CUDA EGS Manual

Version 1.0.0

Jonas Lippuner and Idris A. Elbakri

CancerCare Manitoba
675 McDermot Ave
Winnipeg MB R3E 0V9
Canada

January 2012

The CUDA EGS source code and this document are available online at
`http://www.physics.umanitoba.ca/~elbakri/cuda_egs`
and
`http://www.lippuner.ca/cuda_egs`.

Contact: `jonas@lippuner.ca`

Abstract

CUDA EGS is a parallel implementation of EGSnrc's photon transport mechanism for graphics processing units (GPUs) using NVIDIA's Compute Unified Device Architecture (CUDA). EGSnrc is a well-known Monte Carlo simulation package for coupled electron–photon transport that is widely used in medical physics applications. CUDA EGS is specifically designed for imaging applications in the diagnostic energy range and does not model electrons. No approximations or simplifications of the original EGSnrc code were made other than using single floating-point precision instead of double precision and a different random number generator. The photons leaving the simulation geometry are propagated to a virtual detector, and separate images for primary and scattered photons are generated, allowing a detailed analysis of the amount and distribution of x-ray scattering. CUDA EGS was found to be 20 to 40 times faster than EGSnrc while producing equivalent results.

Table of Contents

Abstract ii

Table of Contents ii

1 Introduction **1**

1.1 Overview 1

1.2 Features and Limitations 2

1.3 License 2

2 Installation **2**

2.1 Requirements 2

2.2 Compiling CUDA EGS 3

2.3 Configuring CUDA EGS 3

3 Running CUDA EGS **4**

4 Input File **5**

4.1 General 6

4.2 Source 7

4.2.1 Spectrum File Format 7

4.2.2 A Note about Geometry 8

4.3 Detector 8

4.4 Data 8

5 Output **9**

5.1 Timing Statistics 9

5.2 Output Files 10

5.3 Count and Energy Output Files 10

5.4 Image Output Files 11

6 Error Messages **11**

7 Notes to Developers **12**

7.1 Source Code Overview 12

7.2 Random Number Generator 13

8 Version History **14**

References **14**

1 Introduction

This manual provides instructions on how to install and use CUDA EGS and it also contains notes to developers who wish to alter or expand the CUDA EGS code. This manual does not explain the basic functionality of CUDA EGS and the reader is referred to section 3.1 of the paper [1] that introduced CUDA EGS (although not with that name). When referring to CUDA EGS in academic work, please cite that paper. An author edited version of the paper is available free of charge at http://www.lippuner.ca/publications/Lippuner_CUDA_EGS_2011_color.pdf.

1.1 Overview

CUDA EGS is a CUDA implementation of the photon transport mechanism of EGSnrc [2]. EGSnrc is a Monte Carlo simulation package designed to simulate coupled electron–photon transport through an arbitrary geometry and CUDA is a programming framework developed by NVIDIA to write programs that can execute parallel code on graphics processing units (GPUs). The main purpose of CUDA EGS is to provide a fast implementation of the highly trusted photon transport mechanism of EGSnrc to simulate x-ray Compton and Rayleigh scattering arising in imaging applications. CUDA EGS was found to be 20 to 40 times faster than EGSnrc while preserving the accuracy of EGSnrc [1].

CUDA EGS does not implement all aspects of EGSnrc. Most notably, only the photon transport mechanism of EGSnrc is implemented in CUDA EGS, i.e. electrons or positrons are not simulated at all. Furthermore, only basic Klein-Nishina Compton scattering is implemented, i.e. bound Compton scattering is not implemented. However, photon propagation, basic Compton scattering and Rayleigh scattering are implemented faithfully following the mechanism of EGSnrc. Furthermore, the data model and overall simulation techniques of EGSnrc have been kept, so that CUDA EGS can be extended in the future to implement other aspects of EGSnrc as well.

In addition to implementing EGSnrc's photon transport mechanism, CUDA EGS also keeps track of the number of scatter events that a photon undergoes during the simulation. When a photon leaves the simulation geometry, it is propagated to a virtual detector specified by the user, that records the total number of photons and the total energy deposited in each pixel. Binary files as well as images containing these results are created.

