.1 Application structure

4.2.1.1 The correlator pipeline

The correlator combines telescope data and performs signal-processing tasks. The finite impulse response (FIR) filter and FFT blocks transform a wide frequency band into narrow frequency channels. The delay compensation block applies phase corrections to the data that are necessary to follow a source on the sky. The bandpass correction applies an amplitude correction to the data, correcting errors made by another filter near the receivers. The correlate block itself multiplies the data from each pair of receivers, and integrates the products over short periods of time (typically around one second). Especially for a large number of receivers, the correlate block is computationally the most expensive block.

The focus in DEEP-EST was put mostly on improving the performance of the correlation operation. The other operations can be improved when NVIDIA's cuFFTDx library becomes available. As this library will perform FFTs directly on the GPU (unlike cuFFT, which initiates FFTs from the CPU), the first four operations can then be collapsed into one single GPU kernel instead of four kernels, so that only one pass over the data is made, reducing memory bandwidth use. Unfortunately, a pre-release version of cuFFTDx did not compute correct results, so that the integration of cuFFTDx has to be postponed to after the DEEP-EST project. Below, we only report on the correlation operation, which is, especially for large numbers of receivers, the most time-consuming operation anyway.

The correlator combines the data from all receivers by multiplying samples from each pair of receivers and integrating the products over time (typically a second). More specifically, for each frequency channel and each integration period, a matrix with observed, filtered samples S is multiplied by its own Hermitian: $V := S * S^H$. Hence, the output V is also Hermitian, and only the upper- or lower-diagonal triangle needs to be computed and stored, as the output matrix is symmetrical in the diagonal (apart from a minus sign). The BLAS function CHERK computes either side of the diagonal, but stores the output data in a rectangular matrix, wasting GPU memory and PCIe/network bandwidth. Thus, instead of using BLAS, the correlator implements the matrix multiplication itself and stores the data in a triangular data structure.

4.2.1.2 The Tensor-Core Correlator

The correlator uses new GPU tensor core technology to significantly improve performance[71]. Tensor cores are limited-precision matrix-matrix multiplication units designed to speed up deep learning (training and inference), but ASTRON uses them for signal-processing operations that can be expressed as matrix multiplication, such as a correlator or beam-former. For signal processing, the 32-bit precision provided by regular GPU cores is overkill, as Analog-to-Digital Converters near the receivers typically have an accuracy of 4 to 14 bits. Hence, the limited precision of tensor cores does not affect the correctness of the computed results.

The three major implementation challenges were the following: First, tensor cores only operate on real-valued data while the correlator works with complex-valued data. In particular, negating and swizzling real and imaginary parts, which is necessary to perform a complex multiplication, is performed by regular GPU cores because tensor cores cannot do so. Second, the output data structure is a triangular data structure, which is not supported by the tensor core API. Hardware-dependent tricks (with a portable but slower fallback solution) were necessary to write output data quickly. Third, the tensor cores compute so quickly that it is difficult to provide them fast enough with input data. Efficient caching at all levels in the memory hierarchy, as well as coalesced memory accesses, were necessary to keep the tensor cores busy.

The correlator is implemented as a library, to simplify its use in the pipelines of various radio telescopes. Major facilities like the Canadian CHIME and South-African MeerKAT (an SKA precursor) already included the tensor-core correlator library in the processing pipelines that they develop for their correlator upgrades. ASTRON will use the library for their AARTFAAC correlator upgrade (a LOFAR derivative).

[71] John W. Romein. The Tensor-Core Correlator. Astronomy & Astrophysics, 2021, to appear.

4.2.2 Application mapping

Due to the high data rates between the pipeline components of the correlator (in excess of PCIe bandwidth limits), it is not reasonable to separate the operations and run them on different DEEP-EST modules. All operations are performed consecutively on the GPUs of the DAM or ESB (see Figure 4.2).

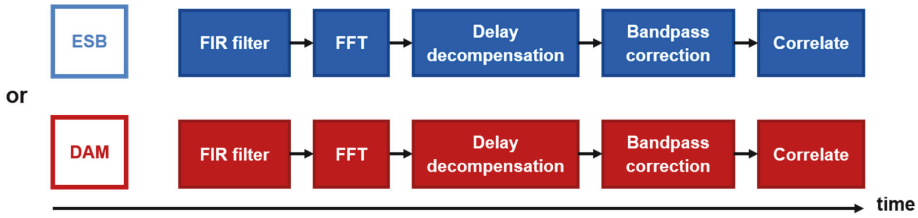

Figure 4.2: Schematic workflow of the correlator in the MSA

4.2.3 Performance comparison

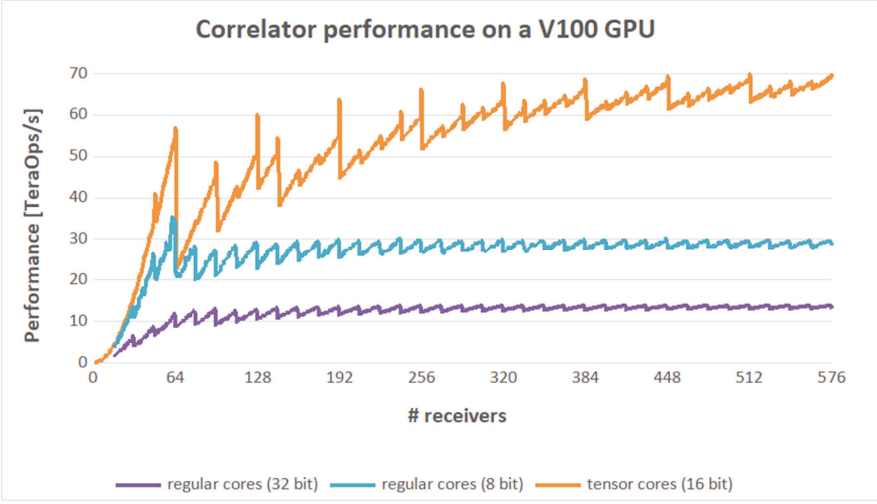

Figure 4.3: Performance of the tensor-core correlator and a legacy GPU correlator on a V100 GPU, as function of the number of receivers, for various precisions of the input data

Figure 4.3 shows the enormous performance benefits of using tensor cores, which depends on the required precision. If the telescope observes with 16-bit samples, the tensor-core correlator performs up to 5.4 times better than the legacy code that runs on regular GPU cores of the NVIDIA V100, at 32-bit precision. The results are obtained on an NVIDIA V100 GPU in a DAM node. The tensor-core correlator is benchmarked

against xGPU[72], commonly considered to be the most efficient correlator library for regular GPU cores. xGPU supports 8-bit and 32-bit input data. The ripples in each of the curves are due to the fact that, depending on the number of receivers, the work cannot always be distributed optimally over the GPU cores or tensor cores.

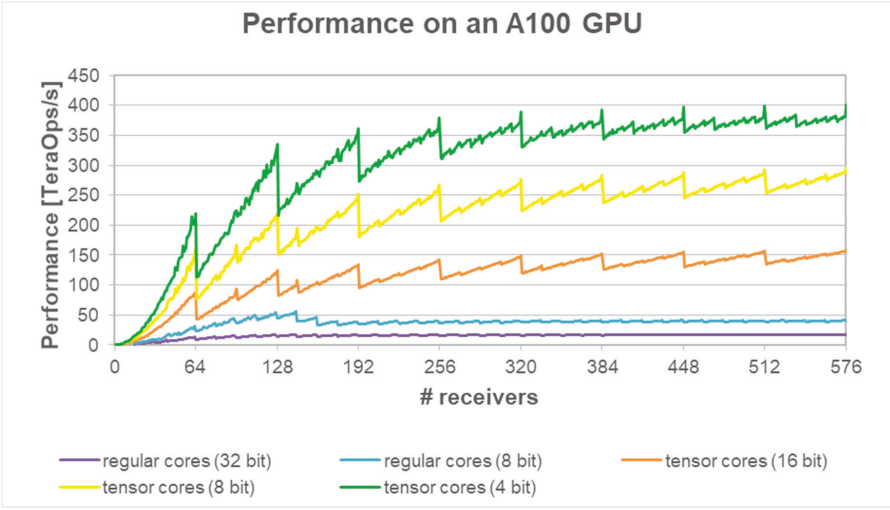

Figure 4.4: Performance of the tensor-core correlator and a legacy GPU correlator on an A100 GPU, as function of the number of receivers, for various precisions of the input data

