Journal of Physics: Conference Series **513** (2014) 052001 doi:10.1088/1742-6596/513/5/052001

The Telescope Array Middle Drum fluorescence detector simulation on GPUs

Tareq Abu-Zayyad[1], Telescope-Array Collaboration

[1]Physics Department, University of Utah, Salt Lake City, UT 84112

E-mail: tareq@cosmic.utah.edu

Abstract. In recent years, the Graphics Processing 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 we report on a Monte Carlo program used for the simulation of the Telescope Array Fluorescence Detector located at the Middle Drum site which uses GPU acceleration. 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. A detailed description of the code implementation is given, and results on the accuracy and performance of the simulation are presented as well. Improvements in computational throughput in excess of 50× are reported and the accuracy of the results is on par with the CPU implementation of the simulation.

1. Introduction

The Telescope Array (TA) detector was designed to study cosmic rays at energies of $\sim 10^{18}$ eV and higher. TA comprises three fluorescence detectors (FDs) [1, 2] and a large surface detector [3]. A cosmic rays particle interacts near the top of the atmosphere and generates an extensive air shower (EAS). Charged particles in the EAS ionize the Nitrogen molecules in the air and cause them to emit fluorescence light. FDs are built using light collectors (mirrors), photomultiplier tubes (PMTs) for camera pixel elements, and high-speed electronics which allows them to form an image of the EAS development through the atmosphere. In addition to fluorescence photons, Cerenkov photons are also produced by the high-energy shower particles, and contribute to the detected signal.

Fluorescence detectors rely on Monte Carlo (MC) simulations to calculate the detector aperture, and to check the validity and accuracy of event selection and reconstruction procedures. In addition, event reconstruction typically employs an inverse Monte Carlo fitting method. Calculating the detector response to a single shower (one "event") 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 inverse 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/months to generate and reconstruct enough MC to produce a physics result. Therefore, accelerating the data processing is a highly desirable and useful development.

General-Purpose computing on Graphics Processing 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 five years, after the introduction of CUDA (Compute Unified Device Architecture)[4, 5] by NVIDIA. The introduction of a "high" level interface to the GPU hardware made programming GPUs accessible to a wider audience. 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 be 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. Also, each event is simulated independently of other events, again ideal for GPUs. The simultaneous calculation of the detector response to multiple events improves the efficiency and utilization of the GPU resources.

The paper is organized as follows: Section 2 gives an overview of the simulation physics. Section 3 gives an overview of the MC program flow, along with the steps required to simulate one event. Section 4 describes the basic concepts of the CUDA programming model as it relates to the problem at hand. Section 5 introduces the MC program design considerations and implementation details. Finally, in section 6 we present results of some sample simulations and discuss performance.

2. FD Simulation Physics

The calculation of an FD response to an air shower requires modeling a number of physical systems and processes:

- The atmosphere:
  - Pressure and density profiles as a function of altitude: typically radiosonde data.
  - Ozone layer: using a table of ozone partial pressure versus height.
  - Aerosols: a simplified model of aerosols distribution and light scattering properties.
- Shower development:
  - Longitudinal development: parametrization of shower size as a function of atmospheric depth and in terms of shower age.
  - Lateral distribution of shower particles: parametrization based on the NKG [6, 7] function.
  - Energy deposit: based on Corsika [8] simulations.
- Light production:
  - Air fluorescence: emission by nitrogen molecules excited by shower electrons.
  - Cerenkov light emission by high-energy charged particles: "direct Cerenkov".
  - Scattered Cerenkov light: Rayleigh and aerosols scattered into the FOV of the detector.
- Light propagation:
  - Attenuation due to Rayleigh/aerosols scattering and ozone absorption
- Light detection:
  - Detector geometry: position and orientation.
  - Detector optics: mirror shape and area, shadowing by camera and support structure.
- Detector response:
  - PMT response: quantum efficiency and 2D response profile.
  - Electronics signal processing for a single channel.
  - Trigger logic: event formation.

3. FD Simulation Procedure

The 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 the GPU the copy initialization data to GPU and 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. This is done on the CPU.
- Reduce the data set (on the CPU) by checking whether the generated shower track is in the field of view of any FD telescope. If not, the event is set aside, as no trigger can be expected.
- If running on the CPU then process each shower event sequentially, otherwise copy the reduced set to GPU memory and run the shower simulation in parallel. To accomodate arbitrary size event sets, the simulation is done in "batches":
  - use a batch size of one on the CPU, and ∼768 on the GPU depending on available memory on the GPU card.
  - if using the GPU then copy the detector response for the current event batch back to main memory
  - at the end of processing of a batch, output event data is written to file.
  - repeat until all events are simulated.

An individual shower simulation proceeds as follows:

- Subdivide the shower-track into segments: The track is divided into 832 track segments of equal atmospheric slant depth steps, ∼1-2 g/cm^2 each depending on shower zenith angle.
- Generate Shower Profile: 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 fluorescence 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 geometrical 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 during this time interval.
- Perform ray tracing: Ray trace photons from the track segment through the telescope optics and 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.

