Fluorescence detector simulation on GPUs

TAREQ ABUZAYYAD[1] FOR THE TELESCOPE ARRAY COLLABORATION
[1]*University of Utah, Salt Lake City, UT, 84112*
tareq@cosmic.utah.edu

Abstract: In recent years, the graphics processor unit (GPU) has been recognized and widely used as an accelerator for many scientific calculations. In general, problems amenable to parallelization are ones that benefit most from the use of GPUs. The Monte Carlo simulation of fluorescence detector response to air showers presents many opportunities for parallelization. In this paper I will present a fluorescence detector simulation program which runs on the GPU. All of the physics simulation from shower development, light production and atmospheric attenuation, as well as, the realistic detector optics and electronics simulations are done on the GPU. I will describe the implementation of the code and present results on the performance of the simulation. Improvements in computational throughput in excess in excess of $50\times$ are reported and the accuracy of the results is on par with the CPU implementation of the simulation.

Keywords: UHECR, fluorescence, detector simulation, GPU, CUDA.

1 Introduction

Fluorescence detectors (FD) are ultra high energy cosmic rays (UHECR) detectors. Cosmic rays interact near the top of the atmosphere and generate an extensive air shower (EAS). Charged particles in the EAS ionize the Nitrogen molecules in the air and cause them to emit flourescence light. FD's are built using light collectors (mirrors), photomultiplier tubes (PMTs) for camera pixel elements, and high speed electronics that allows them to form an image of the EAS development through the atmosphere. Cerenkov light is also produced by the shower particles, and contributes to the detected signal. The detector response to an observed shower is refered to as an ”event”.

Fluorescence detectors (FD) rely on Monte Carlo (MC) simulations to calculate the detector aperture, and check the validity and accuracy of event selection and reconstruction procedures. In addition the event reconstruction itself typically employs an inverse Monte Carlo fitting method. The detector MC simulates (a) the shower development in the atmosphere, (b) the flourescence and Cerenkov light production, (c) the propagation of the light from the shower to the light collector (taking into account atmospheric losses and the collection efficiency of the detector), and (d) the response of the detector electronics to the PMT's signals.

Calculating the detector response for a single shower is reasonably fast, on the order of a few seconds. However there is a need to generate a large number of showers in order to understand the detector and the data. The event reconstruction procedure using the inverce MC method is more time consuming than event simulation since the response to many trial showers has to be evaluated. It could take up to a few weeks to generate and reconstruct enough MC to produce a physics result. Therefore accelerating the data processing is a highly desirable and useful development.

General Programing on Graphics Processor Units (GPGPU) has been a topic of research for a decade now but has really gained momentum in the scientific computing field over the past four years, after the introduction of CUDA [1] (C for Unified Device Architecture) by Nvidia. The creation of a “high” level interface to the GPU hardware made programming the GPU accessible to a large number of non-experts (author included). The main drawback to using CUDA is that not all computers are equipped with GPUs from Nvidia. For this reason, the program described in this paper was written in such a way that it would run with or without GPU acceleration.

GPGPU is most useful when dealing with computational problems that can easily by parallelized. The simulation of the FD response to one shower involves performing a large number of independent calculations that can be executed in parallel, making it ideal for GPU acceleration. In addition, each event is simulated independently of other events (trivial parallelism), again ideal for GPUs. The simultanuous calculation of the detector response to multiple events improves the efficiency and utilization of the GPU resources.

Section 2 below gives an overview of the MC program flow, along with the steps required to simulate one event. Section 3 describes the basic concepts of the CUDA programming model as it relates to the problem at hand. Section 4 discusses introduces the MC program design considerations and implementation details. And in the final section we present results of some sample simulations and discuss performance.

2 FD Simulation Procedure

The FD MC program is invoked to simulate a *set* of shower events. The first step of the simulation is to initialize the detector configuration and set the relevant physics models based on user input. Once initialization is done, the requested set of events is generated either serially (CPU) or in parallel (GPU). Finally, the generated parameters for all simulated showers, along with the detector response for those showers which trigger the detector, are written to an output file.

