CUDAMCML

User manual and implementation notes

Erik Alerstam*, Tomas Svensson, Stefan Andersson-Engels
Department of Physics
Lund University, Sweden
* erik.alerstam@fysik.lth.se

July 6, 2009
First release: April 3, 2009

Contents

1 Introduction **3**

2 CUDAMCML for users **4**
2.1 What is CUDA and GPGPU? . . . 4
2.2 Installation . . . 4
2.2.1 Windows . . . 4
2.2.2 Linux . . . 4
2.2.3 Mac OS X . . . 5
2.3 Running CUDAMCML . . . 5
2.4 Limitations . . . 6
2.4.1 Deviations from MCML . . . 6
2.4.2 Limitations related to CUDA/GPGPU . . . 7
2.5 Performance . . . 7
2.6 Validation . . . 8
2.6.1 Simulation 1 . . . 9
2.6.2 Simulation 2 . . . 9

3 CUDAMCML for programmers **11**
3.1 Parallel programming using CUDA . . . 11
3.1.1 CUDA computational considerations . . . 11
3.2 Program overview . . . 12
3.3 Memory structure . . . 14
3.3.1 Global device memory space . . . 14
3.3.2 Constant device memory space . . . 14
3.3.3 Multiprocessor registers . . . 15
3.4 Handling layers . . . 15
3.5 Massively parallel random number generation . . . 16

3.6 Simulating an exact number of photons . . . 17
3.7 Photon detection . . . 17
3.7.1 Basic problem . . . 17
3.7.2 Current solution . . . 17
3.7.3 Atomic float operations . . . 18
3.7.4 Fast memory caching . . . 18
3.7.5 Implication of the current solution . . . 19

4 Licence **19**

A CUDAMCML functions **20**
A.1 Global device functions . . . 20
A.1.1 LaunchPhoton_Global() . . . 20
A.1.2 MCd() . . . 20
A.2 Device functions . . . 20
A.2.1 rand_MWC_oc() and rand_MWC_co() . . . 20
A.2.2 LaunchPhoton() . . . 21
A.2.3 Spin() . . . 21
A.2.4 Reflect() . . . 21
A.2.5 PhotonSurvive() . . . 21
A.2.6 AtomicAddULL() . . . 22

B Structures **23**
B.1 SimulationStruct . . . 23
B.2 DetStruct . . . 23
B.3 LayerStruct . . . 24
B.4 PhotonStruct . . . 24
B.5 MemStruct . . . 25

C Multiply-With-Carry Random Number Generator **26**
C.1 MWC basics . . . 26
C.1.1 Base . . . 26
C.1.2 Multiplier . . . 26
C.1.3 Finding suitable multipliers . . . 26
C.1.4 Seeding the generator . . . 28
C.2 Implementation . . . 29
C.2.1 Setting up the generators . . . 29
C.2.2 CUDA MWC implementation . . . 29

D How to compile **31**

E Release Notes **32**

References **33**

1 Introduction

This document serves as a manual for CUDAMCML and also contains some implementation details that hopefully will be of interest to anyone hoping to write custom Monte Carlo programs for execution on GPU's.

CUDAMCML is a software for Monte Carlo simulation of photon migration in multilayered media. The code is designed to solve the same problems as the original MCML as written by Wang and Jacques [1]. While MCML is run sequentially on a CPU, CUDAMCML is executed on a graphics processing unit, taking advantage of the parallelism of photon migration Monte Carlo.

The concept of General Purpose computing on Graphics Processing Units (or GPGPU) for photon migration Monte Carlo simulations was introduced to the field of Biomedical Optics in the paper by Alerstam *et al.*[2]. There, a simple case of time-of-flight Monte Carlo simulations in a semi-infinite homogenously scattering media was studied and it was found that GPGPU could help accelerate the simulations by about three orders of magnitude. Although being of great importance in time-domain photon migration, this work covers only a very specific (simple) problem in biomedical optics. In order to demonstrate the use of GPUs for more general (complex) photon migration problems, we have now implemented MCML using CUDA. Doing so, we show that GPUs can speed up simulations even in the more complex case of multilayered structures. Since MCML is a de facto standard in photon migration modelling, we believe this is a very good illustration of the usefulness of GPGPU for photon migration Monte Carlo. We anticipate that GPU-based Monte Carlo will become an important tool in the field of Biomedical Optics. To promote this development, our code is publicly available at our webpage, and should be easy to understand and modify by a large group of scientists and students.