Many telescopes observe with 8-bit or even 4-bit samples though, and these precisions are not naturally supported by the first-generation tensor cores of the V100. Support for 8-bit, 4-bit, and 1-bit was added later in the second-generation tensor cores of Turing GPUs and third-generation tensor cores of Ampere GPUs, which provide even higher performance. ASTRON further developed the tensor-core correlator to also support 8-bit and 4-bit correlations on the newer GPUs. Figure 4.4 shows performance results obtained on a recently introduced Ampere A100 (PCIe) GPU, the successor to the V100 GPU. The figure not only shows order-of-magnitude performance improvements for 16-bit, 8-bit, and 4-bit correlations, but it also shows that the performance gap between tensor cores and regular GPU cores has widened. Thus, the use of tensor-core technology, which ASTRON started exploring on the DEEP-EST system, will become even more important in future systems.

[72] M.A. Clark, P.C. La Plante, and L.J. Greenhill, "Accelerating Radio Astronomy Cross-Correlation with Graphics Processing units", [arXiv:1107.4264 [astro-ph]].

4.2.4 Energy consumption

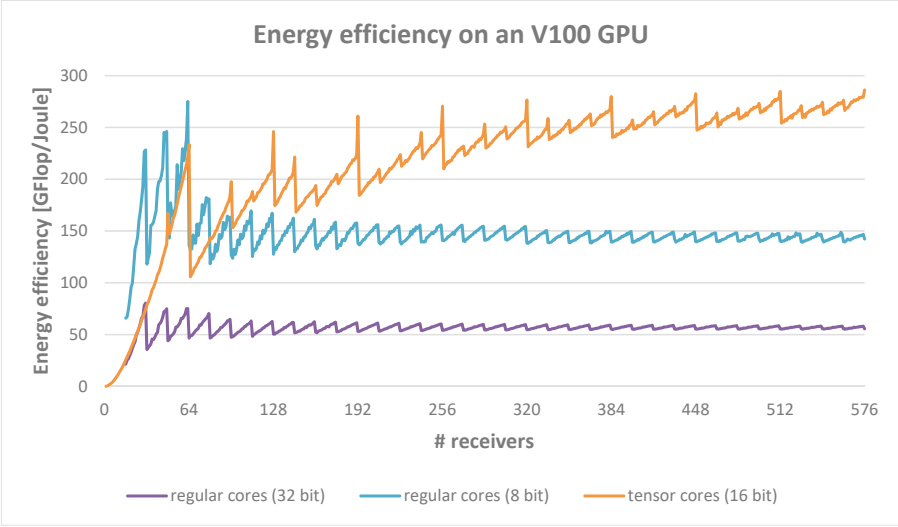

Figure 4.5: Energy efficiency of the GPU correlator on a V100

Figure 4.5 shows that the use of tensor cores is also beneficial to obtain high energy efficiency: in typical use cases, the tensor-core correlator is over 5 times more energy efficient than a GPU correlator that runs on regular GPU cores. The energy efficiency was measured on a V100 GPU in a DAM node. The reported energy efficiency is for the GPU only; system-level measurements are reported in Section 4.2.6. The tensor-core correlator library can accurately measure the GPU's energy use through NVIDIA's NVML library.

The tensor-core correlator is actually limited by the maximum amount of power that the GPU may consume: the GPU's clock frequency is lowered so that the energy use remains below 250W. Even though its power use is high, the energy efficiency is very good, because it performs a very high amount of computations per Joule. Since the correlator drives the GPUs in the ESB to its maximum heat production, this application was used in DEEP-EST also to test a cooling-control plugin developed within the project[73].

[73] Moschny, et al. (2021) D5.5 – Software Support Report

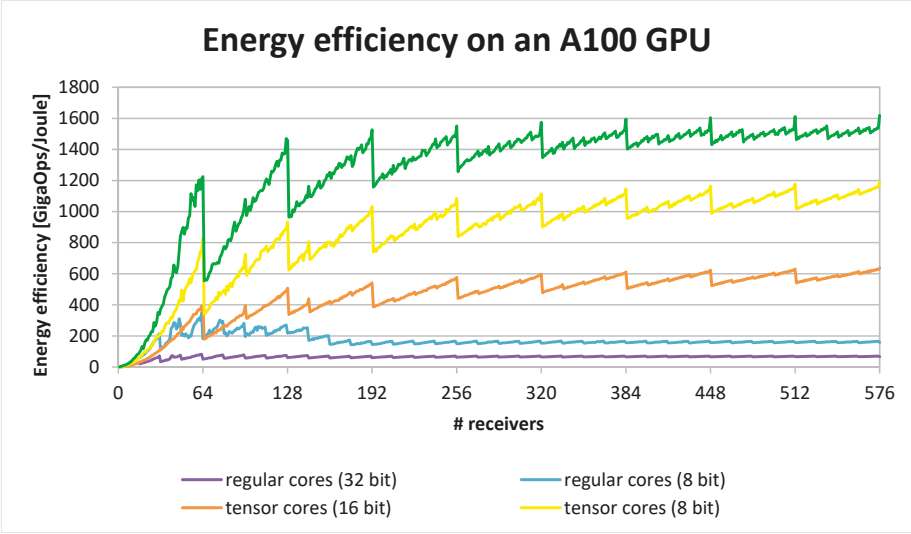

Figure 4.6: Energy efficiency of the tensor-core correlator and a legacy GPU correlator on an A100 GPU, as function of the number of receivers, for various precisions of the input data

Figure 4.6 shows the energy efficiency for the A100 GPU. Yet another leap in energy efficiency is made compared to the V100. On 8-bit and 4-bit input, the energy efficiency exceeds 10^{12} operations per Joule, a new milestone.

4.2.5 Porting experience

Roughly 35% (about 12 PM) of the ASTRON effort was used to develop and optimize the tensor-core correlator. The library was developed from scratch in CUDA, so no additional effort was needed to port the software to the DEEP-EST DAM and ESB nodes. The performance-critical part is only 589 lines of CUDA code, but it is highly complex due to the use of low-level tensor-core intrinsics.

4.2.6 Scalability

The correlator is parallelised over multiple nodes using independent processes, hence the correlator is trivially parallel. Older correlators (such as the LOFAR correlator) used to be MPI programs that performed (real-time) any-to-any transposes of input data across all correlator machines, but newer instruments (e.g., AARTFAAC) perform this transpose outside the correlator, on the way between the receivers and the correlator nodes on packet-switching Ethernet switches. This saves on network hardware costs and simplifies the correlator application.

