7 High Energy Physics with CMSSW

Viktor Khristenko, Maria Girone,
European Laboratory for Particle Physics, CERN, Switzerland
viktor.khristenko@cern.ch

7.1 Introduction

The Compact Muon Solenoid (CMS) detector located at the Large Hadron Collider from CERN is a general-purpose particle detector consisting of several components: tracker, electromagnetic and hadronic calorimeters, magnet and muon systems. Each component (usually addressable as sub-detectors) accomplishes a different task. For instance, tracker (both Pixel and Strip parts) is the closest sub-detector to the interaction point and responsible for identifying the trajectories of charged particles. Calorimeters measure energy depositions of the particles passing through.

Two different applications were evaluated on the DEEP-EST Modular Supercomputer Architecture (MSA): CMS event reconstruction and CMS event classification. CMS event reconstruction refers to the Compact Muon Solenoid Software framework (CMSSW) data processing pipeline aiming to reconstruct a full LHC collision event. CMS event classification is an analytics workflow, which aims to train several Machine Learning (ML) models and perform a multi-event classification

7.2 Application structure

7.2.1 CMS event reconstruction

The process of reconstruction consists of three consecutively applied stages: digitization, local and global reconstruction. Each stage has a mix of GPU and CPU based algorithms.

7.2.1.1 Digitization

Upon recording the response of the CMS detector, physics data is packed in a highly efficient binary format that requires unpacking before it can be dealt with. The actual content of this format is the raw electrical signals that correspond to the amount of digitized charge. Digitization is the first phase in the reconstruction chain.

7.2.1.2 Local reconstruction

In order to perform physics analysis, it is necessary to reconstruct the actual physical quantities of interest. Therefore, digitized signals are converted (or reconstructed) into physical quantities such as energy, time and position. This conversion is performed on a per sub-detector base.

Local reconstruction applies to a particular component of the CMS detector (sub-detector). For instance, a hadronic calorimeter contains thousands of channels and energy deposition, within each is computed a sophisticated regression procedure. Regression algorithms are typically implemented using third party libraries (e.g. Eigen) which incorporate optimised linear algebra routines. However, certain functionality has to be manually ported to CUDA/OpenCL in order to preserve the algorithm itself and utilize the heterogeneous resources provided with the MSA.

7.2.1.3 Global reconstruction

Global reconstruction is the process of combining information from several components of the CMS detector in order to build high-level physics objects such as electrons, photons, jets, etc.

This operation drastically improves the precision of the measurements of properties of high-level objects.

7.2.2 CMS event classification

A typical Machine Learning (ML) pipeline consists of three phases: feature engineering, model training (including cross-validation) and evaluation (inference).

7.2.2.1 Feature engineering

In a typical ML application, input data does not correspond one-to-one to the model's input. Therefore, a certain transformation algorithm has to be applied in order to prepare the input in a certain format. Apache Spark is used to perform Extracting Transforming and Loading (ETL) operations. The transformation involves taking collections of various particles (photons, electrons, etc.) and building an abstract two-dimensional image representing an event.

7.2.2.2 Model training

The model training phase is usually the most time-consuming part of the analytics workflow. At the current stage, GPUs provide the highest performance.

7.2.2.3 Model evaluation

Upon completing the training phase and finding the appropriate hyper parameters, inference is performed. The input data needs to be split at the previous stage so that a classifier does not see the data on which the inference is to be performed. The goal is to find the model giving the highest classification accuracy.

7.3 Application mapping

CMS event reconstruction workflow is a completely data parallel workload, where each event is independent, therefore the distribution of processing across MSA is quite trivial, i.e., there is no communication. Each node processes a completely different set of events and produces output data products. Within the DEEP-EST project, several time consuming parts of CMSSW (i.e. Hadron and Electromagnetic calorimeters) were identified and ported to utilize NVIDIA GPUs. Figure 7.1 below provides a basic overview of the distribution strategy. The idea is to use all the available resources and if possible the more performant one, i.e. if both CPUs and GPUs are available, the latter are chosen to run the codes parts that support them. It is important to note that overall adapting CMSSW to heterogeneous computer architectures is an ongoing activity.

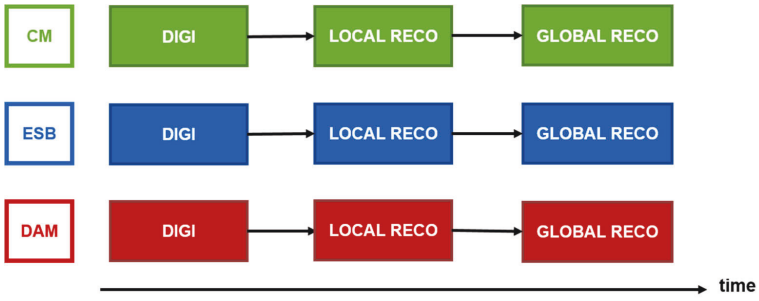

Figure 7.1: Schematic workflow of the CMS event reconstruction in the MSA

CMS event classification workflow (Figure 7.2) is a distributed deep learning training workflow that utilizes PyTorch for the training part. The distribution is implemented using the NNLO package[88], which uses MPI to communicate the weights. Furthermore, it also incorporates the use of Horovod for the purpose of distribution and communication to enhance the more basic Master-Worker approach implemented in NNLO package. More specifically, the model tested out on the DEEP-EST prototype

[88] https://github.com/vlimant/NNLO

is called JEDI-net, which stands for Jet Identification algorithm based on interaction networks. Jets are typically thought of as collimated cascades of particles which are abundant in hadron collisions, such as proton-proton collisions at LHC. Within the CMS event classification, we employed the JEDI-net neural network, which is trained to identify different types of such jet clusters.

