EPJ Web of Conferences **337**, 01070 (2025) https://doi.org/10.1051/epjconf/202533701070
CHEP 2024

Evaluating FPGA Acceleration with Intel® oneAPI Toolkit for High-Speed Data Processing

Alberto Perro[1,2,*], *Paolo* Durante[1], *Flavio* Pisani[1], and *Eleni* Xochelli[1,3]

[1]CERN, Geneva, Switzerland
[2]Aix Marseille Univ, CNRS/IN2P3, CPPM, Marseille, France
[3]University of Thessaly, Volos, Greece

Abstract. The LHCb Experiment employs GPU cards in its first level trigger system to enhance computing efficiency, achieving a data rate of 32 Tb/s from the detector. GPUs were selected for their computational power, parallel processing capabilities, and adaptability.
However, trigger tasks necessitate extensive combinatorial and bitwise operations, ideally suited for FPGA implementation. Yet, FPGA adoption for compute acceleration is hindered by steep learning curves and very different programming paradigms with respect to GPUs and CPUs. In the last few years, interest in high level synthesis has grown because of the possibility of developing FPGA gateware in higher-level languages.
This study assesses the Intel® oneAPI FPGA Toolkit, which aims to simplify the development of FPGA-accelerated workloads by offering a GPU-like programming framework. We detail the integration of a portion of the current pixel clustering algorithm into oneAPI, address common implementation challenges, and compare it against CPU, GPU, and RTL implementations.
Our findings showcase promising outcomes for this emerging technology, potentially facilitating the repurposing of FPGAs in the data acquisition system as compute accelerators during idle data-taking periods.

1 Context

The LHCb Experiment during Run 3 (2022–2026) employs a trigger-less readout system combined with a full software trigger [1]. Detector data is acquired at the LHC bunch clock rate of 40 MHz, resulting in a maximum bandwidth of 32 Tb/s. This data is received by 445 PCIe-based Field-Programmable Gate Array (FPGA) readout cards, reconstructed into events within the Event Builder, and subsequently processed by a two-stage trigger system. The first stage, High-Level Trigger 1 (HLT1), primarily utilizes inclusive one- and two-track algorithms for event selection. To meet its objectives, HLT1 performs full track reconstruction at maximum throughput, efficiently reducing the input rate by a factor of 30–60, depending on the operational configuration.

The HLT1 algorithms are inherently parallelizable and can be efficiently implemented on many-core architectures, such as Graphical Processing Units (GPUs) [2]. Leveraging GPUs as accelerators for HLT1 enables a streamlined LHCb data acquisition architecture.

*e-mail: alberto.perro@cern.ch

GPUs can be integrated directly into the servers of the Event Builder farm, further reducing infrastructure costs. This approach has been successfully realized through the Allen framework [3], which demonstrates the feasibility of a high-throughput GPU-based trigger system. Allen represents the first implementation of such a system for a high-energy physics (HEP) experiment.

Given the large number of FPGAs available in the data acquisition system, which remain idle during periods without data-taking, this work explores the potential of exploiting these FPGAs to further accelerate the reconstruction and selection processes.

1.1 Challenges with FPGAs

A substantial portion of the HLT1 computational workload involves combinatorial and bitwise operations. These tasks are well-suited to FPGAs. However, FPGA development presents a steep learning curve. This is due to the different programming paradigms, complex hardware description languages, and cumbersome debugging tools, which make finding skilled developers difficult.

High-Level Synthesis (HLS) aims to address these challenges. It automates the translation of high-level system designs, typically written in subsets of C, into the register-transfer level structures required for FPGA deployment. Despite this, HLS still requires a deep understanding of FPGA architecture and debugging techniques. Its higher level of abstraction can further complicate the development process.

1.2 Intel® oneAPI FPGA Toolkit

To address these challenges, Intel introduced the Intel® oneAPI FPGA Toolkit [4], which simplifies FPGA development by providing a GPU-like programming environment. Central to this toolkit is the SYCL language, a platform-agnostic abstraction layer built on pure C++. SYCL enables a single-source programming model that integrates host and accelerator code in one file, using templated functions to distinguish between them. Its similarity to the CUDA Runtime API allows developers with GPU programming experience to adopt SYCL more easily.