For analyzing the amount and distribution of Compton and Rayleigh scatter, the photons are assigned to one of the following categories for which separate output files are created; see section 5.2 for more information:

- **Primary:** photons that have never been scattered
- **Compton:** photons that have been Compton scattered exactly once
- **Rayleigh:** photons that have been Rayleigh scattered exactly once
- **Multiple:** photons that have been scattered more than once

The propagation of the photons to a detector and scoring them in different categories according to the number of scatter events that they experienced during the simulation is the same feature that the EGSnrc user code Epp [3] offers. The output files of CUDA EGS and Epp are identical.

1.2 Features and Limitations

Features

- All input parameters are conveniently specified in one input file.
- Command line options make it easy to run several instances in parallel on different GPUs with different random number seeds using the same input file.
- Phantoms are defined by `*.egsphant` files and media properties by PEGS4 files.
- The total number of photons and the total energy deposited in each pixel of the detector are recorded.
- Different output files are created for primary photons, Compton scatter, Rayleigh scatter, multiple scatter and total photons.
- Monoenergetic sources and sources with a tabulated energy spectrum are supported.

Limitations

- No electrons or positrons are simulated.
- Bound Compton scattering is not implemented, only basic Klein-Nishina Compton scattering.
- Atomic relaxations are not implemented.
- Multithreading over multiple GPUs is not implemented, but can be achieved manually.
- An NVIDA GPU with CUDA capability 2.0 or higher is required.
- Only Windows is supported at this point, but porting to other platforms may not be too hard.

1.3 License

2 Installation

CUDA EGS has been developed with Microsoft Visual Studio and only supports Windows at this point. However, it may not be too hard to port CUDA EGS to other platforms that support CUDA. The source code of CUDA EGS is available online at `http://www.physics.umanitoba.ca/~elbakri/cuda_egs` and `http://www.lippuner.ca/cuda_egs`.

2.1 Requirements

CUDA EGS requires an NVIDIA GPU with CUDA capability 2.0 or higher. To find out whether your GPU supports that, visit `http://developer.nvidia.com/cuda-gpus`. The current version of CUDA EGS has been tested with the CUDA Toolkit 4.0, but it should also work with CUDA Toolkit 3.x. The source code is distributed as a Microsoft Visual Studio 2010 solution, hence it is most easily compiled using Microsoft Visual Studio 2010 or Microsoft Visual C++ 2010 Express. It should be possible to compile CUDA EGS with other compilers as well, but no alternatives have been tested.

2.2 Compiling CUDA EGS

To install CUDA EGS on your CUDA enabled Windows system, follow these steps:

1. Make sure you have a working installation of Microsoft Visual Studio and CUDA. CUDA EGS was developed with Microsoft Visual C++ 2010 Express, hence it is preferable to use Microsoft Visual Studio 2010 or Microsoft Visual C++ 2010 Express, but earlier versions might work as well. Microsoft Visual C++ 2010 Express can be downloaded free of charge from `http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express` and CUDA can be obtained from `http://www.nvidia.com/getcuda`, also free of charge. Consult the CUDA Getting Started Guide for Windows (available at `http://developer.nvidia.com/nvidia-gpu-computing-documentation`) for information on how to install CUDA.
2. Open the CUDA EGS solution (`CUDA_EGS.sln`) with Microsoft Visual Studio or Microsoft Visual C++ Express.
3. Configure CUDA EGS as described in section 2.3. Note that you **must at least** set the constants `WARP_SIZE` and `NUM_MULTIPROC` to their correct values.
4. Build the Release version of the solution with Microsoft Visual Studio or Microsoft Visual C++ Express.

2.3 Configuring CUDA EGS

Some options must be specified at compile time because they are hard coded. These options are specified in the section `CONFIGURATION` at the beginning of the file `CUDA_EGS.h`. The following options are available.

`USE_ENERGY_SPECTRUM`

If `USE_ENERGY_SPECTRUM` is defined, the photon source will use a tabulated energy spectrum instead of a constant energy.

`DO_LIST_DEPTH_COUNT`

If `DO_LIST_DEPTH_COUNT` is defined, the average number of different simulation steps performed in the inner loop will be measured. This gives an estimate for the divergence of the threads within the simulation kernel. See the last paragraph of section 3.1.3 in [1] for more details. Note that this option requires slightly more shared memory.

`WARP_SIZE`

The constant `WARP_SIZE` must be set to the warp size of the GPU used for the simulations. Use the sample program *Device Query* included with the CUDA SDK to find the warp size of your GPU.