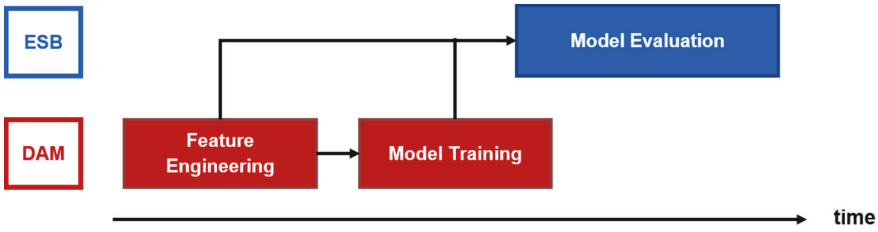

Figure 7.2: Schematic workflow of the CMS event classification in the MSA

7.4 Porting experience

For the CMS event reconstruction, the main porting effort was adapting the CMSSW framework to run on GPUs and optimizing selected time-consuming workflows to NVIDIA V100 in particular. At the start of the project, CMSSW contained CPU-based algorithms only, and there was minimum machinery available that had to be implemented in order to optimize it for heterogeneous resources (e.g., minimize host-device transfers, minimize device memory allocations, etc…). Furthermore, since CMSSW is a framework by itself, it already had certain intrinsic architectural choices and it was important to evolve without breaking the existing infrastructure.

The algorithms to be ported were those on the most time-consuming parts of the code. Figure 7.3 shows a breakdown of how much time is spent in a particular algorithmic part. Hadron and Electromagnetic calorimeters (labelled *HCAL local reconstruction* and *ECAL local reconstruction* in the figure) are two similar parts of the reconstruction that constitute around 24% of the total. The core parts of both algorithms were mathematically identical and employed Fast NNLS for the purpose of energy regression, although there was still quite a large amount of source code porting that completely differs for respective calorimeters. Therefore, these calorimeters were selected for porting and optimization to CUDA.

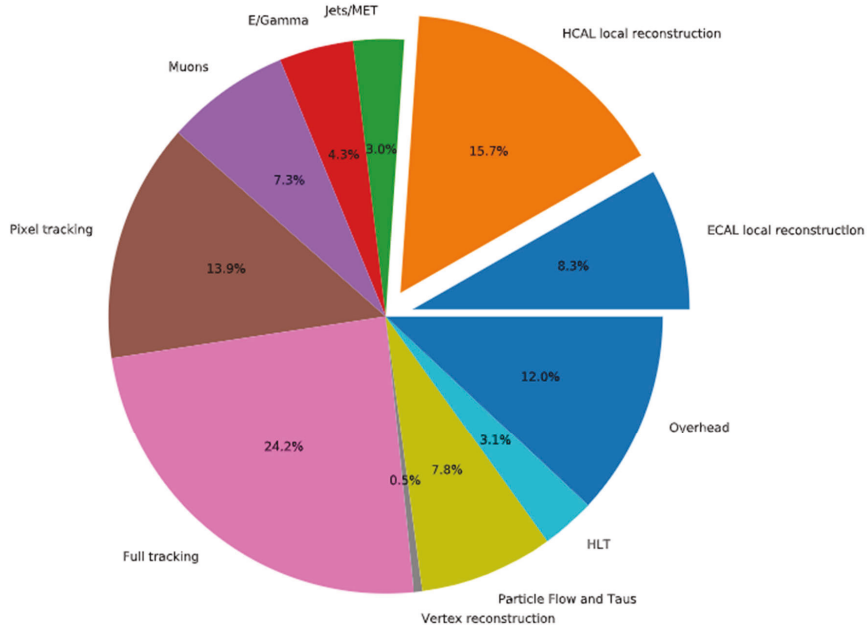

Figure 7.3: Time spent in different algorithmic parts of CMSSW

Before describing the necessary changes and efforts, we outline the problems faced. There are two main issues. First, the pure size and complexity of the source codebase. It is, of course, possible to use isolated small mockups, but then transferring the results to the CMS experiment and its community would be very difficult. The initial code for either Hcal or Ecal calorimeters amounted to O(10K) lines each. Even if isolated computational parts produce the largest impact, it is crucial to stress the importance of integration for software in the scale of CMSSW. Second, the existing software stack was written years ago (HEP experiments tend to last decades) and there was limited documentation about the actual algorithms and time vs. space complexities. In other words, the algorithms first had to be reverse engineered in order to deduce all of the available parallelism, which is quite an important aspect when dealing with GPUs.

The efforts required to port CPU-based source code to CUDA vary depending on the nature of the algorithms. For instance, if there are for-loops with large independent computations per iteration (i.e. data parallel), this trivially maps to CUDA kernel invocations, provided that all the code inside of this for-loop is supported by CUDA (e.g., the C++ version matters) or can be made supportable (e.g. mark functions as

either `constexpr` or `__device__`). This was actually the case with Hcal and Ecal source codes. However, the problem with this approach is that, although trivial to port, it might not lead to the desired performance (e.g., because memory was not aligned to suit GPUs) and the optimizations needed quickly became a full code-rewrite. Nevertheless, we believe that such a simple approach is a very good starting point, especially for people who are not experienced with CUDA and NVIDIA GPUs.

