Journal of Physics: Conference Series **513** (2014) 052013 doi:10.1088/1742-6596/513/5/052013

High energy electromagnetic particle transportation on the GPU

P Canal[1], D Elvira[1], S Y Jun[1], J Kowalkowski[1], M Paterno[1] and J Apostolakis[2]

[1]Fermilab, MS234, P.O. Box 500, Batavia, IL, 60510, USA
[2]CERN, PH Department, J27210, CH-1211 Geneva, Switzerland

E-mail: syjun@fnal.gov

Abstract. We present massively parallel high energy electromagnetic particle transportation through a finely segmented detector on a Graphics Processing Unit (GPU). Simulating events of energetic particle decay in a general-purpose high energy physics (HEP) detector requires intensive computing resources, due to the complexity of the geometry as well as physics processes applied to particles copiously produced by primary collisions and secondary interactions. The recent advent of hardware architectures of many-core or accelerated processors provides the variety of concurrent programming models applicable not only for the high performance parallel computing, but also for the conventional computing intensive application such as the HEP detector simulation. The components of our prototype are a transportation process under a non-uniform magnetic field, geometry navigation with a set of solid shapes and materials, electromagnetic physics processes for electrons and photons, and an interface to a framework that dispatches bundles of tracks in a highly vectorized manner optimizing for spatial locality and throughput. Core algorithms and methods are excerpted from the Geant4 toolkit, and are modified and optimized for the GPU application. Program kernels written in C/C++ are designed to be compatible with CUDA and OpenCL and with the aim to be generic enough for easy porting to future programming models and hardware architectures. To improve throughput by overlapping data transfers with kernel execution, multiple CUDA streams are used. Issues with floating point accuracy, random numbers generation, data structure, kernel divergences and register spills are also considered. Performance evaluation for the relative speedup compared to the corresponding sequential execution on CPU is presented as well.

1. Introduction

In the last few years, the hardware landscape has significantly changed. After relying for almost 30 years on the ever increasing performance of a single CPU core, performance improvements must leverage the increase in number of cores per die. In addition the code developed and used within the HEP has not yet taken advantage of the increased length of the processing units' vector pipelines [1]. With the advent of the many-core platforms and General Purpose Graphics Processing Units (GPGPU), a complete re-design of the software architecture appears necessary to take advantage of those many cores and in parallel to increase the code's efficiency on the existing platforms. To that end, we decided to investigate the implementation of some of the tasks needed as part of a high energy physics detector simulation on a GPGPU.

2. GPU Prototype

The current prototype is conceived as an independent sub-component of HEP detector simulation designed for GPU and coprocessor accelerators. It consists of the particle transportation process in electromagnetic fields, the electromagnetic (EM) physics package, and the geometry package. These three elements were implemented using the parallel programming and computing platform CUDA from NVIDIA. The goal is to develop and study different strategies and algorithms that will enable detector simulation to make efficient use of a large number of SIMD/SIMT computational threads. In particular we developed a prototype to be deployed in hybrid systems with both shared and distributed memory components such as CPUs and GPUs. In this section, the primary components of the GPU prototype are briefly described.

2.1. Particle Transportation

The particle transportation process is an essential component of Geant4 [2][3][4] and is a significant part of the total aggregate runtime for a typical HEP detector simulation. A good choice for a method of numerical integration involved in charged particle transportation in a magnetic field keeps the Streaming Multiprocessor (SM) occupancy high while hiding the memory latency to reduce the ratio of the arithmetic instructions to off-chip memory operands. The default numerical algorithm in the current Geant4 transportation engine is the fourth-order Runge-Kutta algorithm [5] in conjunction with an adaptive step size control. The two other algorithms tested on the GPU prototype are the Cash-Karp [6] and the Runge-Kutta-Nystrom [7] integrators. In regards to the relative performance of the code in a GPU versus a CPU, the Runge-Kutta-Nystrom algorithm gives the largest speed-up, due to a low arithmetic calculations intensity, while the Cash-Karp and default Runge-Kutta integrators showed a comparable speed-up performance. A summary of performance of numerical algorithms is shown in table 1.