`NUM_MULTIPROC`

The constant `NUM_MULTIPROC` must be set to the number of multiprocessors of the GPU used for the simulations. Use the sample program *Device Query* included with the CUDA SDK to find the number of multiprocessors of your GPU.

`SIMULATION_WARPS_PER_BLOCK`

The constant `SIMULATION_WARPS_PER_BLOCK` defines the number of warps used for each thread block of the simulation kernel. Use the *CUDA GPU Occupancy Calculator* included with the

CUDA SDK to determine the optimal value for this number. The simulation kernel of the current implementation of CUDA EGS uses 63 registers per thread, which limits the number of warps per block to 16 for GPUs with CUDA capability 2.0. This constant has a big influence on the amount of shared memory required by one block. To see the number or registers and amount of shared memory used, pass the command line option `--ptxas-options="-v"` to `nvcc`. This is done by default in the Visual Studio solution included with the source code. Note that it may be necessary to set *MSBuild project build output verbosity* (under Tools > Options… > Projects and Solutions > Build and Run) to a higher level in Microsoft Visual Studio in order to see the information in the Output window.

`SIMULATION_BLOCKS_PER_MULTIPROC`

The constant `SIMULATION_BLOCKS_PER_MULTIPROC` defines the number thread block launched per multiprocessor for the simulation kernel. The *CUDA GPU Occupancy Calculator* will show how many thread blocks can be simultaneously active per multiprocessor based on the number of registers and shared memory used by the kernel. For GPUs with CUDA capability 2.0 this is 1. There is probably no significant benefit to choosing a much larger number of blocks per multiprocessor than the maximum number of blocks that can be simultaneously active on one multiprocessor.

`SIMULATION_ITERATIONS`

The constant `SIMULATION_ITERATIONS` defines the number of iteration of the outer loop that are performed in the simulation kernel [see 1, section 3.1.3]. A larger number will increase the performance of the simulation because fewer kernels launches will be necessary, which all have an overhead cost. However, a larger number will also increase the accumulative effect of single precision rounding errors, thus potentially decreasing the accuracy of the simulation.

3 Running CUDA EGS

To run CUDA EGS, run the following command on the command line (arguments in square brackets are optional).

```
CUDA_EGS -i input_file [-o output_prefix] [-s RNG_seed] [-g GPU_id]
```

`input_file` is a relative or absolute path to the input file defining all parameters of the simulation, see section 4.

`output_prefix` is the prefix of all output files (log file and simulation results). The prefix contains the path to the output directory as well as a prefix for the output files. For example, `C:\path\to\output\directory\Simulation_1_`. The path can also be a relative path with respect to the working directory. Make sure that the output directory exists.

`RNG_seed` is an unsigned integer (a number between 0 and 4294967295) that is used to initialize the random number generators.

`GPU_id` is the id of the GPU that is to be used for the simulation. Use the sample program *Device Query* included with the CUDA SDK to see a list of the available GPUs and their id's.

The command line arguments `-h` and `--help` will display the usage information shown above.

Note that the optional arguments can also be specified in the input file, but the values given as command line options take precedence over the values specified in the input file. Using these command line options, one can run the same simulation on different GPUs simultaneously using the same input file. Simply define all the simulation parameters in the input file, and then run CUDA EGS once for each GPU. For each instance, define a different output prefix, a different RNG seed and the corresponding GPU id of the GPU that should be used. Note that the results will not be automatically combined; this has to be done manually.

Before the actual simulation is started, CUDA EGS will display all the simulation parameters. When the simulation is running, the progress and ETA will be shown. Note that this is updated every time a simulation kernel has finished. Since the simulation kernel might take quite a while to run, the progress information is not updated very frequently and the simulation may appear to be frozen. The length it takes for one simulation kernel to run depends on the GPU and on the constants `SIMULATION_BLOCKS_PER_MULTIPROC` and `SIMULATION_ITERATIONS` described in section 2.3.

After the simulation is finished, CUDA EGS will display the number of different simulation steps that were performed, the average number of different simulation steps performed in the inner loop if `DO_LIST_DEPTH_COUNT` is enabled, and finally, some timing statistics. See section 5.1 for details about the timing statistics.

Note that all the information displayed on the command line by CUDA EGS is also written to the log file `[output_prefix]log.txt`, unless an error occurs before the log file could be created.