This document is divided into two major parts: The first part, 2. CUDAMCML for users, is simply a users manual for anyone interested in using CUDAMCML for standard MCML-simulations. Since MCML has its own manual [3], covering most aspects of photon migration Monte Carlo, this manual covers only the issues where CUDAMCML differs from MCML. This first part, covering installation and execution of CUDAMCML, will also describe some of the limitations of CUDAMCML, compared to traditional MCML induced by the execution on a GPU instead of a CPU. This part will also briefly describe the performance benefit of using CUDAMCML and show some validation results.

The second part, 3 CUDAMCML for programmers, is intended for the experienced GPGPU programmers who would like to understand the implementation details of CUDAMCML in order to be able to modify the program to perform other kinds of simulations. The content of this part is supported by three appendices, describing different technical aspects. As the source-code for CUDAMCML is supplied on our webpage, the best way to understand the implementation is of course to read the code, but this document will explain some of the GPGPU-specific implementation details and hopefully make the code easier to understand and modify. Once again, MCML specifics are not covered in this document as the MCML-manual covers all those issues [3].

The authors would like to clarify, that the current version of CUDAMCML is not the final version and several improvements over the current implementation can be expected in the future. With this said, use CUDAMCML at your own risk, and make sure to validate the results produced by CUDAMCML for your own applications.

2 CUDAMCML for users

2.1 What is CUDA and GPGPU?

General Purpose computing on Graphics Processing Units (GPGPU) is the concept of performing calculations on your graphics processing unit (GPU) instead of your CPU. The major benefit of doing so is that the GPU is much faster than the CPU when dealing with parallizable problems, i.e. problems that can be divided into many smaller and independent parts. Photon migration Monte Carlo is an excellent candidate for GPGPU and the user can expect several orders of magnitude faster simulations when performed on a GPU compared to a CPU.

CUDA, on the other hand, is nothing but a software and hardware architecture that allows users to perform GPGPU on Nvidias graphics cards.

2.2 Installation

First, to run CUDAMCML you need a computer with a modern Nvidia graphics card supporting CUDA, and CUDA compute capability 1.1. Such cards are most of the Nvidia GeForce 8xxx series and all better cards (including mobile versions) as well as all Tesla cards. For a specific list, see the Nvidia website: `http://www.nvidia.com`. Our software executable, as well as source code, can be found at our webpage:

`www.atomic.physics.lu.se/biophotonics/`

or directly at:

`www.atomic.physics.lu.se/fileadmin/atomfysik/biophotonics/software/CUDAMCML.zip`

2.2.1 Windows

Please unpack the zip-file in a directory of your choice. Make sure you have the latest Nvidia drivers installed. Currently the 180.xxx drivers are recommended. Drivers are available at the Nvidia website:

`http://www.nvidia.com`

Also, CUDAMCML requires a CUDA library contained in a file called `cudart.dll`. This dll file is distributed via the CUDA toolkit. However, if you don't have the CUDA toolkit installed, just unzip the file `cudart.zip`, included in the `CUDAMCML.zip`-file and make sure the file `cudart.dll` ends up in the same directory as the `CUDAMCML.exe`-file.

2.2.2 Linux

CUDAMCML includes a makefile which should make compiling straightforward on any Linux system with the CUDA toolkit and SDK installed in the default directories. CUDAMCML have so far been successfully tested on Ubuntu 8 and 9.

2.2.3 Mac OS X

CUDAMCML is written to be platform independent and should therefore run and compile without any problems on Mac OS X. An OS X compatible makefile is in the works for future releases.

2.3 Running CUDAMCML

Running CUDAMCML is very similar to running ordinary MCML, although a few more options is given the user. First of, the input and output files of CUDAMCML are fully compatible with the MCML standard with some very minor exceptions (see section 2.4).

To start CUDAMCML the user is required to supply the name of the MCML-input file as the first argument. For example if the name of the MCML-input file is `sample.mci`, CUDA MCML is started by the command:

```
> CUDAMCML sample.mci
```

CUDAMCML provides two optional arguments, `-S` and `-A`. The `-S` argument is related to the seed of the initial state of the random number generator used. If no seed is supplied by the user CUDAMCML will use the current time to seed the generator, similar to MCML. If the user would like to provide a seed, for example the seed 12 the command to start CUDAMCML would be:

```
> CUDAMCML sample.mci -S12
```