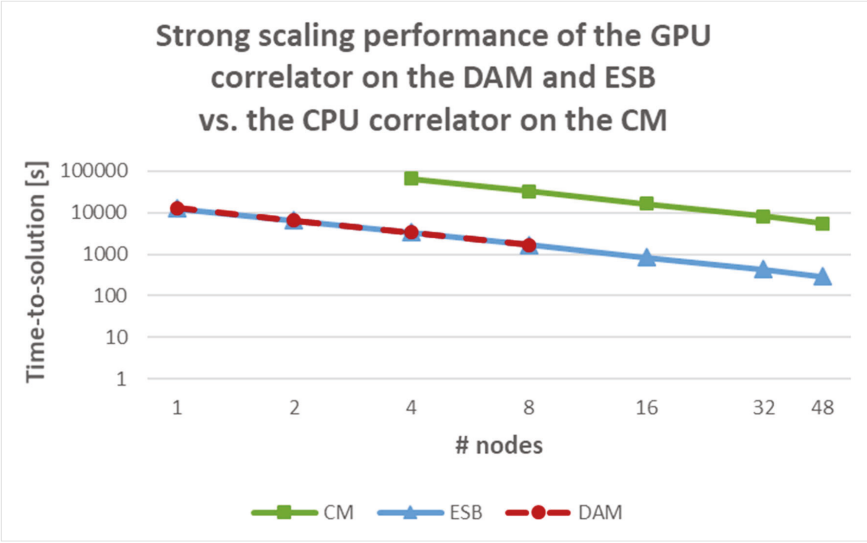

Figure 4.7: Strong scaling results for the tensor-core correlator: runtimes for the GPU correlator on the DAM and ESB, as well as the legacy CPU correlator on the CM

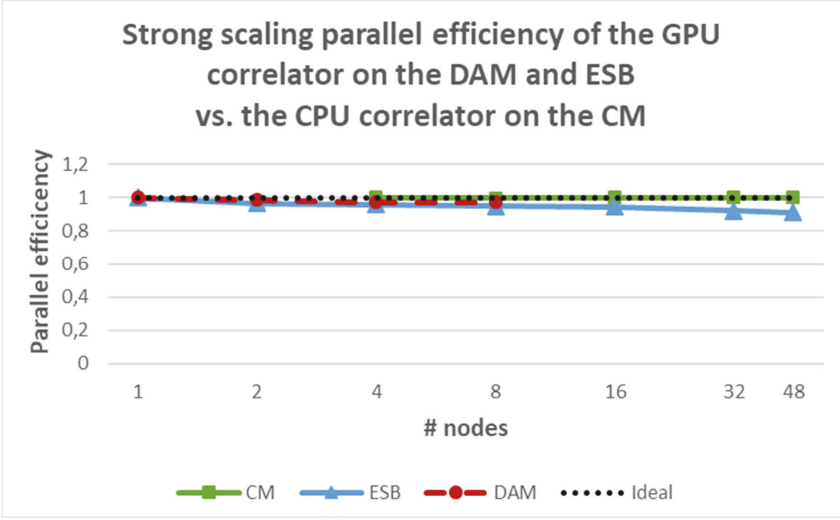

Figure 4.8: Strong scaling results for the tensor-core correlator: parallel efficiency for the GPU correlator on the DAM and ESB, as well as the legacy CPU correlator on the CM

Figure 4.7 shows scaling results from the tensor-core correlator on the DAM and ESB nodes. For comparison, we also included scaling results obtained when running on the CM a slightly modified version of the legacy CPU correlator that was developed for

Xeon and Xeon Phi processors in the predecessor DEEP-ER project. The legacy CPU correlator uses AVX512 intrinsics to achieve optimal CPU performance. For all performance measurements same input data (a large amount) is processed, hence the results are strong scaling results. The single and dual-node measurements of the CPU correlator are omitted, as their runtimes exceed the 20-hour time limit imposed by Slurm.

As both the tensor-core correlator and the CPU correlator processes run independently from each other, the performance scales almost perfectly, as displayed by the parallel efficiency plot in Figure 4.8. The performance on the DAM and ESB is nearly identical, because both modules use GPUs of the same type. ASTRON did notice some variation in GPU speed (up to 8% on the ESB and 3.4% on the DAM), probably caused by different thermal conditions, as all GPUs run as fast as they can within their 250W power limit; the obtained speed depends on the GPU temperature. The graph also shows that the tensor-core correlator processes the same amount of data 19 times faster than the CPU correlator. This atypically high factor is due to the use of tensor cores in the GPU correlator.

Note that in real telescope systems, the correlator is a real-time application that processes streaming data, and only needs to keep up with the incoming data streams. In practice, the correlator GPU hardware will be over-dimensioned so that it can process data faster than the data flows in, frequently stalling the GPUs as they wait for new data to arrive.

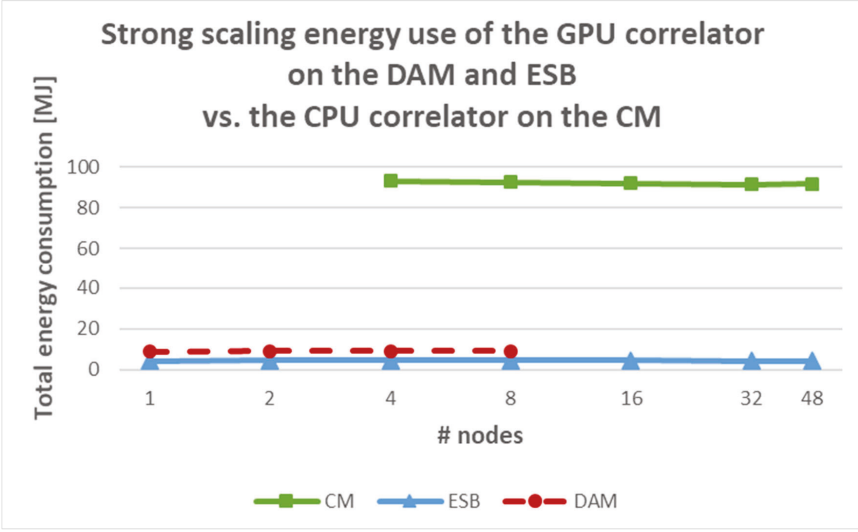

Figure 4.9: Strong scaling results for the tensor-core correlator: energy use of the GPU correlator on the DAM and ESB, as well as the legacy CPU correlator on the CM

Figure 4.9 shows the total energy used by the above-mentioned scaling benchmarks. The total energy consumption is independent of the number of nodes used. The DAM nodes are far less energy efficient than the ESB nodes, due to the presence of (unused) hardware (such as an FPGA, DCPMM DIMMs, powerful CPUs etc.) and of a less energy-efficient air-based cooling mechanism (the ESB is water cooled). The GPU correlator on the ESB is 22 times more energy efficient than the legacy CPU correlator on the CM. Again, this atypically high difference is due to the use of tensor cores.

4.2.6.1 Our path to Exascale

The use of tensor cores resulted in a giant leap in computational performance and energy efficiency, although an Exascale correlator would still need a MegaWatt for the computations alone. The remaining challenge is the part that we moved outside the correlator application and did not investigate within this project: the data exchange between the receivers and the correlator on packet-switching Ethernet switches. At least in theory, such switching systems can be built arbitrarily large by using multiple layers of switches.

The increasing correlator input data rates constitute another challenge. Current instruments typically use UDP/IP (unreliable datagram packets) to send data from the FPGAs near the receivers to the (central) correlator, possibly over dedicated Wide-Area Network links. As we move to the 100+ Gb/s era, the current practice to receive these packets through the kernel stack is no longer sustainable because the system call overhead becomes prohibitively high. ASTRON and its partners currently investigate how to use RoCE (RDMA over Converged Ethernet) to stream packets directly from remote FPGAs into a GPU correlator, without operating system involvement in the critical path. This is not trivial, because RoCE is a much more complex protocol than plain UDP, while the sending side of the protocol should be simple enough to be implemented on FPGAs.

4.2.7 Conclusion

The use of tensor-core technology will have a disruptive impact on correlators (and beam formers) of (near-)future instruments, due to their order-of-magnitude increase in performance and significant energy efficiency compared to the use of regular GPU cores. DEEP-EST provided the means to explore this technology.

4.3 The GPU Imager

4.3.1 Application structure

ASTRON also worked on a relatively new imaging application that creates sky images from calibrated correlations (visibilities) produced by the correlator. Unlike traditional methods that create the image almost fully in the Fourier domain, the new method performs the gridding step (explained below) in the image domain. This way, corrections for direction-dependent effects (e.g., caused by ionospheric disturbance) can be applied efficiently, which improves the image quality, especially when observing at low radio frequencies.

In the past years, the imager has become a mature application and is able to generate sky images for various radio telescopes worldwide. As shown during the project, the application runs much more efficiently on GPUs than on CPUs, because it performs many sine/cosine operations, which are expensive on CPUs and essentially free on (NVIDIA) GPUs.