IMPORTANT NOTICE

The simulation kernel will typically require several seconds to complete. During this time the GPU will not be available to draw the Windows screen. If the same GPU is used that draws the Windows screen, the screen will appear frozen while the simulation kernel is running. Starting with Windows Vista, Windows will detect this issue and abort the GPU kernel. The effect is that CUDA EGS will crash and an error message saying something like “Display driver stopped responding and has recovered” will be displayed.

To prevent Windows from aborting the simulation kernel, TDR must be disabled. This is most easily accomplished by creating a key of type `DWORD` called “TdrLevel” with value 0 in the Windows Registry in `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GraphicsDrivers`. For more information, visit `http://msdn.microsoft.com/en-us/windows/hardware/gg487368`.

4 Input File

All simulation parameters are specified in the input file. The input file follows the INI file format, which defines different sections that contain key-value pairs. Comments lines can be inserted by starting a line with a semicolon. The following example illustrates the basic syntax for an input file.

```
[Section]
key 1 = value
key 2 = value
; this is a comment

[Another Section]
key 3 = "a string value"
```

An example input file `CUDA_EGS_example.ini` is included with the source code. The input file consists of four sections which are described below.

4.1 General

The section `General` defines some general simulation parameters.

```
[General]
output prefix = "CUDA_EGS_example_"
egsphant file = "data\CUDA_EGS_phantom_32.egsphant"
pegs file = "data\CUDA_EGS_phantom.pegs4dat"
histories = 100000000
RNG seed = 1325631759
GPU = "GeForce GTX 470"
GPU id = 0
```

`output prefix` is the prefix of all output files (log file and simulation results). The prefix contains the path to the output directory as well as a prefix for the output files. For example, `C:\path\to\output\directory\Simulation_1_`. The path can also be a relative path with respect to the working directory. Make sure that the output directory exists. This key is optional, but a warning will be displayed if it is missing. This key can be overridden by the command line argument `-o output_file`.

`egsphant file` is the absolute or relative path to the `*.egsphant` file defining the phantom. For more information about the `egsphant` file format, see section 15.6 of the DOSXYZnrc manual [4].

`pegs file` is the absolute or relative path to the PEGS4 file defining the media used in the `*.egsphant` file. For more information about PEGS4, see section 6 of the EGSnrc manual [2].

`histories` is the number of histories that should be simulated. This number must lie between 1 and 9223372036854775807.

`RNG seed` is an unsigned integer (a number between 0 and 4294967295) that is used to initialize the random number generators. This key is optional, but a warning will be displayed if it is missing. If no seed is given, a seed based on the current time will be generated. This key can be overridden by the command line argument `-s RNG_seed`.

`GPU` is the name of the GPU that is to be used for the simulation. The name must be exactly as it is shown by the sample program *Device Query* included with the CUDA SDK. If more than one GPUs with the same name are available, the first will be used. This key is optional, but a warning will be displayed if no GPU is specified using either this key or the key `GPU id`. This key is overridden by the key `GPU id` or the command line argument `-g GPU_id`.

`GPU id` is the id of the GPU that is to be used for the simulation. Use the sample program *Device Query* included with the CUDA SDK to see a list of the available GPUs and their id's. This key is optional, but a warning will be displayed if no GPU is specified using either this key or the key `GPU`. This key is overridden by the command line argument `-g GPU_id`.

4.2 Source

The section `Source` defines the photon source of the simulation. Currently, the only implemented source is a point source with a virtual rectangular collimator. The source is either monoenergetic or uses a tabulated energy spectrum.

```
[Source]
spectrum file = "data\tungsten-80kVp-4mmAl.spectrum"
energy = 0.08
position = 0 0 -30
collimator rectangle = -6.4 -6.4 6.4 6.4
collimator z = -6.4
```

`spectrum file`	is the absolute or relative path to the file specifying the tabulated energy spectrum of the source. If `USE_ENERGY_SPECTRUM` is defined (see section 2.3), this key is required, otherwise it is ignored. See below for information about the format of the spectrum file.
`energy`	is the energy in MeV of the photons. If `USE_ENERGY_SPECTRUM` is not defined (see section 2.3), this key is required, otherwise it is ignored.
`position`	are the x, y and z coordinates in cm of the source point.
`collimator rectangle`	are the minimum and maximum x and y coordinates in cm of the source collimator. The first two values are the minimum x and y coordinates and the last two values the maximum x and y coordinates.
`collimator z`	is the z coordinate in cm of the source collimator.

