Bit Level Data Unpacking Using Heterogeneous Architecture Setups

M. N. J. Momed[1,⋆], F. Blekman[1,2], M. Buschmann[1], C. Wissing[1], and P. Neumann[1,2]

[1] Deutsches Elektronen-Synchrotron DESY, Hamburg, Germany
[2] University of Hamburg, Hamburg, Germany

Abstract. Modern high-performance computing centers increasingly opt for heterogeneous system designs, integrating general-purpose computing cores with accelerators, to deliver high computing performance and high efficiency. Scientific computing codes need to adapt to this change in cluster architecture, which introduces the challenge of hardware portability. In the following, we introduce a hardware portable code for the implementation of a data unpacking algorithm, a key part of the data processing system for particle detector readout systems, in the environment provided by the C++ abstraction framework with the header-only alpaka library. This concept has great relevance for the high-energy physics research at CERN, where different computer accelerators from diverse vendors are used. This code is portable to CUDA, HIP, OpenMP, and serial code, requiring no tailoring to a specific platform. We validate the approach using the high-throughput data processing requirements of the CMS experiment at CERN and verify the conservative, lossless nature of the given data unpacking procedure. The performance improvement using heterogeneous hardware exceeds the average throughput by more than 25 times compared to the reference code run on 8 CPU threads.

Keywords: Heterogeneous HPC · Hardware Portable Libraries · Data Unpacking.

1 Introduction

1.1 The Evolving Landscape of High Performance Computing

High-performance computing (HPC) is the backbone for computational science across many research domains. The HPC systems are known for their magnificent processing capabilities and huge storage capacities, enabling researchers to cope with problems that would be impossible with conventional hardware [1]. These systems are designed to deal with operations at petascale and exascale levels.

Over the past three decades, the supercomputing field has witnessed remarkable advancements with modern architectures crossing the exascale threshold, which enables scientists to deal with quintillion calculations per second through

⋆ mohammad.nasir.jan.momed@desy.de

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

the coordinated operations of millions of compute cores. This drastic scaling is a reflection of both increased processor efficiency and fundamental changes in the system architectures, driven in part by the explosive growth of data-intensive applications in machine learning, scientific simulations, and big data analytics.

HPC systems have undergone an important evolution: the paradigm shift towards heterogeneity in computing. Today's high-performance computers integrate general-purpose CPUs with specialized accelerators such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs). According to the November 2025 TOP500 list [2], the vast majority of the top 100 systems now employ heterogeneous architectures, pairing multi-core CPUs from vendors such as AMD, Intel, or IBM with GPU accelerators from NVIDIA or AMD. Rather than using a single powerful CPU. This architectural evolution takes advantage of the complementary strengths of different processor types: CPUs handle complex sequential tasks requiring control flow and branching logic, while GPUs, with their thousands of cores, excel at performing the same operation on large datasets in parallel.

The decision to use accelerators is motivated not only by performance enhancement but also by energy efficiency. GPUs provide a better ratio of FLOPS/Watt, a metric used to rank supercomputers in the GREEN500 list [3], with the most power-efficient modern systems achieving approximately three to four times the energy efficiency of designs from just a few years ago.

The hardware diversification benefits the research community by providing greater choice. However, it creates a challenge for software portability, as applications must adapt to optimally function across various hardware configurations.

1.2 Adapting Data Processing for Heterogeneous Systems

The widespread adoption of heterogeneous architectures presents particular challenges for data-intensive workflows, especially those involving raw data unpacking and decompression. In various scientific domains, data acquisition systems generate highly compressed bit-packed formats, which are optimized for minimizing the storage requirements as well as the transmission bandwidth. The step for the conversion of these compact representations into a usable numerical format involves an essential preprocessing leap that must be performed efficiently to avoid bottleneck creation in overall pipeline processing stages.

Traditional data unpacking routines were typically designed for sequential executions on CPUs and struggle to exploit the massive parallelism available in modern accelerators. Adapting these algorithms for execution in heterogeneous setups requires precise considerations of memory access patterns, due to the nature of the decompression operations often involving irregular data dependencies rather than complicated parallel decompositions. The potential performance gains from offloading suitable portions of unpacking workloads to GPUs or other suitable accelerators can substantially reduce overall processing latency.

