Performance study of the CLUE algorithm with the alpaka library

Andrea Bocci[1], Antonio Di Pilato[1,4], Luca Ferragina[1], Matti Kortelainen[2], Juan Jose Olivera Loyola[1], Felice Pantaleo[1], Aurora Perego[1], Marco Rovere[1], Wahid Redjeb[1,3], on behalf of the CMS collaboration

[1]CERN, European Organization for Nuclear Research, Meyrin, Switzerland
[2]Fermilab, Fermi National Accelerator Laboratory, Batavia, IL, USA
[3]RWTH Aachen University, III. Physikalisches Institut A, Aachen, Germany
[4]CASUS, Center for Advanced Systems Understanding, Görlitz, Germany

E-mail: antonio.di.pilato@cern.ch

Abstract. The CLUE (CLUsters of Energy) algorithm is designed to optimize the clustering step in the event reconstruction chain of granular detectors, which can achieve an unprecedentedly high segmentation. Therefore, a fast and fully-parallelizable clustering algorithm is required to process such a large amount of data efficiently. In the context of the CMS experiment, meeting the Phase-2 High Level Trigger requirements necessitates reducing the computational load. To achieve this goal, we tested the CLUE algorithm on multiple accelerators and hybrid platforms using alpaka as the performance portability library. The latest results demonstrate that the alpaka implementation of CLUE can fully exploit the available hardware and deliver high performance.

1. Introduction

The Large Hadron Collider (LHC) at CERN is embarking on a new phase of particle physics exploration. The forthcoming High Luminosity configuration of the machine (HL-LHC) [1] will generate an unprecedented rate of proton-proton collisions, resulting in a much larger dataset than previous runs. The main objective of this upgrade is to achieve more precise measurements of known processes and to detect very rare interactions predicted by physics beyond the Standard Model.

To capitalize on the higher collision rate and manage the increased amount of data per bunch crossing, the CMS experiment [2] is pursuing diverse strategies on both the software and hardware fronts. The CMS detector will feature a novel endcap calorimeter for Phase-2 - the High Granularity Calorimeter (HGCAL)[3] - capable of providing exceptionally detailed information on particle showers. As shown in the simulation depicted in Figure1, the complexity of an event in the HGCAL detector is extremely high, making online data reconstruction a formidable challenge for the CMS software.

In the pileup 200 scenario, obtaining high-level particle information from digital signals in each sub-detector requires significant computing resources. The complexity of this task poses a significant challenge to online reconstruction and trigger processes, given their limited time

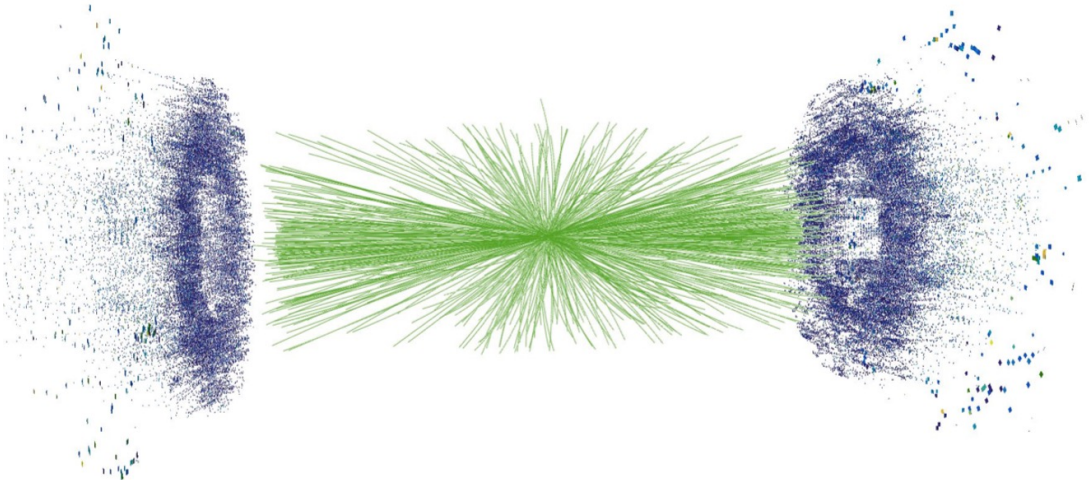

Figure 1. Simulation of a $t\bar{t}$ event in the CMS Phase-2 detector.

and memory management capabilities. Traditional algorithms and computing hardware are inadequate to meet the demands of the CMS Phase-2 challenge.