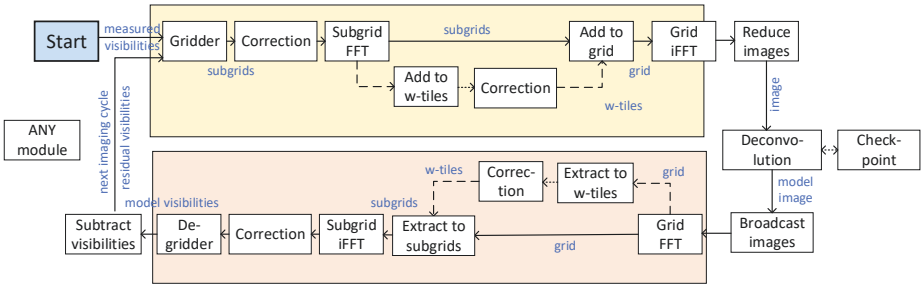

Figure 4.10: The mockup imager

Imaging is an iterative process, consisting of three main steps: 1) gridding, 2) deconvolution, and 3) degridding (see Figure 4.10). Gridding and degridding are the computationally most expensive steps. Hence, they are the focus of this study. In the gridding step, visibilities (measured correlations) are gridded onto a regular grid. The deconvolution step is used to detect sources in the image, and these are added to a model image. In the degridding step, visibilities are computed taking an image as input. By subtracting the model visibilities from the measured visibilities, faint sources become visible. Thus in each iteration the model of the sky is refined by adding increasingly fainter sources. This process is repeated until the sky model has converged.

The GPU imager comprises of two main components: IDG (Image-Domain Gridding) and WSClean. IDG provides gridding and degridding routines, and WSClean provides

deconvolution data handling (visibility input, image output). ASTRON created a mock-up imager around IDG gridding and degridding that emulates the full imager. The mock-up imager is an MPI application that distributes the input data (visibilities) over the nodes. Every node processes a block of data (a number of visibilities, say for an hour of observation) and creates a partial image. These images need to be combined (added together) to attain high signal-to-noise, such that deconvolution can detect faint sources in the image. The image combination step is implemented as a parallel reduction. The deconvolution step is not part of the mock-up imager. Every now and then (say every hour of processing), a checkpoint is made, see Section 4.3.4 for details. The next step is to distribute the model image to all compute nodes such that they can proceed with degridding.

4.3.2 Application mapping

For two reasons, the imager should run on the DAM: the presence of accelerators and the large amount of available memory. As the gridder performs many sine/cosine operations, it should definitely run on GPUs or FPGAs.

The subgrids can be Fourier transformed efficiently on the GPU as well. Addition of the FFTed subgrids to the grid can be done either on the GPU or host CPU of the DAM; in our experience, it is somewhat more efficient to perform it on the GPU. The best place to perform the final inverse FFT of the grid is not yet determined: it seems to depend on the image size and on the efficiency of the FFT library for a particular architecture. The best place can be the GPU, FPGA, or CPU of the DAM, or even one of the other DEEP-EST modules.

Another reason to run the imager on the DAM is the presence of 3D XPoint DIMM modules, which may be used to save huge sky images that do not entirely fit in DRAM. In this case, the DRAM transparently caches the “hot” parts of the grid that are stored in the larger-capacity 3D XPoint DIMMs. On top of that, the even smaller GPU device memory will be used to transparently cache the “really hot” parts of the grid, using NVIDIA's Unified Memory technology. The imager is a particularly interesting application to demonstrate the usefulness of transparent caching (both 3D XPoint and Unified Memory) as the access pattern to the grid becomes complex (but with sufficient locality).

For low-resolution images, which require less memory, the GPU imager can also run on the ESB.

4.3.3 Porting experience

The mock-up imager can run on the GPUs of the DAM or ESB, as well as on the CPUs of the CM. ASTRON expected the GPU imager to run on the DAM or ESB GPUs with

no or little additional effort, as the application was already tested on a number of other GPU-based systems, for small and medium-sized images. However, the imager initially performed sub-optimally on these small and medium-sized images, while on large images, the application simply hung. These problems were solved by making changes to the application and by using an updated cuFFT library.

Roughly 15% of the ASTRON effort (6 PMs) was spent on the GPU imager.

4.3.4 Checkpointing

The input for an imaging cycle consists of two data products: measured visibilities (i.e., calibrated correlations) and model visibilities. The model visibilities are created by running degridding on the model image, after which these model visibilities are subtracted from the measured visibilities. In case of a crash, the measured visibilities can easily be recovered as these are typically read-only. To restore the model visibilities, the model image is needed. When the mock-up imager recovers from a crash, the model image is loaded from the checkpoint and distributed to the compute nodes, after which imaging proceeds as normal.

At first, ASTRON considered using the OpenCHK checkpointing library. This library is pragma-based and requires only a few additional lines of code. However, it requires a different compiler and compilation flow, which was found to be incompatible with IDG. It turned out to be easier to implement checkpointing using the FTI (Fault Tolerant Interface) library, which is internally used by OpenCHK as well.

4.3.5 High-resolution imaging

There is a direct relation between the resolution of a sky image and the geographical distance between the two outermost receivers used in an observation. With the expansion of LOFAR stations all across Europe, there is a need to create increasingly higher-resolution images of tens of thousands of pixels high and wide, consuming hundreds of Gigabytes per image. Through the DEEP-EST project, ASTRON studied and handled the issues of creating such large images.

For wide-field imaging, baselines (receiver pairs) cannot be assumed to be in one plane, due to the curvature of the earth. Similarly, for wide fields of view, the sky cannot be approximated as a flat plane. Together, these effects must be corrected for and require large convolution kernels (W-kernels) during gridding and degridding. Gridding the longest baselines becomes computationally very expensive then. A technique called W-stacking alleviates this by effectively gridding onto multiple grids (one for each so-called W-layer). Short baselines are gridded onto the lowest W-layer, while the

longest baselines are gridded on the highest W-layer. This technique, however, requires a lot of memory, as the image that is being constructed consists of many layers.

The original plan was to explore the use of DCPMM memory in the DAM nodes to store the W-layers. However, recently, we started realizing that an algorithmic change would significantly reduce the memory footprint to create an image. This technique, called *W-tiling*, requires the use of only one grid (one W-layer), and a cache of so-called W-tiles. A W-tile represents a small part of a W-layer, and an algorithm maps the subgrids to W-tiles. Some additional computations are needed to 'project' a W-tile onto the one W-layer, but for large images, W-tiling strongly relaxes the need for impractically large amounts of memory, and hence we consider it a good trade-off. Therefore, there is no need to use DCPMM anymore.

The first implementation of W-tiling is a hybrid (CPU + GPU) imaging mode where gridding takes place on the GPU and W-tiling takes place on the CPU. In the second implementation of W-tiling, most of the W-tiling operations are performed on the GPU. As with gridding, where a phase shift is applied to place a visibility on a subgrid, a phase shift is applied to place a subgrid onto a W-tile and to project a W-tile to the W-layer (the "w=0 plane"). Since these phase shifts involve many sine/cosine computations, W-tiling runs much faster on the GPU compared to running it on the CPU.

Figure 4.11 shows the runtime for both W-tiling implementations. The runtime comprises two parts: the dark bar represents the time spent in gridding, the lighter part the time spent in flushing the W-tiles (constructing the image). For v1, the runtime becomes prohibitive for grids larger than about 28,000×28,000 pixels. The latest implementation (v2) scales to much larger images.