Additional complexities are introduced while reading and interpreting specialized data acquisition formats on heterogeneous hardware [4]. These formats

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

frequently employ variable-length encoding schemes, domain-specific compression, or nested structures optimised for a particular detector or sensor characteristics. The implementation of a parser with the capability to operate effectively across different processor architectures demands abstraction layers that map high-level operations to architecture-specific primitives without performance loss.

1.3 Performance Portability Through Abstraction Frameworks

The massive increase of hardware vendors and architectural variations within HPC systems creates a significant software maintenance burden that can be hard to deal with. Developing and maintaining separate implementations optimized for each target platform, whether NVIDIA GPUs, AMD GPUs, Intel accelerators, or even multi-core CPUs, multiplies the development efforts and introduces potential inconsistencies among the code paths. This aspect is a motivation for the adoption of performance portability ecosystems such as alpaka[5], KOKKOS[6], RAJA[7], and so on that enable a single codebase to execute efficiently across multiple hardware.

Among the most famous portability ecosystems, one is alpaka, which represents one of many libraries for such an abstraction layer designed specially for heterogeneous computing using C++. This framework provides a combined programming interface that abstracts away hardware-specific details while preserving access to architecture-specific optimizations. A code written in alpaka can target multiple backends, including CUDA for NVIDIA devices, HIP for AMD hardware, OpenMP or standard threading for CPU executions through compile and runtime configurations rather than source code modification [8].

The so-called *write once, run anywhere* approach offers conclusive advantages for scientific software projects. The algorithm's correctness can be a focus point for the development teams rather than repeatedly implementing the same code in vendor-specific languages. In case of the availability of new hardware or the upgrade of existing systems, the same application code can be recompiled with approperiate backend selection flag for leveraging new capabilities and not having to go through extensive refactoring. To prove that decompression algorithms and format parsers have the potential to run at a comparable pace regardless of architectures, with the help of abstractions such as alpaka, this strategy would be justified for the use in scientific applications.

In this paper, we demonstrate the potential for implementing data unpacking algorithms within a hardware-portable ecosystem. We achieve significant performance gains by offloading suitable parts of the code to GPU accelerators while maintaining a single code base, which allows for easy adaptation in the event of hardware vendor or accelerator changes. Section two of this paper describes the methods used for the study. It introduces the case study environment, the data format and hierarchy, the parallelization strategy employed, and how the workload is divided among the different hardware accelerators. Section three presents the results of the study, validating our approach initially within the case study's framework, and discusses the performance improvements in terms of execution time. Later on, the core unpacking extraction logic is discussed for

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

further analysis and generalization purposes; the extracted unpacking logic is validated outside the case study framework. We also report on the performance enhancements of the extracted unpacking code beyond this case study environment. The paper concludes with a brief summary of the achieved results and outlines planned future work arising from this study.

2 Methods

2.1 Environment

The presented study has been developed in the context of the CMS Experiment [9]. CMS is a multipurpose particle physics experiment that is being operated at the Large Hadron Collider (LHC) at CERN. The CMS Collaboration brings together members of the particle physics community from across the globe in a quest to advance humanity's knowledge about fundamental laws of elementary particles and their interactions. CMS has over 6000 particle physicists, engineers, computer scientists, technicians and students from around 240 institutes and universities from more than 50 countries [10]. The CMS detector consists of many custom-built subcomponents, one being the so-called "outer tracker", a large device used to measure trajectories from charged particles that are created in proton-proton collisions. The outer tracker consists of 13297 double-layer modules of two kinds, modules with two silicon strip sensors on both sides (2S modules) and modules with one strip sensor layer and a pixel layer (PS modules) [11].

Particles traversing through the tracker volume deposit clusters of charge in the silicon detectors that get digitized, and parameters of recognized clusters are read for triggering purposes and for long term storage. Data gets transported over custom-built electronic boards and links employing a dedicated internal protocol with highly packed bit patterns. In order to cope efficiently with the data, a well-performing unpacking routine is key to achieve good throughput for the downstream processing of the clusters.

2.2 Input Data Structure

The input for the unpacking algorithm consists of raw binary data organized hierarchically across multiple data links from custom-made top-level boards referred to as Data Trigger and Control (DTCs). In the setup of this case study in total 216 DTCs and 864 links are present. Each DTC handles 4 links, where each link transmits information from 18 readout channels, with the data arranged in fixed-width bit-packed words.