Table 1. Execution times for the evaluation of a magnetic field and integration of the equation of motion for 100,000 tracks using Intel® Xeon E5-2620 (CPU) and NVIDIA® K20M [8] (GPU).

Algorithm	CPU [ms]	GPU [ms]	CPU/GPU
Classical Runge-Kutta	78.6	1.7	47.4
Cash-Karp	87.9	1.6	55.2
Runge-Kutta-Nystrom	30.9	0.7	46.9

2.2. Geometry Navigation on the GPU

Particle transport makes heavy use of the geometry subsystem. The current prototype supports a minimal set of geometric and material objects to assist in the particle transportation. A simple detector similar in structure to the CMS central electromagnetic calorimeter (ECAL) [9] is constructed on the host side, and then the entire memory buffer of the geometry description is transferred to the global memory on the device side with appropriate mapping of memory access pointers for mother and daughter physical volumes [10]. In the standard Geant4 tracking algorithm, each dynamic particle keeps track of the history of geometry access in a navigator object. The reference pointers used to refer to logical and physical geometry volumes allocated in the CPU and the GPU memories are currently different thus the navigator for each particle ought to be created on both the CPU and GPU. Re-initializing a navigator object for each track on the host memory and copying it to the GPU is an expensive operation due to the size of this object. We confirmed that it is the best to create one navigator per thread on the GPU and

reuse it for all tracks processed within the same thread. This solution avoids a large latency penalty due to unnecessary global memory access.

2.3. Electromagnetic Physics

Implementing EM physics on the GPU may increase the computational intensity and makes multiple stepping possible. Since most of the secondary particles produced in primary interactions are electrons (∼75%) or photons (∼20%), all processes that describe their electromagnetic (EM) interactions with matter must be implemented in order to track them through the detector. Since the structure of the Geant4 physics processes is very complex and heavily relies on virtual inheritance, we implemented a specific set of physics models for the standard EM physics processes to simplify the problem. This CUDA code includes sampling of secondary particles and the generation of simulated hits. A minimal set of classes associated with materials and atomic elements have been ported to interface with the physics and the geometry aspects of the simulation. A summary of all the EM physics processes and models that are implemented for electrons and photons is included in table 2.

Table 2. Electromagnetic physics processes and models implemented for electron and photon processes in the GPU prototype.

Primary	Process	Model	Secondaries	Survivor
e^-	Bremsstrahlung	SeltzerBerger	γ	e^-
	Ionization	MollerBhabhaModel	e^-	e^-
	Multiple Scattering	UrbanMscModel95	–	e^-
γ	Compton Scattering	KleinNishinaCompton	e^-	γ
	Photo Electric Effect	PEEffectFluoModel	e^-	–
	Gamma Conversion	BetheHeitlerModel	e^-e^+	–

3. Performance Considerations

A CUDA kernel is executed by a grid of thread blocks. Primary considerations for improving the performance of the GPU prototype are to minimize communication cost per unit of work, thread divergence and global memory accesses. In this section, we summarize different strategies to maximize instruction throughput and data locality for the GPU prototype.

3.1. Data Transfer

Data transfers between the host and the device are usually very expensive due to the limited bandwidth of the PCI bus. The necessary input data allocated onto the device memory for the particle transportation on GPU is of two different types: one-time data transferred at the start and used during the entire program execution (magnetic field map, detector geometry, physics tables, etc.) and recurrent data transfers for each kernel execution (bundle of tracks, output of secondary tracks, etc.). The latter is an important overhead and uses asynchronous transfers using multiple CUDA streams to overlap kernel execution with the data transfer. When transferring data from the host to the device and back, it is also better to use a single large transfer rather than many small transfers.