Note that in the current implementation of CUDA EGS, the source collimator is always perpendicular to the z-axis.

4.2.1 Spectrum File Format

The tabulated energy spectrum is defined as a histogram with N bins. p_i is the probability that a new photon has an energy between E_{i-1} and E_i for $i = 1 \ldots N$. The energy distribution within one bin, i.e. between E_{i-1} and E_i, is uniform.

The spectrum is defined by a file containing the following lines.

```
name
N E_0 mode
E_1 p_1
E_2 p_2
...
E_N p_N
```

`name`	is some name describing the spectrum.
`N`	is the number of bins N.
`E_0`	is the lower energy E_0 in MeV of the first bin.

`mode`	is either 0 or 1. If mode is 0, the probabilities p_i are considered to be counts per bin, and if mode is 1, they are considered to be counts per MeV.
`E_i`	is the upper energy E_i in MeV of the i-th bin.
`p_i`	is the probability p_i associated with the i-th bin.

4.2.2 A Note about Geometry

The direction of photon propagation is along the positive z axis. All photons start at the position of the source point. The initial direction of a photon is determined by choosing a random point on the source collimator (uniformly distributed) through which the photon will travel. To account for the fact that the source is an isotropic point source, the statistical weight of the photon is adjusted based on the point on the collimator that was chosen. Therefore, to get photons travelling in the positive z direction, `collimator z` must be larger than the z coordinate of the source position. The location of the phantom is specified by the voxel boundaries given in the `*.egsphant` file.

4.3 Detector

The section `Detector` defines the virtual detector to which all photons leaving the simulation geometry are propagated.

```
[Detector]
position = 0 0 30
size = 512 512
pixel size = 0.1 0.1
```

`position`	are the x, y and z coordinates in cm of the centre of the detector.
`size`	is the number of pixels on the detector in the x and y directions.
`pixel size`	is the size of the individual pixels in cm in the x and y directions.

Note that in the current implementation of CUDA EGS, the detector is always perpendicular to the z-axis. Also note that the detector must be outside of the phantom. Furthermore, the detector is sensitive on both sides.

4.4 Data

The section `Data` defines where external data files are located.

```
[Data]
data directory = "data"
MT parameter file = "MTGP_3217_0-8191.bin"
photon xsections = "si"
atomic ff file = "pgs4form.dat"
```

`data directory`	is an absolute or relative path to the directory where the data files are located. A relative path must be relative to the working directory.

MT parameter file is the relative path (with respect to the data directory) to the file that contains the parameters for the Mersenne Twister random number generators, see section 7.2 for more information. The file `MTGP_3217_0-8191.bin` contains 8192 parameter sets that were generated using the *MTGP Dynamic Creator* available at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MTGP/`. Note that the data in the file is in little endian format.

photon xsections is the prefix of the data files containing the photon cross section data. The files `x_photo.data`, `x_pair.data`, `x_triplet.data` and `x_rayleigh.data` must exist in the data directory, where `x` is the value of `photon xsections`. The Storm–Israel files are included with the source code. See section 2.2.5 of the EGSnrc manual [2] for more information.

atomic ff file is the relative paht (with respect to the data directory) of the file containing the coherent scattering form factors. The default file `pgs4form.dat` of EGSnrc is included with the source code. See section 2.2.4 of the EGSnrc manual [2] for more information.

5 Output

5.1 Timing Statistics

The output on the command line and in the log file contains some timing statistics. An example is shown below.

```
Timing statistics
  Elapsed time  . . . . . . . 14752.00 ms (100.00 %)
  Total CPU/GPU . . . . . . . 14392.00 ms (97.56 %)
  Simulation kernel . . . . . 13955.97 ms (94.60 %)
  Summing kernel  . . . . . . 25.37 ms (0.17 %)
  Copying . . . . . . . . . . 16.00 ms (0.11 %)
  Other . . . . . . . . . . . 754.65 ms (5.12 %)
  Initialization. . . . . . . 360.00 ms (2.44 %)

Histories per ms: 6898.507592
```

Elapsed time is the total elapsed time from when the program was started to when everything except the timing statistics was finished.

Total CPU/GPU is (approximately) the total time that the CPU and GPU spent on the simulation. It is the elapsed time minus the initialization time.

Simulation kernel is the time the simulation kernel was running on the GPU.

Summing kernel is the time the summing kernel was running on the GPU.