Any seed can be used, although the program will only read a 64-bit unsigned integer. Not all seeds are acceptable (see Appendix C and section C.1.4 in particular) and the program will quit if the seed is not valid. The `-A` argument is used if the used is not interested in the internal photon distribution (denoted the A-matrix in MCML). This will give a significant additional performance boost due to reasons related to GPGPU programming (see section 3.7). CUDAMCML will still give an ordinary output file, but the A-matrix and all related physical quantities will be set to zero. The Reflectance and Transmittance quantities will still be recorded in an ordinary fashion. See section 2.5 for more details on performance boost using the `-A` option. As an example, running the same simulation as above using the `-A` option, the command would be:

```
> CUDAMCML sample.mci -S12 -A
```

When running CUDAMCML, the feedback on the screen will be slightly different from MCML. The major reason for this is that simulations are much faster, and hence the time to completion does not have to be predicted with great accuracy. The program will often, about every second, report three numbers: The number of the current run, the number of photons terminated and the number of active threads. An example of this output is shown here:

```
Run 201, Number of photons terminated 98298780, Threads active 17920
Run 202, Number of photons terminated 98787964, Threads active 17920
Run 203, Number of photons terminated 99276397, Threads active 17920
Run 204, Number of photons terminated 99765265, Threads active 17920
Run 205, Number of photons terminated 99995514, Threads active 4486
Run 206, Number of photons terminated 100000000, Threads active 0
```

```
Simulation done!
Simulation time: 72.44 sec
```

Each tread simulates one photon propagating through the medium in an MCML style manner (Hop, Drop and Spin). Whenever a photon is terminated, a new photon is launched for that thread, assuming there are photons left to launch (so that the total number of simulated photons exactly add up to the specified number). In this case (code optimised for a Nvidia 8800GT card) 17920 (or 16128) threads are invoked on the GPU. During the last few runs, some threads are deactivated as there are no more photons to simulate. Each run comprise a predefined number of photon steps (currently 1000), where each step is a finite step of the photon, either to the next point where some weight is dropped and a new direction is determined or a step to a boundary, where it is determined of the photon is reflected or not. The reason for dividing the simulation into batches of 1000 steps are part to provide feedback to the user and part of technical nature, see section 3.2 for details. For the experienced MCML user, please note that a "run" in CUDAMCML is not the same as a "run" in MCML. The former is, as described above, 1000 steps for all threads, and the latter is an entire simulation.

2.4 Limitations

This section describes some limitations of the current CUDAMCML implementation, of interest to the ordinary user. Most of these are implementation specific and could be implemented rather easily (and probably will be in future versions). However some limitations are due to CUDA and/or GPGPU programming specifics and may be very hard to correct.

2.4.1 Deviations from MCML

CUDAMCML exhibits a few minor deviations from the standard MCML program:

- The binary output is currently not supported. CUDAMCML will simply ignore a request to have the output in binary instead of ASCII.
- CUDAMCML does currently not support having the first layer made out of glass. The program will still execute, but the specular reflection due to the first glass/tissue boundary will be reported as diffuse reflectance.
- The maximum number of photons per simulation is currently limited to $2^{32} - 1 \approx 4.294 \times 10^9$ (see section 3.7.3). We recommend to keep the number of photons per simulation at or below 4×10^9. Also, the number of photons cannot be fewer than the number of threads (currently fixed to 17920 or 16128).
- CUDAMCML does currently not perform all the testing of the input data that MCML does to make sure the data makes sense. Hence, it is up to the user to make sure the input data is actually correct.
- The maximum number of layers is currently set to 100 (including the non-optional above and below layers). For details, see section 3.3

- As previously stated, all calculation are performed using 32-bit floating point precision (single) as compared to the 64-bt precision (double) used by MCML. This may affect the results slightly. See section 2.6 for more details on validation.

2.4.2 Limitations related to CUDA/GPGPU

- Currently, CUDAMCML does not optimize the number of threads automatically. This will not have a major effect on the performance of plain CUDAMCML simulations, but if the `-A` option is used, the effect will be significant.
- CUDAMCML only supports one GPU and will not make an effort to select the most appropriate one if several are present.
- The simulation results are not guaranteed to be deterministic, even if the same seed is used. This is due to the fact that the order of execution of all the treads is undefined [4].

2.5 Performance

The major motivation of using CUDAMCML over the traditional CPU based implementation is of course speed. The exact performance of CUDAMCML over traditional MCML will depend on the problem definition and the GPU/computer used. As a rule of thumb, CUDAMCML is roughly two orders of magnitude faster when comparing a high-end GPU to a high end CPU or comparing low-end GPU's and CPU's and so on. As a second rule of thumb, using the `-A` option gives another order of magnitude of speedup.