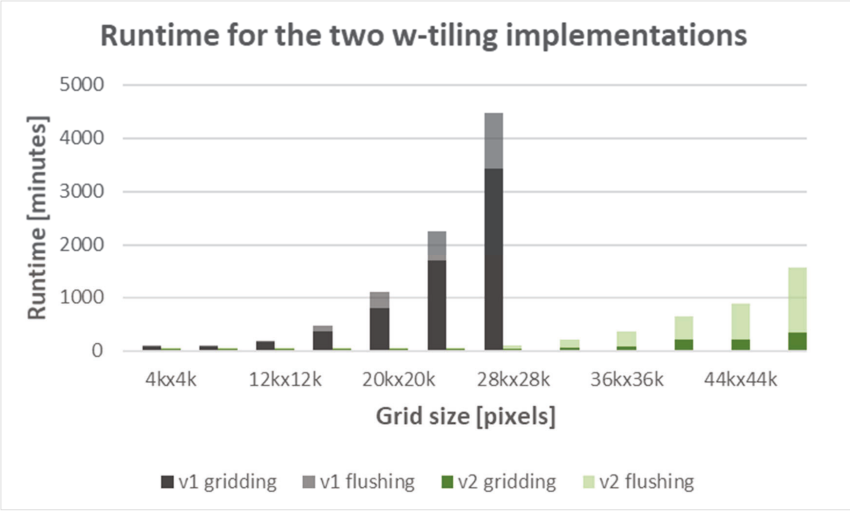

Figure 4.11: Comparison of runtime for the two w-tiling implementations

The method has been demonstrated to work for much larger grids (up to more than 100,000×100,000 pixels) as well, but in these cases the throughput becomes bound by the limited bandwidth of PCIe: the GPU spends a few hundred milliseconds in gridding (at over 10 TFLOP/s), after which several seconds are spent copying W-Tiles to the host (either explicitly, or by Unified Memory page migrations).

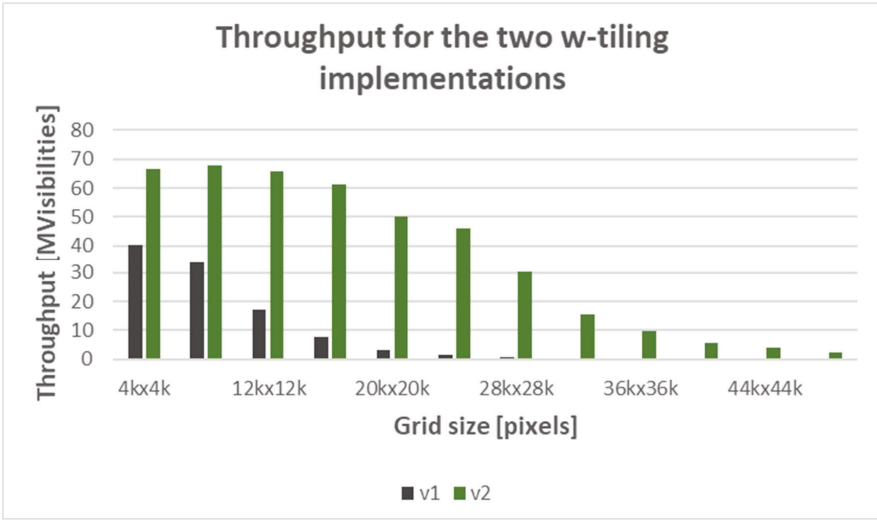

Figure 4.12: Comparison of throughput for the two w-tiling implementations

Figure 4.12 shows the throughput achieved by the two W-Tiling implementations. As explained above, the runtime for the CPU-only implementation (v1) becomes prohibitive for grid sizes larger than 28,000×28,000 and we therefore omit throughput results for larger grid sizes. For these cases, throughput is well below 1 million visibilities/s. The achieved throughput for the GPU- accelerated implementation (v2) also goes down as the grid size increases, but it attains much higher overall throughput.

Imaging is a lot more challenging for large images, which is reflected in the reduced throughput. Still and most importantly, DEEP-EST enabled ASTRON to speed up IDG such that large images (32,000×32,000 pixels) are now feasible.

4.3.6 Scalability

ASTRON evaluated the scalability of the mock-up imager on the ESB (with GPUs, see Figure 4.13) and on the CM (CPU-only, see Figure 4.14). To this end, a simulated dataset for a 12-hour observation was used, based on all of the proposed SKA1 Low station coordinates.

On the CM, the runtime of the imager is dominated by the gridding and degridding times. The overall runtime is inversely proportional to the number of nodes used. The time spent in communication (the grid-reduce and grid-broadcast) is negligible.

As explained before, IDG runs much more efficiently on GPUs than on CPUs, which is also reflected in about a 20-fold reduction in runtime. GPUs are much faster because they compute sine/cosine operations very efficiently *in hardware*, while on CPUs, we have to resort to library functions to compute sine/cosine *in software*. Even though these libraries (e.g., Intel MKL) are highly optimized, the CPU spends 80% of the time computing sines and cosines. On GPUs, where gridding and degridding run so fast, the time spent in communication becomes noticeable, especially when more than 8 nodes are used.

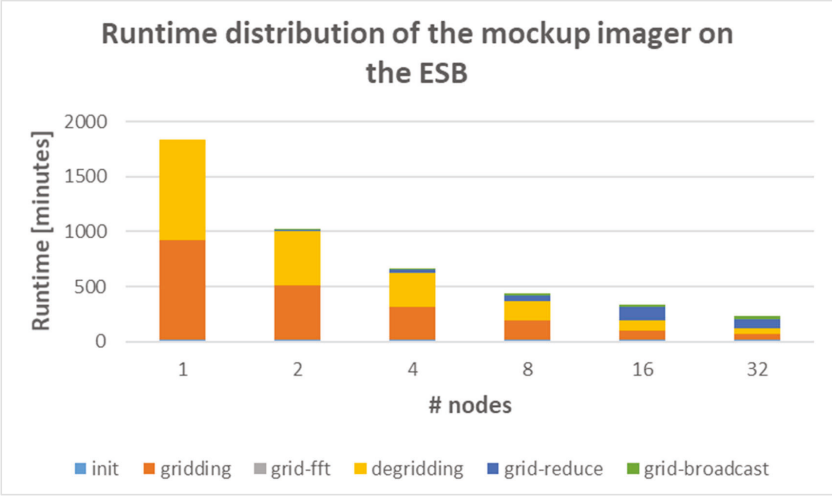

Figure 4.13: Runtime distribution of the mockup imager on the ESB

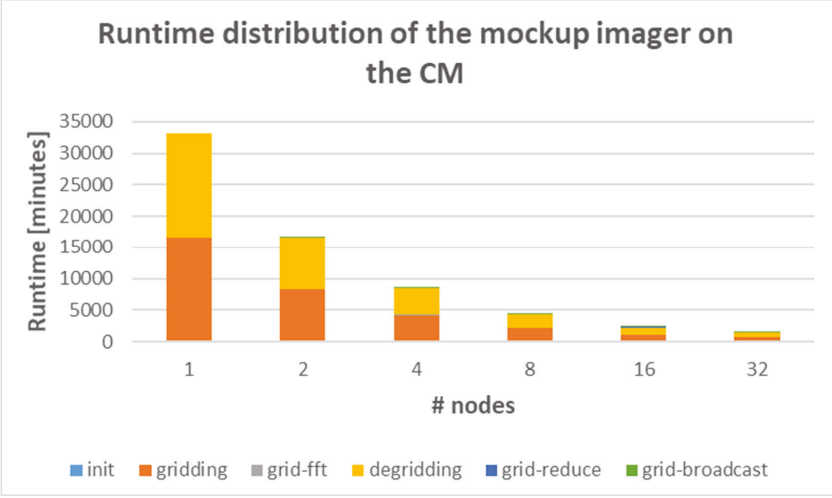

Figure 4.14: Runtime distribution of the mockup imager on the CM

4.3.6.1 Our path to Exascale

In its basic form, the imager is trivially parallel, as different frequency bands are processed independently by independent processes. Hence, the mock-up imager will scale perfectly. Even in the case that the work for one frequency band is distributed over multiple nodes, good scalability is achieved, as shown in Section 4.3.6.

When scaling to thousands of nodes, there is no need to run the imager as a single MPI application across all nodes: visibilities from different frequency bands can be imaged independently by multiple (groups of) imaging processes. The frequency-dependent images still need to be merged to a final image, but this is not in the critical path.

That is not to say that the whole imaging processing pipeline, which contains several other applications, is ready for Exascale yet: the imager itself, which used to be the slowest processing step (as it is the computationally most expensive step) has become so fast now that it reveals several new bottlenecks. Due to the amount of work involved, it was not possible to remove all new bottlenecks within the scope of the DEEP-EST project.