Figure 1 further illustrates the data hierarchy. DTCs are the highest level data transfer controllers managing data from multiple serial links. This hierarchical organization reflects the physical topology of a typical modern detector readout system, where data concentrators aggregate information from numerous front-end modules before transmission to the downstream processing.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

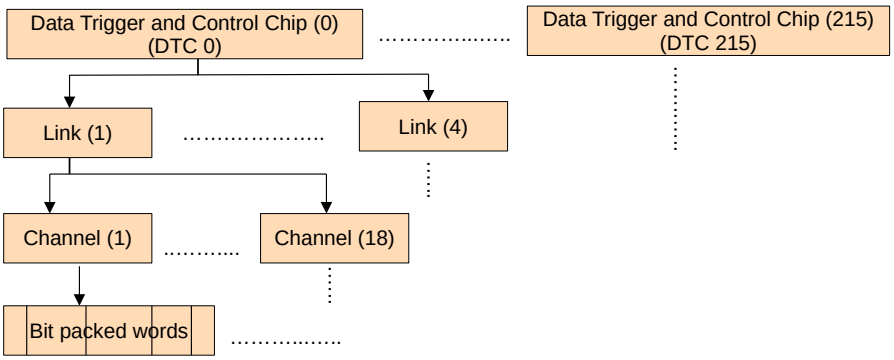


Fig. 1: Example data hierarchy of the case study, starting from the top-level DTC boards to the bit-level packed words containing data. In this case study, the number of DTCs is 216, where each DTC processes data from 4 links and each link processes data from 18 channels.

Figure 2 further shows the contents of the bit-packed words. The format includes a header section that contains metadata and offset tables, followed by variable-length payload regions for each channel. Such a structure is typical for a data acquisition system used in detector systems, where bandwidth optimization requires compact binary encoding.

For each of the two detector types, there exists a detector-specific bit-width encoding for optimal usage of its respective physical characteristics. The unpacking algorithm needs to parse these format-specific encodings and route the extracted cluster data to appropriate output structures based on the type of detector module associated with each channel.

Each channel data begins with the header word containing some metadata and the count for strip and pixel cluster words present in the payload. These counts are extracted through bit masking operations applied to predetermined bit positions within the header of a word. The counts determine the total number of bits to be read from the subsequent payload region and the number of data words for that channel.

The process of data unpacking involves traversing through an offset table to find where the varied-length payload for each channel is found in the raw data stream. The most challenging aspect is the recovery of bit fields that are varied in width from fixed-width words since the clusters lie contiguously in a non-aligned manner, often extending beyond a word boundary. This is achieved by tracking state variables for bit positions.

The raw input data stored in memory is a contiguous array of 32-bit words with preserved stream. The data is flattened into a single linear byte array with a separate offset array, dissolving the DTC and link hierarchy before the GPU transfer. Each S-link is located via an offset table entry, and within each S-link,

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

an offset table stores 16-bit channel offsets packed two per 32-bit word. These offsets locate variable-length payload regions for each channel.

This layout results in variable-length regions per channel, which are located through the offset table rather than computed via fixed strides. Consequently, while the data for channels within the same link are accessible via two-level addressing, their starting positions and lengths differ. In the parallel implementation, where one thread processes one link, consecutive threads access different link regions. This provides good spatial locality at the link level, but within a link the per-channel accesses are irregular due to variable payload sizes, limiting fully coalesced global memory accesses on GPUs.

On CPUs, the same layout allows largely sequential traversal within each link, enabling effective cache-line reuse and hardware prefetching. Therefore, the overall performance reflects a combination of memory access behavior and bit-level unpacking operations, with the balance depending on detector occupancy and resulting payload variability.

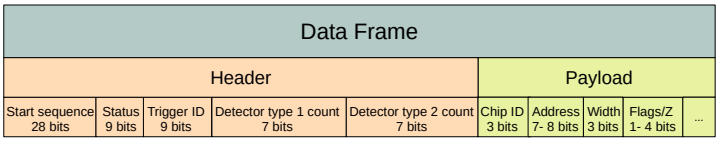


Fig. 2: Bit-packed words contain a header with metadata: Start sequence (28 bits), Status and Trigger ID (9 bits each), and Detector type 1/2 counts (7 bits each) indicating the number of payloads per detector type. The payload holds the physical data: Chip ID (3 bits), Address (7–8 bits depending on detector type), Width (3 bits), and a final column storing z-coordinate and detector-specific flags. Continuity arises because a header can span multiple payloads, as illustrated in the figure.