Modern computing farms and data centers employ heterogeneous architectures that integrate conventional CPUs with hardware accelerators, such as GPUs and FPGAs, to provide efficient solutions for executing complex software and algorithms. Furthermore, despite the use of powerful devices, this approach helps reduce overall energy consumption and cost. CMS has adopted a similar solution to tackle the Phase-2 challenge for the High-Level Trigger [4] (HLT). The HLT has been restructured as a heterogeneous farm, with GPU parallel processing utilized for a portion of the reconstruction process. However, a variety of different hardware devices are available on the market, and the primary GPU vendors (NVIDIA, AMD, and Intel) use their own programming models to leverage their hardware (e.g. NVIDIA CUDA, AMD ROCm/HIP, Intel OpenCL). To leverage technological advancements in computing devices, CMS must ensure compatibility with different hardware available on the market that is suitable for the Phase-2 reconstruction challenge.

2. Performance portability

The CMS High Level Trigger farm must manage the complexity of multiple versions of the same algorithm for each available hardware architecture, which results in significant code duplication. To address this issue, the CMS collaboration has explored several performance portability solutions, including software libraries that enable the creation of a single source code that can be compiled for different backends and executed on the target platform. Adopting performance portability libraries simplifies code maintenance and ensures continued support for future devices, as long as the chosen library is updated by developers. After careful evaluation, CMS has selected Alpaka for future use due to its close-to-native performance and ease of integration with the CMS software.

Alpaka [5] (Abstraction Library for Parallel Kernel Acceleration) is a C++ header-only library developed and maintained at HZDR (Helmholtz-Zentrum-Dresden-Rossendorf)[6] and CASUS (Center for Advanced Systems Understanding)[7]. It supports a wide range of compilers and several computing architectures by abstracting each backend through a hierarchical parallelism model similar to CUDA. Several backends are currently available, including CPU serial and parallel execution, through C++ threads or Threading Building Blocks (TBB)[8], parallel execution on NVIDIA GPUs through CUDA[9], and on AMD GPUs through HIP/ROCm [10].

An experimental SYCL [11] backend has also been tested and studied. In addition to the CUDA-like hierarchical model, Alpaka provides an additional level of abstraction called "elements," which is mainly introduced to exploit vectorization.

Alpaka is a fully-templated library, meaning that each backend has its own entities (i.e., platform, accelerator) that can be selected at runtime. Since these entities are separate from each other, it is also possible to configure an application such that jobs are split among multiple backends and run in parallel without any backend conflict (i.e., if both AMD and NVIDIA GPUs are available on the same machine). Memory management efficiently relies on smart pointers, to avoid possible memory leaks due to the non-destruction of objects in the code. Pointers to device memory are passed to *kernels*, which are executed by *queues* (entities similar to *cudaStreams*). *Queues* execute additional tasks, such as memory copy between *host* and *device* (or device and *device*), while data are allocated in and transferred between Alpaka *buffers*. Since these objects have no default constructors, it is necessary to treat them as *optional* when the memory allocation is performed at a later stage. Finally, Alpaka kernels are defined as functors templated on the accelerator, called through an *enqueue* function that assigns it to a queue object and executes it in parallel through a valid *work division* (number of grids, blocks, threads, elements).

3. A real use case: the CLUE algorithm

CLUE [12] is a fast and fully-parallelizable algorithm developed for future high-granularity calorimeters under high-occupancy conditions. It has been tested and its performance studied in the high-pileup scenario for the CMS HGCAL detector. The algorithm is based on energy density, specifically clusters targeted by CLUE have an energy density profile with a maximum in the center and decreasing radially. CLUE has been designed to build small clusters that resemble energy deposits of particles crossing each layer of the detector.

Furthermore, two additional requirements have driven the design of CLUE:

(i) Linear scaling with the number of points.

(ii) Parallel execution capability.

