EPJ Web of Conferences **245**, 09015 (2020) https://doi.org/10.1051/epjconf/202024509015
CHEP 2019

Geant Exascale Pilot Project

Philippe Canal[1,*], *Elizabeth* Sexton-Kennedy[1], *Jonathan* Madsen[2], *Soon Yung* Jun[1], *Guilherme* Lima[1], *Paolo* Calafiura[2], *Yunsong* Wang[2], *Seth* Johnson[3]

[1]Fermi National Laboratory, Batavia, IL, USA
[2]Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[3]Oak Ridge National Laboratory, Oak Ridge, TN, USA

Abstract. The upcoming generation of exascale HPC machines will all have most of their computing power provided by GPGPU accelerators. In order to be able to take advantage of this class of machines for HEP Monte Carlo simulations, we started to develop a Geant pilot application as a collaboration between HEP and the Exascale Computing Project. We will use this pilot to study and characterize how the machines' architecture affects performance. The pilot will encapsulate the minimum set of physics and software framework processes necessary to describe a representative HEP simulation problem. The pilot will then be used to exercise communication, computation, and data access patterns. The project's main objective is to identify re-engineering opportunities that will increase event throughput by improving single node performance and being able to make efficient use of the next generation of accelerators available in Exascale facilities.

1 Introduction

The upcoming wave of High-Performance Computing supercomputers to be deployed by the Department of Energy (DOE) labs will both bring exciting opportunities and significant challenges. Reaching more than a quintillion (10_{18}) floating point operations per second, this new generation of supercomputing will open up the path to a new scale of application ranging from precision medicine to regional climate, water use to materials science, nuclear physics to national security. Exascale computing has the potential to drive discoveries across the spectrum of scientific fields. Taking advantage of this performance will however require effort to adapt the existing code and algorithms to the highly heterogenous architecture being deployed. In particular it is expected that the ratio of graphical processing unit (GPU) coprocessors to legacy CPU to be in the order of 4 to 1.

To ensure an enduring exascale computing capability for the nation, the DOE launched in 2016 the Exascale Computing Project (ECP) [1] to bring together research, development, and deployment activities as part of a capable exascale computing ecosystem. Related to this effort, DOE Office of Science's High Energy Physics (HEP) program is also investing towards the use of exascale computing (and thus GPUs) by steering its experimental

* Corresponding author: pcanal@fnal.gov

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

programs to use in the upcoming years mostly exascale machines and running software that make efficient use of all the resources provided on those machines.

As a step in this direction, this Geant Exascale Pilot project, investigating general purpose detector simulation on GPU, has been started as a collaboration between HEP and ECP, including members from Fermilab, Lawrence Berkeley Lab, Oak Ridge National Lab with interest from researchers associated with US-ATLAS and US-CMS.

One of the main goals of the Geant Exascale Pilot Project is to perform an initial evaluation of the Geant4 [2] toolkit for high-energy physics simulation and determine the scope and feasibility of porting a significant enough subset of its functionality to GPU. This work is motivated by the increasingly large computational requirements of the HEP accelerator and detector projects, in particular the High Luminosity Large Hadron Collider (HL-LHC) upgrade, which will increase the rate of events in the ATLAS and CMS detector by a factor of ten [3].

Geant4 is a toolkit for the simulation of the passage of particles through matter. Its areas of application include high energy, nuclear and accelerator physics, as well as studies in medical and space science. Since Geant4 is an extremely general particle transport toolkit, its application scope is immense. For the pilot application, we are targeting a specific subset of Geant4's application space: modelling high energy event simulation in the CMS and ATLAS detectors. The workflow for LHC analysis begins an event generator. Each event usually consists of a set of multiple source high energy particles emitted from a primary set of collisions. Roughly, each event generates about 5 MB of data that is later postprocessed into possible detector responses. Despite this large amount of output, the simulation application is not I/O bound. Some profiling of Geant4, indicated that a typical Geant4 simulation run, represented by the Geant4 HepExpMT benchmark using a CMS geometry, is tightly memory bound, using a small fraction of the available FLOPS (for example 0.12% of peak performance on the Haswell nodes on the Cori supercomputer [4])

GeantV [5] original idea was to explore whether SIMD vector operations would make a significant difference in performance for the execution of the type of simulations implemented by Geant4. The project found that solitary physics kernels exhibited speedups on the order of four to eight (in line with the underlying available instructions), and overall performance on some example applications improved by a factor of two.

However, it appeared in practice that most of the speedup is due to improved data and instruction locality rather than vectorized math operations. Newer processors with larger L3 caches exhibited improved times with Geant4, so that the overall GeantV speedup moved closer to 1.5 times.

