# Performance of Heterogeneous Algorithm Scheduling in CMSSW

*Andrea* Bocci[1], *Christopher* Jones[2], and *Matti J.* Kortelainen[2,*]

[1]CERN, Geneva, Switzerland
[2]Fermi National Accelerator Laboratory, Batavia, IL, USA

**Abstract.** The CMS experiment started to utilize Graphics Processing Units (GPU) to accelerate the online reconstruction and event selection running on its High Level Trigger (HLT) farm in the 2022 data taking period. The projections of the HLT farm to the High-Luminosity LHC foresee a significant use of compute accelerators in the LHC Run 4 and onwards in order to keep the cost, size, and power budget of the farm under control. This direction of leveraging compute accelerators has synergies with the increasing use of HPC resources in HEP computing, as HPC machines are employing more and more compute accelerators that are predominantly GPUs today. In this work we review the features developed for the CMS data processing framework, CMSSW, to support the effective utilization of both compute accelerators and many-core CPUs within a highly concurrent task-based framework. We measure the impact of various design choices for the scheduling of heterogeneous algorithms on the event processing throughput, using the Run-3 HLT application as a realistic use case.

## 1 Introduction

CMS started to utilize Graphics Processing Units (GPUs) to accelerate the online reconstruction and event seleection in its High-Level Trigger (HLT) at the beginning of the LHC Run 3 [1]. The projections for both online and offline computing for LHC Run 4 and beyond foresee significant use of compute accelerators in addition of traditional CPUs. The Run 3 gives valuable development, maintenance, and operational experience on working with GPUs for CMS for the long term.

In this work we review the features that were developed for the CMS data processing framework, CMSSW [2–6], to effectively utilize GPUs and many-core CPUs simultaneously [7]. CMSSW implements multi-threading using the oneAPI Threading Building Blocks [8] library, and utilizes tasks as the concurrent units of work. Users implement the physics algorithms as modules within the framework, and the framework orchestrates the concurrent execution of the modules following their data dependencies. The GPU APIs are used directly from the modules. In practice so far the GPU API has meant NVIDIA CUDA, but CMS is adopting Alpaka portability library [9–11] during Run 3 to ease the development and maintenance of algorithms between CPU and NVIDIA GPUs [12]. Alpaka also should give CMS a straightforward path to target GPUs from other vendors. Given that underneath

[*]e-mail: matti@fnal.gov

Alpaka uses CUDA for NVIDIA GPUs, the findings of this work translate directly to Alpaka as well.

This paper is organized as follows. The scheduling and synchronization strategies that were tested are described in Section 2. The measurement methods and results are discussed in Section 3. The conclusions are given in Section 4.

## 2 Scheduling and synchronization strategies

### 2.1 Synchronous, one CUDA stream

The simplest scheduling strategy is to queue all work, i.e. kernels and data copies, into one CUDA stream. Usually this stream would be the so-called default CUDA stream, as that is the simplest way to use the CUDA API. In this work an explicitly created CUDA stream was used as a proxy, because that was easiest to achieve when starting from an application supporting multiple CUDA streams. The host and device are synchronized with a call to `cudaStreamSynchronize()` function that blocks the calling CPU worker thread until the work on the device has finished.

The main downsides of this simple approach are all work being queued in one CUDA stream, that doesn't allow expressing the potential concurrency from indenepdent work, and the synchronization blocking the CPU worker thread, that could do other work while the device is running the kernels.

### 2.2 Multiple CUDA streams

The simplest improvement to the pattern described in the previous subsection is to queue independent work to different CUDA streams. In CMSSW each non-branching chain of framework modules within an Event uses a separate CUDA stream, as depicted in Figure 1. In addition, each event being processed concurrently has its own chains and therefore CUDA streams. The CUDA stream management is implemented as a shared cache of CUDA streams that are dynamically picked by the first CUDA-using modules in the sub-graphs. In case the module dependency graph has a branch, i.e. a CUDA data product is consumed by many modules, one of the consuming modules re-uses the same CUDA stream, and the others pick a new CUDA stream from the cache. With this approach the available concurrency in the application is maximally expressed to the CUDA runtime, that can then schedule the work as it sees fit.

