

# Asynchronous Offloading in Gaudi

Paolo Calafiura<sup>1</sup>, Julien Esseiva<sup>1</sup>, Xiangyang Ju<sup>1</sup>, Charles Leggett<sup>1</sup>,  
Beojan Stanislaus<sup>1</sup>, Vakhtang Tsulaia<sup>1</sup>

<sup>1</sup>Lawrence Berkeley National Laboratory, Berkeley, California, USA

E-mail: bstanislaus@lbl.gov

**Abstract.** The ATLAS and LHCb experiments share a common data processing architecture called Gaudi. In Gaudi, data processing workloads are ultimately split into units called Algorithms, and a smart scheduler (the Avalanche scheduler) is used to schedule these Algorithms on a fixed pool of CPU threads managed by Intel's TBB.

Here we present a prototype of an addition to this scheduler which places such GPU-accelerated algorithms on a separate pool of dedicated threads. By making use of lightweight Boost Fibers which can be suspended without suspending the underlying OS thread, we can run the GPU workload asynchronously, without blocking the thread. This allows more efficient use of the CPU resources and, where the work offloaded by a single Algorithm doesn't fill the GPU resources available, can also improve GPU-efficiency by making use of separate CUDA streams.

## 1 Introduction

### 1.1 Gaudi

Gaudi [1] is the common basic data processing framework used by the ATLAS [2] and LHCb [3] experiments, among others. Gaudi is used to process a series of independent events (corresponding to bunch collision events for ATLAS and LHCb) by running a predefined series of transformations (called Algorithms) over each event. These algorithms are scheduled onto a pool of threads (one per CPU core) managed by TBB [4], but Gaudi controls this scheduling by only adding a TBB task when there is a free thread for the task to be executed on. This scheduling mechanism is called the Avalanche or Hive scheduler.

To date almost all algorithms used in ATLAS have been local CPU algorithms. Under these conditions the Avalanche scheduler makes good use of the available CPU resources, keeping all CPUs busy as far as possible. This is aided by being able to run work from a subsequent event on a given thread if there is insufficient parallelism available within an event.

### 1.2 GPUs and Other Accelerators

The situation is, however, changing. Particularly on HPCs, the majority of the FLOPS are now provided by GPUs and therefore the LHC experiments must be able to take advantage of these accelerators in order to make prudent use of the limited financial resources available. As a result work is being done to port parts of the ATLAS workflow to GPUs.

When algorithms that perform their computational work on a GPU or other accelerator are introduced, the Avalanche scheduler is no longer able to make good use of the available CPU resources. The scheduler must ensure that no algorithm is run before all of its data dependencies are available. Since each algorithm is responsible for any offloading it does, and the data produced by an algorithm are considered to be available as soon as the algorithm has finished running, an algorithm must wait for any work it is running on an accelerator to be complete before it can return its thread to the scheduler. As a result CPU time is wasted, since the core occupied by an algorithm that has offloaded its work remains idle. This model is illustrated in Figure 1.

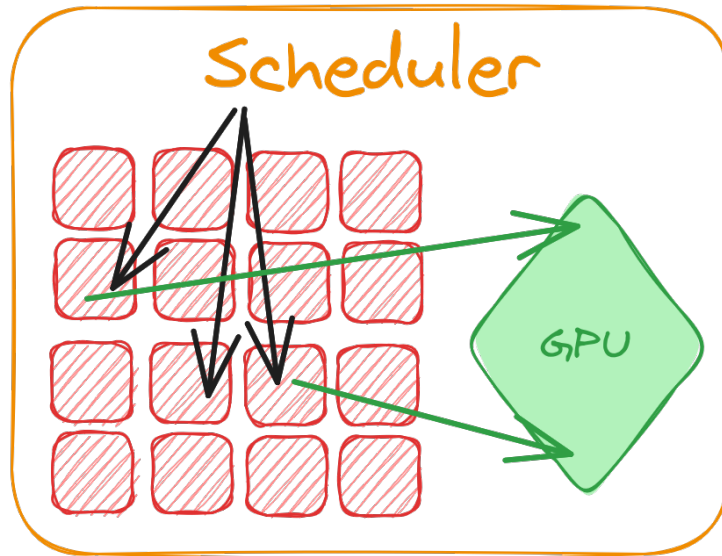


Figure 1: The model currently used by Gaudi for scheduling of work offloaded to GPUs. The black arrows represent algorithms while the green arrows represent GPU kernels.

## 2 Explicitly GPU-Aware Scheduling

In order to avoid this wasted CPU time, we desire a scheduler that is somehow aware of whether an algorithm runs on the CPU or the GPU, and can deal with this appropriately. The overarching model is illustrated in Figure 2.