Therefore, here we outline the steps performed to arrive at an implementation that proved to be a good starting point for further optimizations:

1) Profile/Trace CPU code to identify hot spots.
2) In parallel with 1) deduce (reverse engineer) the algorithm and identify all the available parallelism. This also requires reasoning about how to align data in memory to better utilize GPU's compute units.
3) Typically, as the result of identifying the available parallelism, it will be apparent how many kernels are required and what the dependencies are between them.
4) Implement the required kernels for the GPUs.

Once point 4) is completed and implementations of separate kernels are available, it is crucial to evaluate these unoptimised versions to make sure that results (in the case of CMS event reconstruction it is physics quantities like energy) are validated with respect to what was obtained using CPU-based reconstruction algorithms. This allows to debug the ports early on before starting the optimization, which very often requires rewriting some compute-heavy routines, e.g., various mathematical operations. At this point, it is important to add that both CMS Hcal and Ecal CPU-based algorithms employed the Eigen library for the linear algebra computations. Although Eigen does feature some CUDA support, it did not cover the routines that were used in our algorithms. Therefore, we extended the functionality that was required to make the code work on GPUs (a very similar procedure was later applied to enable using Eigen from within Intel oneAPI kernels and is described below).

Once we verified that results of CPU vs. GPU reconstruction either match or are within certain precision (note that results of floating point computations on CPU and GPU can differ: they should not differ dramatically, but differences at certain tiny precision could be expected and were observed), we employed NVIDIA profiling tools, NVIDIA Nsight Systems and Nsight Compute. The first one, Nsight Systems, gives an overall system level view of what runs on a single node and identifies kernels that are the most time-consuming and therefore would be the first ones to undergo further optimization. For the purpose of kernel-level optimization we employed NVIDIA Nsight Compute. The most time-consuming kernel (originally ~90% of the total time) was the kernel responsible for the actual Fast NNLS energy regression. There are many potential approaches with regard to what to look for and how during the optimization. It is also

important to consider the amount of resources to be utilized during the execution – in other words, we could give more GPU resources to a kernel, but given that we run multiple streams (multiple events are reconstructed and run the same kernels but on different data), the overall performance would not go up, just the performance per single kernel/stream. Therefore, for the purpose of optimization, we tried to keep the amount of resources used per kernel fixed and minimize the runtime of this kernel when running a single CUDA stream. One of the first things we did was to rewrite the majority of Eigen operations, which were generating quite large stack frames. Also, given that Eigen is a header-only template-heavy library, it featured quite deep function call stacks. This made it very difficult to use the sampling-based features of Nsight Compute, which can help navigate to lines of code sampled more often and identify reasons for pipeline stalls, e.g., due to memory dependencies. Another important optimization was to not just use shared memory, but rather reuse it for different stages of the kernel execution, reducing the stack frame size of the kernel. All of that allowed, in turn, to reduce the number of registers used per thread, which is really important for parallelism of warps (more warps could be running on a given streaming multiprocessors (SM).

Before moving forward to describe the experience with our second application, we would like to outline our experience with Intel oneAPI as an approach for portability, which is based on the SYCL C++ language extension. First, a few words about why one would need such a layer, using the CMSSW framework as an example for this discussion. Consider that we rewrote and optimized substantial amount of C++ to CUDA (O(10K) lines of code). For the CERN/CMS collaboration, this essentially implies that these parts can only be run on nodes equipped with NVIDIA GPUs – in other words we are stuck with the choice of the vendor (critical for large collaborations such as LHC experiments). Of course, within the project we are optimizing for the given DEEP-EST prototype, but CMS will always strive to exploit as many compute resources as possible, therefore locking into a single vendor is not optimal, especially considering that there will be machines with AMD and Intel GPUs in the future. Another aspect is that given large source bases, rewriting parts for different accelerators could take months, which implies a lengthy development process, sometimes requiring some reengineering effort as some accelerators might not expose the same primitives. Overall, portability frameworks should prove useful in particular for large-scale scientific software development targeted to run on accelerators, provided the performance drop is minimal when using a portability framework with respect to using the native toolchain.

For the purpose of performing a minimal evaluation of Intel oneAPI, we used a standalone electromagnetic calorimeter reconstruction implementation that does not require the CMSSW framework and was ported from plain C++ to CUDA in the

beginning of the project. In short, it was a rather easy process to turn a CUDA-based implementation into oneAPI-compatible one. Here are the steps performed:

1. Employ the Intel DPC++ Compatibility Tool to transform our CUDA-based implementation into DPC++ compliant;
2. Fix all the issues raised by the Compatibility Tool;
3. Fix all the issues that come up during the compilation/linkage stages.

