Explorations in heterogeneous computing

August 2019

Author:
Davide Marcato

CMS Patatrack Group
INFN Laboratori Nazionali di Legnaro
Università degli Studi di Padova

Supervisors:
Felice Pantaleo
Antonio di Pilato

CERN Openlab Summer Student Report 2019

Abstract

High Luminosity LHC (HL-LHC) is set to produce in the following years an exponential increase in the amount of data to be analyzed, while the computing power is not expected to become significantly cheaper. Patatrack group inside CMS is tackling this challenge by offloading the most expensive workloads to GPUs. While this approach is providing promising results, it implies some software engineering drawbacks that need to be addressed. This report will describe some of them and how they are being solved. In particular the problems of code duplication and the dependency on fixed CUDA kernel launch parameter will be presented along with possible solutions.

The last chapter will describe a real world code porting from CPU to GPU for the HGCal calorimeter, the problems to overcome and the performance improvements achieved.

Table of Contents

1 Introduction

1.1 High Luminosity LHC

High Luminosity LCH is an upgrade of the Large Hadron Collider (LHC) and currently one of the major project being developed at CERN. The main goal is to increase peak luminosity from $1.2 \times 10^{-34} cm^{-2} s^{-1}$ obtained in 2016 to $5 \times 10^{-34} cm^{-2} s^{-1}$ in a 10 year time frame [7]. The integrated luminosity is then expected to reach $250 fb^{-1}$ per year with the goal of $3000 fb^{-1}$ in about a dozen years. For reference, all of the hadron colliders in the world before the LHC have produced a combined total integrated luminosity of about $10 fb^{-1}$, while the LHC delivered nearly $30 fb^{-1}$ by the end of 2012 [7].

This will be achieved with a series of upgrades scheduled for the first half of the 2020s, as shown in Figure 1.

Figure 1: LHC timeline showing the energy of the collisions (red line) and luminosity (green lines).

1.2 CMS Upgrade

In this context the LHC experiments (mainly CMS and ATLAS) face the non trivial task of coping with such high luminosity. The number of collisions, in fact, depend on both on the frequency of bunch crossing (which for LHC is fixed at $40 MHz$) and on the pileup. This parameter measures the number of different proton-proton collisions in the same bunch crossing. Upgrading the luminosity means increasing

the spatial density of the protons inside the bunches, thus increasing the probability of a collision. For this reason the pileup is expected to go from ~ 50 of today to a maximum of 200.

While this is good news for physics, as more events means more statistical evidence of any new measure, it poses many challenges for detectors, in particular for the trigger and reconstruction software, which must deal with a greater amount of data and with more complex events. In fact, not all the software analyses may scale linearly with the pileup, as more complex events may require much more effort to be correctly reconstructed. In the case of CMS this means that the required computing power is going to increase exponentially in the following years, as shown in Figure 2

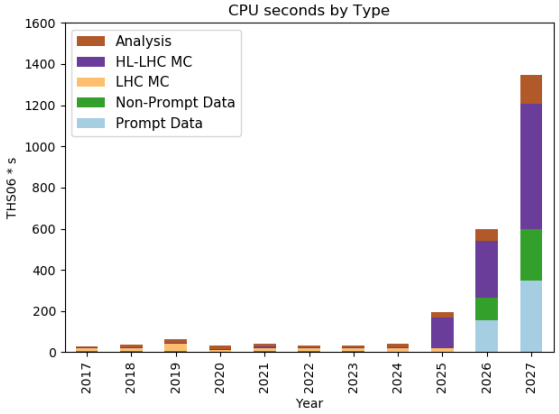

Figure 2: CMS computing requirements. In 2027 required computing capacity will be 22x the current level. Source [8]

At the same time the CERN budget that can be allocated for new computing resources is basically flat, meaning that it is not possible to scale just adding more and more hardware. Over the last few decades most of the improvements have been made possible by Moore's law, which has enabled a exponential decrease in the cost of computing power over the years. Unfortunately the trend has been slowing down during the last few years, as shown in Figure 3, as the industry leaders struggle to keep shrinking the manufacturing process towards the limit of the Silicon.