3.2. Texture memory for magnetic field evaluation

Texture memory is cached on chip and designed for memory access without spatial locality. The magnetic field map used is a 2-dimensional grid (900×3200 points in $r \times z$, equivalent to 46M

bytes) which may be suitable for using the texture memory. We compared the performance of interpolating the magnetic field either using the Texture memory or using explicit CUDA code accessing the data from the GPU main shared memory. We found that the interpolation made using the Texture memory was slightly less accurate (1% to 3% differences) than the explicit calculation.

The interpolation using the Texture memory is twice as fast as the explicit interpolation, but only for random access. The access latency of the Texture memory is much less than the access latency of the GPU main shared memory. However when the interpolation is made for locations that are close to each other, the part of the field map that is required is small enough to be kept in the level-1 cache and reused for many successive interpolations and the extra cost of the memory fetch is amortized over several interpolations. And indeed in our current small-scale tests with CMS ECAL, there was no noticeable difference for three evaluations of the fourth-order Runge-Kutta algorithm with or without using the texture memory.

3.3. Global Memory Access and Data Structure

Since a typical HEP detector simulation usually requires intensive memory transaction, efficient global memory access is a critical factor to hide latency. The implementation of the EM physics code was followed by an evaluation of the computing performance of each process that included memory transactions to store secondary particles into a stack of arrays on the GPU global memory. These studies established that using fixed size pre-allocated memory as a temporary buffer for secondary particles outperformed other options which were based on memory allocated dynamically on a per-thread or per-block basis.

The alignment of data structures is another important factor to achieve coalesced global memory access. Position and momentum of the track are frequently queried and updated throughout the transportation process and should be aligned for efficient data access. The Struct of Arrays (SoA) is more efficient than the Array of Structs (AoS) for various structures of track data on GPU as shown in table 3.

Table 3. Execution times for the evaluation of loading data (structs of 4-doubles and 8-doubles) and storing them back to the global memory (65536 accesses) using NVIDIA® K20M GPU.

Data Type	Array of Struct [ms]	Struct of Array [ms]	AoS/SoA
4-doubles	0.065	0.056	1.17
8-doubles	0.144	0.093	1.56

3.4. Floating-point Consideration

Another important aspect of the particle transportation algorithm is the choice of format for floating–point quantities in order to achieve the best possible time performance without loss of the required precision. The use of double-precision data types would allow more precision but with a cost in memory through-put and number of arithmetic instructions per unit of time. The cost would be very high if there was, in addition, register spilling that led to a sudden performance cliff. Studies of the precision of particle propagation algorithms were performed, based on two of the main algorithm parameters such as the size of the truncation errors and the "distance to the chord", defined as the distance from the curved trajectory to the middle point of the segment that connects the start to the end point. These studies confirmed the need to use double-precision types in all integration algorithms used for particle transport in GPUs. For

parameters or numeric data requiring low precision such as the local coordination of geometry, input physics tables and the magnetic field map, the float should be used whenever appropriate.

3.5. Random Number Generators

Random numbers are used heavily in Geant4, and it is therefore important to find efficient thread-safe generators to use in the prototype. Delivering uncorrelated sets of pseudo–random number sequences to concurrent threads simultaneously is critical to generate statistically independent samples. Since the simulation of HEP detectors requires high-quality random numbers, we only consider pseudo–random engines for use in the GPU prototype. We use CURAND [11], a CUDA library that provides simple and efficient algorithms to generate pseudo–random and quasi–random numbers. Preliminary results of the performance tests of the CURAND random number engines on the GPU prototype are summarized in table 4. Initialization and generation are factorized into two kernels for maximum performance since

Table 4. Performance of the CURAND random number generator used on the GPU prototype to generate 10,000 pseudo–random numbers (64 blocks and 256 threads).