The following bullets outline the flow of the MC program:

- Parse user input; set detector configuration, atmospheric parameters and physics models.
  - If using GPU: copy initialization data to GPU; Initialize RNG on GPU
- Start data set simulation: Either randomly generate shower profile/track geometry for all showers in the set, or read in the shower profile/track geometry from an input file. The latter option is used in hybrid simulations were the same set of showers already used in a surface detector simulation is passed to the FD simulation.
- Reduce the data set by checking on whether the generated shower track is in the field of view (FOV) of any FD telescope. If not, the event is discarded as no trigger can be expected. In addition, if this is a hybrid event, then check the event time stamp to see if it occurs at a time when the FD is running (recall FD's have a 10% duty cycle)
- If running on the CPU then process each shower event sequentially, otherwise copy the reduced set to the GPU memory and run the shower simulation in parallel.
- If using the GPU then copy the detector response back to main memory
- Write out simulation results.

In the FD MC, an extenisve air shower is defined by eight parameters: Four parameters define the shower track. Another four parameters define the shower energy and development profile, following the Gaisser-Hillas (GH) parameterization. These eight variables form the input to an individual shower simulation. The simulation proceeds as follows:

- Subdivide the shower-track into smaller track segments: The shower track is a line segment which extends from the CR first interaction point to the ground along the shower path. This track is divided into 832 track segments of equal atmospheric depth steps, 1-2 g/cm^2 each depending on shower zenith angle.
- Generate Shower Profile: Given the GH profile paramaters, evaluate the shower size (number of shower electrons) and related variables at the center of each segment.
- Calculate fluorescence and Cerenkov light production: This involves calculating the shower energy deposit, and the flourescense yield. Note also, that Cerenkov light refers both to local production at each track segment and the accumulated Cerenkov beam as the shower develops. The latter is calculated in a separate step in order to make best use of the GPU.
- Light propagation and collection: With a track segment acting as a point source calculate the wavelength dependent atmospheric attenuation along the path to the observing telescope. Also account for gemetrical, and wavelength dependent, collection efficiency of the detector elements.
- Prepare electronics simulation: Evaluate the time duration for which the electronics simulation needs to be done and add the sky noise background photons contribution to the PMT signals in this time interval.
- Perform ray tracing: Ray trace photons from the track segment through the telescope optics and into a determine if a PMT is hit.
- Electronics simulation: The PMT signal is filtered and amplified. Also, check for a "tube trigger" which occurs when the signal voltage crosses a certain threshold.
- Trigger logic: Check the numbers and pattern of triggered tubes to see if the conditions for a "mirror trigger" are satisfied. If one or more mirrors trigger that defines a detector trigger.

3 CUDA Programming model

The reader is refered to [2] for a full introduction to CUDA. Here, I will introduce a few key conepts that are relavent to this paper.

A computer program written with CUDA executes on the CPU, *the host*, and treats the GPU, *the device*, as a coprocessor. The program flow is controlled by code that runs on the CPU. The GPU is invoked from the host by calls to special functions, *kernels*, identified in the source code by a keyword `__global__`. Kernels can call *device* functions, identified by the keyword `__device__`. These keywords (function qualifiers) are introduced as extensions to the C programing language and are interpreted by the CUDA compiler (nvcc). A function can be defined to be executed on the CPU *and* on the GPU by giving it a name qualifier `__host__ __device__`.

The CUDA parallel execution model is refered to as single-instruction multiple-thread (SIMT). A group of 32 threads, collectively referred to as a *warp*, execute a single instruction, each thread on its own private data. Best performance is obtained when all threads in a warp execute the same instruction. CUDA defines a thread hierarchy as follows:

- Threads are organized in *thread blocks*. The thread identifier is a 3 component vector allowing for 1D, 2D, or 3D blocks with upto 1024 threads.
- A collection of blocks, up to 65k, make up a *grid*

The user specifies the total number of thread-blocks to run, and the size and dimensionality of each block. A kernel launch is requested for a grid of thread blocks. The management of all the threads: scheduling, resource allocation, etc. is handled by the hardware.