1.3 A new approach for CMS software

It is clear that a new approach was needed to solve the scaling issues of the current CMS software. The Patatrack group inside CMS has been working in these years to tackle this challenge by pushing the use of GPUs accelerators and renewing

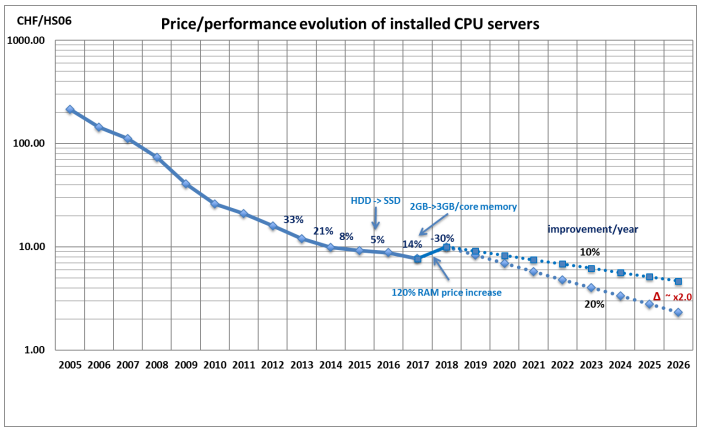

Figure 3: The trend of CPU price/performance is slowing down in the last few years. Source [10]

the focus on software optimization. This has proved to be a successful approach and has already delivered the first results. For example, in [3] a new algorithm is presented for the clustering of HGCal calorimeter which improves the performance by almost a factor of $30\times$ by exploiting a different GPU-friendly data structure and its parallel implementation in CUDA.

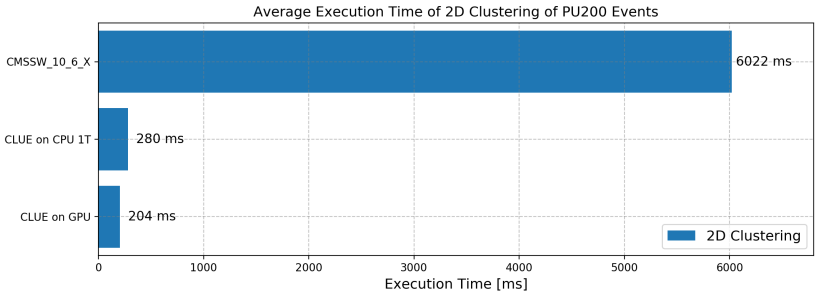

Figure 4: Parallel Clustering in CMS-HGCAL. Source [3]

While this approach is effective in solving the scaling issue, it requires a certain investment in software engineering to port a large code base to GPUs, using the CUDA library, and to do it in the optimal way. In fact, to obtain the maximum performances usually it is necessary to fundamentally rethink the whole workflow with GPUs in mind.

The following chapters will present some problems that arises when dealing with such porting and the solutions that have been pursued during the summer student period.

2 CUDA to Cupla

One of the first effects of porting an algorithm to CUDA is code duplication. In fact, the upgrade with GPUs of the Worldwide LHC Computing Grid (WLCG), where a large portions of the analysis and simulations are currently run, could require years and some nodes may still opt to remain CPU only. For this reason the CPU code, not only can't be removed, but must be actively maintained and updated. Furthermore, the CUDA coding paradigm is sufficiently different from host code that it is difficult to share common parts and this leads to multiple re-definitions of the same pieces of code.

As one can easily understand, this is far from the ideal, as code duplication leads to less maintainable, readable and upgradable software, slowing the development of new feature and wasting resources and efforts. For this reason a number of different projects are trying to develop libraries to write accelerator-independent code, allowing to write the code once, and run it on different devices like CPUs, GPUs and maybe even FPGAs or other specialized hardware. In this case, two of those libraries have been evaluated and tested: Alpaka and Cupla.