Copying is the time that was spent copying data to and from the GPU, excluding the time spent copying during the initialization.

Other is the elapsed time minus the time spent in the simulation and summing kernels and minus the time spent copying. This is the time not spent on the actual simulation, i.e. the time spent for the initialization and writing the output files.

`Initialization` is the time spent for the initialization.

`Histories per ms` gives the number of histories that were simulated per millisecond, based on the total elapsed time.

5.2 Output Files

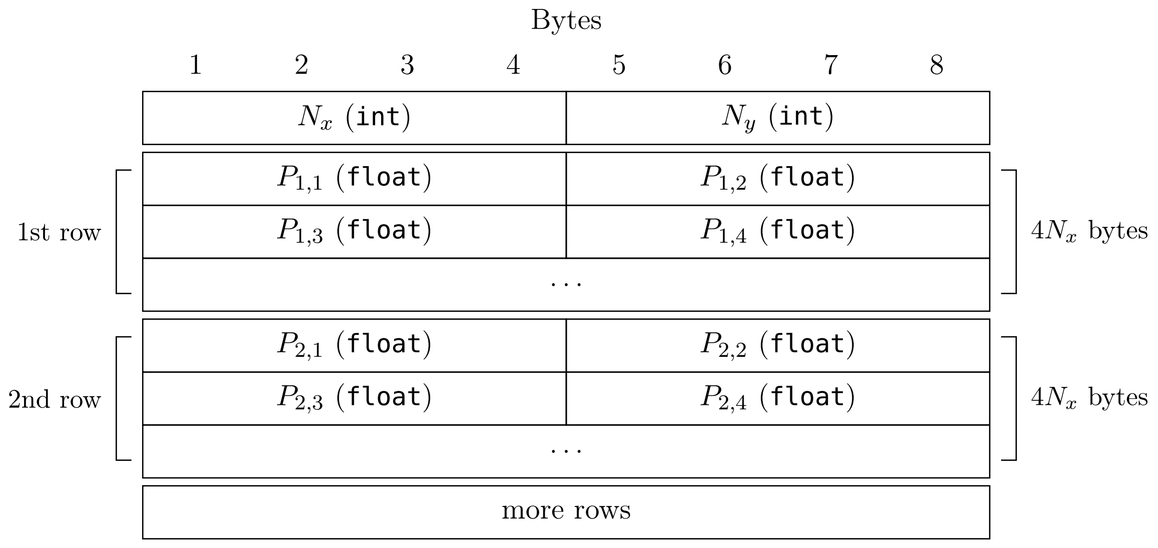
CUDA EGS records the number and total energy of photons that reach the detector. Separate output files for the photon categories primary, Compton, Rayleigh, multiple scatter and total are created and labelled with the suffixes `p`, `c`, `m`, `r` and `t`, respectively.

Due to the way that the photon source is implemented, one cannot simply count discrete photons on the detector; instead one has to sum the statistical weights of the photons. Similarly, for recording the total energy deposited in the detector, the energy of each individual photon has to be multiplied by its statistical weight and then this product will contribute toward the total energy. The sum of the statistical weights or the sum of the energies multiplied by the statistical weights must then be divided by the average statistical weight of all photons generated by the source. This is done automatically by CUDA EGS and it is necessary to obtain meaningful values so that the sum of the statistical weights—even though it is a fraction—corresponds to the number of photons reaching the detector.

5.3 Count and Energy Output Files

The count (`[output_prefix]count_[p|c|m|r|t].bin`) and energy (`[output_prefix]energy_[p|c|m|r|t].bin`) output files contain the number of photons—actually the sum of the statistical weights as explained above—and the total energy deposited in each pixel of the detector. The information is stored in binary and for each pixel there is one 4-byte single precision floating point number. The pixels are arranged in row-major order, i.e. one full row is stored after another. For one row the pixels are stored from left to right and the rows are stored from top to bottom.

In addition to the data for each individual pixel of the detector, the first eight bytes of the file contain two 4-byte integers that specify the number of pixels in the x direction (columns) and the y direction (rows). The following diagram illustrates the format of a count or energy output file.



$\boldsymbol{N_x}$, $\boldsymbol{N_y}$	are the number of pixels in the x direction (columns) and the y direction (rows).
$\boldsymbol{P_{i,j}}$	is the total number of photons or total energy that was deposited in the pixel in the i-th row and j-th column, $i = 1 \dots N_y$, $j = 1 \dots N_x$.