The toolkit also offers powerful debugging capabilities. Developers can run their code on any host without an FPGA accelerator, as the system emulates the accelerator at a high level. This eliminates the need for cycle-accurate simulations, drastically reducing development time. Standard debugging tools like `GDB` and `valgrind` can be used as if debugging regular software. Furthermore, an estimated resource usage report for the FPGA can be generated within minutes, avoiding the need for full compilations. Only after the algorithm is fully tested and optimized, the user can proceed to run a full compilation, which can take several hours. The resulting code is a single executable that initializes and runs the FPGA-accelerated algorithms with no user intervention. Additionally, the toolkit includes performance profiling tools to monitor and benchmark the algorithms without incurring host overhead.

An algorithm used in HLT1 is detailed in Section 2 alongside its GPU implementation. Section 3 describes the process of porting this algorithm to an FPGA and the corresponding implementation architecture. Finally, benchmark results for the FPGA implementation are presented.

2 Algorithm

Incoming data from the subdetectors is encoded in formats specific to each subdetector. This *raw data* must be decoded before it can be used in reconstruction algorithms. This paper

focuses on a specific subdetector, the Vertex Locator (VeLo), and its data decoding and initial processing stage known as clustering.

2.1 Vertex Locator

The VeLo is a silicon vertex detector and the closest subdetector to the interaction point in LHCb. Its primary role is to enable LHCb to reconstruct displaced vertices, while also contributing significantly to tracking. The subdetector is constructed from 52 modules positioned on either side of the beam line and oriented perpendicular to it. Each module consists of 4 pixel sensors, and each sensor contains 3 chips with a grid of 256×256 pixels.

Encoding the data from every single pixel as raw data would require an enormous amount of bandwidth, far exceeding what is available. To address this, the designers grouped pixels into *SuperPixels* (SPs) consisting of 8 pixels each, encoding only the active SPs. This approach is efficient since the average occupancy of the subdetector is below 0.1%. Each SP is encoded along with its location coordinates relative to the sensor.

The VeLo raw data is organized into 208 *raw banks*, with each bank corresponding to a specific sensor. Each raw bank contains an *SP header* followed by a series of *SP words*. Within the SP word, there are fields to specify the position of the SP and the hitmap.

2.2 VeLo Clustering

Particles flying through the subdetector produce signals that often activate several neighboring pixels, a phenomenon that must be considered during reconstruction. The goal is to reconstruct what is called a cluster, which is defined as a set of neighboring active pixels. A neighbor is any pixel that touches another pixel by one of its edges or corners, a connectivity pattern known as *8-connectivity*. After reconstructing the cluster, the averages of the rows and columns, as well as the total size of the cluster, must be computed.

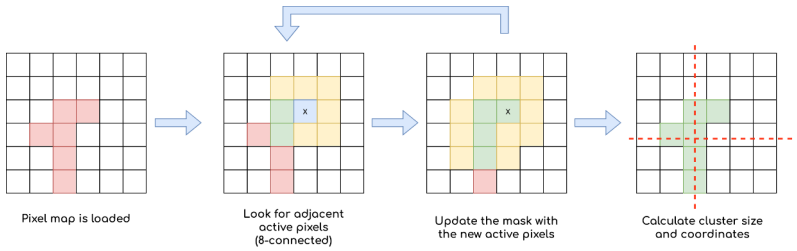

Figure 1. Illustration of the mask clustering algorithm.

The algorithm, referred to as *Mask Clustering*, is provided with a set of 208 raw banks and a set of candidates—active pixels likely to form a cluster—pre-selected during a preprocessing stage described in [5]. The algorithm, illustrated in Figure 1, follows these steps:

1. The SP containing the candidate active pixel is loaded into memory, along with its neighboring SPs, forming a map of three columns and four SPs each, for a total of 96 pixels. The candidate SP is positioned at $(1, 1)$.

2. A bit-mask is created around the cluster candidate, including the candidate pixel and its 8-connected neighbors.

3. A logical AND operation is applied between the mask and the map, which selects the active neighboring pixels.

4. A new bit-mask is generated around the newly identified active pixels, and Step 2 is repeated with this updated mask. This process continues iteratively until no new pixels are added to the cluster.

5. The cluster is finalized and its coordinates and size are computed.

The bit-mask itself is computed in constant time using eight shift operations combined with logical OR operations.

The algorithm described is highly parallelizable because there are no dependencies between sensors. This enables storing only a subset of SPs in local registers. GPU implementations typically utilize one or more routines, known as *compute kernels*, to handle computations. In the GPU implementation of this algorithm, a block-and-thread approach is adopted, where multiple kernels are launched to process candidates, banks, and events independently.

3 Architecture

The algorithm outlined in this work is available in three implementations: CPU, CUDA, and SYCL. The latter two are specifically optimized for execution on GPUs. While the SYCL implementation is marketed as cross-platform, capable of running on both CPUs and GPUs, additional modifications are necessary to adapt the code for execution on FPGA accelerators.