The first requirement comes from the evaluation of how the available clustering algorithms based on energy density generally scale. For example, Clustering by Fast Search and Find Density Peak [13] groups points together within a KD-Tree data structure [14] which has a query complexity of $\mathcal{O}(n \log n)$. To address this, CLUE employs a spatial indexing technique (shown in Figure 2) that leverages the full granularity of the calorimeter, enabling fast queries and overall complexity of $\mathcal{O}(n)$. This is achieved using a tiled data structure, which simplifies the parallelization of each query and step of the clustering algorithm, unlike the KD-Tree data structure. This addresses the CMS need for parallelization of existing algorithms as described in the previous section, and meets the second requirement.

CLUE's logic is described in Figure 3. The first step is to arrange points in the tiled data structure, which is necessary for the next two steps: the calculation of the local energy density and the search for the "nearest higher" points respectively. For the local density, a window of configurable size is opened around each point, and the only candidate points to contribute to such computation are those included in the tiles touched by this window (from a minimum of one to a maximum of four). In this way, the complexity of such computation for each layer is $\sim \mathcal{O}(n)$, as only a few candidates are considered for each point, instead of all the points present on the layer (which would result in an $\mathcal{O}(n^2)$ loop). The same logic applies for the search for the *nearest higher* (nearest point with higher energy density) and the computation of its distance, but with a different (usually higher, necessary to distinguish seeds and outliers in the next step) distance threshold.

In the third step, *seeds* and *outliers* are identified among all the points, with the first being the "centers" of the clusters (points with high local density and without a nearest higher within

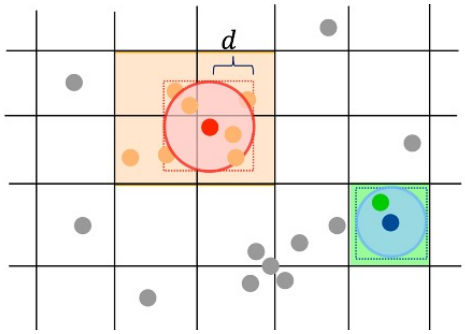

Figure 2. Tiled data structure used for CLUE spatial indexing [12].

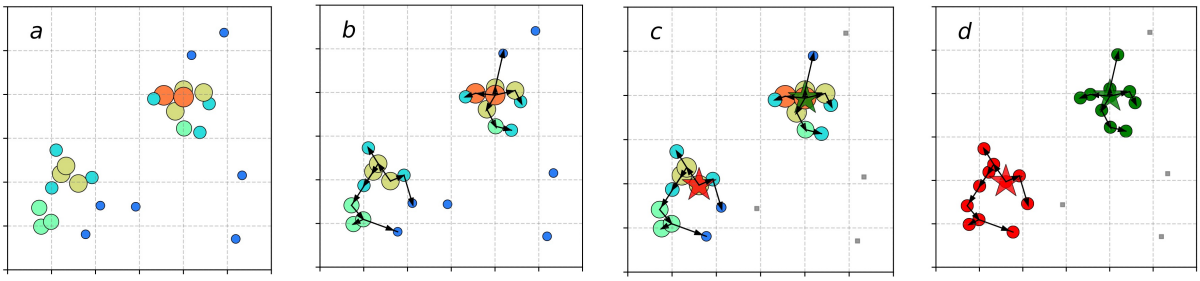

Figure 3. Illustration of the CLUE procedure. First (a), CLUE calculates the local energy density ρ for each point. The color and size of points represent such values. Then (b), the nearest-higher nh (nearest point with higher energy density), and its distance δ are computed for each point. The black arrows represent the relation from the nearest higher point to the point itself. In the third step (c), CLUE promotes points as seeds (represented as stars, these are the centers of clusters) or demotes them to outliers (represented as grey squares, considered as noise). In the final step (d), CLUE propagates the cluster indices from seeds through their chains of followers (points having such seeds as their nearest higher. The color of points represents the cluster indices [12].

the selected distance threshold) and the second being isolated points that can be interpreted as noise. Finally, points that are neither seeds nor outliers are defined as *followers* of their respective nearest highers. Such a definition is crucial in the last step, where each seed is assigned a unique cluster index, which is then propagated iteratively through the chain of followers.