The very first step is probably the easiest one here, basically we just need to run a single command with all the initial host/device sources, and the Compatibility Tool will produce new sources that are now based on oneAPI. This conversion takes care of things like device memory buffers, allocations and transfers. CUDA streams and kernel invocations are mapped to the usage of queues and command groups. In terms of error handling, CUDA uses C style by reporting errors through error codes, whereas oneAPI is more C++ like and uses exceptions. The conversion between both is automatically handled by the tool as well, and step 2 above essentially refers to fixing whatever the compatibility tool was not able to convert. Kernels that use certain features specific to NVIDIA GPUs, for instance Tensor Cores, must be reimplemented. This is actually an important point overall and will require further investigation. For our small standalone evaluation we did not observe any difficulties after the conversion – the code was converted almost to 100% and required minimal changes. The third step was a bit more involved for us and should be of interest to other developers. Our implementation makes significant use of Eigen's primitives and therefore it is important that Eigen's routines work inside of the kernel and are supported. We found a couple of things that had to be adapted; first, Eigen is a header-only template-heavy library, therefore it utilizes advanced features of C++ templates and also makes heavy use of macros to configure the compilation process. When compiling things with DPC++, similar to when using the NVCC compiler, there are essentially two modes of compilation: kernel and host modes. The idea is that pre-processor directives will be configured differently based on the mode. The host mode in Eigen is for regular CPU execution and kernel mode is what sits inside of our oneAPI kernels. The main issue overall is that there is a set of restrictions applied to the code that goes into the kernel part, which can be cumbersome to fix when trying to use functionality from some library in your kernels. For the case of Eigen, one of the most difficult features to resolve, we had to disable dynamic stack allocation explicitly and also make sure that inline assembly instructions are not added, not even in comments. For the most part, we utilized the Ahead Of Time (AOT) compilation flow of Intel oneAPI toolchain. One of the minor drawbacks of the kernel mode compilation is quite poor error reporting: just reporting the error without explicitly stating parts of the code causing it makes it really difficult to debug, especially if trying to port a library one is not the author of.

In terms of results, we compared an implementation that is pure C++ based with one that is oneAPI based. For the purpose of evaluation, we just used a Virtual Machine (VM) from CERN's cloud. We have been able to reproduce exactly the same physics results (energy) and performance-wise the two versions were comparable, considering that oneAPI was able to utilize the multi-core VM and the plain C++ was written having single thread execution in mind.

For the CMS event classification, there was minimal porting effort as it is mostly an ML-based workload and does not require changing a lot of application logic in the code, compared to the CMS event reconstruction workload, where we had to produce O(20K) lines of code from scratch in order to have CMSSW-compliant implementations. However, here emphasis is more on the system integration and on the proper installation and configuration of the required software. For instance, Horovod requires a multithread aware version of MPI, which might be unavailable by default. Furthermore, the usage of the underlying communication method is quite important, i.e. using RDMA versus TCP for MPI. In other words, proper usage of the system is crucial for workloads that perform Deep Learning and similar activities, and a lot of time can be spent identifying why certain things do not perform as expected even if the actual porting of the code was trivial.

Overall, approximately 24 PMs were spent doing the actual development, porting, testing, validation for both of the applications combined. In terms of the sheer code size, around 20K lines of code were developed just for CMS Hcal and Ecal Local Reconstruction as the final footprint, not including the iterations involving the optimizations. These numbers do not include scripting side code for the purpose of analysis and benchmarking of applications.

7.5 Scalability

As it has already been emphasized, the CMS event reconstruction is a data parallel workload. As a consequence, there is no real communication across compute nodes. Therefore, the critical metric for this application is weak scaling because our goal is to use as many resources as we can get on the system without degrading the performance per unit. In addition, the CMSSW data processing constantly requires getting more and more input data (when running in production), by reading either from shared storage or from a remote location, and also produces output. This input/output data flow is where the bottlenecks for scalability are lying at the moment, and this is currently being investigated outside the DEEP-EST project.

Figure 7.4 shows how, by using all three types of compute nodes (CM, ESB, and DAM) and almost all of the corresponding nodes, the throughput increases. This figure does

not demonstrate the scalability, but it allows to view how adding more nodes does increase throughput and also dissects the contributions of different types of nodes.

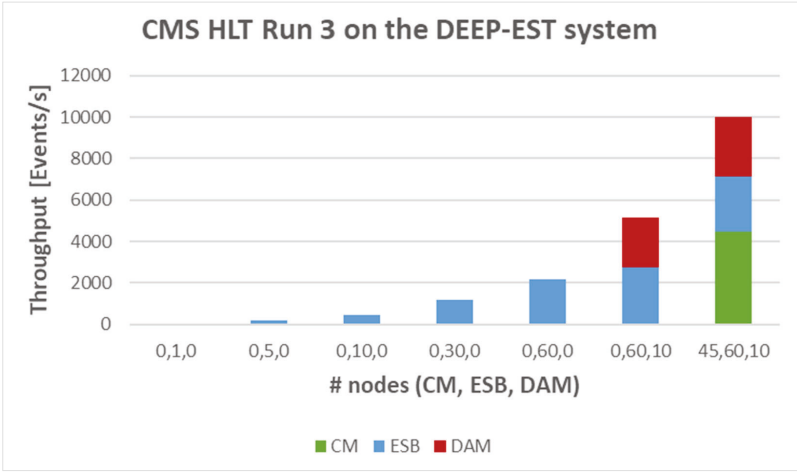

Figure 7.4: CMS reconstruction using the whole DEEP-EST system

For the purpose of demonstrating weak scaling for the CMS Event Reconstruction workflow, we essentially tried loading as many available nodes as possible, in the expectation that performance per node does not degrade. Throughout all the measurements we used the BeeGFS shared storage system for storing/ingesting input data. Figure 7.5 shows the distribution of throughput when employing all the CM nodes. As we can see, there are very few outliers and for the most part performance per node is quite stable.