3.1 Single Task Kernels

FPGA accelerators do not natively support the concept of blocks and threads, which are intrinsic to GPUs. Consequently, kernels cannot be executed directly via the GPU-specific `parallel_for` mechanism, which offloads kernel dispatching and argument handling. Instead, FPGA systems require the use of the `single_task` method, which schedules a single kernel to the accelerator. This necessitates manual implementation of the `parallel_for`-like logic and corresponding modifications to the kernel. This represents the primary difference between GPU and FPGA implementations within the context of SYCL.

Upon making these modifications, the algorithm was successfully executed on the FPGA accelerator, specifically a BittWare IA-840F card, and produced correct results, which validated the feasibility of the approach. The initial implementation phase was completed in under a month, despite the developer's limited prior experience with SYCL or oneAPI. While the Proof of Concept (PoC) was functional, its performance was suboptimal, even lagging behind the CPU-only version as presented in Section 4.2.

3.2 Memory Transactions

A major distinction of FPGA accelerators is the limited on-board memory and slower external RAM relative to GPU architectures. This limitation makes it more efficient to perform memory transactions via Direct Memory Access (DMA) to system memory, rather than copying data to and from device memory.

Efficient DMA pipelining over the PCIe bus is not a trivial task, requiring the development of two additional kernels. These kernels are responsible for transferring data from host memory to the worker kernel –the kernel performing the actual computation– and returning

the results to host memory. These kernels are typically referred to as the *producer* and *consumer*. This design simplifies the code structure and reduces compilation time, as only the modified kernels are recompiled.

Analysis of the compiler reports revealed that the generated hardware design was inferring a high number of Load-Store Units (LSUs), the logic responsible for handling memory transactions. Excessive LSUs led to the insertion of an arbiter, which significantly hindered performance. To address this issue, the memory architecture was restructured so that the source data was encoded in a contiguous memory region, while the producer reads it sequentially. This modification allowed the system to infer a single LSU, thus using the available PCIe bandwidth and eliminating memory bottlenecks.

3.3 Pipelines

The design of a multi-kernel pipeline necessitates a mechanism for transferring output data from one kernel to the input of the next kernel. In GPU architectures, this transfer typically occurs through shared or global memory. However, in FPGA designs, this transfer is facilitated using *pipes*, a feature of the Intel® oneAPI Toolkit. Pipes are unidirectional data channels that connect two kernels and map to hardware as First-In, First-Out (FIFO) buffers.

Pipes can hold a predefined number of data elements before reaching capacity, at which point they stall, thus functioning as buffers that can mitigate back-pressure. Pipe operations can be either blocking or non-blocking. In the blocking case, the write operation is halted if the pipe is full, and similarly, the read operation is blocked when the pipe is empty, mimicking the behavior of FIFO memory.

3.4 Multiple Workers and Synchronization

Once memory transactions were optimized, compute operations became the next bottleneck. Compiler reports indicated that plenty of resources remained unused on the FPGA. At this stage, the algorithm was utilizing a single worker for computation. The next step was to parallelize the computation by introducing multiple workers. While this is straightforward on GPUs, implementing this on an FPGA is significantly more complex.

To achieve this, two new kernels were introduced. The first kernel distributes data across memory banks, while the second kernel distributes data across candidate workers. These kernels interface with the producer kernel on one side and the n-th worker on the other. Work is distributed in a round-robin manner at the single raw bank level. Additionally, a kernel was needed to collect the results from the workers and pass them to the consumer kernel. All kernels are connected with pipes tuned to avoid stalling due to back-pressure.

However, this setup addresses only the data flow. Synchronization between parallel kernels is not inherently guaranteed on FPGAs. To resolve this, a synchronization arbiter kernel was introduced. Its role is to coordinate event boundaries and prevent data corruption.

The synchronization arbiter communicates with the producer, consumer, and worker kernels using pipes. The execution flow is as follows:

1. The producer reads an event from system memory and dispatches it to the workers via the distribution kernels. The producer then halts, waiting for the release signal from the synchronization arbiter.

2. Each worker processes its assigned task, sends a completion signal to the synchronization arbiter, and then waits for the release signal.