2.1 Introducing Alpaka and Cupla

Alpaka [11] is an open source `c++` library that aims to abstract different accelerator specific libraries while not hiding their functionalities. It supports host CPUs, CUDA GPUs and Intel Xeon Phis, as well as different backend libraries depending on the device, like CUDA, TBB and OpenMP. As a programming paradigm the library follows the CUDA grid-blocks-threads abstraction, with different blocks running multiple threads with can communicate inside the block via shared memory. The code is shared between all different accelerators, while the target device depends only on the compiler and its configuration.

Cupla [5] is a simpler user interface built on top of Alpaka which resembles as much as possible the CUDA API, in order to minimize the porting efforts when starting from working CUDA code. For this reason this library was chosen to be tested, with the aim of reducing at the minimum the impact on the existing code base and the training required to write new software.

2.2 Porting a first test

As a first exercise, the porting of a CMSSW test was attempted. The chosen file was `RecoPixelVertexing/PixelTrackFitting/test/testEigenGPUNoFit.cc`, which is

independent from the rest of the code base, but depends on Eigen, a third party library for linear algebra.

The main steps required to port the code can be summarized as follows:

- Replace all the CUDA-related imports and replace them with cupla ones
- Port the CUDA kernels to cupla functors. This is done by transforming the `__global__` functions to structures with a templated const `operator()`, which must have the `ALPAKA_FN_ACC` prefix and a first templated argument called acc. This is used by cupla to pass all the accelerator specific information.
- Update the kernels host side calls using the `CUPLA_KERNEL` macro.
- Transform the `__host__ __device__` functions to `ALPAKA_FN_ACC` and add the acc templated parameter, both to the function definition and call. This parameter is needed only when alpaka functions like `blockIdx` or `atomicMax`, ... are used.
- Update the build configuration to compile for host or device, or both.

For a detailed guide refer to the documentation[6] or to the Patatrack Wiki[9], where all the details of this project have been documented as a reference for future developments.

The main problem highlighted by this first exercise were mostly related to the building configuration and to the includes to use.

2.2.1 Build configuration

The versions of Cupla and Alpaka used are header only libraries, so in the build configuration the includes path must be correctly updated to add these libraries. For this example the `CXXFLAGS` have been used to directly add the path with the `-I` option, but eventually they will have to be integrated as CMSSW externals.

`scram`, the build tool used by CMSSW, also needs to updated. In fact, it automatically detects which compiler to used based on the file extensions (`.cc` for host code, `.cu` for CUDA files) and there is not the option to build the same file twice with two different compilers. As the goal is to build the same code for multiple accelerators, `scram` would need to be modified to add this functionality. For this test a symbolic link has been used to give two names with different extension to the same file.

2.2.2 Correctly include the headers

Cupla porting guide suggests to remove all the headers related to CUDA and replace them with `<cuda_to_cupla.hpp>`. In practice, we found that we needed to include also device specific headers (eg: `<cupla/standalone/CpuSerial.hpp>` for CPU and `<cupla/standalone/GpuCudaRt.hpp>` for CUDA). Since we want a single source file that can be built for both devices, we created a `cms_cupla.h` header with includes one of the two depending on the target device.

cms_cupla.h

```
#ifdef FOR_CUDA
#include <cupla/standalone/GpuCudaRt.hpp>
#else
#include <cupla/standalone/CpuSerial.hpp>
#endif
#include <cuda_to_cupla.hpp>
```

By including this file instead of `<cuda_to_cupla.hpp>` we managed to use a single header file for all the target devices. This requires to declare the `FOR_CUDA` macro in the build configuration when building for GPU.