2.3 Parallelization Strategy

The unpacking workload of this case study employs parallelization at the link levels. Each of these links, visualized in Figure 1, can be processed individually due to no data sharing dependencies between these links. In the implementation, in this case, we assign one thread to each data link, with each thread being responsible for iterating through all channels within its assigned link, parsing headers, extracting cluster payloads, and writing results to output buffers.

Thread indexing follows the alpaka range-based parallel loop abstraction: Each thread iterates over links assigned by a uniform elements abstraction, which distributes work evenly across threads. This approach ensures full workload coverage independent of the relationship between link count and available parallelism.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

In this study, clusters are groups of adjacent activated detector channels indicating where a particle traversed the sensor. The management of the output buffer must coordinate between multiple threads running in parallel in order to prevent race conditions. The employed technique uses an atomic counter operation to allocate a sequence of output slots in advance for each of the channels' clusters. Each thread computing its number of clusters for output atomically increments a counter and obtains the old counter value as its offset for writing. Thus, each channel performs exactly one atomic operation, regardless of how many clusters it contains. For a typical event with moderate occupancy, this results in an order of thousands of atomic operations across all threads, which does not constitute a performance bottleneck given the coarse granularity of one atomic per channel.

For each thread, in each channel, processing is sequential. It first looks at the type of the module for the channel in process, ignoring the channels with undefined modules. For the defined module types, it reads the channel header to look at the strip and pixel clusters, then allocates local arrays based on the maximum possible amount and calls the bit extraction function for each type in sequence. Allocation of local arrays is in stack memory, with compile-time sizes, to eliminate the overhead of runtime memory allocation when inside the kernel.

These decoded properties are then placed in the reserved output positions. Bit shifts and bitmasks are used to selectively access each field, and transformations are performed before placing them into output arrays.

The Structure of Array (SoA) format is used due to its superior parallelization capabilities, allowing threads to access data more efficiently.

2.4 Heterogeneous workflow scheme

The unpacking of the raw data has been implemented with the help of the alpaka portability library so that it can run on a CPU or a CPU/GPU setup together with a single codebase. The process here is configured to use the heterogeneous setup, hence the workload is divided between CPU and GPU according to their capabilities. These processes with their subprocesses are shown in Figure 3.

On the CPU, a series of initialization tasks takes place. The lookup tables that translate readout channels into the interested region of detector module identities are created. The system initializes 216 DTCs, each controlling four links. The types of modules are determined so that either 2S or PS architectures, which have different logic for data unfolding, can be distinguished. Furthermore, the acquisition of the raw data for each event, as well as data flattening using a flat indexing scheme, is implemented on the CPU side.

The kernel for unpacking data, which has the ability to be run on either a CPU or a GPU, executes the links depending on the architecture serially or concurrently for CPU or GPU, respectively. In the case of execution on GPU, each thread is given a link with 18 channels. Figure 3 also indicates the link parallel execution on the GPU. Within the threads, the five-stage pipelined execution reads as follows: 1) The header is processed to extract the offset table. 2) The channels are looped through. 3) The clusters are derived from the bit-packed

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

payload. 4) The payload is decompressed into the properties of the clusters. 5) The results are written to the output buffers; in this stage, the results are made thread-safe using an atomic counter. These five stages are processed sequentially.

The output data is stored as a Structure of Arrays (SoA), where the arrays are contiguous in memory for detector ID, z position, cluster width, module type, cluster x coordinates, cluster y coordinates, seed indicators, and additional flags depending on the sensor type. This helps to optimally use the cache during the iteration of individual properties of the clusters performed during the downstream track reconstructions.

Kernel functions in alpaka are realized by templated callable structures that automatically adapt to different kinds of accelerators at compile time: thread indexing and atomic operations are automatically mapped to the respective platform primitives, as well as launch configurations. Memory management utilizes the buffer abstraction of alpaka, which ensures proper device-side allocation. Most of the involved data transfers are automatically orchestrated before the kernel launch using asynchronous transfers, while the selective output transfer minimizes this overhead by copying only those parts of the result arrays actually being populated from the cluster count.