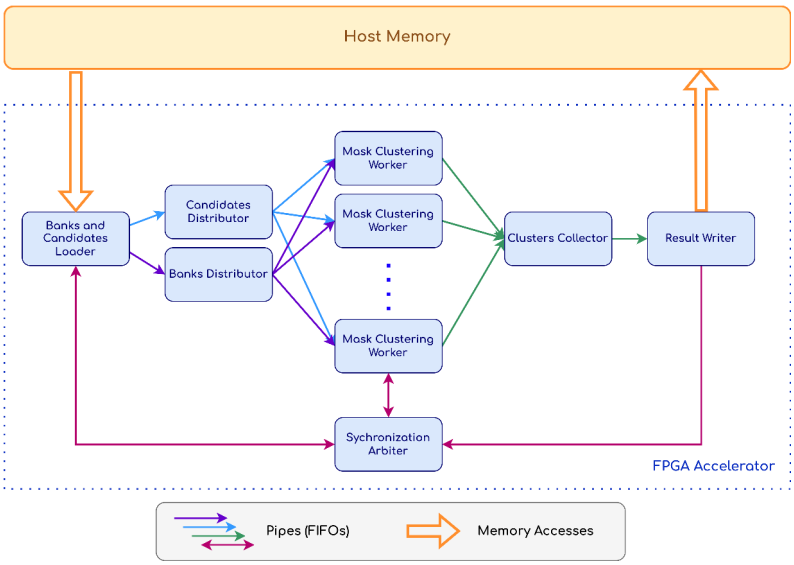

Figure 2. Synopsis of the FPGA optimized architecture. From the left, the *producer* kernel reads from the host memory candidates and banks. These are distributed to the workers via distributor kernels in a round-robin manner. Multiple workers apply the mask clustering algorithm. Results are collected by a dedicated kernel and written back to host by the *consumer* kernel on the right. The whole pipeline is kept synchronized by the synchronization arbiter.

3. The consumer collects results from the workers, sends a completion signal to the synchronization arbiter, and waits for the release signal.

4. Upon receiving all completion signals, the synchronization arbiter broadcasts a release signal. This allows the system to process the next event.

This synchronization mechanism is known in the literature as a *central barrier*. It ensures that each event is processed in a synchronized manner. This naive implementation guarantees correct results and high throughput, fully utilizing the resources of the FPGA. The synopsis of the complete architecture is shown in Figure 2.

4 Results

The proposed multi-kernel pipeline design was evaluated using the BittWare IA-840F FPGA Acceleration Card, featuring an Altera Agilex 7 F-Series FPGA. To assess platform stability and compatibility, tests were conducted on two separate hosts: one equipped with an Intel Xeon Gold 6326 CPU, and another with a newer Intel Xeon Gold 6426Y CPU. Both hosts provide a PCIe Gen4 x16 connection to the accelerator.

The kernel design is modular, allowing the number of workers to be specified at compile time. This modularity enables optimal utilization of FPGA resources.

4.1 Worker Scaling

The initial set of tests was designed to determine the optimal configuration of the FPGA accelerator by varying the number of worker kernels. Each test involved processing 100,000

events, using a batch size of 100 events repeated 1,000 times. The evaluation focused on two metrics: the event processing rate and PCIe input throughput. Output throughput measurements were omitted due to the runtime issue detailed in Section 4.3.

The results shown in Table 1 demonstrate that the system scales effectively, achieving a maximum event rate of 107.9 kHz with 16 worker kernels. While the performance improvement from 4 to 8 workers was nearly linear, the increase from 8 to 16 workers yielded only a 50% improvement, indicating the presence of a bottleneck. PCIe throughput analysis revealed that the kernel utilized less than one-fifth of the theoretical maximum bandwidth of 256 Gbps, confirming that the implementation is compute-bound rather than bandwidth-limited.

# workers	Event Rate (kHz)	PCIe Throughput (Gbps)	Speedup
4	37.5	15.8	1.0
8	71.4	30.0	1.9
16	107.9	45.3	2.9

Table 1. Worker scaling analysis results. The event rate, PCIe throughput, and corresponding speedup are presented for configurations with 4, 8, and 16 workers.

4.2 Platform Benchmarks

A comparative analysis of the algorithm's performance across various platforms was conducted under consistent test conditions: processing 100,000 events with a batch size of 100 events repeated 1,000 times. The platforms evaluated were as follows:

- **CPU:** Execution on an Intel Xeon Gold 6326 CPU using 16 compute threads.
- **GPU:** Execution on an NVIDIA RTX A5000 GPU hosted on a production server used for HLT1.
- **FPGA PoC:** An initial FPGA implementation based on the GPU algorithm with no architectural modifications, executed on the IA-840F card in the Xeon Gold 6326 host.
- **FPGA Optimized:** The optimized pipelined FPGA implementation described in this work, executed on the IA-840F card.