One solution would be an explicitly GPU-aware scheduler, which would exactly follow the model illustrated in Figure 2. CPU algorithms would be scheduled as TBB tasks as they are now. GPU algorithms, on the other hand, would be scheduled onto a pool of GPU “threads” (e.g. CUDA streams), and Gaudi would track CPU resident and GPU resident data, and migrate as needed. However, this has a number of disadvantages:

1. Only local GPU offloading would be supported by this solution
2. Gaudi would be forced to select and specify which GPU technology is supported, since it is providing so much of the infrastructure
3. A significant amount of complexity would be added to the scheduler. The scheduler would be required to coordinate migration of data between the CPU and GPU, and track and manage the use of GPU computational capacity and memory. This is further complicated by the fact that a GPU cannot be divided into a number of independent cores in the way a CPU can.
4. As a result of the scheduler providing so much of the infrastructure, GPU algorithms would probably have to be limited to a single kernel.

## 3 Asynchronous Algorithms

The above disadvantages can be avoided by instead taking the approach of providing a general ability to schedule asynchronous algorithms, which are not expected to use much CPU time. These asynchronous algorithms would otherwise have a great deal of flexibility as to where data is kept, and what technologies are used for offloading. Meanwhile the complexity added to the Gaudi scheduler is minimal, requiring only that the idle CPU cores that would otherwise be blocked by an algorithm waiting on a GPU be reused. This, therefore, is the model we have chosen, and which is described in these proceedings.

### 3.1 Overcommitting the CPU

A variation of this idea is to simply overcommit the CPU, that is, use more threads in the thread pool than the number of available CPU cores. Modern operating systems, such as Linux, which is used for

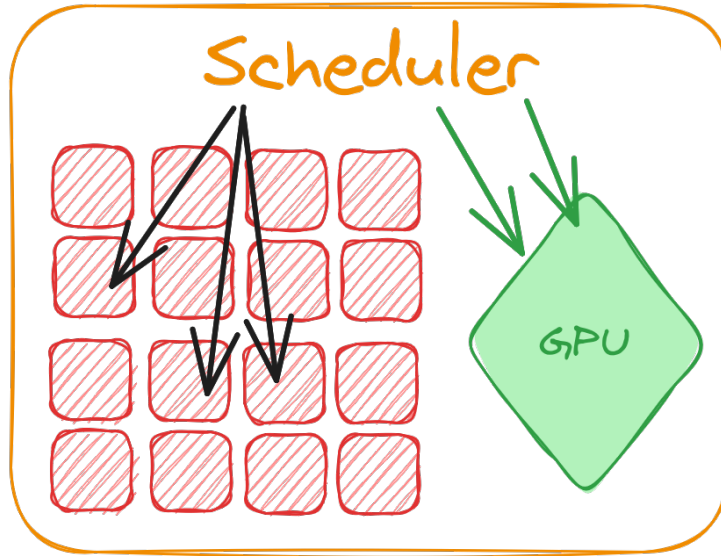


Figure 2: The desired scheduler model, with Gaudi aware of whether an algorithm runs on the CPU or the GPU and scheduling it accordingly. The black arrows represent CPU algorithms while the green arrows represent GPU algorithms.

all production data processing in ATLAS, will time-share the CPU over all threads. Threads which are waiting on an accelerator could just yield their timeslice. While this implementation requires no changes to Gaudi, it imposes an artificial limit on the number of GPU algorithms that can run at one time. In addition, if too many GPU algorithms are run at one time we begin silently losing CPU efficiency, since the number of simultaneous CPU algorithms will fall below the number of available cores.

### 3.2 Suspendable Boost Fibers

The solution selected instead utilizes suspendable fibers.<sup>1</sup> We have a small pool of OS threads on which we are able to schedule a large number of fibers. These fibers cooperatively multitask, with fibers permitted to continue running until they either complete or voluntarily suspend and yield the OS thread.

A schematic of the design is presented in Figure 3.

First we create two pools of threads. The first, managed by TBB, contains one thread per CPU core. All CPU algorithms are scheduled onto this pool, in the manner currently adopted by the Avalanche scheduler.

The second pool, managed by Boost Fiber [5], consists of only two or three total threads. GPU (or otherwise asynchronous) algorithms, which are not expected to consume much CPU time, are each scheduled onto new fibers on this thread pool. Since Boost Fibers are lightweight, and context switches are very fast [6], it is not necessary to throttle or otherwise manage the creation of these fibers.

The GPU algorithms are expected to internally manage data movement between the CPU and GPU, and the launching of work on the GPU. Whenever the algorithm is waiting for data to be copied to / from the GPU, or waiting for work to be completed on the GPU, it is able to suspend the fiber on which it is running, allowing the OS thread to be used for the next fiber in the queue.

While there are no strict constraints placed on the technologies used for offloading, most GPU algorithms, at least in the near future, are likely to use CUDA. We therefore provide a few convenience features for use with CUDA, including:

1. A wrapper around a Boost Fiber feature that allows the fiber scheduler to check if a CUDA stream being awaited is ready before rescheduling a suspended fiber. This avoids the need to keep scheduling

<sup>1</sup>These can also be called lightweight threads, or stackful coroutines. TBB tasks are also an example of the same concept, though until oneTBB they could not be suspended, and the suspension functionality is still not as developed as in Boost Fiber.