In the comparisons below, a lower mid-end (∼$100) GPU (Nvidia 9800GT) running CUDAMCML was used and compared to a low-end CPU system and a high-end CPU system, both running MCML recompiled for the current system with maximum optimisation (which in itself gives a major performance boost over the distributed executables). The low end system comprised an Intel Pentium 4 HT 3.4 GHz CPU while the high-end system featured an Intel Core i7 2.66 GHz CPU. In the case of the multi-core high-end system a single core was used.

To illustrate the performance boost given by CUDAMCML over traditional MCML, two baseline simulations were defined and run

To exemplify the performance variations with respect to problem definition, two baseline simulations were performed using traditional MCML:

- *Simulation 1*: A simulation in a homogenous, very thick slab: $n = 1.4$, $\mu_a = 0.1$ [cm^{-1}], $\mu_s = 90$ [cm^{-1}], $g = 0.9$ and d=100 [cm]. The detection window was fixed to 5x2 cm while the grid resolution was varied.
- *Simulation 2*: A simulation in a slab made out of two different materials: Material 1: $n = 1.5$, $\mu_a = 0.1$ [cm^{-1}], $\mu_s = 90$ [cm^{-1}], $g = 0.9$. Material 2: $n = 1.2$, $\mu_a = 0.2$ [cm^{-1}], $\mu_s = 50$ [cm^{-1}], $g = 0.5$. Five layers of each material, alternating (Material 1, Material 2, Material 1 and so on), each layer 1 mm thick (total thickness 1 cm). The detection window was fixed to 5x1 cm while the grid resolution was varied

All simulations were performed with 10^7 photons. The results of simulation 1 and 2 performed on the two different CPUs and the GPU (with and without the `-A` option) are presented in Table 1 and 2 respectively.

dr, dz	MCML (Core i7)	MCML (P4)	CUDAMCML	CUDAMCML `-A`
0.01 cm	3736 s	7667 s (0.5x)	63.9 s (58x)	13.7 s (273x)
0.05 cm	3751 s	7614 s (0.5x)	31.9 s (117x)	13.5 s (276x)
0.1 cm	3781 s	7635 s (0.5x)	27.6 s (137x)	13.9 s (272x)

Table 1: The time in seconds to perform *Simulation 1* with different grid size resolutions for MCML (on the two different CPU's) and CUDAMCML, with and without the -A option. The number within the parenthesis shows the corresponding speed-up compared to traditional MCML on a single core of the Core i7 CPU.

dr, dz	MCML (Core i7)	MCML (P4)	CUDAMCML	CUDAMCML `-A`
0.01 cm	843 s	1724 s (0.5x)	15.2 s (55x)	6.9 s (122x)
0.05 cm	836 s	1746 s (0.5x)	13.4 s (62x)	6.7 s (124x)
0.1 cm	835 s	1751 s (0.5x)	13.6 s (61x)	6.7 s (124x)

Table 2: The time in seconds to perform *Simulation 2* with different grid size resolutions for MCML (on the two different CPU's) and CUDAMCML, with and without the -A option. The number within the parenthesis shows the corresponding speed-up compared to traditional MCML on a single core of the Core i7 CPU.

From the results presented in Table 1 and 1, several conclusions can be drawn regarding the performance of CUDAMCML vs. traditional MCML. In all simulations, it is obvious that traditional in MCML the simulation time is independent on the detection geometry, while CUDAMCML features strong dependence on the grid resolution in particular. This result is very important to the user, as the a coarser grid gives better performance. The reason for this is discussed in section 3.7.3. Similarly, a smaller detection window also gives better performance (results not shown here). *Simulation 2* illustrates that CUDAMCML gives a major performance boost, even in case of complex structures. The performance boost is, however, reduced due to the increased divergence among the threads. Looking at the results of using the `-A` option we can see that this gives a significant performance advantage. In simulation 2 the reason for the smaller speed-up is probably due to overhead, noticeable due to the short simulation time. The explanation for the huge performance gain when using the `-A` option is given in section 3.7. However, keep in mind that when using `-A` CUDAMCML does not record the absorbed photon weight inside the tissue which makes this comparison slightly unfair.

2.6 Validation

While validation for Monte Carlo code is very important, it is also a very difficult task to do validation for all cases of interest. At this stage, we choose to have a closer look at the two simulations performed in the previous section. We would like to emphasize that the current CUDAMCML version is development stage code. For critical applications, please perform validate your own results!