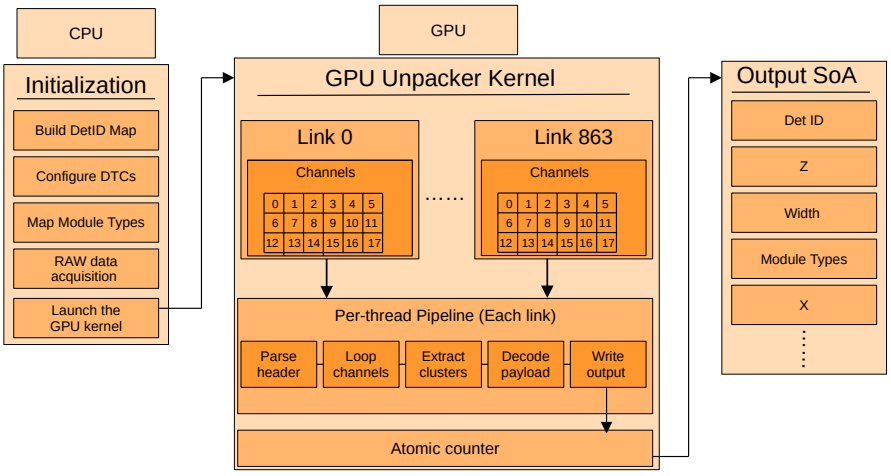


Fig. 3: Workflow of heterogeneous CPU–GPU raw data unpacking: the CPU initializes a 216-DTC lookup table (link → detector ID, z, width, module type) and flattens raw data; the GPU performs parallel link-level unpacking (1 thread/link, 18 channels) via a five-stage pipeline: header & offset extraction, channel iteration, bit-packed cluster extraction, cluster property decompression, and atomic-protected output writing. Detector ID, z, width, module type, x, y, and additional flags are stored as a Structure of Arrays (SoA) for cache-efficient track reconstruction.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

3 Results

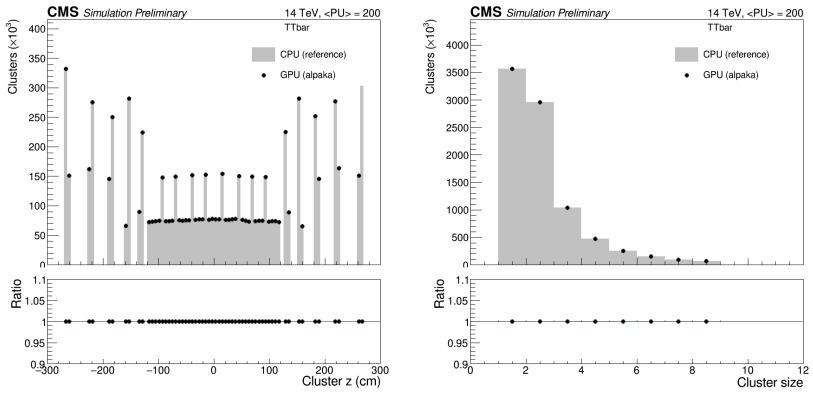
3.1 Validation and Cluster Unpacking Quality

The performance and accuracy of the hardware portable unpacking algorithm have been evaluated through extensive validation studies against a reference unpacking implementation that runs solely on the CPU [12]. In these comparisons, the reference code serves as the ground truth, and the alpaka code performs unpacking using a combination of a CPU and a GPU.

The comparison of cluster characteristics is performed for the two types of detector modules, 2S modules and PS modules, for a number of cluster variables.

In the case of strip modules (2S), Figure 4.*a* shows one of the spatial coordinates, measuring longitudinal position, the z-coordinate of the clusters. For the pixel modules used in PS modules, we have shown in Figure 4.*b* the number of pixels within a cluster.

The validation plots, Figure 4.*a* and Figure 4.*b*, explicitly with the ratio plotted in the bottom sections, show a complete one-to-one agreement of the CPU reference code and the alpaka code executed on heterogeneous architecture, proving the correctness of the GPU-enabled implementation.



(a) Z coordinate 2S modules (b) Cluster size PS

Fig. 4: Example comparisons between variables unpacked by the CPU reference implementation (gray histogram) and the alpaka code (black dots). (a) z coordinate distribution for the 2S modules. (b) cluster size distribution for the PS modules. Result of both unpacking codes are identical.