On this other hand, several groups have ported successfully some subset of the Geant4 physics processes used in very specific applications. Those use cases have the particularity that one or a few physics processes are taking a considerable fraction of the execution time of a typical simulation and that have inherently fewer conditional branches (this is unlike HEP simulations where there is no such concentration of run-time and where branching is frequent). Compared to those efforts we attempt to tackle the more generic problem. Unfortunately, those packages are often closed source and we cannot take full advantage of their experiences.

For example, the following teams have reported very good speed-ups. MPEXS-DNA [6] is a simulator for track structures and radiation chemistry at subcellular scale. It includes EM Physics and diffusion and chemical reactions for molecular species for micro dosimetry simulation. They report a speed-up of 2900x (using NVidia Volta). GATE [7] uses GPU to simulated Preclinical and Clinical Scans in Emission Tomography, Transmission Tomography and Radiation Therapy. They use GPU for voxelized phantom (very time consuming) and report a speed-up of 60x. For PET and CT applications (validated in Bert et al. 2013) they report a speed-up of 70x when using GPUs. G4CU [8] is Geant4 Based

Simulation of Radiation Dosimetry in CUDA that focuses on ion dose calculations for radiation therapy. They run EM Physics in H2O in voxel geometry on the GPU and report a 75~97 speedup. Opticks [9] is a GPU Optical photon simulation for Particle Physics with NVDIA's OptiX that reports a 200x speedup with mobile GPU (GeForce GT 750M).

2 Pilot application

2.1 Goals and strategies

The project will study and characterize the possible architecture and approaches to target an efficient use of HPC exascale machine and their GPU coprocessors when executing Monte Carlo simulations of HEP detectors. We will encapsulate, at first, a minimum set of physics and software framework processes necessary to describe a representative HEP simulation problem. This will then be used to exercise and research improvement of communication, computation and data access patterns. The main outcome should be the identification of re-engineering opportunities that will increase event throughput by improving single node performance and being able to make efficient use of the next generation of accelerators available in exascale facilities.

As we investigate how to best use GPUs for full HEP simulation, an important focus will be to reach an estimate of the higher limit of speed-ups that could be reached by leveraging the GPUs. From past experiences [5][10], it is clear that we will have to explore several options and design for memory access, computation ordering, and CPU/GPU communication patterns. This will require flexibility in the data structure and code implementation. Another important step is to avoid over-simplification and look at the whole Geant simulation chain not just a few components. In past attempts we have seen very encouraging early results for individual components, being fatally diluted when scaled to the larger, more complex use cases.

The pilot application should demonstrate the components and architecture necessary for a generic Monte Carlo transport simulation framework that executes its workload on either the CPU or GPU or combination of both. For example, we might use the GPUs for parts of the code where the speed-up has been demonstrated and run on the CPU the parts where speed-up is highly unlikely (e.g. physics with significant branching; particles with short-lifetimes and/or a relatively small average number of interactions).

As one of the goals is to understand the upper limit, when considering trade-offs, we shall lean on the side of performance. For example, we are going to push beyond the scope of this project any consideration of backward compatibility with existing framework or code base or any upstreaming of our exploratory work.

In order to bootstrap the project, we plan on reusing or leveraging as many components as practical, this includes newer components like VecGeom [11], the latest magnetic field and physics implementation for Geant4 or GeantV. At first, we will only support NVidia hardware, later in the project we will need to understand AMD and Intel programming direction; we are likely to switch to a library like Kokkos [12] or SYCL [13]

2.2 Requirements

After some iterations our team converged on a list of requirements for the pilot application. The goal was to target an HEP physics application as simple as possible that could simultaneously emulate the complexity of a real HEP simulation. The complexity in Geant4 is essentially a result of its infinite extensibility, but three key contributors are the complex

geometry definition, the extensive list of physics models and particles they act upon, and the open-endedness of user-specified detectors.

Profiling runs from an example that produced medium-energy (1--10 MeV) leptons and hadrons showed that the electromagnetic particles dominated the run time. The initial set of targeted particles and processes are listed in **Table 1.** The requirements also included propagating charged particles in a uniform electromagnetic field and was to be expanded to more realistic electromagnetic field description.

Table 1. Initial proposed physics models.

Particles	Physics interaction processes
e-, e+	Ionization, bremsstrahlung, multiple scattering
γ	Photoelectric effect, Compton scattering, pair production

Instead of being hardcoded to a single simulation, the pilot application should be configurable for different:

1. Physics lists, for the selection of allowable particles and physics models
2. Detector (geometry) descriptions, probably using GDML input description
3. Input event types, i.e. Primary particle descriptions.