4 Program design considerations

One of the main design considerations was that the program would run on any computer, with or without a GPU. The build system, based on CMake, compiles the program to run on the CPU by default. However, it can be invoked with an option `WITH_CUDA`=ON in which case the program is built with GPU acceleration. Where appropriate a preprocessor variable is defined to select the CPU (host) path or the GPU (device) path.

To simplify code maintinance in the long term and to minimize the chance for the CPU and GPU code to be modified separately and inconsistently, most calculations are performed inside functions defined to run on both the CPU and the GPU. This is accomplished by using the CUDA constructs `__host__ __device__` and making sure that the functions can be compiled on the more restrictive environment of a GPU (e.g. GPU code can not use C++ STL classes). If the program is built on a computer that does not support CUDA, care was taken so that the above keywords are replaced by white-spaces and the source code then looks like standard C++ code. The MC also has support for the CERN ROOT framework [3]. Many classes in the code library can be loaded into a ROOT session for interactive calculations. A small complication arose when adding ROOT support, in that the rootcint preprocessor would not ignore the key words `__host__ __device__` (undefined outside of CUDA). It did however accept (and ignore) the symbol `__HOST_DEVICE__`, and this symbol was used as a workaround.

Another consideration is that while some GPUs support half-speed double precision calculations (as compared to single precision), many consumer grade video cards only support $1/8$ or $1/12$ double precision speeds. On the CPU we always use double precision. To get the same function to run in double precision on CPU and single precision on GPU, C++ templates are used throughout the program. The accuracy of the single precision calculation was verified by comparing the results from the GPU to double precision calculations done on the CPU. The following code sample illustrates the basic structure of the source files and classes.

```
#ifdef __CUDACC__
#  define __HOST_DEVICE__  __host__ __device__
#else
#  define __HOST_DEVICE__
#  include <iostream>
#endif

namespace utafd {

  // Vector3 class definition _____________________

  template <typename real_t>
  class Vector3 {
  public:
  __HOST_DEVICE__
  Vector3() : x_(0), y_(0), z_(0) {}

  //...
  __HOST_DEVICE__
  inline real_t dot(const Vector3& v2) const;
  //...

   protected:
    real_t x_, y_, z_;
  };

  // Vector3 class Implementation _________________

  //...
  template <typename real_t>
  __HOST_DEVICE__
  real_t Vector3<real_t>::dot(const Vector3& v2) const
  {
    return x_*v2.x_ + y_*v2.y_ + z_*v2.z_;
  }
  //...
}
```

The preprocessor section starting in `#ifdef` makes it possible to compile the source code on a machine without CUDA. The template paramater real_t is replaced by either float or double depending on where the code is executed.

Random number generation is done differently on the CPU and the GPU. On the CPU we use a routine from Numerical Recipes [4] to generate a single sequence of psudorandom numbers. On the GPU side, each thread gets it's own sequence using a MWC generator [5]. The actual implementaion used was developed by the authors of a program described in [6].

Wrapper functions are used to hide the different RNG implementations, so for example the function:

```
namespace utafd {
  template <typename real_t>
  __HOST_DEVICE__ inline
  real_t random_uniform(unsigned long long* rng_x=0,
                        unsigned int* rng_a=0)
  {
#if !defined(__CUDA_ARCH__)    // host (cpu) path
    double r;
    rangen_(r);       // numerical recepies ran2()
    return (real_t) r;
#else                        // device (gpu) path
    return rand_MWC_co(rng_x, rng_a);
#endif
  }
}
```