### 2.3 External worker

The next improvement is to replace the blocking synchronization with a callback-style solution, called *External Worker* in CMSSW. Traditionally the framework modules have had one function called by the framework. In case of producers the function is called `produce()`. Modules that use the External Worker mechanism have instead two functions, `acquire()` and `produce()`, depicted in Figure 2. The framework calls first the `acquire()` function, that should launch all the asynchronous work. In case of CUDA-using modules, this means queueing all the work into the CUDA stream provided by the framework. The `acquire()` function is also given a reference-counted holder object that holds the oneTBB task that will make the framework call the `produce()` function. The last action in the `acquire()` function is to queue a callback function to the CUDA stream with `cudaStreamAddCallback()`, passing also the holder object to the callback function. CUDA runtime calls the callback function after all the work queued in the CUDA stream in the `acquire()` function has finished, and

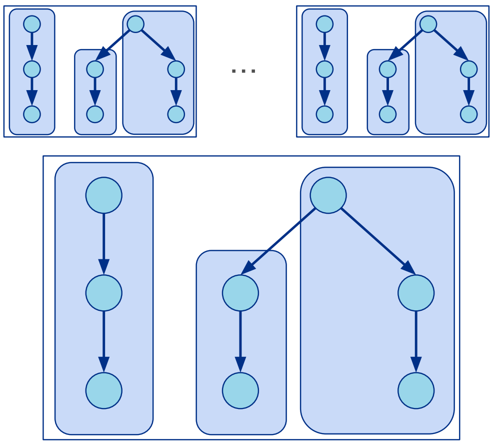

**Figure 1.** Example of chains of framework modules where 3 CUDA streams are being used in each event. The circles represent the framework modules, and the arrows the data flow between modules. The rectangles with rounded corners show the non-branching chains of modules, that each have their own CUDA stream. The rectangles depict the events.

the callback function notifies the holder object all the work has completed. The holder object then schedules the oneTBB task. The External Worker mechanism can propagate exceptions from the asynchronous work to the rest of the framework.

In CUDA documentation [13] the `cudaStreamAddCallback()` is listed as a deprecated function, but its replacement `cudaLaunchHostFunc()` does not report errors from the asynchronous processing. In case of data processing errors it is imperative to catch them and terminate the application early to avoid wasting computing slots by stalled jobs.

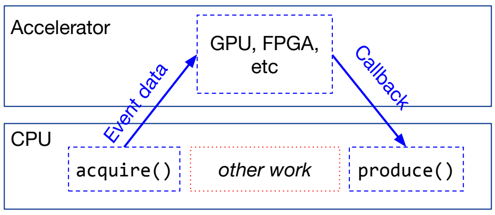

**Figure 2.** Illustration of how the External Worker mechanism allows the CPU thread to do other work while the accelerator, e.g. GPU, is doing work. The `acquire()` function queues data copies and kernels to be run on the GPU, and also a callback so that the module's work on the `produce()` function can resume after the offloaded work has completed.

## 2.4 Use external worker only when really needed

In practice it turned out that the only reason to synchronize the host and the device is after copying data from the device to the host. Often only the last CUDAmodule of a chain of CUDA-using modules needs to copy data from the device to the host, and the earlier modules only queue kernels or host-to-device data copies into the CUDA stream. Therefore having every CUDA-using module to synchronize the host and the device implies unnecessary work, and therefore avoiding these unnecessary synchronization calls should improve the data processing throughput, as depicted in Figure 3, because the subsequent modules can queue their work concurrently to the GPU already running the kernel(s) of the earlier module in the chain.

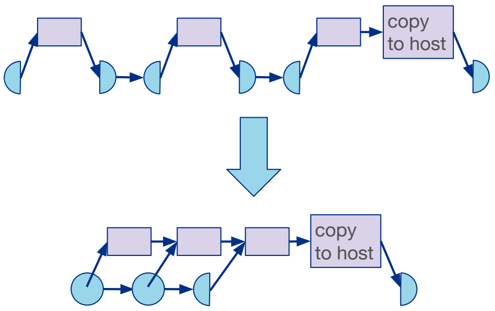

**Figure 3.** Illustration how limiting the use of the External Worker mechanism only for modules copying data from device to host could improve the data processing throughput. The circles denote the modules, with circle halves denoting the `acquire()` and `produce()` functions. The rectangles denote the kernels queued by the modules, with the rightmost rectangle being the device-to-host data copy, that requires the synchronization. Arrows denote the data flow.