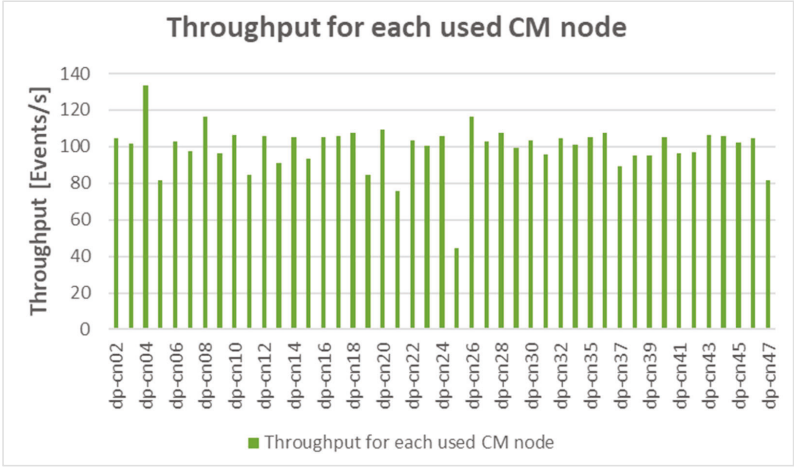

Figure 7.5: Throughput distribution by node

CMS Event classification is a typical distributed ML training workflow with a strong scaling objective. Figure 7.6 shows the execution time as a function of the nodes used for training. The training was performed using the ESB nodes. The most important outcome of these measurements is the fact that this distributed training workload shows good strong scaling features when using more and more nodes. In particular this is important when we compare running training on ESB to other systems that contain special NVIDIA Inter-GPU links and other optimizations. Employing the ESB we can scale up quite flexibly the number of nodes available for training.

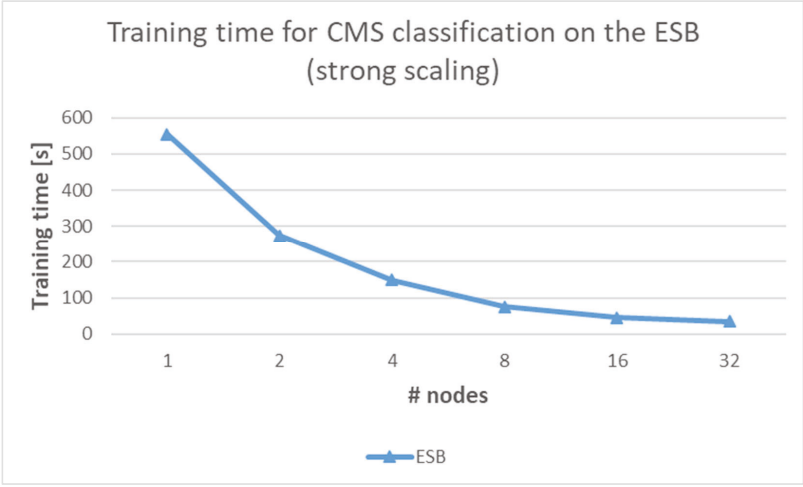

Figure 7.6: Training time for CMS event classification on the ESB

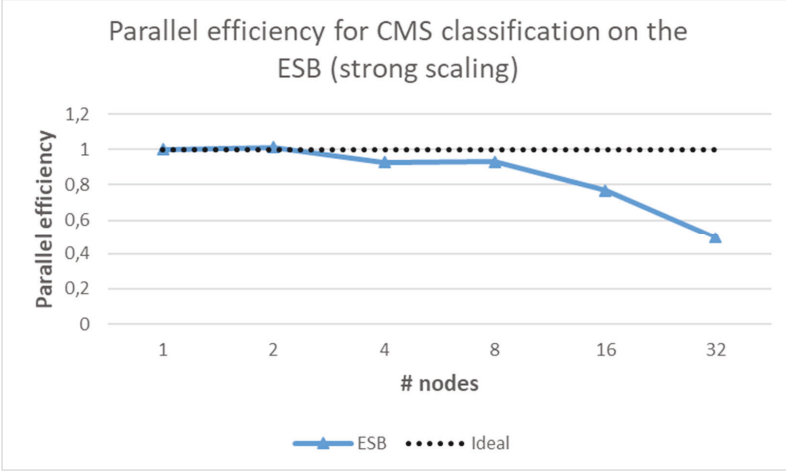

Figure 7.7: Parallel efficiency for CMS event classification on the ESB

7.5.1 Our path to Exascale

Overall, we think that it is important to understand first what the future is going to look like before addressing our path towards that future. For LHC, the CMS experiment and HEP in general, the future will bring significantly more data and more complex structure of collision events, both of which in turn mean that physicists have to find more efficient ways of handling these massive amounts of information. To quantify this, more data means processing O(500PB) by the CMS experiment per year. These numbers are obtained using the existing production level workflows.

Within the DEEP-EST project we tackled the very first part of this journey – making our software more efficient by using heterogeneous resources available (or soon to be available) at HPC facilities. Our contributions have been integrated into the CMS Experiment's framework and will be used in production starting with this year's production campaign. It is crucial to note that these algorithms are not just running at an HPC centre in an offline manner (where someone launches jobs and collects results), but are also used at CMS experiment's online farm during the data-taking, in order to identify events of interest for the future physics analysis, which means that they must be robust, performant, and error-free all the time. Heterogeneous resource utilization is one of the key features for any scientific software when targeting Exascale as it allows us to make the code more efficient. For CMS, it means higher throughput per single node.

For the purpose of data processing, all large scale LHC experiments, CMS included, utilize the world-wide distributed computing grid, which allows us to perform data processing across many different computing sites. The reason for this distribution is that a single site would not be able to cope with requirements imposed by HEP workflows. Furthermore, by dividing the processing infrastructure into multiple sites, the movement of data has to be properly handled. This problem of having to move around O(500PB) data per year is one of the central challenges of the path to Exascale for the CMS experiment.