CURAND library	Description	Initialization [ms]	10K PRNG [ms]
XORWOW	xor-shift family	4.09	7.55
MRG32k3a	L'Ecuyer's Multiple Recursive	6.16	22.17
MTGP32	Mersenne Twister	0.69	35.96

the initialization of the random generator states generally requires more memory than the generation. While XORWOW is efficient but statistically inferior for most of the HEP applications, the Mersenne Twister algorithm proved to be the best for generating a small set of random numbers within a thread, balancing performance and quality of the random stream.

3.6. Concurrent Streams and Multiple Kernels

Using multiple streams enhances the GPU performance for parallel tasks by executing kernel and transferring data simultaneously. Using pinned memory and asynchronous data streams, the current GPU prototype efficiently handles batching data transfers for small chunks of tracks (down to 4096) without much overhead. This strategy is critical to maintain the occupancy of resident warps when the track collection is depleted.

Increasing arithmetic intensity would make the kernel of the GPU prototype more complicated and introduce more divergent branches in kernel executions. We conducted a study in which algorithms were decomposed into multiple kernels to avoid divergence and reduce a potential performance drop due to register or local memory spills. Since the downstream transportation paths are very different for electrons (charged) and photons (neutral) in a magnetic field, the separation of kernels by particle type improved the overall performance by about 30%.

4. Performance

As a measure of the performance, we define "relative speedup" as the ratio between the time taken by the whole GPU card and by a single CPU core to execute the same task. A test geometry is based on a simplified CMS ECAL and a magnetic field map excerpted from the CMS software. Elapsed times on both CPU and GPU were measured for transporting 65536 electrons for one-step with all standard physics processes listed in table 2. Secondary particles were sampled and stored on a temporary stack in the global memory. Hardware platforms and performance results are summarized in table 5.

Table 5. The "relative speedup" as the ratio between the CPU and the GPU times for simulating the one-step process with standard EM physics processes for 65536 electrons using NVIDIA® GPUs.

Host	GPU Device	CPU [ms]	GPU [ms]	CPU/GPU
AMD Opteron^{TM} 6136	M2090 [12]	748	37.8	19.8
Intel® Xeon E5-2620	K20M [8]	571	30.4	18.7

The relative speedup depends significantly on the thread organization, the arithmetic intensity and the number of memory transactions involved in the kernel operation. The thread-block combination used for M2090 are 32 blocks and for K20M are 128 threads and 26 blocks and 192 threads, respectively.

5. Validation

As the GPU prototype evolves and expands, it becomes critical to understand the computing performance of each component and verify the accuracy and precision of the results. Since the current code is designed to be compatible with both the CPU host and the GPU device, all the prototype's kernels, except random number generation, can be executed in the exact same way on both GPU and CPU, allowing us to verify the results produced on both. To test the correctness of the algorithms' implementation, we extended the validation to execute also the same operations using the original Geant4 library. We call this software the "back-ported code". The back-ported code also provides a way to compare the performance of the CUDA host code with that of the original Geant4 implementation, and gives additional insight on how to optimize the device code. Simulated quantities such as the final position of the primary track, the amount of energy loss, the number of secondary particles and their kinematical distributions have been compared and validated within statistical uncertainties.

6. Connection to a scheduler

The current implementation goals for running detector simulation code on a GPU are intentionally limited to the propagation of only electrons and gamma particles within a limited set of geometrical solids. In order to execute a complete simulation we need to insert this code within a framework that can handle the other task, including propagating the other types of particles, handling other types of solids and generating the initial steps. To that end, we elected to collaborate with the detector simulation Vector prototype being developed at CERN [15]. The Vector prototype natively organizes the track to be propagated in baskets associated with each logical volume of the detector. We implemented a connector acting as one of the tasks of the Vector prototype which can consume baskets and schedule work on the GPU.