4. CLUE Performance

CLUE has been successfully ported to GPUs using CUDA and to alpaka for testing the performance of the portability library. Tests were conducted on a CMS-HLT machine equipped with a dual socket AMD EPYC 7763 64-core and an NVIDIA Tesla T4 GPU using a standalone framework that resembles the CMSSW coding structure. The algorithm was run for 10,000 events, split among several CPU threads through TBB, and the usage of EDM streams as in CMSSW was employed to compute multiple events concurrently. Operations were performed asynchronously on the GPU, and the copy of the results back to the CPU was not considered. The performance of CLUE is shown in Figure 4, evaluated in terms of throughput.

The alpaka implementation on the CPU shows the same linear scaling with the number of threads as the serial implementation, demonstrating close-to-native performance and confirming alpaka as an excellent choice for code portability on different platforms. The alpaka version with

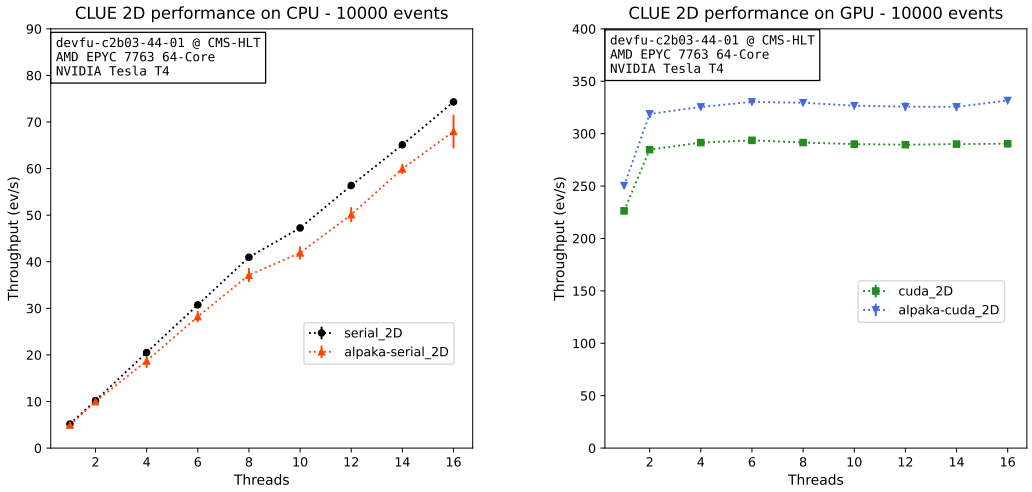

Figure 4. Performance of CLUE on CPU (left) and on GPU (right). A comparison between the native and alpaka implementation on the same backends is shown.

the CUDA backend outperforms the native CUDA version for the GPU implementation, which was unexpected. An investigation is required to identify the reason for the slower performance of native CUDA. Additionally, the scaling with the number of threads is limited by the I/O operations and computing capability of the GPU beyond 4 threads, indicating the need for further optimization.

5. A work in progress: CLUE3D

A 3D version of the CLUE algorithm, called CLUE3D, has been implemented to reconstruct particle showers in multi-layer high granularity calorimeters. A serial implementation is currently available in CMSSW as one of the pattern recognition algorithms for this step of the event reconstruction. The algorithm has been directly ported to alpaka to run on the GPU, bypassing a native CUDA implementation. The performance of CLUE3D has been studied on the same machine as before, and the results are shown in Figure 5. Alpaka provides close-to-native performance with the serial backend, while the throughput on the GPU is more than 20 times higher than on the serial CPU for 2 threads. However, the scaling on the GPU seems to be limited, and further investigation is necessary. CLUE3D is an important achievement to demonstrate the advantages of adopting alpaka as a performance portability solution.

CLUE3D performance has been studied on the same machine as described in the previous section, and Figure 5 shows the results. Also in this case, alpaka provides close-to-native performance with the serial backend, showing the same linear scaling trend with the number of concurrent threads. For the GPU version, there's no available comparison with a native CUDA implementation in this case. However, the throughput is more than 20 times higher on GPU than on serial CPU for 2 threads, while the scaling on GPU seems to be limited as in the previous case and is the object of further investigation. The experience with CLUE3D is an important achievement to prove the advantages of adopting alpaka as a performance portability solution, as it allows for efficiently exploiting the available hardware and the knowledge of parallel programming with CUDA, due to the similar hierarchical model and programming strategy.