Cross-backend validation has further extended such comparisons over all alpaka backends that support the execution of alpaka programs. Evaluation of the same set of inputs with CUDA, HIP, OpenMP, and serial backends has yielded the same numerical results for both z-coordinates of the clusters as well as the

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

sizes of these clusters. Such results indicate that the abstracting layer provides a sufficient platform to filter out differences in the scheduling of threads and memory accesses that are specific to backends supporting atomic operations in an efficient manner while avoiding computational divergence.

3.2 Performance Analysis

Efficiency and Comparison metrics Efficiency in this work refers to the ability to sustain high throughput under increasing workload relative to resource usage. As the unpacking algorithm in this case study is intended for use in the CMS High-Level Trigger (HLT), an online real-time system, the primary performance metric is event throughput. Throughput is defined as the number of events processed per unit time:

$$\text{Throughput} = \frac{N_{\text{events}}}{t_{\text{total}}}$$

where N_{events} is the total number of events processed and t_{total} is the total execution time measured from the end of the job start-up time to completion of the unpacking process for all events in the batch. Throughput is reported in events per second (events/s).

Secondary metrics commonly used in heterogeneous computing, such as energy efficiency (events per watt), GPU occupancy, and CPU utilization, are acknowledged as important for complete system characterization. However, they are not reported in this study, as the primary constraint for the HLT is real-time throughput rather than power or utilization optimization. The focus is therefore on end-to-end event processing rate, which directly determines the trigger's ability to keep pace with the LHC collision rate.

The performance analysis was conducted across three different configurations: the reference CPU code running exclusively on CPU, the alpaka ported code running on CPU only, and the alpaka ported code running on a heterogeneous setup utilizing both CPU and GPU resources. These configurations are labeled in Figure 5 as CPU (reference), CPU (alpaka), and GPU (alpaka), respectively.

This analysis has been done using an Intel Xeon Gold 6130 CPU hardware architecture (dual-socket configuration with 32 physical cores and 64 threads, base clock frequency of 2.10 GHz and maximum turbo frequency up to 3.70 GHz, with a total system memory of 92 GB distributed across two NUMA nodes) for both the CPU (reference) and CPU (alpaka) configurations. And the GPU (alpaka) configuration uses an Intel Xeon Gold 6130 CPU together with an NVIDIA Tesla T4 GPU hardware architecture (Turing architecture, 15 GB GDDR6 memory, PCIe Gen3 x16 interface, compute capability 7.5, base graphics clock of 585 MHz and boost clock up to 1590 MHz), with tasks distributed between both processing units according to the workflow described in Figure 3.

Performance was evaluated using weak scaling studies where we examined the performance as a function of increasing workload while keeping resources constant. In this study, we analyzed the execution time of the end-to-end unpacking

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

process as a function of the number of input events in an 8-threaded configuration , which is the present standard in CMS for multi-thread configurations. The end-to-end unpacking process mainly consists of the pre-processing on the host side, buffer allocations, host-to-device copy operations, launching and processing of the unpacking kernel, device-to-host copy operations, and the post-processing. One "event" is the collection of clusters that are created from one proton-proton collision inside the detector. All events are independent of each other and are processed individually. Figure 5 demonstrates this analysis, showing the CPU (reference) code exhibits a nearly linear scaling. The CPU (alpaka) configuration demonstrates moderate improvements, which can be attributed to benefits from the SoA data structure, that was introduced in comparison to the CPU reference code. The GPU (alpaka) heterogeneous setup is most efficient, and execution times rise the least as a function of the number of input events. The achieved throughput increases to an improvement of a factor of more than 25 for the ported code compared to the reference code using 8 CPU threads. However, the actual performance for the heterogeneous backend in this specific case is limited by reading the input data, which was checked using a configuration that only loads the data into memory without any processing.

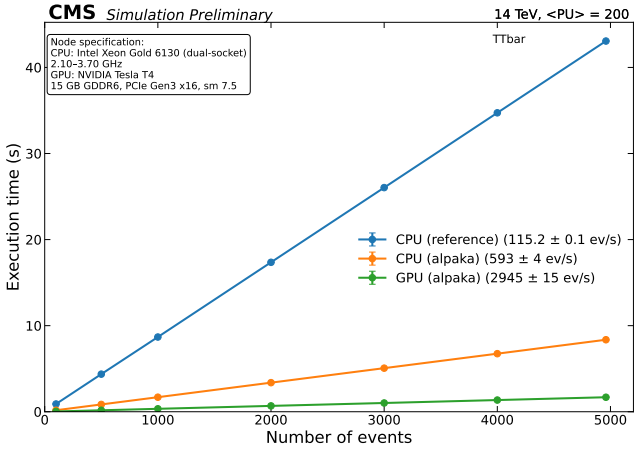


Fig. 5: Execution time as a function of events for different implementations: CPU reference implementation in blue, CPU alpaka in orange, and GPU alpaka in green. The legend also includes the throughput of each code at 5000 events.