When using external libraries which uses CUDA internally, as Eigen, the order of the includes is important. We realized that by including Eigen after cupla, the macro definitions of cupla completely broke Eigen, thus it is important to include it earlier. This is something to keep in mind when considering moving a large code base with many dependencies as all of those may have to be patched to be cupla compliant.

2.2.3 Results

This first example proved the feasibility of porting a piece of code from CUDA to cupla, but still highlighted multiple problems to be solved. The code builds and runs correctly even though many warnings are still present.

Performance wise no difference between CUDA and Cupla is expected, as the binary file (when built for GPU) should be theoretically the same. The first measures seems to confirm this hypothesis, as seen in Figure 5, but eventually more testing is required, in particularly using more compute intense examples.

2.3 Porting production code from CMSSW

After the first example described above, the porting of a plugin from `RecoLocalTracker/SiPixelClusterizer` was attempted.

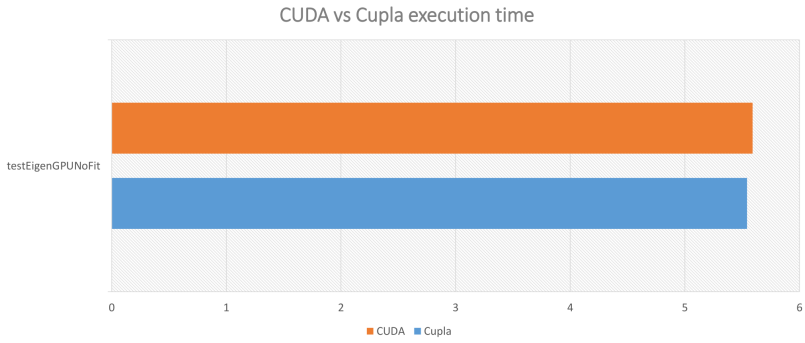

Figure 5: Performance are not affected by Cupla abstraction

The difference with the previous one is that now the code has much more dependencies spread all over the CMSSW code base. This means that to build just one file, many different parts of the project had to be updated. In practice the first step of the conversion process is to identify all the included files and apply the porting procedures as explained above. Most of the work in this first phase can even be automatized with a script, as shown in [9].

2.3.1 Problems to overcome

During this porting many problems strictly related to the CMSSW code base arouse, eventually leading to a stop where it was no longer possible to continue before a broader re-evaluation. Here the major problem are presented, along with the possible solutions to be adopted.

- Some specific CUDA functions are not available on Cupla
  - For example `__syncthreads_or()` and `__syncthreads_and()`
  - But they are defined in Alpaka, so one could extend Cupla to define them

```
#define __syncthreads_or(...)
 ::alpaka::block::sync::syncBlockThreadsPredicate
 <::alpaka::block::sync::op::LogicalOr>(acc, __VA_ARGS__)
#define __syncthreads_and(...)
 ::alpaka::block::sync::syncBlockThreadsPredicate
 <::alpaka::block::sync::op::LogicalAnd>(acc, __VA_ARGS__)
```

- Other functions are not available even in Alpaka
  - For example `cudaHostAlloc()`
  - Opened an issue on github and received a possible workaround

```
#define cudaHostAlloc(...) cuplaHostAlloc(__VA_ARGS__)

CUPLA_HEADER_ONLY_FUNC_SPEC
cuplaError_t cuplaHostAlloc(void **ptrptr, size_t size, unsigned int flags) {
#if ALPAKA_ACC_GPU_CUDA_ENABLED == 1
#undef cudaHostAlloc
// if compiling for CUDA, use the native allocation functions
auto tmp = (cuplaError_t) cudaHostAlloc(ptrptr, size, flags);
#define cudaHostAlloc(...) cuplaHostAlloc(__VA_ARGS__)
return tmp;
#else
// otherwise, use cuplaMallocHost
return cuplaMallocHost(ptrptr, size);
#endif
}
```

- `cudaCompact.h` tries to do the same job as cupla.
  - To be avoided
  - Some keywords are reserved by Alpaka
    * `blockDim, gridDim, elemDim, blockIdx, threadIdx, dim3 ...`
  - Rewrite the code base to be cupla compliant. For the moment the code from that file has been removed with an ifdef.

```
#if !defined __CUDACC__  &&  !defined CUPLA_KERNEL
```

- `cuda/api_wrappers.h` is not compatible with cupla.
  - `cuda::throw_if_error()` is undefined.
  - `cuda::stream_t` is undefined.
  - Some part could be easily rewritten, but we can't afford to completely rewrite and then maintain a third party library.