Creating very large sky images (e.g. 100,000×100,000 pixels in size) remains challenging. Unlike on POWER8/NVLink-based systems, a PCIe-based connection to the CPU provides too little bandwidth to keep the GPU busy. The new W-tiling implementation alleviates, but does not resolve this bottleneck.

Faster links between GPUs and (host) memory (e.g., CXL, NVLink) will increase the imaging throughput. Alternatively, using many small GPUs may be better than using fewer big GPUs, to take advantage of the larger total PCIe bandwidth. Moreover, small GPUs typically consume less power.

DEEP-EST provided the means to create sky images at much higher resolutions than before, which are required by future Exascale instruments like the SKA, but also current ones such as LOFAR, which is now also capable of creating sky images of 30,000×30,000 pixels in size.

4.3.7 Performance and energy comparison

A comparison of the runtime and energy consumption for the mock-up imager on CM and ESB is shown in Figure 4.15 and Figure 4.16, respectively. Unsurprisingly, the ESB is much more (energy) efficient.

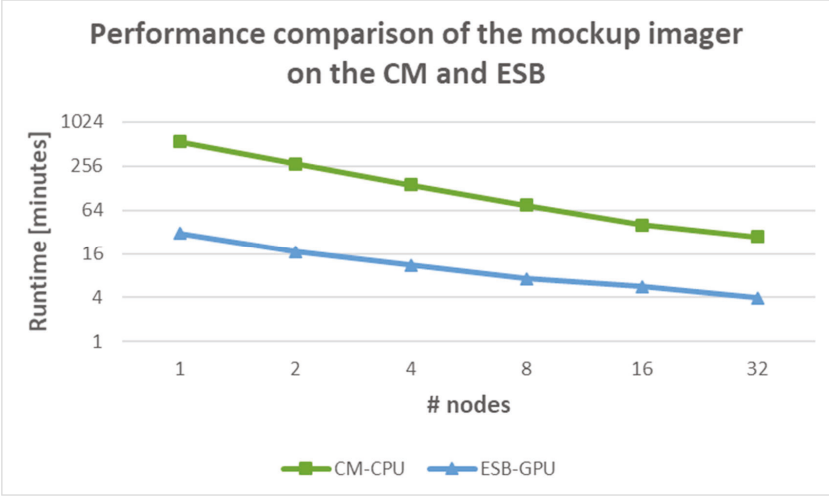

Figure 4.15: Performance comparison of the mockup imager on the CM and ESB

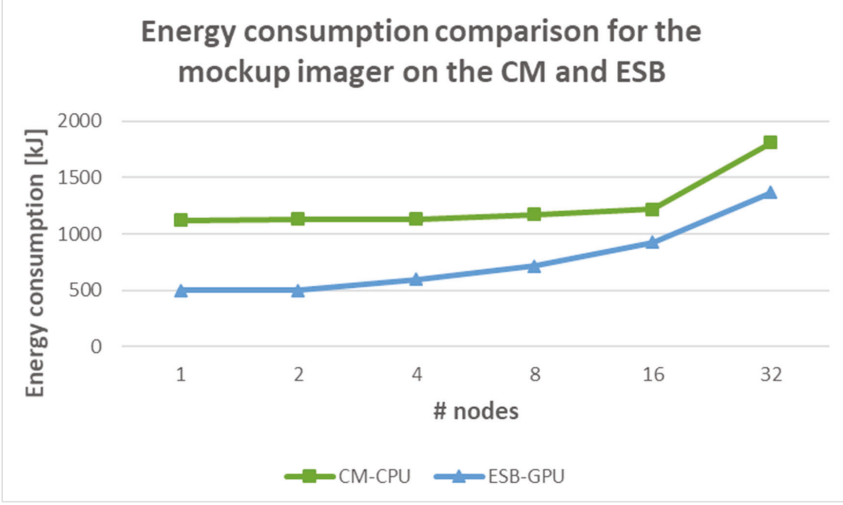

Figure 4.16: Energy consumption comparison for the mockup imager on the CM and ESB

4.3.8 Conclusion

IDG runs highly efficient on GPUs thanks to their native support for sine/cosine operations. Now that the GPU kernels (such as the gridder and degridder) run so fast, every operation in the imager that does not run on the GPU tends to become a bottleneck. ASTRON explored a new technique, called W-tiling, that significantly reduces the amount of memory used to create (very) large sky images, at the expense of a minor increase in computations. By implementing W-tiling on the GPU, image creation of up to ~30,000×30,000 pixels in size has become practical and fast, so that the painstaking effort of stitching hundreds of facets together belongs to the past. The biggest bottleneck remaining is the fairly limited bandwidth between the host DRAM and GPU memory. Similarly, the mock-up imager becomes bound by the network bandwidth between the compute nodes, especially when IDG is used with GPU acceleration. Future algorithmic improvements on deconvolution may help to reduce the amount of communication between nodes, and therefore help to improve scalability. All in all, DEEP-EST enabled us to improve the overall performance of the imager and brings us a big step closer to Exascale imaging.

4.4 The FPGA Imager

4.4.1 Application Overview

A mock-up imager able to run of FPGAs and performing only the most compute-intensive tasks of the full imaging pipeline (described in more detail in Section 4.3.1) has also been developed. Figure 4.17 shows these tasks: gridding visibilities onto 32×32 subgrids, a correction for direction-dependent and other effects, and a 2D FFT over each subgrid. The optional addition to W-tiles and associated corrections were not implemented on the FPGA, as these algorithmic improvements were explored in the GPU imager during the final months of the DEEP-EST project and there was no time left to implement and optimize them on the FPGA. The FPGA does not have sufficient on-board DDR4 memory to store the full (Fourier-transformed) image, thus the next two steps, accumulation into the full grid and the final inverse FFT over the full grid, cannot be performed on the FPGA. Instead, the Fourier-transformed subgrids are transferred back to the CPU for further processing.

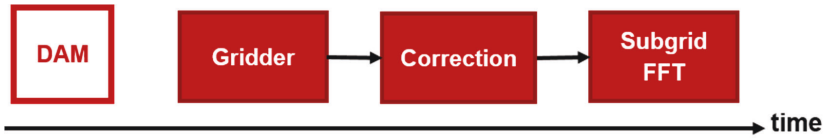

Figure 4.17: Schematic workflow of the FPGA Imager in the MSA

4.4.2 Porting experience

The Intel OpenCL/FPGA toolkit was used to implement the FPGA imager, before oneAPI became available. Originally, the imager was developed for the (mid-range) Arria 10 FPGA, as (high-end) Stratix 10 boards were not available at the start of this project. The performance obtained on the Arria 10 was fair: much better than CPUs, but not as good as GPUs. A paper comparing CPU, GPU, and FPGA imaging with respect to architectures, programming models, optimizations, performance, energy efficiency, and programming effort received the Euro-Par'19 best paper award[74].

As soon as the Stratix 10 FPGA boards were installed in the DAM nodes, ASTRON started porting the Arria 10 imager to the Stratix 10. The Stratix 10 is not only larger and theoretically able to run at higher clock speeds, but it is also characterized by an on-chip interconnect that is architecturally different from the interconnect in an Arria 10. As a result, Intel advises to not use blocking channels (FIFOs, an OpenCL extension), because this reduces the maximum clock speed at which an FPGA application runs on a Stratix 10. And indeed, the original imager was initially very slow, both because of excessive resource usage and because of a low clock at which the application ran.