For validation we choose the method used by Lo et al. during validation of their FPGA-accelerated Monte Carlo model [5]. In good agreement with their results we found that the default pseudo-random number generator (PRNG) in MCML, `ran3` introduced odd biases in the internal photon absorption probability maps. Hence the PRNG in MCML was exchanged for a fast and high quality PRNG, the Multiply-With-Carry PRNG [7]. The MCML results with the new PRNG

were cross validated with results using several other high quality PRNG's, ensuring no biases were present due to the PRNG choice. This problem was the source of the minor deviations found during validation of the previous CUDAMCML code. Thanks to William Lo for suggesting that an inappropriate PRNG could give rise to problems like these! Throughout this document, all MCML results are results produced with the updated PRNG.

2.6.1 Simulation 1

Figure 1 illustrates the relative difference in the internal photon absorption probability. Looking

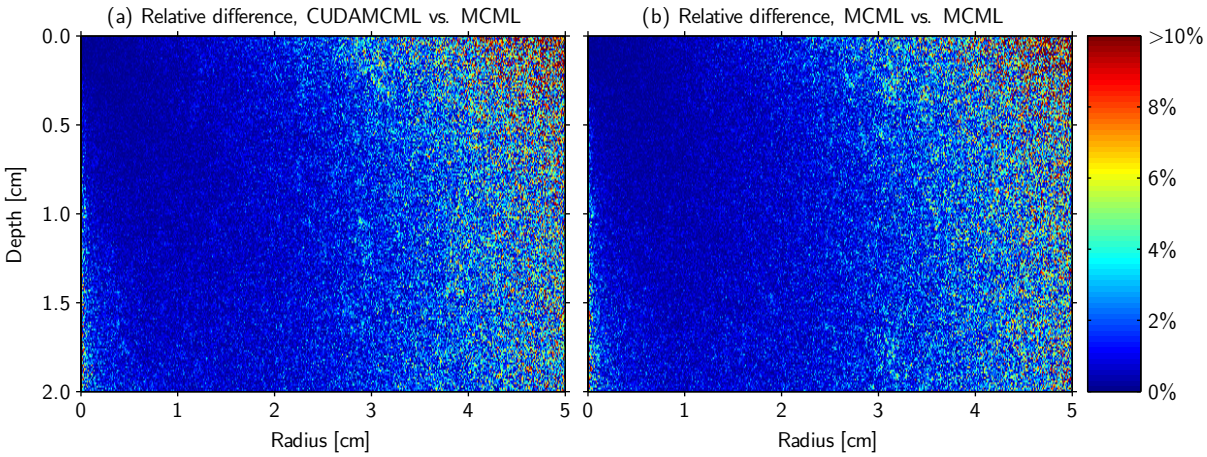

Figure 1: Illustrations of the relative difference in the internal photon absorption probability when comparing two simulations. *(a)* shows the relative difference between a CUDAMCML and a traditional MCML simulation. *(b)* shows the relative difference between two traditional MCML simulations. The simulation performed was *Simulation 1* as defined in the previous section. The colorbar illustrates the relative difference in percents [%]. If the relative difference was greater than 10%, the color is the same as for 10 %.

at figure 1, it is evident that the relative difference in CUDAMCML vs. MCML *(a)* is very similar to that of MCML vs. MCML *(b)*, suggesting that the results produced by CUDAMCML are statistically equivalent to those produced by traditional MCML, despite the lower (32- vs. 64-bit) numerical precision as well as the precision used during photon detection calculations (see section3.7). Similarly, comparing the other detected quantities, no significant deviations between CUDAMCML vs. MCML could be found.

2.6.2 Simulation 2

Figure 2 illustrates the relative difference in the internal photon absorption probability. Looking at figure 2, it is evident that the relative difference in CUDAMCML vs. MCML *(a)* is very similar to that of MCML vs. MCML *(b)*. Looking at the data carefully, extending the detection window to 5x2 cm, it is evident that a few detection events ("drop"'s) have occurred outside of the medium. The reason that this can happen is once again the limited precision used in the simulations, and the calculation of the coordinates of the detection in particular. However, the error is very small. The fraction of the weight detected outside the medium to the total detected weight is 2.2×10^{-9}.