The file `ReadCUDAEGSOutput.m` included with the source code can be used to read these output files directly into MATLAB.

5.4 Image Output Files

The images (`[output_prefix]count_[p|c|m|r|t].bmp`) and (`[output_prefix]energy_[p|c|m|r|t].bmp`) are bitmap images of the count and energy results. The pictures are in gray scale with zero photons/energy being black and the highest number of photons/energy in that category being white. All values in between are linearly scaled.

6 Error Messages

CUDA EGS performs a lot of error checking and attempts to display meaningful error messages. Each error has a number of the form $E \times 1000 + i$ where E is a positive integer representing a group of errors and i is a positive integer between 1 and 999 representing an individual error within the group E. The purpose of the error number is to make it easier to find the location in the source code where the error occurred. The following table shows where in the source code the different error groups occur and what issues they are related to.

E	File	Function	An error occurred while ...
1	`CUDA_EGS.cu`	`main`	reading the command line arguments
2	`CUDA_EGS.cu`	`main`	reading the input file
3	`init.cu`	`init_MTs`	initializing the Mersenne Twister random number generators
4	`init.cu`	`init_stack`	initializing the stack
5	`init.cu`	`init_source`	initializing the source
6	`init.cu`	`init_detector`	initializing the detector
7	`init.cu`	`init_regions`	initializing the regions
8	`init.cu`	`init_phantom`	reading the `*.egsphant` file and initializing the phantom
9	`init.cu`	`free_all`	freeing all allocated memory
10	`media.c`	various	reading the PEGS4 file and initializing the media data
11	`CUDA_EGS.cu`	`main`	running the simulation
12	`CUDA_EGS.cu`	`read_step_counts`	copying the simulation step counters from the GPU

Most errors are issued with `error([error_number], ...)`. Errors related to CUDA are issued with `ce([error_number], ...)`. To find find where an error occurs in the source code, simply search for the full error number, e.g. 3002 or 11007, in the code.

7 Notes to Developers

The basic functionality and method of CUDA EGS is not described in this manual. The reader is referred to the original paper [1], especially section 3.1. In the following we give a brief overview of the source code. The code itself contains many comments and should be fairly self-explanatory. The comments in the code and the following discussion assume that at least section 3.1 of [1] has been read, since the terminology introduced there will be used.

There are many chunks of code that have been copied from other existing code. The main source is the original MORTRAN/FORTRAN code of EGSnrc, there is some code from the EGSnrc C++ Class Library and some from the Mersenne Twister for Graphic Processors (MTGP). The sources of such code are indicated with comments in the CUDA EGS source code. Those code chunks usually do not contain many comments and the variable names may not be as self-explanatory as in the original CUDA EGS code. For more information about code taken from the EGSnrc code system, see the EGSnrc manual [2] and comments in the EGSnrc code. For details about code taken from the EGSnrc C++ Class Library, see its documentation [5] and source code. Finally, more information about MTGP can be found in [6] and the source code of an implementation is available at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MTGP/`.

7.1 Source Code Overview

In this section, a brief description of the different source files is given to provide an overview of the source code.

`CUDA_EGS.cu`

This is the main file of CUDA EGS, containing the function `main`. `main` reads the command line arguments and the input file. Then it initializes the whole simulation and launches the simulation kernel followed by the summing kernel until the simulation is complete. Then it copies the results from the GPU, creates the output files and writes all the command line output.

`CUDA_EGS.h`

This is the main header file CUDA EGS containing all the definitions of constants, data structures and variables. Pointers that point to memory residing on the device (the GPU) are usually prefixed with `d_`, while pointers to memory residing on the host (the CPU) are usually prefixed with `h_`.

`init.cu`

This file contains different functions that initialize the Mersenne Twister random number generators, the stack, the source, the detector, the regions and the phantom.

`kernels.cu`

This file contains all the code that is executed on the GPU. Most importantly, it contains the simulation and summing kernels as well as various helper functions. The different simulation steps of the simulation kernel are all implemented in separate functions. Most of the code in this file comes from the original EGSnrc code or the EGSnrc C++ Class Library code.

media.c

This file handles everything connected to reading a PEGS4 file and initializing the media data. Most of the code was taken from the EGSnrc code.

output.c

This file contains some functions for writing strings and error messages to the command line and the log file. It also contains the functions that create the output files.

bmp.c

This file creates the Bitmap output images. It was taken from Xplanet available at `http://xplanet.sourceforge.net/`.