ASTRON already foresaw that the port to the DAM FPGA would be a major effort. Essentially, the OpenCL code was fully re-implemented. The original code consists of hundreds of small (partly replicated) OpenCL kernels connected by blocking channels (FIFOs); the new imager consists of one single, highly complex OpenCL kernel that performs all operations and essentially fills the whole FPGA. Both the single-kernel and many-kernel imagers were optimized further, in order to find out which one would be the better approach. Many Stratix 10 specific optimizations were necessary, such as avoiding the use of double-pumped memory (i.e., memory that transfers two words per cycle per port), limiting the use of memory to only one read location and one write location. In many cases this made the code more complex. Another important Stratix 10 specific optimization is to write code in such a way that it allows the compiler to enable hyper-optimized handshaking; this increases the clock speed at which the design will run. Furthermore, many optimizations were implemented that reduced the resource usage other than multipliers (one does want to use as many multipliers as possible: the more multipliers are used, the more simultaneous computations can be performed). Reducing resources makes it easier to fit the design onto the FPGA.

[74] Bram Veenboer and John W. Romein. Radio-Astronomical Imaging: FPGAs vs GPUs. Euro-Par'19, Göttingen, Germany, August 2019

The sine/cosine operations are implemented using a lookup table. A lookup table is less accurate than the compiler-built-in sine and cosine operations, but sufficiently accurate for this application. The lookup tables do not require the use of multipliers, which can be used to perform more gridding (and other) operations instead. However, the table uses a large amount of internal memory (16% of the total available memory blocks), because the compiler (automatically) replicates the table 320 times to provide enough memory bandwidth for 320 simultaneous sine and cosine operations per cycle. The table is compressed, meaning that for each table entry (a pair of single-precision floating point numbers), 13 of the 64 bits are always the same and are not stored in memory.

Although both FPGA imagers were eventually successfully implemented and optimized, there were many workarounds for compiler issues necessary. Some issues were obvious compiler bugs (e.g., a crash when compiling legal program code), but most cases were on code constructs that were not well handled or optimized by the compiler (e.g., a sharp increase in memory use when shrinking the width of a table from 60 to 51 bits). Yet in other cases, the compiler could not be blamed; some constructs cannot be handled efficiently in any way by an FPGA. Finding solutions for all issues turned out to be a very time-consuming effort, especially when it requires the synthesis of a full FPGA design, which can take a full day, even on a fast computer. Several compiler bugs were reported to Intel and were fixed or will be fixed in subsequent compiler releases. Over the years, the tools improved significantly, but it takes a long time for them to reach full matureness.

Roughly half of ASTRON's effort (about 18 PM) in DEEP-EST was spent on implementing the FPGA imager. The many-kernel imager consists of 835 lines of performance-critical OpenCL code, the single-kernel imager about 560 lines, not counting the generated sine/cosine lookup tables. Thousands of small modifications to these programs were made to minimize resource usage, identify and reduce idle times, optimize clock frequency, and work around compiler issues.

As we implemented and optimized Image-Domain Gridding for CPUs, FPGA, and GPUs, we found differences and similarities with respect to architecture, programming model, necessary optimizations, performance, energy efficiency, and implementation effort. We discuss them in the sections below.

4.4.3 Performance

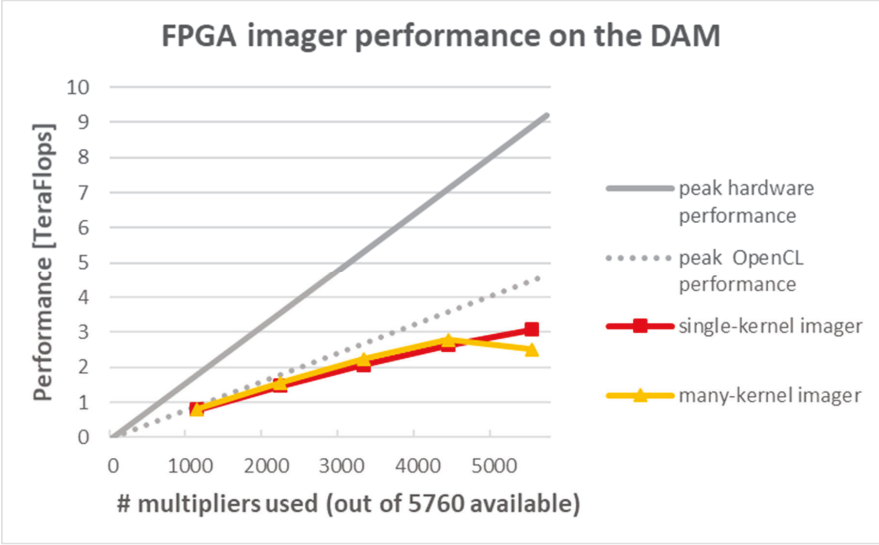

Figure 4.18: Performance of the FPGA imager on the Stratix 10

Figure 4.18 shows the performance of the FPGA imager on the Stratix 10 FPGA board in a DAM node, as a function of the number of multipliers used. The FPGA hardware has a peak clock frequency of 800 MHz, which translates to the “peak hardware performance” line in the figure. However, the OpenCL board support package does not run at more than 401 MHz, limiting any OpenCL program to at most 50% of the theoretical performance, as reflected by the “peak OpenCL performance” line in the figure. Two other curves show the performance of the multi-kernel imager and the single-kernel imager, respectively. The performance is reported in terms of floating-point operations per second, where only all multiplications, additions, and subtractions are counted, not the sine/cosine lookups. Each measured value in the figure is the result of a series of compilations over a large number (50–200) of random seeds, and the performance of the design that runs at the highest clock speed is reported in the figure.

Figure 4.18 tells that the performance is a factor of three from the advertised hardware peak performance. Eventually, the single-kernel imager is slightly faster than the many-kernel imager. On a full design where almost all multipliers are used, the logic use of the many-kernel imager is so high that the design is difficult to place and route; if the placement succeeds at all, the frequency at which the design runs is no more than 236 MHz. The single-kernel imager runs at 289 MHz for a full design.

4.4.4 Energy consumption

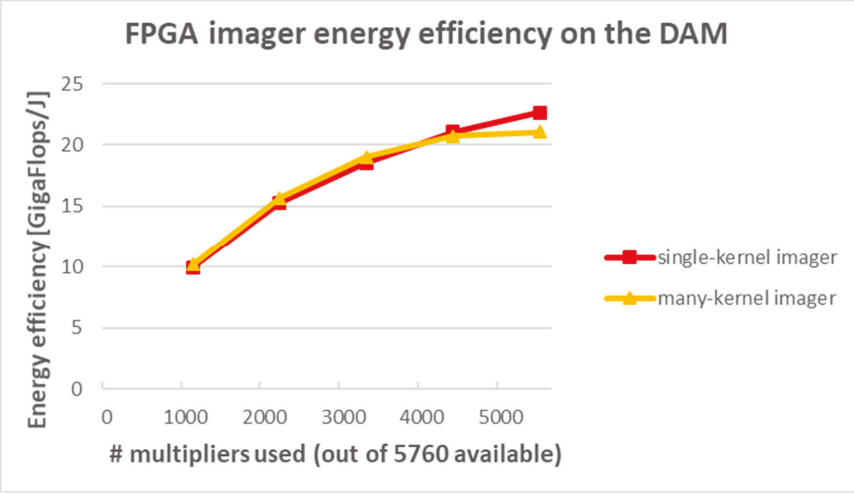

Figure 4.19: Energy efficiency of the FPGA imager on the Stratix 10 FPGA

The Stratix 10 FPGA board can measure its own power use. Figure 4.19 shows the energy efficiency for the multi-kernel and single-kernel imagers. Unsurprisingly, a full design is more energy efficient than a partially-used FPGA. The maximum energy efficiency is 22.7 GigaFlops/Joule. The section below discusses how this compares to other devices.