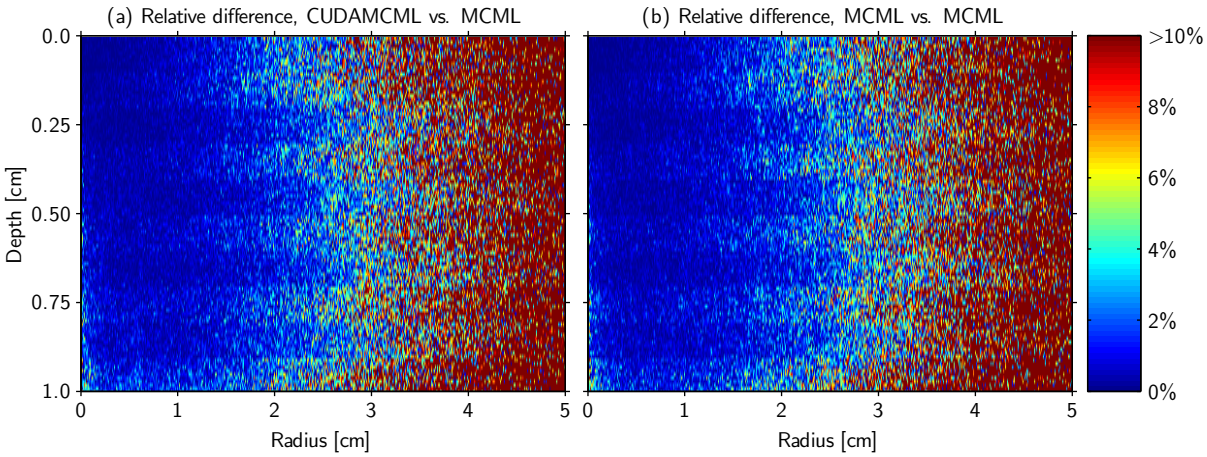

Figure 2: Illustrations of the relative difference in the internal photon absorption probability when comparing two simulations. *(a)* shows the relative difference between a CUDAMCML and a traditional MCML simulation. *(b)* shows the relative difference between two traditional MCML simulations. The simulation performed was *Simulation 1* as defined in the previous section. The colorbar illustrates the relative difference in percents [%]. If the relative difference was greater than 10%, the color is the same as for 10 %.

Hence the issue can be neglected.

Similar to the results of *Simulation 1*, no deviations in the other detected quantities could be found

3 CUDAMCML for programmers

This section will give details on the implementation of CUDAMCML and discuss issues related to the implentation of Monte Carlo simulations using GPGPU and CUDA in particular. This section is indeed intended for programmers, and in particular programmers with some experience of Nvidia's CUDA technology. Some sections will assume prior knowledge of CUDA and a thorough read of the CUDA programming manual [4] is highly recommended. This document will serve as a help for people interested in how to write efficient GPGPU Monte Carlo code, in particular for the specific problem of photon migration. It will also provide vital information and as a starting point for anyone interested in writing their own CUDA Monte Carlo software. As stated in section 4 modification and use of the provided code is encouraged.

3.1 Parallel programming using CUDA

Writing code for GPGPU, using CUDA is rather simple for the experienced C programmer. CUDA provides a minimal set of extensions to the C programming language, which together with the CUDA toolkit provide a simple, yet powerful, architecture for GPGPU. However, programming for parallel execution of code, as well as for an environment where the programmer is in charge of managing memory access, imposes some rules on how to write efficient code. While these rules and limitations are described in the CUDA programming manual [4], the implications of those rules for a specific problem are not always clear. This section aims to highlight some of the performance-limiting barriers one may run into when writing a GPGPU Monte Carlo simulation program.

3.1.1 CUDA computational considerations

Comparing GPU's and CPU's it's evident that GPU's can provide several orders of magnitude better raw computing power in terms of operations per second. To unlock this potential performance one has to keep several things in mind when designing a parallel algorithm. Here are a few of those computational considerations listed, as they are imperative to the understanding of the implementation:

- First of, current GPU's excel in 32-bit floating point computations only! Hence all integer- and double precision (64-bit) floating point calculations should be avoided whenever possible. (In fact 64-bit floating point precision is not even supported in many GPU's). The limited precision of 32-bit floating point representation may impose a problem in some calculations but in many cases these problem can be solved. To fully utilize the computational power of the GPU, CUDA provides a set of faster GPU-specific functions for some floating point operations. The price one has to pay for the increased performance is reduced computation accuracy, when using numbers in certain ranges. For example when doing a simple floating point division of two variables, `a` and `b`, this can be done either by `a/b` or by `__fdividef(a,b)`. The later is significantly faster and the two methods deliver the same result as long as `b`$< 2^{126}$.
- Keeping in mind that some operations are more costly (take longer time) than other operations, one should try to minimize the use of slow operations whenever possible. For example, division, either by `/` or `__fdividef()` is more costly than a simple multiplication. `/` takes 36 clock cycles, `__fdividef()` takes 20 clock cycles and a floating point multiplication takes 4 clock cycles. In many cases it is possible to exchange a division with a multiplication. For example when determining the step length, s, in an MCML-style simulation one has to perform

the calculation:

$$s = -\log(\xi)/\mu_t$$

where ξ is a random number between 0 and 1 and μ_t is the sum of the scattering- and the absorption coefficient. Instead of storing μ_t in memory and performing the costly division each step it is beneficial to store $\mu_{t,r} = 1/\mu_t$ and instead doing the calculation:

$$s = -\log(\xi) \times \mu_{t,r}$$

each step. This method can be applied to many calculations within the code, and the only cost is reduced clarity of the code.