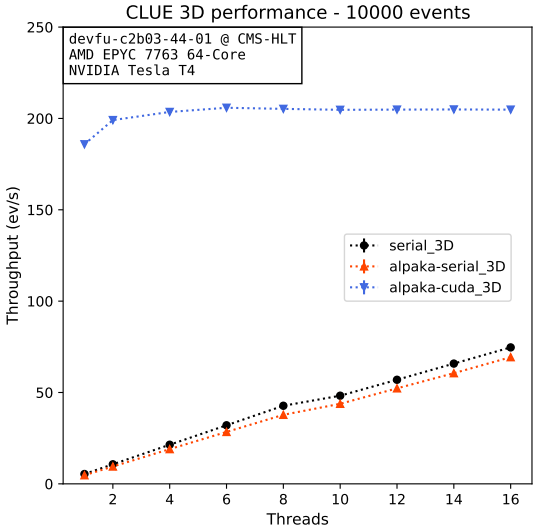

Figure 5. Performance of CLUE3D's native implementation on CPU (serial) and of its alpaka implementation on both CPU and GPU.

6. Related projects and conclusion

The alpaka performance portability library is an interesting solution in the era of heterogeneous computing for several reasons. It shows close-to-native performance, which is not always guaranteed by other libraries, and new backends are in constant development (i.e., SYCL). CLUE represents a useful testbed for performance portability solutions as it is a simple application designed to achieve high performance. Tests have been conducted on its implementation with native SYCL/oneAPI, and its performance is expected to be studied along with the alpaka implementation running on the SYCL backend. A Python library has been developed to generalize CLUE to N dimensions while using Python bindings to the C++ implementation to preserve high performance. Finally, CLUE3D is the first algorithm of the CMS software reconstruction that has been ported to alpaka without passing through an intermediate CUDA version, providing a useful example for future porting of other algorithms within the context of the heterogeneous computing era at the CMS experiment.

Acknowledgements

This work has been supported by CASUS, the Center for Advanced Systems Understanding in Görlitz, Germany, where the alpaka library has been developed in collaboration with the Helmholtz Zentrum Dresden-Rossendorf (HZDR). The mentioned work on the SYCL/oneAPI version of CLUE has been funded by CERN openlab, while the python library has been developed in the context of Google Summer of Code program (credits to Simone Balducci, University of Bologna).

References

[1] Apollinari G, Alonso I B, Brüning O, Fessia P, Lamont M, Rossi L and Tavian L 2017 URL `https://cds.cern.ch/record/2284929`
[2] CMS Collaboration 2008 *JINST* **3**
[3] Collaboration C 2017 URL `http://cds.cern.ch/record/2293646`

[4] 2007 CMS High Level Trigger Tech. rep. CERN Geneva revised version submitted on 2007-10-19 16:57:09 URL https://cds.cern.ch/record/1043242
[5] Zenker E, Worpitz B, Widera R, Huebl A, Juckeland G, Knüpfer A, Nagel W E and Bussmann M 2016 *CoRR* **abs/1602.08477** (*Preprint* 1602.08477) URL http://arxiv.org/abs/1602.08477
[6] Helmholtz Zentrum Dresden Rossendorf website URL https://www.hzdr.de/
[7] Center for Advanced Systems Understanding website URL https://www.casus.science/
[8] Intel Threading Building Blocks URL https://software.intel.com/en-us/intel-tbb
[9] *CUDA C Programming manual*
[10] *HIP Programming Guide v4.5*
[11] The Khoronos SYCL Working Group, SYCL 2020 Specification (revision 2) (2021)
[12] Rovere M, Chen Z, Di Pilato A, Pantaleo F and Seez C 2020 *Frontiers in Big Data* **3** 41
[13] Rodriguez A and Laio A 2014 *Science* **344** 1492–1496 ISSN 0036-8075 URL https://science.sciencemag.org/content/344/6191/1492
[14] Bentley J L 1975 Multidimensional binary search trees used for associative searching vol 18 (ACM) pp 509–517 ISSN 0001-0782