7.5.1.1 What are the limitations? – Can they be fixed?

From the perspective of an LHC (or probably even High Luminosity LHC) experiment, it is not a question of limitations, but rather a question of what has to be developed in order to unleash the full scientific potential of the upgraded detectors when moving to High Luminosity LHC. As was mentioned for the CMS event reconstruction workflow, it is necessary to keep ingesting data and also storing the output data products, which means that I/O subsystem could be seen as a limitation at Exascale. Therefore, here are two important areas of work that are currently being investigated by the whole HEP community:

- At Exascale it is important to consume computing resources without degrading the performance per node. There could be two potential limitations in here: the network and the storage itself. For instance, there could be too many clients trying to do I/O from the shared storage system. It is true that data could be pre-staged to the compute nodes first and then processed, but again the limit on the network could be reached with the huge number of nodes at Exascale. On the other hand, there are new types of shared storage systems (e.g., object storage) that are significantly more robust for applications that heavily utilize the I/O subsystem. To run production-type of workflows these “limitations” need to be further tested and resolved.
- The previous point covered the network and storage that are fully internal to an HPC facility. The next item to consider is the external connection of an HPC facility. Considering that HEP experiments cannot store all of their data at an HPC site, it is crucial that there are efficient means of enabling data flow to/from such HPC centres. There are two main reasons why all of the data cannot be preserved at an HPC site. First, there will be as much as O(500PB) for a single experiment. Second, HEP collaborations by themselves are quite large entities and data collected is one of their main products, therefore storage and preservation of this information is of crucial importance to HEP. Delegating this responsibility will not be feasible. Overall, this means that it will be necessary to find efficient ways to enable dynamic and scalable flow of data to/from HPC centres. Such a system would need to be capable of:
    - Overlapping processing with bringing in more data;
    - Talking to the outside world: request more data, inform of what is missing, etc.;
    - Being aware of what is running, which data can be purged, what has to be requested and brought in or taken out.

Overall, this requires having a system in place that runs at an HPC site and handles the external data flow activity.

7.5.1.2 How to use future Exascale systems

Overall, the way for HEP community to exploit Exascale systems is by maximising the efficiency every available compute node. Having in mind that our target is weak scaling, we have to stay efficient when scaling out. This is mainly achieved by embracing the usage of accelerators and software reengineering, which has been successfully done within the DEEP-EST project for the CMS experiment’s workflows.

Another important aspect to stress is data. We will not have less data in the future, only more. And this applies to many other data driven sciences, not just HEP (e.g., SKA,

see Section 4 of this volume). The network bandwidth, although increasing as well, will not keep up with the pace of data volumes. This is in particular crucial for the large scale experiments that cannot store their data at an HPC site. The dynamic component that is responsible for bringing data in and taking processed output outside of an HPC centre in a scalable fashion will play an important role for such applications. We would also like to note that this is an area of work for HPC centres as well, as they are aware of the shifting requirements of the applications that would like to utilize their facilities.

7.5.1.3 Where did the DEEP-EST project help on the way to Exascale?

Within the DEEP-EST project, one of the more crucial development contributions overall was the porting of 20-25% of CMS High Level Trigger (HLT) to utilize NVIDIA V100 GPUs. There are several reasons why this contribution is important:

- The algorithm implementations developed in DEEP-EST will be running in data-taking production for the CMS experiment starting this year. They are not just performant, but they reproduce physics results with good precision, therefore having no effect on downstream physics performance.
- The very positive experience serves as motivation for other members of the CMS collaboration to join the effort of code modernization and porting to heterogeneous architectures. In parallel to the CMS Hcal/Ecal ports, the software from another detector was ported by other members of the CMS collaboration, which allowed developers to interact and discuss ideas for implementations and optimizations.
- Finally, developed functionality will not just be running for the CMS HLT, but also for offline production workflows, which will be utilizing HPC resources. Through this effort, we increased the efficiency (i.e. throughput) of our applications when targeting future Exascale systems.

Given that HEP community traditionally did not use HPC machines for reconstruction workflows, this work allowed to gain overall confidence that we can run efficiently these data-driven types of workflows on such large machines, not just simulation workloads that do not really require any input data. Another aspect is the ability to test out our production workflows when running on larger node counts using a prototype system, not a production one – we are able to tweak things, test, change, and do it again, which is quite important during the development life cycle. This applies to data analytics type of workflows even more, as these require tweaking the configuration to find the appropriate parameters on a given system.

7.6 Energy consumption

Energy consumption is an important metric in particular when thinking of Exascale workflows. It is clear that in order to be sustainable, we cannot just think of performance without addressing the issue of power consumption, as well as costs associated with resource utilization. For HEP production types of workflows, with a weak scaling objective, it is important to be as efficient as possible per node, given the tendency to consume as many resources as there are available. This is where software reengineering potentially helps not only performance but also energy efficiency.

All the energy utilization metrics were collected using various DEEP-EST prototype sensors, which allow for frequent probing of quantities, which in turn enables fine-grained downstream analysis. Figure 7.8 shows the total energy consumed when running the full CMS Event Reconstruction workflow on CM nodes (green) and ESB nodes (blue) as a function of the number of nodes used for the reconstruction.