Kernel	tpb	nb
generate track segments	32	NE/tpb
shower profile	64	$NE \times NS/tpb$
shower photons (step 1)	32×32	$NE \times NS/tpb.x$
shower photons (step 2)	32	NE
mirror photo-electrons	64	$NME \times NS/tpb$
prep electronics	32	NME/tpb
init electronics	256	MLM
ray tracing	64	$MLM \times 4$
electronics	256	MLM

Table 1: Kernel configuration: tpb = threads per block, nb = number of blocks per launch, NE = total number of shower events, NS = number of track segments, NME = total number of mirror-events, MLM = maximum number of mirrors per launch.

can be called from a host function, omitting the parameters x, and a. On the device, these parameters specify the sequence unique to the calling thread. The preprocessor variable `__CUDA_ARCH__` is used to select the section of the code to be included in the function when compiled for the CPU or GPU.

The kernel launch configuration is modified for each part of the calculation in order to achieve best performance. The implementation for the TA MD site simulation is summarized in table 1. Due to the limited amount of memory on the GPU card, 1 GB in our case, the maximum number of shower events simulated at one time is limited to 768 events. The electronics simulation requires 8MB per mirror and is therefore limited to 32 mirrors at a time with multiple launches for the whole set.

5 Results and Performance

Accuracy of floats vs doubles: during development test runs were made were the same simulation was run on the CPU in double precision and on the GPU in single precision. Intermediate results from each stage of the simulation was saved and compared for accuracy. Where differences were found they were on the order of 0.01%.

Execution speed comparison: A test simulation of the TA Middle Drum detector was done to measure the relative performance of the program. A simulation of a set of 1500 events, $10^{19} eV$ showers, was run on both the CPU (Intel Q8300, 2.50GHz) and GPU (Nvidia GTX 460, 763MHz). The total run times were 6min 16s (CPU) and 6.1 sec (GPU), for an overall speedup of about 60×. The output of the simulations is not identical since the RNG's used are different, however the results were consistency. As an example, the total number of triggered events: 310 vs. 308 with about 300 triggered events being the same shower events.

6 Acknowledgements

The Telescope Array experiment is supported by the Japan Society for the Promotion of Science through Grants-in-Aid for Scientific Research on Specially Promoted Research (21000002) "Extreme Phenomena in the Universe Explored by Highest Energy Cosmic Rays", and the Inter-University Research Program of the Institute for Cosmic Ray Research; by the U.S. National Science Foundation awards PHY-0307098, PHY-0601915, PHY-0703893, PHY-0758342, and PHY-0848320 (Utah) and PHY-0649681 (Rutgers); by the National Research Foundation of Korea (2006-0050031, 2007-0056005, 2007-0093860, 2010-0011378, 2010-0028071, R32-10130); by the Russian Academy of Sciences, RFBR grants 10-02-01406a and 11-02-01528a (INR), IISN project No. 4.4509.10 and Belgian Science Policy under IUAP VI/11 (ULB). The foundations of Dr. Ezekiel R. and Edna Wattis Dumke, Willard L. Eccles and the George S. and Dolores Dore Eccles all helped with generous donations. The State of Utah supported the project through its Economic Development Board, and the University of Utah through the Office of the Vice President for Research. The experimental site became available through the cooperation of the Utah School and Institutional Trust Lands Administration (SITLA), U.S. Bureau of Land Management and the U.S. Air Force. We also wish to thank the people and the officials of Millard County, Utah, for their steadfast and warm support. We gratefully acknowledge the contributions from the technical staffs of our home institutions and the University of Utah Center for High Performance Computing (CHPC).

References

[1] `http://www.nvidia.com/object/cuda_home_new.html`
[2] `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`
[3] `http://root.cern.ch`
[4] William H. Press, et al.: 1992. Numerical Recipes in FORTRAN (2nd Ed.). Cambridge University Press, New York, NY, USA.
[5] Marsaglia G., Random number generators, J. Mod. Appl. Stat. Meth. 2003, **2**, 213
[6] `http://code.google.com/p/gpumcml/`