The statistical uncertainties on each data point in Figure 5 was calculated as $\sigma_{\text{mean}} = \sigma/\sqrt{n}$, where σ is the standard deviation and n is the number of individual measurements. Over seven iterations, the uncertainties were for the CPU (reference) code ± 0.002 seconds, for the CPU (alpaka) implementation ± 0.02 seconds, and for the GPU (alpaka) implementation ± 0.01 seconds on average in execution time.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

Strong scaling studies, which analyze performance as a function of increasing resources allocated for a job, are currently ongoing. Initial results indicate that the performance of the alpaka serial implementation scales well, benefiting from the framework's parallelization strategies, and compares favorably with the CPU reference code in single-socket cases, and the heterogeneous alpaka implementation shows a roughly constant throughput largely independent of the number of CPU threads, as the computation is performed on the GPU. This later configuration achieves a rate that is more than two and a half times the peak performance of the reference CPU code. The details of the behavior of these implementations still require deeper understanding, particularly when transitioning to multi-socket cases. This further motivates the extraction of the alpaka unpacking algorithm from the CMS framework to allow for a more detailed and independent study of these behaviors in the future.

Moreover, the ongoing work includes a roofline analysis, which is used to construct a performance model that identifies computational bottlenecks related to memory bandwidth constraints. Initial results indicate that, in addition to the demonstrated gains achieved by porting in terms of portability and performance, we are still more than one order of magnitude away from reaching the strict upper roofline limit. This aspect will be addressed in future work focused on optimizing the code.

3.3 Core unpacking extraction

For further performance analysis and fully understanding of this implementation's behavior in multi-socket and shared NUMA domains cases, the core unpacking logic was extracted from the complex case study setup in order to implement more optimization techniques and further generalization possibilities.

At the initial stage, the standalone heterogeneous unpacking algorithm and its configuration eliminate dependencies specific to the CMS software framework(CMSSW) [13]. A validation of the standalone setup is performed by comparing the results from the CMSSW setup and the standalone implementation. These validations over a set of variables confirm the accuracy of the standalone version with a hundred percent match, making it suitable for further optimization studies in this case and for generalization in other unpacking algorithms across different experiments.

To provide an understanding of the performance of the unpacker under different execution models, we provide a comparison in terms of the throughput expressed in *events per second*. From Figure 6, we illustrate the obtained kernel-level implied throughput. This is calculated from the alpaka unpacking kernel execution time per event only and does not account for the other operations involved in the unpacking procedure, such as buffer allocation, copying operations between host and device, the IO operations, the framework effects, and so on. Here, we have plotted three configurations: the CMSSW alpaka-based serial unpacker in the single-threaded case, the standalone alpaka-based serial unpacker, and the standalone alpaka-based heterogeneous unpacker. Hence, we are able to

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

obtain the results specific to the implementation and execution models of the unpacker kernel solely.

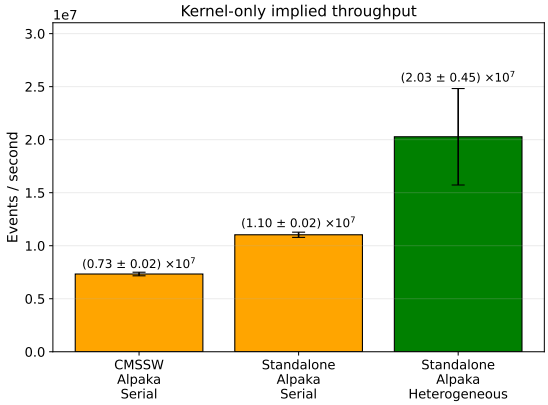


Fig. 6: Unpacking throughput measured in events per second, presenting the kernel-level implied throughput, calculated as a multiple of the per-event kernel running time, comparing the performance of the CMSSW alpaka serial unpacker to the standalone alpaka serial and heterogeneous implementations.