The output of the pilot application would be representative of data from a typical HEP experiment, e.g. detector responses for each event integrated over all tracks. Finally, since the pilot application's purpose is to explore and measure performance, including scalability, event throughput, and memory usage, profiling instrumentation must be included to the application.

2.3 Approaches

Two approaches to developing a pilot application were debated among the team. The top-down approach was to develop directly in a fork of either the Geant4 or GeantV code bases so that the end product could be merged upstream. This approach was in reaction to past "mini applications" developed for Geant that were far too simple to be representative.

The bottom-up approach was to start with a blank slate and build up with a minimum of components. This would make the pilot application simpler and more understandable, and it would force all design decisions to be considered as the code grew, rather than defaulting to those of previous generations of Geant.

In the end, a compromise was reached where we began with a subset of auxiliary functionality from Geant4 and agreed to use the VecGeom and VecMath libraries for geometry, math functions and random numbers, respectively. Both of which support simultaneous CUDA and host C++ functionality.

2.4 Dependencies

The initial pilot application is configured with several third-party libraries; they can be installed externally or downloaded and configured automatically:

- GoogleTest: Test harness framework. [14]
- PTL: Multithreading tasking system. [15]
- TiMemory: CUDA-enabled profiling utility. [16]
- VecCore: GeantV-related CUDA-enabled utility code [17]
- VecMath: GeantV-related CUDA/SIMD-enabled math functions [18]
- VecGeom: GeantV-related CUDA-enabled Geant geometry [11][19]

The application requires CMake 3.8 or higher to support CUDA as a CMake native/first-class language, and we decided on a minimum requirement of C++14, the newest version (at the time) supported by the compilers deployed on HPC clusters.

2.5 Current Investigations

Current detector simulation generic toolkit (e.g. Geant4) relies on polymorphism to support a wide range of particles and physics models and to allow customization of the simulation; this includes for example custom physics lists (of processes and models) and experiment specific integrator of the equation of motion in a magnetic field.

In C++ polymorphic (virtual) functions are implemented via the inclusion of a virtual function table pointer inside the memory allocated for the object (often but not always at the start of the object's memory). This virtual function table is itself an array of member function pointers that contains the information about which instances of a given virtual function to call for the specific type of the object. Uploading of data from a CPU to GPU is done, for example in CUDA, via a simple copy of the memory. This means that once it is in the GPU memory, the object still contains the address of this table, but this address is reference to a memory location in the CPU. Consequently, without further post-processing to update this pointer, any device code that attempts to call one of the virtual functions of the object, will try to deference this pointer and fail; and it will definitively not have the intended effect of causing the execution of the device code version of the virtual function.

To be able to transfer the objects back and forth without having to postprocess them, we are investigating techniques involving static polymorphism. In those techniques, one moves the decision of which type specific function to execute from run-time to compile-time. This can be accomplished in C++ by leveraging its class and function template paradigms. However, if we solely rely on those technique, one can end up in a situation where all the code is templated resulting in both highly complex code and code that takes significant resources to compile and store (because of assembly code duplication inherent to template techniques).

To avoid this explosion, we are exploring "Partial Static Polymorphism" where some of the code uses templates and part of the code still rely on virtual function. One of the elements is an hybrid generic and type-specific container, a C++ collection that can hold an heterogenous set of objects but offers interfaces (template based) that allow the execution of operation on an homogenous subset of the objects [20]. Another element is a virtual eraser face function which given a generic pointer and knowing the actual derived type will call the appropriate function without using the virtual function table [20] One of the ways we might use this technique is to have the hybrid container holding tracks and retaining the information of which type of particle the track corresponds to. We can then have some device kernel that request only the track corresponding to, for example, electrons and when requesting information about the particle it will pass the generically typed pointer to the particle from the track object and use its functionality via the "virtuality eraser".

Another aspect of customization is the selection of the set of physics lists. Typically, this list is only known at run-time and its use require loops and calls to virtual functions that the

compiler has no chance to optimize. We are investigating replacing this list by a compile-time configuration mechanism base on C++ template and trait patterns. In this scenario the user would exchange the flexibility inherent to run-time configuration with the potential optimization that the compiler could do upon seeing most of the operations needed to process the particles through the requested physics list.