- Thread divergence is a potential performance killer and should be avoided if possible. This means that the entire structure of the original MCML code is ill-suited for execution in a parallel environment. An excellent pedagogical example of this is the dual functions for taking a step in MCML, one for taking a step in tissue and one for taking a step in glass. Since the calculations in the two functions are basically the same, a common function for the two should be employed. All divergent functions, such as reflection calculations should be kept as brief and fast as possible and aim to return to the main execution path with minimal delay.

- To maximize the use of the GPU computing potential, one must make sure to efficiently use the available resources. The number one resource to keep track of is the number of registers the kernel uses. This can be done by analyzing the .cubin files produced by the compiler (using the `-keep`-flag) or simply by having the compiler print the used resources when compiling (using the `-Xptxas -v`-flag). The current implementation of CUDAMCML (64-bit Linux) uses 25 registers and a GPU of compute capability 1.1 has 8192 registers per multiprocessor. This means that 8192/25=327.7 threads can run simultaneously on a multiprocessor. Since the warp size is 32 threads, we set the block-size to 320 to maximize the use of the registers and hence have as many threads running as possible. For a compute capability 1.3 GPU, which has 16384 registers per multiprocessor, the best block-size would still be 320. However, in this case two blocks can run simultaneously instead of just one. According to the CUDA programming manual, the number of threads per block should always be 192 or more (and of course a multiple of 32). The number of blocks should be used to provide software scalability. As a guideline, it is often advantageous to use a multiple of the number of multiprocessors on the GPU and the number of blocks that can run simultaneously on each multiprocessor. Also, there are other things to consider, such as memory limitations, execution time of a single kernel call and the total number of threads. One should also keep in mind that the exact register useage may vary with platform. For example, CUDAMCML requires 25 registers when compiled on 64-bit Linux and requires 26 registers when compiled on 32-bit Windows XP. Hence the block-size on Windows should be 288, and the total number of treads $56 \times 288 = 16128$.

3.2 Program overview

The CUDAMCML source code was written with two objectives in mind. 1. To optimize performace and 2. to be clear and easy to read for anyone interested in modifying CUDAMCML

to do non-standard simulations. In contrast to MCML, which featured a flora of sub-functions, CUDAMCML instead has just a few, more general sub-functions. Keeping the functions general helps the interpretation of the code, and makes it much easier to modify, as there is little of no overlap in calculations between functions. Also, this approach is advantageous from a divergence-minimization point of view. This section will provide an overview of the code, and perhaps explain some of the GPGPU/CUDA specific implementation details. More info on the structures and the different sub-functions can be found in Appendix B and A respectively. This section is intended as a complement to reading the actual source code to help the user/programmer better understand the implementation details.

Simplistically, CUDAMCML is built up of three layers. The first layer is the `main()`-function, which will handle all the global, i.e. non-simulation specific tasks. This includes reading all the simulation data from the MCML-input file, an storing this as an array of `SimulationStruct`-structures, initializing and seeding all the RNG's (one per thread), allocating and freeing memory etc. The heart of the `main()`-function is a loop, which runs once per defined simulation. Inside this loop, the next layer is called, the `DoOneSimulation()`-function.

The `DoOneSimulation()` handles all the simulation-specific host tasks. The main argument of the `DoOneSimulation()`-function is a pointer to a single `SimulationStruct`-structure which defines the current simulation. `DoOneSimulation()` allocates memory structures in device memory and copies all the data from the `SimulationStruct` (and the RNG-data) to the structures in device memory (both global and constant memory space). When all data is copied, the first CUDA kernel is called, `LaunchPhoton_Global()`, which initializes all photons i.e. stores their initial state in global memory. At this stage, the device is fully set up to do the simulation.