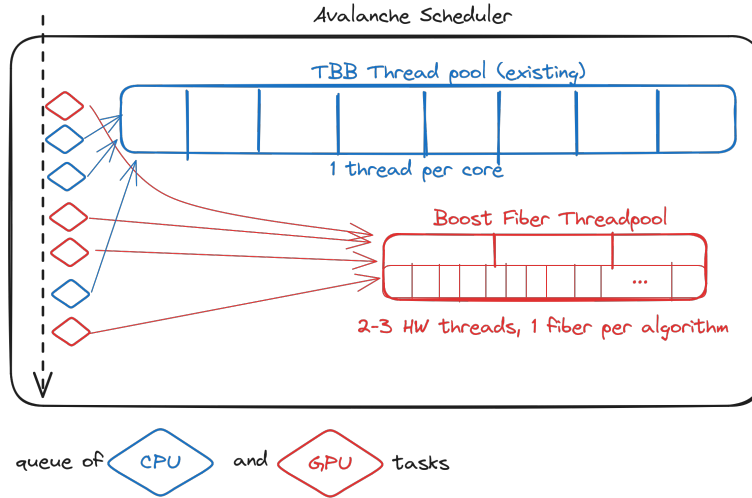


Figure 3: Schematic of the asynchronous algorithms design, using Boost Fibers.

a fiber, only for the fiber to yield again because the stream is not yet ready. The availability of this features was one reason why Boost Fiber was chosen for this task.

2. The ability to re-use CUDA streams, since the time to destroy a stream and construct a new one is non-trivial.
3. A wrapper for allocating GPU memory, which yields the fiber instead of failing if the GPU is out of memory. We also take care of freeing the allocated memory when the algorithm completes execution. This cleanup does however prevent data from being left on the GPU between algorithms, so we would later need to add explicit functionality to allow this if desired.

#### 4 Performance

Finally in Figure 4 we show a demonstration of the performance of this solution on an artificial GPU cruncher task. We can see that we were able to run 48 simultaneous CUDA streams on a pool of two OS threads, and make full use of the GPU compute and memory.

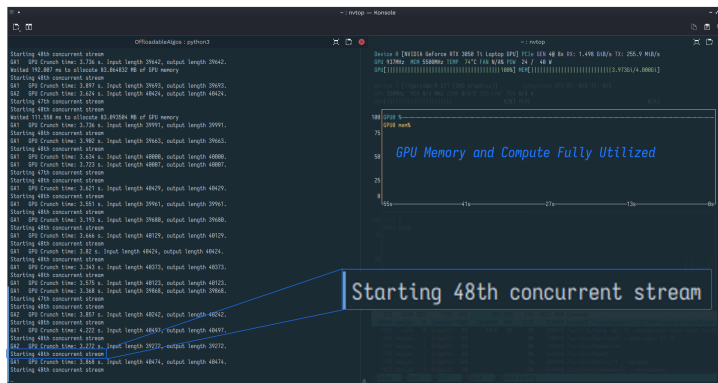


Figure 4: Demonstration of Gaudi running an artificial GPU cruncher asynchronous algorithm.

## Acknowledgements

This work was supported by the DOE HEP LHC ATLAS Detector and Computing Support program at Lawrence Berkeley National Laboratory under B&R KA2102021. We also thank the ATLAS collaboration for its support and cooperation. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

## References

- [1] LHCb Collaboration and ATLAS Collaboration 2024 Gaudi URL <https://zenodo.org/doi/10.5281/zenodo.3660963>
- [2] ATLAS Collaboration 2024 Software and computing for Run 3 of the ATLAS experiment at the LHC version Number: 1 URL <https://arxiv.org/abs/2404.06335>
- [3] Roiser S, Bozzi C and on behalf of the LHCb Computing Project 2018 *J. Phys.: Conf. Ser.* **1085** 032049 ISSN 1742-6596 publisher: IOP Publishing URL <https://dx.doi.org/10.1088/1742-6596/1085/3/032049>
- [4] Intel® Threading Building Blocks Release Notes and New Features URL <https://www.intel.com/content/www/us/en/developer/articles/release-notes/intel-threading-building-blocks-release-notes.html>
- [5] Oliver Kowalke Boost Fiber Documentation URL [https://www.boost.org/doc/libs/1\\_85\\_0/libs/fiber/doc/html/index.html](https://www.boost.org/doc/libs/1_85_0/libs/fiber/doc/html/index.html)
- [6] Oliver Kowalke 2024 boostorg/fiber original-date: 2012-12-05T16:17:03Z URL <https://github.com/boostorg/fiber>