The benchmark results, shown in Figure 3, indicate that the FPGA PoC implementation was the worst performer, achieving only 300 Hz, which is 40 times slower than the CPU implementation. However, the optimized FPGA implementation outperformed the GPU implementation, achieving a 36% higher event rate. While this result is promising, it is important to note that the FPGA directly executes the algorithm, whereas the GPU implementation operates within the Allen framework, using a custom sequence to invoke the *mask clustering* kernel. Additionally, the FPGA benchmarks are influenced by the runtime issue discussed in Section 4.3.

4.3 Runtime Issues

The performance measurements of the optimized FPGA implementation are impacted by a bug in the oneAPI runtime. This bug causes kernel panics when the accelerator writes results back to the host. The issue was isolated and reported to both the hardware vendor and Intel. While they have confirmed its reproducibility, the problem has not yet been resolved at the time of writing.

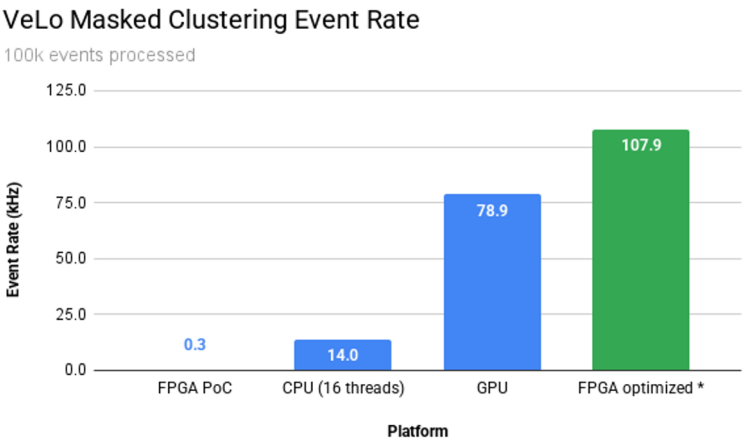

Figure 3. Results of the multi-platform benchmark. Note that the FPGA optimized result is impacted by the issue discussed in Section 4.3.

To circumvent this issue, benchmarks were conducted using a modified consumer kernel. This kernel mimics the behavior of the original but avoids accessing the PCIe interface. Based on input throughput measurements, it is assumed that the performance impact of write operations is negligible. This assumption is supported by the fact that only a fraction of the available PCIe bandwidth is utilized.

5 Conclusions

This work demonstrates the feasibility of employing FPGAs as accelerators for high-throughput data processing tasks. The Intel® oneAPI FPGA Toolkit enabled rapid adaptation of GPU code to FPGA, thanks to a consistent programming model and standard debugging tools. However, while porting the code to FPGA was relatively straightforward, achieving performance targets required wide optimization efforts due to the architectural differences between GPUs and FPGAs.

The modular pipeline design enabled efficient resource utilization, and the optimized FPGA implementation achieved a 36% higher throughput than a high-end GPU. Nevertheless, the development process revealed several limitations in the current state of oneAPI tools, with the most critical being the unresolved kernel panic issue during host writes.

While oneAPI represents a significant advancement in high-level synthesis for FPGA programming, its maturity level remains insufficient for production deployment in its current form. Further refinements to the toolchain and runtime environment are necessary to fully support FPGAs in high-speed data processing applications.

6 Acknowledgments

The authors would like to acknowledge the support from Christian Färber from Altera and Karol Hennessy and Kurt Rinnert from University of Liverpool.

References

[1] LHCb Collaboration, *LHCb Trigger and Online Upgrade Technical Design Report*, Tech. Rep. LHCB-TDR-016 (2014), https://cds.cern.ch/record/1701361

[2] LHCb Collaboration, *LHCb Upgrade GPU High Level Trigger Technical Design Report*, Tech. Rep. LHCB-TDR-021, CERN (2020), https://cds.cern.ch/record/2717938

[3] R. Aaij, J. Albrecht, M. Belous, P. Billoir, T. Boettcher, A. Brea Rodríguez, D. vom Bruch, D.H. Cámpora Pérez, A. Casais Vidal, D.C. Craik et al., Computing and Software for Big Science **4**, 7 (2020), https://doi.org/10.1007/s41781-020-00039-7

[4] *oneAPI: A New Era of Heterogeneous Computing*, https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html, accessed on 2024-09-16

[5] D.H. Campora Perez, Ph.D. thesis, Seville U. (2019), https://cds.cern.ch/record/2718278