EPJ Web of Conferences **245**, 05006 (2020) https://doi.org/10.1051/epjconf/202024505006
CHEP 2019

GPU Usage in ATLAS Reconstruction and Analysis

Attila Krasznahorkay[1,*], *Charles* Leggett[2], *Alaettin Serhan* Mete[3], *Scott* Snyder[4], and *Vakho* Tsulaia[2]

[1]CERN, Geneva; Switzerland
[2]Lawrence Berkeley National Laboratory, Berkeley CA; United States of America
[3]University of California Irvine, Irvine CA; United States of America
[4]Brookhaven National Laboratory, Upton NY; United States of America

Abstract. With Graphical Processing Units (GPUs) and other kinds of accelerators becoming ever more accessible, High Performance Computing Centres all around the world using them ever more, ATLAS has to find the best way of making use of such accelerators in much of its computing.

Tests with GPUs – mainly with CUDA – have been performed in the past in the experiment. At that time the conclusion was that it was not advantageous for the ATLAS offline and trigger software to invest time and money into GPUs. However as the usage of accelerators has become cheaper and simpler in recent years, their re-evaluation in ATLAS's offline software is warranted.

We show new results of using GPU accelerated calculations in ATLAS's offline software environment using the ATLAS offline/analysis (xAOD) Event Data Model. We compare the performance and flexibility of a couple of the available GPU programming methods, and show how different memory management setups affect our ability to offload different types of calculations to a GPU efficiently.

1 Introduction

With the computing landscape rapidly changing in the last few years, the ATLAS Experiment [1] had to renew its investigations into the usage of non-CPU devices for its offline computing. Evaluations for using NVidia GPUs using the CUDA programming language [2] in the ATLAS High Level Trigger have been done during Long Shutdown 1 of the LHC [3]. The conclusion for ATLAS's Run-2 was not to use GPUs as part of the trigger system. Both due to a difficulty in re-writing the complex reconstruction code in a way that CUDA could understand at the time, and due to the high price of General Purpose GPUs (GPGPUs) of the era.

Improvements in those areas, along with significant improvements in the ATLAS offline software infrastructure, the new investigation promises much more positive results for the experiment.

*e-mail: Attila.Krasznahorkay@cern.ch

1.1 The ATLAS Experiment and its Software

ATLAS [1] is one of the general purpose physics experiments at the LHC [4]. Its sub-detectors provide information about the LHC's proton-proton collisions in O(100M) read-out channels. The raw information read out of the experiment during data taking needs to be interpreted by complex algorithms to find what particles were most likely created in any proton-proton collision event, and with what exact properties. The experiment's software also needs to be able to fully simulate how particles arising from different physics processes would interact with the detector, in order to understand the data collected.

Most of the ATLAS software is built on top of the Gaudi [5] software framework. The organisation of the code into "algorithms", which can be executed in parallel on multiple threads using TBB [6] is orchestrated by this core framework code.

The offline software of ATLAS is managed in a single Git repository [7]. All of the code is organised into "packages", which can be built in different combinations into different projects, that have specific goals.

- The `Athena` project is used for the simulation digitisation and data/simulation reconstruction tasks;
- The `AthGeneration` project is used for Monte-Carlo event generation;
- The `AthAnalysis` project is used for the physics analysis of the reconstruction data;
- etc.

The ATLAS offline software repository holds a total of O(4M) lines of C++ and O(2M) lines of Python code.

Keeping and sharing all of the code in a central place allowed us to harmonise all aspects of the code between vastly different operations. It shall also help us with writing "portable" code for different types of hardware in the same way for our simulation, reconstruction and analysis software.

1.2 Heterogeneous Computing

In recent years the importance of non-CPU-based computing has increased significantly. The latest US based supercomputers for instance do and will all provide most of their computing power using GPGPUs.

- Frontier at Oak Ridge [8] will provide AMD CPUs and GPUs;
- Aurora at Argonne [9] will provide Intel CPUs and GPUs;
- The current largest HPC, Summit at Oak Ridge [10] is providing Power9 CPUs and NVidia GPUs.

To make use of these "new types of resources", multiple different approaches exist.