The connection of the two implementations presents several challenges. The Vector prototype was implemented using the geometry classes implementation provided by ROOT [13], whereas our GPU prototype implementation started with a port of the geometry classes provided by Geant4. The two geometry implementations can represent the same detectors but the organization of the logical volume is different. In particular the description of the current location within the logical volumes, and by extension the history of the location of the tracks, is expressed differently. To connect the two prototypes we needed to implement a translation between the two forms, however this has so far proven to be difficult for the general case. The first implementation supports only the simple geometry description used in our initial tests. A full fledged implementation will be needed for the general case and the code in the ROOT Geant4

virtual monte-carlo [14] is being considered as source of implementation for this translation.

Another challenge is the difference in physical layout of the objects passed to and from the CPU memory and the GPU memory thus requiring a transformation before the copy can be initiated. Currently the usual size of the baskets for the Vector prototype and the GPU prototype is significantly different, from few elements for the CPU prototype to a few thousands elements for the GPU prototype. This requires the implementation of a staging mechanism in the connector to accumulate enough work for the GPU out of many baskets. Finally, maximizing the efficiency of the GPU code requires minimization of the thread divergence within a kernel. Minimizing the divergence requires execution of the kernel on a set of tracks that are as similar as possible including same particle types, same detector element and same energy range. This introduce an additional need for the staging of the track to sort them in the appropriate set of staging buckets.

7. Conclusion

We have developed a prototype running on a general purpose graphics processing unit that implements the simulation of the propagation of photons and electrons within a high energy physics detector. This prototype has also been embedded into a vectorized framework for detector simulation. This connection showed the need for greater synergy between the two implementations to reduce the need or the complexity of the required scatter-gather steps.

Our CUDA implementation is now complete enough to provide a very good test bed for various optimization strategies. The current speedup is in the order of 20 for the full code. Some smaller benchmarks have reached even higher speedup (in the order of 100) leading us to believe that there is still much room for improvement.

Our short term strategy includes proceeding with in depth analysis of the most promising optimization opportunities and a significant increase of our collaboration with the Vector prototype in particular to reduce impedance mismatch between the two components and to increase the overall functionality. Our ultimate goal is to have a full fledged simulation that efficiently uses the main CPU and also leverages existing coprocessors whenever they are present as well as gathering a set of techniques and patterns that could be reused in other HEP software areas.

References

[1] Sverre J 2012 Many-core experience with HEP software at CERN openlab *J. Phys.: Conf. Series* **396** 042043
[2] Allison J *et al* 2006 Geant4 Developments and Applications *IEEE Transactions on Nuclear Science* **53** No. 1 pp 270-278
[3] Agostinelli S *et al* 2003 Geant4 - A Simulation Toolkit *Nuclear Instruments and Methods in Physics Research* A **506** pp 250-303
[4] Ahn S *et al* 2013 Geant4-MT: bringing multi-threading into Geant4 production *Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo*
[5] Press W H *et al* 1988 *Numerical Recipes in C* (Cambridge University Press) pp 569-588
[6] Cash J R and Karp A H 1990 *ACM Transactions on Mathematical Software* vol 16 pp 201-222
[7] Cash J R 2005 *SIAM J. Scientific Computing* **26** pp 963-978
[8] NVIDIA Kepler Architecture K20M http://www.nvidia.com/object/nvidia-kepler.html
[9] Chatrchyan S *et al* 2008 The CMS experiment at the CERN LHC *Journal of Instrumentation* **3** S08004
[10] Seiskari O *et al* 2012 GPU in Physics Computation: Case Geant4 Navigation *arXiv:1209.5235*
[11] NVIDIA CUDA Toolkit 5.0 CURAND Guide http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf
[12] NVIDIA Fermi Architecture M2090 http://www.nvidia.com/object/fermi-architecture.html
[13] CERN 1997 ROOT - An Object Oriented Data Analysis Framework (http://root.cern.ch) *Nuclear Instruments and Methods in Physics Research* A **389** pp 81-86
[14] CERN ROOT - Geant4 - Virtual Monte Carlo http://root.cern.ch/drupal/content/vmc
[15] Gheata A 2013 Vectorizing the detector geometry to optimize particle transport *CHEP2013*