The takeaway from this exercise is that before porting such a complex code base to cupla, it is necessary to remove all third party CUDA dependencies which are not cupla compliant and to reorganize the code with cupla in mind.

3 CUDA kernel launch parameters tuning

One of the engineering challenges when dealing with CUDA code bases is to make sure to achieve the maximum performance in every scenario. In fact, one line of code written today and optimized for today hardware could end up executing in ten years in a completely different GPU architecture. This is particularly true in this field, where the lifetime of any project is very long compared to the speed of technological improvement. For example for CMS, much of the code that is being developed now is required for run 4, starting in 2026. In this time-frame GPU architecture could evolve drastically and the data centers where the software will run may not even be planned yet.

One of the key factor to be sure that your code can exploit at his maximum one specific GPU is to tune the kernel launch parameters. These are used to define the level of parallelism of the execution, segmenting the operations between multiple blocks in a grid, and in multiple threads inside a block. The parameters should be chosen so that the GPU streaming multiprocessors are sitting idle waiting as little as possible, or in other word to achieve the maximum occupancy. Usually this is done by accurately profiling an application and tuning the parameters for one specific GPU.

To be able to solve the problem as in the best possible way for many different GPUs and even for future hardware, we tested the use of a specific CUDA function: `cudaOccupancyMaxPotentialBlockSize`. Given a certain kernel, this function returns the maximum block size and the minimum number of block to achieve maximum occupancy of the GPU. Since this is evaluated at run time, in theory the parameters should be always the best possible for the available hardware.

This function was used, as shown in the following code snippet, to automatically select the ideal block size and instead deriving the number of block from the number of elements to be computed.

```
1 int threadsPerBlock = 512;
2 cudaOccupancyMaxPotentialBlockSize (nullptr, &threadsPerBlock, RawToDigi_kernel);
3 const int blocks = (wordCounter + threadsPerBlock - 1) / threadsPerBlock;
```

To test if this was in fact the case a benchmark of CMSSW was run before and after applying all the function on all the possible kernels. The first run was performed on a Nvidia Tesla T4, the current target device of the software, showing no speedup but also no slowdown. This means that the parameters were already well optimized for this graphic card. When running the same code on a fairly different card, the Nvidia GTX 1080ti, a small but clear improvement could be measured, as shown

in Figure 6.

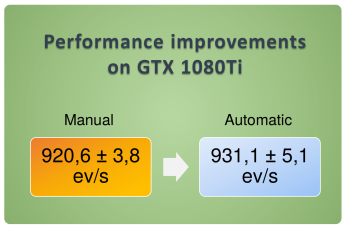

Figure 6: Automatic kernel launch parameters performance on GTX 1080ti.

While this is a small overall improvement, it is nonetheless important as it didn't require any algorithm or data structure improvement, and should now scale on different GPUs without effort.

3.1 Other approaches

Two further approaches have been investigated to reduce the overhead due to the kernel launch time: launch bounds and CUDA graph.

Launch bounds are a way to inform the compiler of the maximum number of threads per block and the maximum number of block that will ever be used to launch a kernel. By adding this information to the kernel declaration, the CUDA compiler is able to optimize the number of register used by the kernel in order to fit the desired kernel in memory, and thus enhancing performance. In practice, minor improvements have been observed, and this technique has been discarded in favour of a more dynamic assignment of the parameters, as described above.