The setup described up to this subsection, i.e. multiple CUDA streams combined with asynchronous execution pattern with External Worker applied only for the modules copying data from device to host memory, is the pattern that was used in production in CMS HLT in 2022 data taking.

### 2.5 Pool of waiting threads

We found out that the use of the callback functions with `cudaStreamAddCallback()`, or `cudaLaunchHostFunc()`, leads to noticeable CPU usage in the case the CPU is only waiting for the work on the GPU to finish. The 2022 HLT application has plenty of work to be done on the CPU, and therefore, in principle, such additional CPU load from synchronization could impact the performance of the CPU-side data processing, especially on a fully loaded compute node. Therefore we explored alternative synchronization mechanisms in case we could find a way to reduce the CPU usage.

We found that the following setup showed both lower CPU utilization in a GPU-bounded test than the setup described in the previous subsection. We created a separate pool of CPU threads that are mostly sleeping. At the end of the `acquire()` function, a CUDA event is recorded, and passed to one of these waiting threads that is currently free, along with the holder object. The waiting thread then calls the `cudaEventSynchronize()` function that blocks the thread. After the `cudaEventSynchronize()` function returns, the holder object is notified from the work completion, and the waiting thread is marked as free. The oneTBB worker thread that ran the `acquire()` function is free to do other work. When the CUDA event is created with the `cudaEventBlockingSync`, the blocking `cudaEventSynchronize()` function uses only very little CPU.

In addition, since `cudaEventSynchronize()` function reports errors from any asynchronous execution via return code, this waiting thread pattern is a viable replacement for the deprecated `cudaStreamAddCallback()`, that guarantees the application will always be terminated, rather than becoming starved, in case of errors on the GPU.

## 3 Results and discussion

The performance of the various scheduling and synchronization strategies described in Section 2 were compared using a setup mimicking an actual node in CMS HLT farm in 2022. The measurements were carried out on a machine like the HLT production nodes, i.e. with dual-socket AMD EPYC 7763 (Milan microarchitecture) CPUs and two NVIDIA Tesla T4 (Turing microarchitecture) GPUs. The node has 64 CPU cores per socket, with 2 hardware threads per socket, giving total of 256 CPU hardware threads.

The number of CPU threads per process and number of processes per node were varied such that their product was always equal to the 256 threads. The number of concurrent events per process was set to 3/4 of the number of CPU threads per process to conserve GPU memory. The HLT menu has enough intra-event parallelism that the event processing throughput is the same as if the number of concurrent events would be equal to the number of CPU threads. The measurements start at 16 CPU threads per process in order to fit all test cases to the 16 GB of memory of the T4 GPU. The event processing throughput are normalized to the throughput of the HLT menu run with CPU only.

The relative event processing throughput for the various test cases are shown in Figure 4. Already the simplest case, synchronous with one CUDA stream (Section 2.1), gave 15-45 % improvement compared to the CPU-only execution, depending on the number of CPU threads per process. The benefit from the GPU execution decreases clearly when number of concurrent events and CPU threads per process is increased. It is a clear sign of a concurrency bottleneck, likely lock contention, in the GPU-enabled application, although the exact cause is not known at the time of writing.

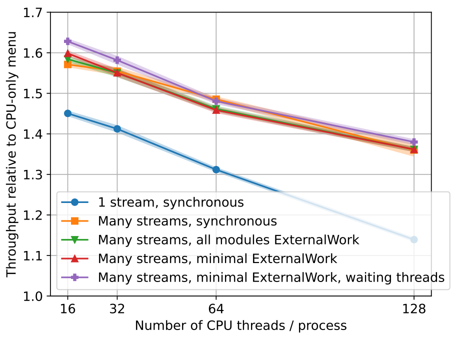

**Figure 4.** Event processing throughput of the tested scheduling and synchronization patterns relative to the HLT menu ran only on CPU as a function of the number of CPU worker threads per process. In each test case the number of events in flight was set to 3/4 of the number of CPU worker threads, and the number of processes was set such that the total number of CPU worker threads corresponded the total number of hardware threads of the node.