1. Program the specific target hardware in a hardware-specific low-level language, taking the strengths and weaknesses of the hardware into account;
2. Write the program in a standardised, higher level language, which hides some of the details of the underlying platform, but allows for fewer optimisations;
3. Make use of high level libraries that translate higher level, more abstract tasks into calculations on different, specific hardware.

While all of these have areas where they are the best possible choice, for writing reconstruction and simulation code for large HEP experiments like ATLAS, the second option seems to be the the only feasable one.

In "high level programming languages" one still has a number of choices, which all have upsides and downsides to them.

- OpenCL [11] is a cross-platform standard meant to provide a uniform programming interface to a wide range of accelerators. Unfortunately its support from AMD and NVidia is practically non-existent by now.
- CUDA [2] is the most "established" out of the available languages. It is supported both by NVidia's own compiler, and, to some degree by LLVM/Clang, but only for NVidia hardware in all cases.
- ROCm/HIP is providing a programming interface part way between OpenCL and CUDA. It is mainly meant as a way for programming AMD GPUs, but at the moment AMD also provides a way to generate code for NVidia devices from ROCm/HIP sources.
- SYCL [12] / oneAPI [13] is a pure C++ interface in the sense that it doesn't require any extensions to the C++ language for its syntax. Like OpenCL it is an open standard, in practice only being supported by Intel at the moment.
- OpenMP / OpenACC also allow "pure" C++ code to be compiled for accelerators. But while they proved very appropriate for many applications, they do not seem to scale to the ATLAS offline software's size.

In this most recent study we investigated the usage of OpenCL, CUDA and SYCL/oneAPI code with the ATLAS offline software. The following will describe our experience with these different programming methods.

2 TBB Based Multi-Threading in Gaudi/Athena

The ATLAS offline software is built on the Gaudi framework, which is implemented in a collaboration between ATLAS, LHCb and the CERN SFT group.

Calculations in Gaudi/Athena are performed by "algorithms", which are classes that need to provide a function that gets called for every "event" being processed. In this function the algorithms can retrieve their necessary inputs using an object whiteboard, perform their operations, and then record new objects into the whiteboard before finishing.

After LHC's Run-2 Gaudi was updated to execute algorithms using the Threading Building Blocks (TBB) library. In this setup all algorithms must explicitly declare what object(s) they require to run, and what object(s) they produce as an output. Using this information a component of Gaudi called the "scheduler" can figure out during a job how to feed TBB tasks to the TBB runtime system such that algorithms would be executed with the highest efficiency.

2.1 (A)Synchronous Accelerator Usage

The most widely used method for running accelerated calculations is the following:

1. Allocate memory on the device for the proceeding calculation, and copy all input data needed for the calculation, to the device;
2. Launch the calculation on the accelerator, and wait in the CPU thread until the calculation is finished;

3. Copy the results of the calculation from the device back to the host, and (possibly) free the memory that is no longer required.

This setup can prove perfectly appropriate for calculations that are primarily happening on the accelerator. And/or when the accelerated calculation is taking much less time than the same calculation would on the CPU.

In the case of HEP software, especially in the current "migration period" while we are working on establishing efficient programming methods for accelerators, we need to be smarter with how we schedule accelerated calculations. We need to make sure that CPU threads spend as little time as possible with setting up accelerated calculations, and especially that they do not ever have to wait for synchronisation points with the accelerator.

Most current high-level languages provide ways of achieving such a behaviour when using multiple CPU threads. In all cases the accelerator programming languages provide ways of notifying the host code about the completions of selected operations on the accelerator. Allowing the host code to schedule the retrieval of the result data from the device and the launch of calculations dependent on that data, at the next convenient step in the TBB task execution. To make use of this, the TBB based code execution has to have a concept of the asynchronous execution happening on accelerators.

3 Asynchronous Gaudi

To evaluate different accelerator programming techniques in the same context, a dedicated software project was set up [14]. It provided a convenient way of experimenting with changes in Gaudi's TBB based scheduling system independent of the full offline software build of ATLAS.

3.1 Code Organisation

The project was set up to build Gaudi, the core analysis Event Data Model code of ATLAS and the code being tested, as a single CMake project.