4. CUDA Programming model

The reader is referred to [9] for a full introduction to CUDA. Here a few key concepts that are relavent to this paper will be introduced.

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 the 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 the name qualifier `__host__ __device__`.

The CUDA parallel execution model is referred 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 up to 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.

5. 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 maintenance 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 [10]. 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 keywords `__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 code sample shown in listing 1 illustrates the basic structure of the source files and classes.

At the top of listing 1, 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 [11] to generate a single sequence of pseudorandom numbers. On the GPU, each thread gets its own sequence using a MWC generator [12]. The actual implementation of the MWC generator used in this program was developed by the authors of a program described in [13]. Wrapper functions are used to hide the different RNG implementations, for example the function shown in listing 2 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.

Listing 1. model class design: all classes sport the same basic design as shown here for a 3-vector class

```
#ifdef __CUDACC__
# define __HOST_DEVICE__ __host__ __device__
#else
# define __HOST_DEVICE__
#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_;
}
//...
}
```

5.1. Service Classes

Service Classes provide properly initialized data and functions describing the geometry, atmosphere, physics options, and detector configuration used in the simulation. As already mentioned the MC can be run on either the CPU or the GPU and templates are used to allow running in single or double precision. This means that four copies (CPU/GPU, single/double) of an object of the service class type is created at the start of the program. For each case we define a class, e.g. PhysicsService, which provides the required functionality. Any function or object which needs access to physics data or needs to call a physics function will do so by first getting the active PhysicsService object as illustrated in the code listing 3. The CPU double version of the service class is initialized based on user input and *copies* (one float on CPU, and two float/double on GPU) of it are created and copied to GPU if required.

Listing 2. different code paths

```
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.

Listing 3. service class example: An object of type ShowerTrack needs to call a service class to perform a calculation which depends on the chosen coordinate system and on the externally supplied atmospheric paramaters.

```
#include "utafd_coordinates.h"
#include "utafd_atmosphere.h"
namespace utafd
{
  //...
  template <typename real_t>
  __HOST_DEVICE__
  void ShowerTrack<real_t>::set_track_ends(...)
  {
    real_t dummy(0);// used to resolve float or double
    const CoordinatesService<real_t>& utafd_coord =
    service::coord::utafd_coord(dummy);
    const AtmosService<real_t>& utafd_atmos =
    service::atmos::utafd_atmos(dummy);
    //...
    int icode = utafd_coord.track_ends(...);
    //...
    real_t deltax = utafd_atmos.xslant(...);
    //...
  }
}
// ... in utafd_atmosphere.cu ...
// under namespace utafd::service::atmos

__HOST_DEVICE__
const AtmosService<double>& utafd_atmos(double)
{
#if !defined(__CUDA_ARCH__)
  return utafd_atmos_d; // host
#else
  return *utafd_atmos_d_dev; //device
#endif
}
```

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.

6. Results and Performance

6.1. 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 RNGs used are different, however the results were consistent. As an example, the total number of triggered events: 310 vs. 308 with about 300 triggered events being the same shower events. NVIDIA profiler "nvpp" was used to examine where the simulation time was spent. The most time was spent doing ray tracing and second was the electronics simulation. Ray tracing was sped up by reducing code divergence (doing more computation) on the GPU, but code

divergence was still the cause of inefficiency. The electronics simulation initialization was also affected by code divergence in the calculation of the Poisson random deviate which uses a "do while()" loop. The memory access pattern was efficient with the inverted layout of the 2D arrays.

Acknowledgments

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] Abu-Zayyad T, Aida R, Allen M, Anderson R, Azuma R *et al.* 2012 *Astropart.Phys.* **39-40** 109–119 (*Preprint* 1202.5141)
[2] Tokuno H, Tameda Y, Takeda M, Kadota K, Ikeda D *et al.* 2012 *Nucl.Instrum.Meth.* **A676** 54–65 (*Preprint* 1201.0002)
[3] Abu-Zayyad T, Aida R, Allen M, Anderson R, Azuma R *et al.* 2012 *Nucl.Instrum.Meth.* **A689** 87–97 (*Preprint* 1201.4964)
[4] https://en.wikipedia.org/wiki/CUDA
[5] http://www.nvidia.com/object/cuda_home_new.html
[6] Greisen K 1956 *Prog. Cosm. Ray Phys.* **3** 1
[7] Kamata K N J 1958 *Prog. Theor. Phys. Suppl.* **6** 93
[8] Heck D, Schatz G, Thouw T, Knapp J and Capdevielle J 1998
[9] http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
[10] http://root.cern.ch
[11] William H Press e a 1992 *Numerical Recipes in FORTRAN* 2nd ed (New York, NY, USA: Cambridge University Press)
[12] G M 2003 *J. Mod. Appl. Stat. Meth.* **2** 213
[13] http://code.google.com/p/gpumcml/