Enabling multiple CUDA streams (Section 2.2) gave a sizable, 7-20 % improvement in the throughput. This finding indicates that majority of the kernels of the application and the size of the data are not sufficient to fully utilize the T4 GPU, and/or data being transferred in parallel with the kernels. All the scheduling options `cudaDeviceScheduleAuto`, `cudaDeviceScheduleSpin`, `cudaDeviceScheduleYield`, and `cudaDeviceScheduleBlockingSync` were tested and gave practically the same throughput. Therefore the default option `cudaDeviceScheduleAuto` was used in Figure 4.

Replacing the explicit host-thread blocking synchronization from each framework module with the asynchronous, external worker mechanism (Section 2.3) gave the same performance as the blocking synchronization within 1.5 %. This observation means the application does not have substantial CPU-only processing need compared to the duration of the chains of kernels and data copies, to offset the cost of the additional complexity of the asynchronous processing.

Removing the external worker -style synchronization from the modules that do not need a synchronization (Section 2.4) had only minimal, 1 % improvement in the throughput, and was visible only in the 16-CPU-thread case.

Replacing the `cudaStreamAddCallback()` with our own pool of waiting threads that call `cudaEventSynchronize()` gave about 2 % higher throughput, that was also the highest for all thread counts.

## 4 Conclusions

In this work we demonstrated the performance impact of the design decisions of the CUDA-using module pattern in CMSSW, using the CMS High Level Trigger application with a menu used in 2022 data taking as a realistic test bed. Already a simple single-stream approach with blocking synchronization gave significantly, (15–45 %, depending on the number of CPU threads per process) higher event processing throughput than a CPU-only menu. Adding support for multiple CUDA streams improved the throughput by 7–20 %, whereas the asynchronous execution features had only small, percent-level impact, and in some cases decreasing the throughput. Interestingly combining the asynchronous execution pattern with an application-level pool of threads used only for blocking synchronization with CUDA events gave the highest throughput on all tested thread counts over blocking synchronization in the CPU worker thread and asynchronous execution with callback functions. We anticipate with more computationally expensive algorithms being ported to GPUs the benefits of the asynchronous processing would become more apparent.

## Acknowledgements

This document was prepared by the CMS Collaboration using the resources of the Fermi National Accelerator Laboratory (Fermilab), U.S. Department of Energy, Office of Science, HEP User Facility. Fermilab is managed by Fermi Research Alliance, LLC (FRA), acting under Contract No. DE-AC02-07CH11359.

## References

[1] G. Parida, these proceedings

[2] C.D. Jones, M. Paterno, J. Kowalkowski, L. Sexton-Kennedy, W. Tanenbaum, *The New CMS Event Data Model and Framework*, in *Proceedings of International Conference on Computing in High Energy and Nuclear Physics (CHEP06)* (2006)

[3] C.D. Jones, E. Sexton-Kennedy, J. Phys.: Conf. Series **513**, 022034 (2014)

[4] C.D. Jones, L. Contreras, P. Gartung, D. Hufnagel, L. Sexton-Kennedy, J. Phys.: Conf. Series **664**, 072026 (2015)

[5] C.D. Jones, J. Phys.: Conf. Series **898**, 042008 (2017)

[6] C. Jones, P. Gartung, these proceedings

[7] A. Bocci, D. Dagenhart, V. Innocente, C. Jones, M. Kortelainen, F. Pantaleo, M. Rovere, EPJ Web Conf. **245**, 05009 (2020)

[8] *oneAPI Threading Building Blocks*, https://github.com/oneapi-src/oneTBB (2023), accessed: 2023-08-25

[9] B. Worpitz, *Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures* (2015)

[10] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W.E. Nagel, M. Bussmann, *Alpaka - An Abstraction Library for Parallel Kernel Acceleration* (IEEE Computer Society, 2016), `1602.08477`

[11] A. Matthes, R. Widera, E. Zenker, B. Worpitz, A. Huebl, M. Bussmann, *Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the Alpaka library* (2017), `1706.10086`

[12] A. Bocci, these proceedings

[13] NVIDIA, *CUDA Toolkit Documentation 12.2 Update 1*, accessed: 2023-08-23, `https://docs.nvidia.com/cuda/`