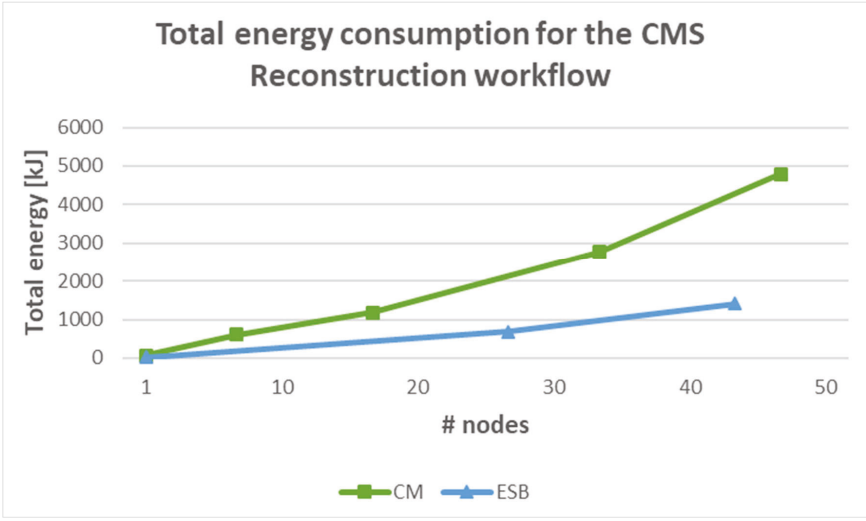

Figure 7.8: Total energy consumption running for the CMS Event Reconstruction

Figure 7.9 in turn shows the same values but averaged over the number of nodes. Given that we are after stable resource utilization when scaling out the number of nodes, here we observe a slight increase when going to 32 and 49 nodes on the CM. This is not too dramatic and could be a result of the outliers observed in Figure 7.5.

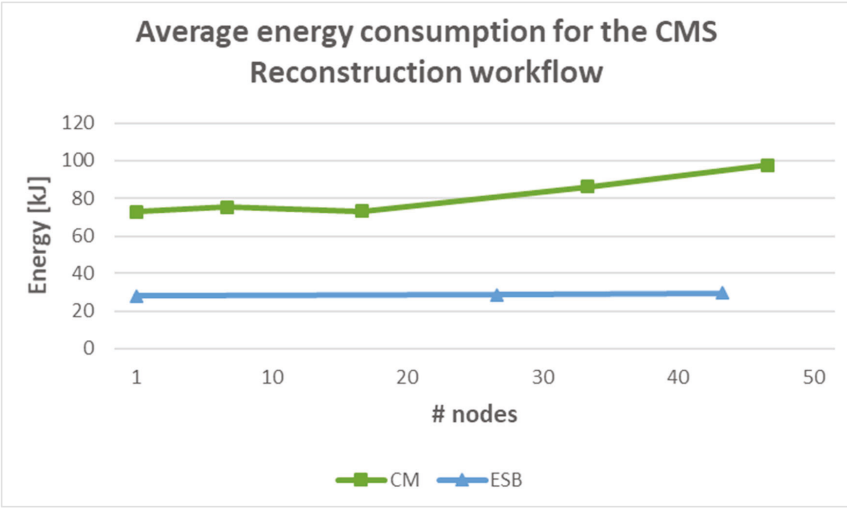

Figure 7.9: Average energy consumption for the CMS Event Reconstruction

When employing NVIDIA GPUs for the CMS Event Reconstruction, the ESB shows nearly perfect linear scaling in total energy when scaling out the data processing from 1 node to 48 (blue line in Figure 7.8). The averaged energy utilization per node, which is the more relevant metric for the workload that aims to scale weakly, is shown in the blue line in Figure 7.9. Overall, here we observe stable resource utilization per node when scaling out our production workload to many nodes equipped with NVIDIA V100 GPUs.

7.7 Performance comparison

7.7.1 CPU vs GPU

For the purpose of performance comparison we were trying to evaluate the maximum throughput, defined in terms of events per second that could be achieved either on CPU or GPU. Although the CMS software framework by itself is multi-threaded, the original CPU-based algorithm implementations are single threaded (i.e. task-based parallelism), therefore they basically scale with the number of available cores (provided there are no other limitations). However, for the case of a GPU-based implementation it is not so straightforward and we essentially are trying to push as many concurrent events into a single card as possible until we reach a limit. Figure 7.10 shows the result of comparing the CMS Ecal Local Reconstruction using NVIDIA V100 vs. using 2-socket Intel Xeon Gold 6148 (32 cores total). The reason for comparing against this particular model of the CPU is that most of the comparisons which we make when

targeting heterogeneous hardware architecture are done against models similar to the ones currently employed at CMS High Level Trigger (HLT). Here we observe a factor of 3-4× speedup with respect to the CPU version.

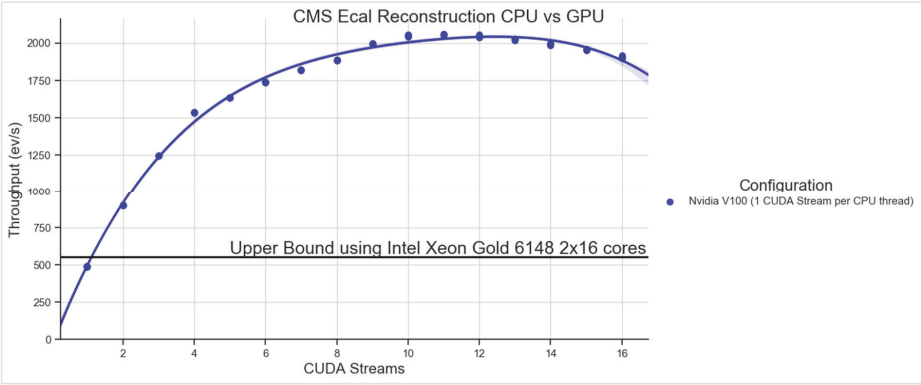

Figure 7.10: CPU vs GPU for CMS Ecal Reconstruction

Similarly, for Hadron calorimeter reconstruction, Figure 7.11 displays the comparison of throughput when running the CMS Hcal Local Reconstruction on NVIDIA V100 GPU vs 2-socket Intel Xeon Gold 6148 (32 cores in total). Here we observe factors of 7-8× speedup when comparing to the baseline CPU version. This could come as a surprise, given that we indicated that the Hcal and Ecal algorithms share the same core routines. However, as mentioned above the core routines, although crucial, are not the only element needed for these algorithms to run successfully within the CMSSW framework. Furthermore, most of the optimizations were first tested using Hcal workflow and then applied to Ecal, which means that Hcal was more heavily optimized. Also the percentage of time spent in different kernels is slightly different for Hcal and Ecal. All this leads to slightly different speed up factors.