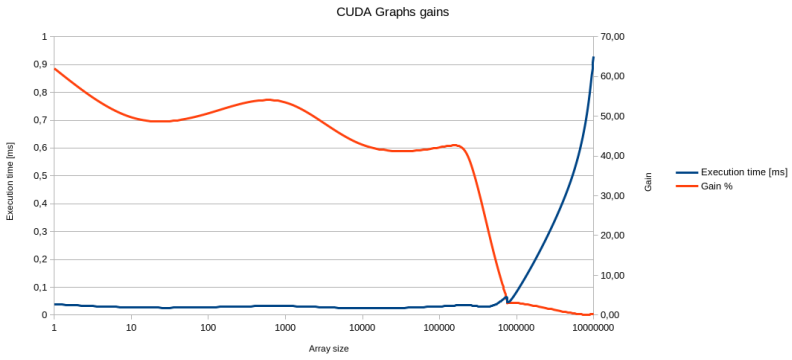

Figure 7: CUDA Graph: executing 6 kernel 1000 times, while increasing the input array size. Good performance improvements can be observed only while the array size is negligible.

CUDA graph is a model to submit kernel execution where the declaration of the workflow and its execution are separate. This means that the overhead related to the kernel declaration can be executed only once, while the kernel can be launched as many times as needed. In practice this gives good performance improvements only for very small kernel, not very computationally intensive, which are called multiple times, as can be seen in Figure 7. Unfortunately CMSSW kernels do not belong to these categories.

4 Updates on HGCal clustering algorithm

HGCal is a High Granularity Calorimeter designed to replace the existing endcap calorimeters in CMS. In fact, the existing calorimeters, essential to measure the energy deposited by a particle, had been designed to withstand an integrated luminosity of $500 fb^{-1}$ and their degradation due to radiation would lead to unacceptable degraded performance. The new project aims to reach a radiation tolerance of $3000 fb^{-1}$, the target value of HL-LHC, by exploiting recent advances in silicon sensors and radiation-tolerant electronics[4].

From the software side point of view, the analysis and reconstruction algorithms need to be updated to fit the new hardware. Here the clustering algorithm and its updates will be discussed. This algorithm, called *CLUE*, is an update of the *Imaging Algorithm* used for the old calorimeter and it is responsible of clustering the cells activations in a layer derived from the same particle interacting with the calorimeter.

It is based on the following steps:

- Build the tiles
  - Each cell is assigned to a tile
- Calculate Local density
  - To each cell is assigned the weighted sum of the energy of his neighbours
- Calculate distance to higher
  - Each cell is assigned the index of the nearest cell with higher energy
- Find Seed and create clusters
  - The seeds are found
  - Each cell is assigned a cluster via a follower mechanism.

For more detailed description refer to HGCal website [1].

After the first implementation of this algorithm for GPU, the CPU code has been updated to account for the differences between the inner part of the calorimeter, silicon based, and the external one, based on scintillators. We have thus started to update the GPU version with the latest improvements.

4.1 Porting challenges

The main problem when moving from CPU to GPU is that many library functions can't be called directly, as they are not `__device__` functions. This was the case in particular for `HGCScintillatorDetId`, a class used to manage the IDs of the cells that is used by the updated CPU code. In order to obtain the same functionality inside a CUDA kernel, three possible alternatives have been considered.

1. Directly copying the code that would be executed by the class inside the kernel
2. Call the class methods from the host, and pass the result to the kernel as a parameter or as application data
3. Port the class to be GPU friendly

The drawback of the first one is that it means to duplicate code, which then has to be validated and maintained. The second solution could be acceptable as a last resort option, but It means to transfer more data to and from the device, leading to higher slowdowns due to memory transfers. So the last alternative is naturally the most elegant one, but usually requires some effort to be sure that all the required methods are `__host__ __device__` functions.