4.4.5 A comparison between CPU, GPU, and FPGA imaging

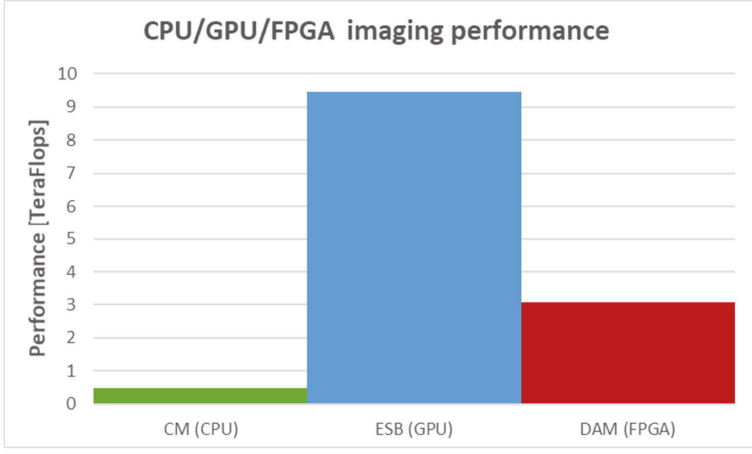

Figure 4.20: A comparison of imaging performance on a CPU in a CM node, a GPU in an ESB node, and an FPGA in a DAM node

As ASTRON has optimized CPU, GPU, and FPGA implementations for the computationally most intensive parts of the full imaging application, the performance and energy efficiency of these processors could be compared. Figure 4.20 shows the performance in TeraFlops (so higher is better) counting only the multiplications, additions, and subtractions, not the sine and cosine computations or lookups. Section 4.3 already showed a huge performance advantage for GPUs over CPUs for sky-image creation. Figure 4.20 shows that the FPGA is somewhere in between. The FPGA computes much faster than the CPU, mostly because the FPGA performs the sine/cosine lookups much more efficiently than the CPU performs these operations. ASTRON also tried the lookup-table approach on a CPU, but this did not improve performance. The FPGA does not perform as well as the GPU. This was expected, as the FPGA has a lower peak performance (9.2 vs. 14 TFLOPS), but the GPU imager runs at 68% of the GPU peak performance, while the FPGA imager runs at 33% of the FPGA peak performance. For the FPGA, the maximum clock rate at which the imager runs, is too low to compete with GPUs.

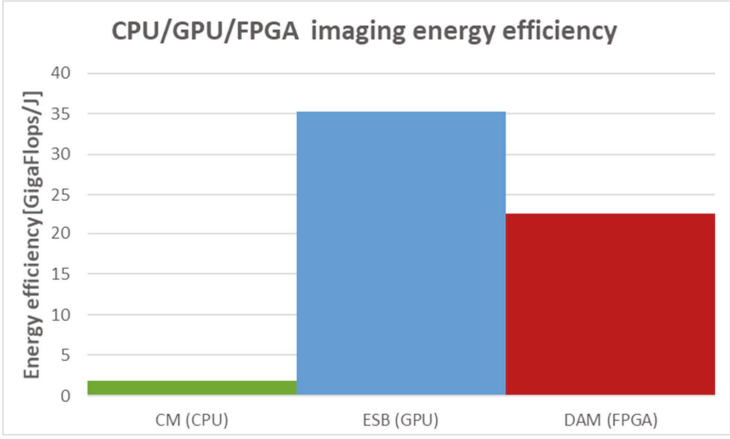

Figure 4.21: A comparison of imaging energy efficiency on a CPU in a CM node, a GPU in an ESB node, and an FPGA in a DAM node

Another (and arguably fairer) way to compare these processors is to analyze their energy efficiency. Figure 4.21 shows the energy efficiency in terms of visibilities per second (GigaFlops/Joule, higher is better). For a fair comparison, only the GPU or FPGA device power is measured, not the total system power, as the DAM contains many unused components that draw power and are not used for this comparison. For the CPU measurements, the processor and DRAM power are measured. The figure shows that the energy efficiency of the FPGA is an order of magnitude higher than the

energy efficiency of the CPU, but still somewhat lower than the energy efficiency of the GPU.

4.4.6 FPGA vs GPU: Lessons learned

The GPU imager was implemented in both CUDA and OpenCL and the FPGA imager only in OpenCL. Even though the GPU and FPGA imagers share a common programming language, hardly any code reuse was possible. This is mostly due to the different programming models: with FPGAs, one builds a dataflow pipeline, while GPU code executes instructions. On the FPGA, the programmer has to think about how to divide the resources (multipliers, memory blocks, logic, etc.) over the pipeline components, so that every cycle all multipliers perform a useful computation. At the same time, the use of other resources (such as logic and memory blocks) should be reduced, while bottlenecks, underutilization, stalls, and constructs that lead to a low clock speed should be avoided. For the imager, this pipeline is complex, and consists of many subpipelines. The speed at which data flows through each of the subpipelines had to be managed carefully. Data should not flow too slowly through any of the subpipelines, or that subpipeline becomes an overall bottleneck. Nor is there any point in making data flow too quickly, because that wastes resources.

There are more differences. On an FPGA, the programmer constructs local memory in some optimal way, while local memory on a GPU has a fixed, banked configuration that the programmer uses. On FPGAs, non-performance-critical operations, such as initialization routines, can consume many resources, while on GPUs, performance-insensitive operations are not an issue. On FPGAs, it is also much more important to think about timing (e.g., to avoid pipeline stalls), but being forced to think about it leads to high efficiency: 96.3% of all multipliers/adders perform a useful operation 96% of the time.

FPGAs have typically less memory bandwidth than GPUs, but we found that with the FPGA dataflow model, where all kernels are concurrently active, it is less tempting to store intermediate results off-chip than with GPUs, where kernels are executed one after another. In fact, our FPGA designs use DDR4 memory only for input and output data; we would not have used the DDR4 memory at all if the OpenCL Board-Support Package would have implemented the PCIe I/O channel extension. In contrast, the cuFFT GPU library even requires data to be in off-chip memory.

Both FPGAs and GPUs obtain parallelism through kernel replication and vectorization; FPGAs also by pipelining and loop unrolling. This is another reason why FPGA and GPU programs look different.

Surprisingly, many optimizations for FPGAs and GPUs are similar, at least at a high level. Maximizing FPU utilization, data reuse through caching, memory coalescing, memory latency hiding, and FPU latency hiding are necessary optimizations on both architectures. For example, an optimization that we implemented to reduce local memory bandwidth usage on the FPGA also turned out to improve performance on the GPU, but somehow, we did not think about this GPU optimization before we implemented the FPGA variant. However, optimizations such as latency hiding are much more explicit in FPGA code than in GPU code, as the GPU model implicitly hides latencies by having many simultaneously instructions in flight. On top of that, architecture-specific optimizations are possible (e.g., the sine/cosine lookup table).

Overall, we found it more difficult to implement and optimize for an FPGA than for a GPU, mostly because it is difficult to efficiently distribute the FPGA resources over the kernels in a complex dataflow pipeline. Even so, we consider the availability of a high-level programming language and hard FPUs on FPGAs an enormous step forward. The OpenCL FPGA tools have considerably improved during the past few years, but have not yet reached the maturity level of the GPU tools, which is quite natural, as the GPU tools have had much more time to mature.

4.4.7 Conclusion

Despite the amount of programming effort that is put into the FPGA imager, the performance is still not as good as we hoped for, but the energy efficiency is fair. The attempt to avoid blocking memory channels, which are known to limit performance on the Stratix 10, resulted in a new program code in which the whole application is implemented as a single, complex OpenCL kernel that fills the entire FPGA. Eventually, the single-kernel imager performs slightly better, but it took a lot of time to implement and optimize the program code. Successive compiler releases managed to recognize some of the complex structures better and better, but it remains difficult for a compiler to efficiently translate such a highly complex kernel.

The FPGA imager is much more (energy) efficient than the CPU imager, mostly because of the CPU's poor support for vectorised sine/cosine computations. However, there is not a single aspect (performance, energy efficiency, programming effort, flexibility, compilation time) where the FPGA surpasses GPUs. Typical FPGA advantages over GPUs, such as strict real-time behaviour, low latency, integrated 100 Gigabit/s Ethernet interfaces, and the ability to interface with other electronics like Analog-to-Digital converters, are not advantages from which the imaging application could profit.

On the other hand, the experience that was obtained with the OpenCL/FPGA toolkit has been very useful. ASTRON now uses this experience for other applications where

FPGAs are indispensable. Eventually, the use of a high-level programming language will significantly reduce programming effort compared to traditional hardware description languages like VHDL.