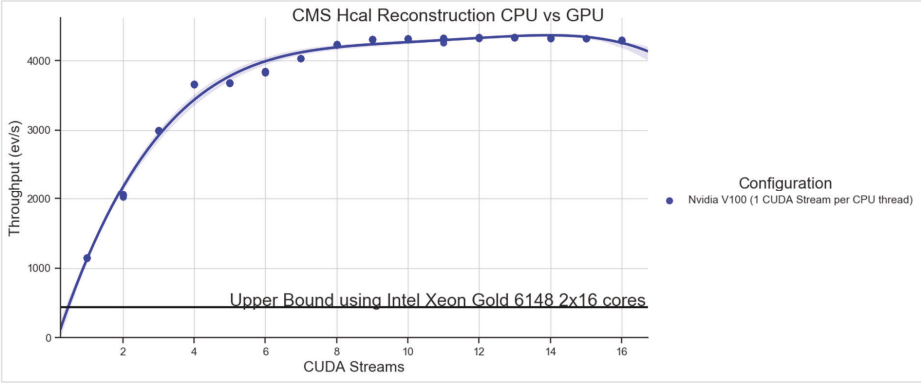

Figure 7.11: CPU vs GPU for Hcal Reconstruction

Finally, one of the more interesting things to test was to use the full CMS HLT workload on each type of the nodes available on the DEEP-EST prototype. This workflow consists of running O(1000) algorithms including Ecal and Hcal. We used the CMS Open Dataset for the purpose of these measurements. Figure 7.12 shows the throughput (events per second) when running CMS event reconstruction on different types of nodes on the DEEP-EST MSA and also either CPU only or combining CPU and GPUs.

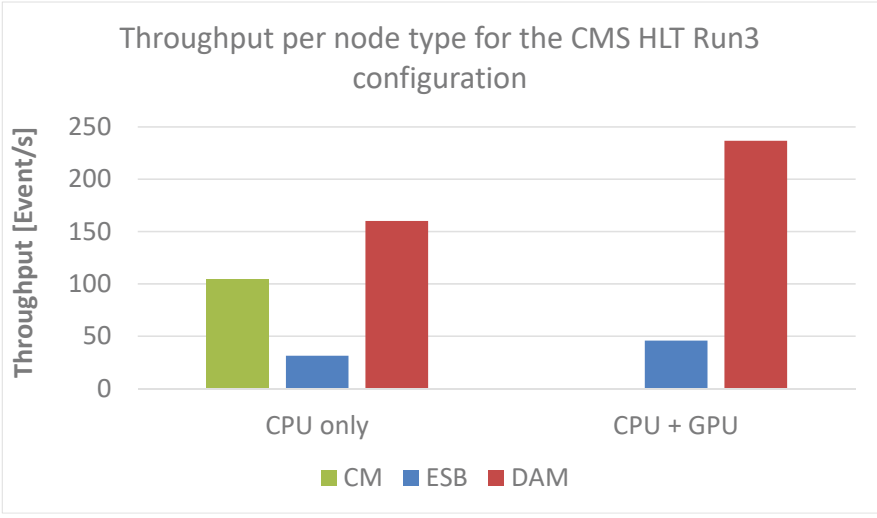

Figure 7.12: Throughput per node type on the DEEP-EST MSA

We have used CPU-only configuration on the CM nodes, but when we targeted either ESB or DAM we could employ both CPU-only and CPU+GPU configurations. With CPU+GPU configurations we achieved 50% speed up in throughput from a single node. This result is quite important to justify the advantage of porting your software to heterogeneous hardware, keeping in mind that not all algorithms do benefit from being ported to accelerators.

7.8 Conclusion

The work carried out in the DEEP-EST project paves the way for the High Energy Physics community to successfully exploit the future Exascale HPC systems for both production data processing workloads, and for analytics driven applications requiring usage of Deep Learning techniques. It is a highly non-trivial task to take such a large software stack such as CMSSW and be able to efficiently run, measure and understand the results when employing high node counts at an HPC site. Although we did not tackle issues related to the I/O subsystem, experience gained within the project about what worked well and which components are going to be required for a successful utilization of large Exascale machines is invaluable for our further investigations.

Experience gained within the project has been conveyed to other members of the CMS collaboration working on porting scientific software to heterogeneous platforms. In particular, the knowledge about NVIDIA Profiling tools, Nsight Systems and Compute, allowed us to optimize CMS Hcal and Ecal Local Reconstruction GPU implementations and to achieve significant speedups (3-4× for Ecal and 7-8× for Hcal) with respect to the CPU-based implementations.

Finally, the developed algorithmic implementations have successfully been integrated into the CMS software framework. Physics validation has been performed and GPU-based algorithms will be exploited during the upcoming data-taking campaign of the CMS Experiment at LHC.