4 Conclusion

In this paper, we presented a hardware-portable implementation of an efficient data unpacking algorithm for particle detector readout systems. The solution is based on the C++ abstraction framework provided by the header-only alpaka library, enabling a unified code base that can be executed across multiple backends, including CUDA, HIP, OpenMP, and serial execution, while maintaining portability and code maintainability.

The results demonstrate the practicality and effectiveness of the proposed approach. The achieved throughput increased to an improvement of a factor of more than 25 in the case of the ported code compared to the reference code using 8 CPU threads, however, being limited by the reading of the input data. Importantly, this performance gain is obtained while preserving a conservative and fully lossless data unpacking procedure, ensuring data integrity under the strict throughput and reliability requirements of the CMS experiment.

This work highlights the growing importance of hardware-portable programming models in modern high-performance computing environments, where heterogeneous architectures are increasingly prevalent. By minimizing hardware-specific optimizations, the proposed approach simplifies software maintenance, improves portability, and facilitates collaborative development across diverse computing platforms.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

Future work of this study will focus on a deeper analysis of the alpaka-ported unpacking algorithm under multi-threaded execution, both within single and across multiple NUMA domains, in order to better understand the scalability of the code and to improve the utilization of available computing resources, and roofline analysis will be done in order to explore the algorithm's performance bottlenecks relative to hardware-imposed memory bandwidth and compute capacity limits. In addition, a further extension of this work will investigate the applicability of the presented methodology to other experimental domains with custom-designed data acquisition and readout systems.

Acknowledgments. This work was supported by the Data Science in Hamburg - Helmholtz Graduate School for the Structure of Matter (DASHH), funded by the Helmholtz Association and DESY. The authors would also like to acknowledge the CMS colleagues who contributed on the making of this work, particularly the High-Level Trigger (HLT) team, the Backend System (BES) colleagues, the Tracker Detector Performance Groups (DPG) and the CMS Phase 2 software group, as well as the supportive CMS colleagues at DESY.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Sterling, T., Anderson, M., Brodowicz, M.: High Performance Computing: Modern Systems and Practices. Morgan Kaufmann (2017)
2. TOP500 Homepage, `https://www.top500.org/lists/top500/2025/11/`, last accessed 2026/01/16
3. Green500 List, `https://www.top500.org/lists/green500/`, last accessed 2026/01/16
4. Bocci, A., et al.: Bringing heterogeneity to the CMS software framework. EPJ Web of Conferences 245, 05009 (2020)
5. E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann, *Alpaka – An Abstraction Library for Parallel Kernel Acceleration*, in Proc. 30th IEEE International Parallel and Distributed Processing Symposium Workshops (AsHES), Chicago, IL, USA, 2016. Available: `https://arxiv.org/abs/1602.08477`
6. C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, *Kokkos 3: Programming Model Extensions for the Exascale Era*, IEEE Trans. Parallel Distrib. Syst., vol. 33, no. 4, pp. 805–817, 2022. doi: 10.1109/TPDS.2021.3097283.
7. D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. Ryujin, and T. R. W. Scogland, *RAJA: Portable Performance for Large-Scale Scientific Applications*, in Proc. IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), Denver, CO, USA, 2019.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43

8. Alpaka — Advanced CMake, https://alpaka.readthedocs.io/en/stable/advanced/cmake.html, last accessed 2026/01/16
9. The CMS Collaboration, "The CMS Experiment at the CERN Large Hadron Collider," JINST 3 (2008) S08004.
10. CMS Collaboration, *Collaboration*, CMS Experiment at CERN. Available: https://cms.cern/collaboration, last accessed 2026-02-16.
11. CMS Collaboration: The Phase-2 Upgrade of the CMS Tracker. Technical Design Report CMS-TDR-014, CERN-LHCC-2017-009, CERN, Geneva (2017). https://doi.org/10.17181/CERN.QZ28.FLHW
12. CMS Phase-2 Outer Tracker Packer and Unpacker Modules, https://github.com/P2-Tracker-BES-SW/cmssw/tree/unpackers_16_0_0_pre1/EventFilter/Phase2TrackerRawToDigi, last accessed 2026/01/29.
13. CMS Offline Software, https://github.com/cms-sw/cmssw, last accessed 2026/02/16

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_43