Alternatively to defining a function as `__host__ __device__`, one can take advantage of the fact that the `constexpr` qualifier, used in combination with the `--expt-relaxed-constexpr` build flag, implies the same behaviour as `__host__ __device__` [2]. Since the base class from which `HGCScintillatorDetId` was derived had most of the methods already defined as `constexpr`, it has been straightforward to follow this route to make it GPU friendly. Most of its methods in fact have been simply defined as `constexpr` and their implementation has been moved directly to the header file, in order to make them visible in all the files that include the header. No major update of the code inside of the methods has been required, but now the same can be called from a CUDA kernel.

The second major challenge in porting the code to GPU has been related to the structure of the original `for` loop. In fact, when executing in CPU, the data is processed layer by layer, and some of them are skipped entirely to speed up the execution when the layer is only composed of silicon sensors. In GPU, each kernel executes on a cell, independently on the layer, and so it is difficult to apply the same optimization: each kernel would have to access the global memory and check if this

cell is silicon or not. To deal with this problem one could think to create different kernels for different type of cells, but this would create problems when the two kind of cells have to communicate together. An alternative approach is to use a shared memory variable in a block to check just once from the global (slow) memory if the layer is all silicon, and then all the kernel of the same block can access this information in a fast way and avoid multiple access to the global memory.

5 Conclusions

For someone fairly new to the world of high performance computing as I am, this has been an exciting and interesting summer. I was able to learn a lot of new things, I had to face different problems and explore multiple solutions for each of them, surely building a wide ranging knowledge.

I would like to thank my supervisors, Felice Pantaleo and Antonio di Pilato for the help and the technical guidance, the Patatrack group as well as Kristina Gunne and all the Openlab Summer Student organization.

References

[1] Clue algorithm. URL http://hgcal.web.cern.ch/hgcal/Reconstruction/clueAlgorithm/.

[2] Cuda toolkit documentation. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#constexpr-functions.

[3] Adriano Di Florio Antonio Di Pilato. Accelerating hep with gpus at lhc. URL https://ssl.linklings.net/conferences/pasc/pasc19/slides/msa302s2.pdf.

[4] CMS Collaboration. The Phase-2 Upgrade of the CMS Endcap Calorimeter. Technical Report CERN-LHCC-2017-023. CMS-TDR-019, CERN, Geneva, Nov 2017. URL https://cds.cern.ch/record/2293646. Technical Design Report of the endcap calorimeter for the Phase-2 upgrade of the CMS experiment, in view of the HL-LHC run.

[5] Rene Widera et Al. cupla - c++ user interface for the platform independent library alpaka, . URL https://github.com/ComputationalRadiationPhysics/cupla/.

[6] Rene Widera et Al. Cupla porting guide, . URL https://github.com/ComputationalRadiationPhysics/cupla/blob/master/doc/PortingGuide.md.

[7] Apollinari G., Béjar Alonso I., Brüning O., Fessia P., Lamont M., Rossi L., and Tavian L. *High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1*. CERN Yellow Reports: Monographs. CERN, Geneva, 2017. doi: 10.23731/CYRM-2017-004. URL https://cds.cern.ch/record/2284929.

[8] James Letts. Htcondor solutions for the future of the cms submission infrastructure. HTCondor Week 2019, May 2019. URL https://agenda.hep.wisc.edu/event/1325/session/14/contribution/7/material/slides/0.pdf.

[9] Davide Marcato. Patatrack wiki - transition from cuda to cupla. URL http://patatrack.web.cern.ch/patatrack/wiki/cuda2cupla/.

[10] Bernd Panzer-Steindel. It technology and market, status and evolution. URL https://twiki.cern.ch/twiki/pub/Main/TechMarketPresentations/tech_market_BPS_May2018_v10.pptx.

[11] Erik Zenker, Benjamin Worpitz, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E. Nagel, and Michael Bussmann. Alpaka - an abstraction library for parallel kernel acceleration. IEEE Computer Society, May 2016. URL http://arxiv.org/abs/1602.08477.