A major part of this project is going to be the investigation of the data memory layout, in particular for track information, that can lead to the best run-time performance. In order to allow for significant refactoring of the organization of the track data in memory, we are planning on splitting completely the (track) data from the interfaces used to access to it. The intent is to completely shield the code that is using the track data from the changes we might make to the tracks' memory layout. This change might include using either an array of structs or a struct of arrays. This implementation might also allow us to have different data layout on the CPU and GPU, where for example the track may be containing only the information actually used by the device kernels.

Those interfaces would be split in two sets, one for accessors and one for modifiers. Each set would itself be split in term of functionality, for example geometry related data, physics related data, etc. This separation will allow to factor out more of the track related functionality without bloating a single class. Thanks to inlining and compilers optimization, we expect those accessors and modifiers functions to be completely cost-free at run-time. Having a clear separation of the different parts of the track interfaces should also allow for a better understanding of the code structure and which part of the track data each part of the code actually uses. This clearer understanding will be one of the major inputs in our decision of how to tweak the memory layout of the track data.

3 Conclusions

We started a new project to explore the use of GPUs for HEP Geant Simulations, targeting the upcoming Exascale class of machines. This is a collaboration between the Exascale Computing Project and the DOE Office of Science's High Energy Physics program involving Fermi National Laboratory, Lawrence Berkeley National Laboratory and Oak Ridge National Laboratory. This combination provides joint expertise in HPC, GPU and Geant4 and GeantV. We are currently setting up the infrastructure classes and simulation functionality. In time for the HL-LHC, we will deliver our ideas and solutions into a production quality simulation tool kit.

Acknowledgement

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations---the Office of Science and the National Nuclear Security Administration---responsible for the planning and preparation of a capable exascale ecosystem---including software, applications, hardware, advanced system engineering, and early testbed platforms---to support the nation's exascale computing imperative.
This research was supported by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

References

1. P. Messina, "The Exascale Computing Project," in Computing in Science & Engineering, vol. **19**, no. 3, pp. 63-67, (2017)
2. S. Agostinelli et al., "GEANT4: A Simulation toolkit," Nucl. Instr. Meth., vol. **A506**, pp. 250-303, (2003)
3. G. Apollinari et al., "High-Luminosity Large Hadron Collider (HL-LHC) : Technical Design Report V. 0.1", CERN, vol **4** p 590 (2017)
4. Y. Wang, "Geant4MT profiling on the Cori system", https://indico.cern.ch/event/707173/contributions/2920706/attachments/1612381/2561530/profiling_AtlasSC_mar_2018.pdf, retrieved March 2020.
5. G. Amadio et al., "GeantV alpha release," J. of Phys. Conf. Ser., vol. **1085**, p. 032037 (2018)
6. S. Okada et al. "MPEXS-DNA: a new GPU-based Monte Carlo simulator for track structures and radiation chemistry at subcellular scale," Med Phys. vol **46** no. 3, pp. 1483-1500 (2019)
7. J. Bert et al., "Hybrid GATE: A GPU/CPU implementation for imaging and therapy applications," 2012 IEEE Nuclear Science Symposium and Medical Imaging Conference Record (NSS/MIC), Anaheim, CA, pp. 2247-2250 (2012)
8. G4CU: https://kds.kek.jp/indico/event/15926/session/30/contribution/110/material/slides/0.pdf
9. S. Blyth, "Opticks : GPU Optical Photon Simulation for Particle Physics using NVIDIA® OptiXTM," EPJ Web Conf., **214,** pp. 02027 (2019) 02027
10. S. Hamilton, T. Evans, "Continuous-energy Monte Carlo Neutron Transport on Gpus In the Shift Code", Annals of Nuclear Energy, vol **128** no. 1, pp 236-247 (2019)
11. J. Apostolakis et al., "A vectorization approach for multifaceted solids in VecGeom," inEur. Phys. J. Web of Conf., vol. **214**, p. 02025, (2019)
12. C. Edwards, "Kokkos," J. of Para. and Dist. Comp. vol. **74**, no 12 (2014)
13. SYCL language, https://www.khronos.org/sycl, retrieved March 2020.
14. Google Testing and Mocking Framework, https://github.com/google/googletest, retrieved March 2020.
15. Parallel Tasking Library, https://github.com/jrmadsen/PTL, retrieved March 2020.
16. TIMemory, https://github.com/NERSC/timemory, retrieved March 2020
17. VecCore, https://github.com/root-project/veccore, retrieved March 2020
18. VecMath, https://github.com/root-project/vecmath, retrieved March 2020
19. VecGeom, https://gitlab.cern.ch/VecGeom/VecGeom, retrieved March 2020
20. https://github.com/Geant-RnD/GeantExascalePilot/blob/master/source/Geant/proxy/test/BasicCpuTransport/TrackManager.hpp