7.2 Random Number Generator

As described in section 3.1.5 of [1], each warp has a different instance of a Mersenne Twister. Specifically, the MTGP [6] implementation is used. The various Mersenne Twisters all have different parameters, so that they will produce independent sequences of random numbers given the same initial seed. Such parameter sets can be generated with the *MTGP Dynamic Creator* available at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MTGP/`. CUDA EGS loads the parameter sets from a given file in the data directory (see section 4.4). This file contains binary data with the following structure.

	Bytes		
	1 2 3 4	5 6 7 8	
1st parameter set	`mexp (uint)`	`bit_size (uint)`	60 bytes
	`id (uint)`	`M (uint)`	
	`sh1 (uint)`	`sh2 (uint)`	
	`tbl[0] (uint)`	`tbl[1] (uint)`	
	`tbl[2] (uint)`	`tbl[3] (uint)`	
	`tmp_tbl[0] (uint)`	`tmp_tbl[1] (uint)`	
	`tmp_tbl[2] (uint)`	`tmp_tbl[3] (uint)`	
	`mask (uint)`	▨ not present	
2nd parameter set	`mexp (uint)`	`bit_size (uint)`	60 bytes
	`id (uint)`	`M (uint)`	
	. . .		
	more parameter sets		

mexp	is the Mersenne prime exponent p ($p = 3217$ in this implementation).
bit_size	is the number of bits w of one random number ($w = 32$ in this implementation).
id	is the number (id) of the parameter set in the file starting with 0.

`M`	is the pick up position (also called `pos` in the MTGP code).
`sh1` and `sh2`	are the two shift parameters.
`tbl[i]`	is the i-th entry of the array `tbl` that will generate the recursion matrix.
`tmp_tbl[i]`	is the i-th entry of the array `tmp_tbl` that will generate the tempering matrix.
`mask`	is a bitmask that sets the r least-significant bits to 0 where $r = w\lceil p/w \rceil - p$ ($r = 15$ in this implementation and so `mask` is `0xFFFF8000`).

Note that a `uint` is an unsigned 32-bit integer. Also note that the field marked "not present" in the above diagram is not simply not present in the file, i.e. the entry `mexp` follows immediately after the entry `mask` of the previous parameter set; the data is **not** aligned to 64 bytes.

The file `MTGP_3217_0-8191.bin` included with the source code contains 8192 different parameter sets.

8 Version History

Version 1.0.0 Mon 09 Jan 2012

- First version with all core features implemented.

References

[1] Lippuner J and Elbakri I A 2011 A GPU implementation of EGSnrc's Monte Carlo photon transport for imaging applications *Phys. Med. Biol.* **56** 7145–7162
`http://www.lippuner.ca/publications/Lippuner_CUDA_EGS_2011_color.pdf`

[2] Kawrakow I, Mainegra-Hing E, Rogers D W O, Tessier F and Walters B R B 2011 The EGSnrc Code System: Monte Carlo Simulation of Electron and Photon Transport NRCC Report PIRS-701 National Research Council of Canada, Ottawa, Canada
`http://irs.inms.nrc.ca/software/egsnrc/documentation/pirs701`

[3] Lippuner J, Elbakri I A, Cui C and Ingleby H R 2011 Epp: A C++ EGSnrc user code for x-ray imaging and scattering simulations *Med. Phys.* **38** 1705–1708
`http://www.lippuner.ca/publications/Lippuner_Epp_Med_Phys_2011.pdf`

[4] Walters B, Kawrakow I and Rogers D W O 2011 DOSXYZnrc Users Manual NRCC Report PIRS-794revB National Research Council of Canada, Ottawa, Canada
`http://irs.inms.nrc.ca/software/beamnrc/documentation/pirs794`

[5] Kawrakow I, Mainegra-Hing E, Tessier F and Walters B R B 2009 The EGSnrc C++ class library NRCC Report PIRS-898 (rev A) National Research Council of Canada, Ottawa, Canada
`http://irs.inms.nrc.ca/software/egsnrc/documentation/pirs898`

[6] Saito M 2010 A Variant of Mersenne Twister Suitable for Graphic Processors. arXiv:1005.4973v2 [cs.MS]