The project uses a common algorithm base class (`ASync::Algorithm`) to allow implementing both "synchronous" and "asynchronous" algorithms. To do this, the base class provides the following interface:

```
namespace ASync {
  class Algorithm : public Gaudi::Algorithm {
    ...
    /// Execute the algorithm on the host or an asynchronous device
    virtual StatusCode mainExecute( const EventContext& ctx,
                                    AlgTaskPtr_t postExecTask ) const;
    /// Run a post-execution step on the algorithm
    virtual StatusCode postExecute( const EventContext& ctx ) const
    /// Implementation of the base class's @c execute function
    virtual StatusCode execute( const EventContext& ctx ) const override;
    /// Check if the algorithm is set up to execute itself asynchronously
    virtual bool isAsynchronous() const;
    ...
  };
}
```

In order for an algorithm to implement an asynchronous interface it needs to:

1. Implement the `mainExecute(...)` and `postExecute(...)` functions, which shall take care of launching an asynchronous calculation, and collecting its results, respectively;

2. Make sure that `tbb::task` object received through the `AlgTaskPtr_t` smart pointer object would get scheduled when the offloaded calculation is finished.

It was possible to implement test algorithms on top of this common base which would either run using only the CPU "synchronously", or use an accelerator for their calculation either "synchronously" or "asynchronously".

The project also provides a custom "Gaudi scheduler" (`ASync::SchedulerSvc`) that would orchestrate the execution of algorithms implementing the `ASync::Algorithm` or `Gaudi::Algorithm` interfaces, in a way that would maximise the CPU usage of the application.

3.1.1 Asynchronous Execution With SYCL/OpenCL

In order to be able to make use of the NVidia GPUs available to us, we had to use a maximum of OpenCL version 1.2 in all of our tests. This means that all of the accelerated code has to be handled separately from the rest of our C++ source code, and we have to ship the OpenCL sources along with our compiled binaries.

Following the design of the user interface provided by `tbb::flow::opencl_node` (which we could not use directly due to an incompatibility of `tbb::flow` with Gaudi's scheduling system with TBB) we experimented with providing a variadic template based programming interface for executing "vanilla" OpenCL code from our algorithms. This effort was however abandoned before its conclusion, as by that time it was clear to us that OpenCL itself would not be an appropriate programming interface for ATLAS.

Instead we started looking at SYCL, more specifically Intel's implementation as part of its oneAPI platform. To perform asynchronous calculations with SYCL, we extracted variables from ATLAS EDM objects individually using `cl::sycl::buffer` objects, and then used basic SYCL code to run calculations on these buffers. Unfortunately in order to receive a callback about asynchronous calculations finishing, we had to rely on OpenCL's `clSetEventCallback(...)` as the SYCL API does not currently provide a C++ interface for such a feature.

3.1.2 Asynchronous Execution with CUDA

To use NVidia GPUs to their full potential, tests specifically using CUDA were written.

For simplifying the memory handling between the host and the NVidia GPUs, a special class (`AthCUDA::AuxStore`) was written that would allow xAOD style objects [15] to offload their data to the GPU, and get the results of a calculation back, with just a few lines of user code.

In order to perform GPU calculations with CUDA asynchronously, every memory copy operation and kernel launch was assigned to CUDA queues, and the TBB based scheduling was notified of the completion of GPU calculations using CUDA's `cudaLaunchHostFunc(...)` function.

Since all CUDA memory creation/deletion operations are managed by a global mutex, extra care had to be taken to serialise the creation and deletion of memory areas both on the device and the host.

3.2 Tests

To test the performance of running calculations through CUDA and SYCL, the same ATLAS Monte-Carlo reconstruction "toy configuration" was used as for the development of Gaudi's

TBB based scheduling. This configuration describes all algorithms taking part in the ATLAS reconstruction, with the time that they each took on a reference CPU, and all the data dependencies and products that the algorithm require and produce.

Using this information "toy" algorithms were written that would execute the number of floating point operations on the CPU or GPU that correspond to the time recorded in the previously described configuration, on the test machine's CPU. This simplicity also meant that higher order effects, that one usually runs into when writing real GPU code, could not be properly emulated by this toy code.

The algorithms running calculations on a GPU assumed that all calculations could be vectorised to 100 parallel threads, and were set up to allow us to run a configurable number of floating point operations with respect to the "ideal" value taken from the previously described calculation.

The algorithms were also set up to transfer a small amount of memory to and from the GPU before and after the calculation. However no fine-tuning was done to adjust the memory amounts of the individual algorithms in any realistic way, they all copied the same small amount of data. Because of this setup the toy algorithms were not set up to be "chained" on the GPU. They all were set up to receive their input data and return their output data from/to the host.

The results of a number of representative jobs are shown in Table 1.

Table 1. CUDA and SYCL (oneAPI) performance results

Row	Test Description	Time [s]
1	CPU-only algorithms	68.3 ± 0.47
2	3 "critical-path" CPU/GPU algorithms, run only on CPUs	68.1 ± 0.66
3	3 "critical-path" CUDA algorithms with ideal FPOPS	54.5 ± 0.47
4	3 "critical-path" CUDA algorithms with 10x FPOPS	151.2 ± 27.2
5	4 "heavy non-critical-path" CUDA algorithms with ideal FPOPS	49.5 ± 1.51
6	4 "heavy non-critical-path" CUDA algorithms with 3x FPOPS	70.3 ± 10.0
7	3 "critical-path" SYCL algorithms with ideal FPOPS	67.14 ± 0.06
8	4 "heavy non-critical-path" SYCL algorithms with ideal FPOPS	73.38 ± 12.13

A couple of conclusions can be drawn from the numbers:

- We have to be careful with offloading algorithms that many other algorithms depend on. Making these inefficient can have big consequences for the job (Row 4 of Table 1);
- Algorithms off of the "critical path" can handle being less efficient on a GPU than on a CPU, but not by much (Row 6 of Table 1);
- An integrated Intel GPU can not currently compete with a modern dedicated NVidia GPU (Rows 7-8 vs. 3 and 5 of Table 1);
- The TBB based scheduling used in the tests was far from perfect. With inefficient GPU based calculations the CPU would often not find enough calculations to run while waiting for the asynchronous (GPU) calculations to finish.

4 Summary

ATLAS, with the rest of HEP, is looking very seriously at using non-CPU computing resources for its calculation needs. Due to the programming methods used in the LHC experiments we have to actively engage the hardware manufacturers to figure out the best way of making use of these new types of hardware.

Executing offloaded calculations from TBB in an asynchronous way is providing encouraging results in our tests so far. But the results also tell us that all offloaded code will have to be written very carefully to be able to achieve the best possible performance.

Effort is currently ongoing to migrate a select list of ATLAS reconstruction algorithms to GPUs – chosen primarily based on coding considerations in this first round, not directly based on this study –, which shall allow ATLAS to run much more elaborate tests with accelerated calculations in its offline software framework in the non too distant future.

References

[1] The ATLAS Collaboration, G. Aad, E. Abat, J. Abdallah, A.A. Abdelalim, A. Abdesselam, O. Abdinov, B.A. Abi, M. Abolins, H. Abramowicz et al., Journal of Instrumentation **3**, S08003 (2008)

[2] J. Nickolls, I. Buck, M. Garland, K. Skadron, ACM Queue **6**, 44 (2008)

[3] P. Conde Muíño, The ATLAS Collaboration, Journal of Physics: Conference Series **898**, 032003 (2017)

[4] T.S. Pettersson, P. Lefèvre (LHC Study Group), Tech. Rep. CERN-AC-95-05-LHC (1995), http://cds.cern.ch/record/291782

[5] G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, G. Corti, M. Frank, G. Gracia, J. Harvey, E. Herwijnen et al., Computer Physics Communications **140**, 45 (2001)

[6] https://www.threadingbuildingblocks.org

[7] The ATLAS Collaboration, *Athena* (2019), https://doi.org/10.5281/zenodo.2641997

[8] https://www.olcf.ornl.gov/frontier

[9] https://www.anl.gov/aurora-announcement

[10] https://www.olcf.ornl.gov/summit

[11] J.E. Stone, D. Gohara, G. Shi, Computing in Science & Engineering **12**, 66 (2010)

[12] https://www.khronos.org/sycl

[13] https://www.oneapi.com

[14] https://gitlab.cern.ch/akraszna/asyncgaudi

[15] A. Buckley, T. Eifert, M. Elsing, D. Gillberg, K. Koeneke, A. Krasznahorkay, E. Moyse, M. Nowak, S. Snyder, P. van Gemmeren, Journal of Physics: Conference Series **664**, 072045 (2015)