The simulation is performed by running a loop calling the main kernel `MCd()`, each time. The `MCd()`-function is the third and last layer of CUDAMCML, and is executed on the GPU. The kernel, containing the main Monte Carlo transport loop (Hop-Drop-Spin) is very simple and should be self explanatory. Briefly, `MCd()` copies the simulation state (photon and RNG data) from global memory to Multiprocessor registers, performs 1000 photon steps for each tread and then copies the simulation state back to global memory. After each `MCd()`-call, the loop in the `DoOneSimulation()` makes a check, counting the number of active threads. When all treads are inactive i.e. they don't have more photons to simulate, the simulation is done and the loop is exited. After this, all the interesting data (the detection matrices and the RNG state) is copied back from the global memory and written to the simulation output file. All memory that was allocated earlier in the the `DoOneSimulation()`-function is freed and the program returns to the `main()`-function.

There are multiple aspects of the solution to call the `MCd()`-kernel multiple times, performing a limited number of photon steps each time. The main reason for this solution, is a Windows-feature, called the Watchdog timer. This timer makes sure no programs may use the GPU for more than approximately 5 seconds if Windows has the desktop extended to a monitor connected to a GPU, as such an execution time for most applications would indicate an error. Hence, the calculations has to be divided into smaller parts for most systems (in fact the watchdog timer sometimes also prevents longer execution time on secondary, non-utilized GPU's). Second, this give the program a chance to give updates to the user regarding the progress of the simulation. There may also be further advantages of this solution, that are currently being investigated and hopefully will be implemented in future CUDAMCML versions.

The downside of this solution is slightly increased simulation time. Since register memory only has the lifespan of the kernel, the entire photon state has to be copied to global memory (which

has the lifetime of the host program) in between executions of the `MCd()`-kernel. This copying data back and forth takes some time, in particular for non-coalesced memory operations. The RNG-state can be copied in two coalesced memory operations (one for `x` and one for `a`), but unfortunately the `PhotonStruct`-structure does not currently allow coalesced memory access as the structure contains mixed field (integers and floats). This will hopefully be fixed in future CUDA versions. However, the delay related to these memory operations is very small and can, along with the time to initiate a kernel, be neglected.

3.3 Memory structure

CUDAMCML uses three different memory spaces in its current form. The motivation for placing different types of information in those different memory spaces, as well as explanation of the advantages and disadvantages of using the different memory spaces are provided below. A more detailed explanation of the different structures used to group variables can be found in appendix B

3.3.1 Global device memory space

The global device memory is the only way to communicate information between the host and device (apart from the arguments to any device kernel, usually used to pass pointers to memory structures in global memory space). Also, as stated in section 3.2 the global device memory works as a non-volatile memory space, where device data can be stored in between kernel (`MCd()`) executions. Hence the the global memory must be set up to be able to store the entire simulation state. A summary of all the memory allocated in global memory space is available in appendix B.5. This section describes the `MemStruct`-structure, which is provided as only argument to the `MCd()`-function and contains pointers to all the arrays, matrices and scalars stored in global memory.

The advantage of the global memory space is that the amount of memory available is rather large (hundreds of Megabytes), and that the GPU-global memory bandwidth is very good, as long as the memory access is done in an ordered, or coalesced, fashion. CUDA devices of compute capability 1.1 or higher also provide the option of atomic memory operations on global memory which is a crucial feature for CUDAMCML (see section 3.6 and 3.7).

3.3.2 Constant device memory space

The constant memory space, is a small (64KB part of the device memory) part of the global device memory. While the host can both read and write to this memory space, it appears as read-only to any device code. The major feature of this small space is that the memory access is cached using a 8KB cache per multiprocessor. As long as the code can predict the memory access (make use of the cache) access to the constant memory space is virtually as fast as reading from a register. Since both the memory space and the cache, are small this memory space is suitable for all data that is shared among all the threads. In the current CUDAMCML implementation, this space is occupied by 4 scalars, one structure and one array of structures:

- `num_photons_dc` is a pointer to an unsigned int scalar stating the total number of photons to be simulated.
- `n_layers_dc` is a pointer to an unsigned int scalar stating the total number of layers in the simulation.

- `start_weight_dc` is a pointer to an unsigned int scalar stating the start weight of each photon i.e. the full photon weight, minus the specular reflection. See sections 3.7.3 and 3.7.5 for an explanation on why this is an unsigned integer instead of an expected float.
- `det_dc` is a pointer to a `DetStruct`-structure, containing all the data on the detection grid. See appendix B.2 for more details on this structure.
- `layers_dc` is a pointer to an array of `LayerStruct`'s. Since the constant memory does not allow dynamic allocation, the `layers_dc` array is preallocated to contain `MAX_LAYERS` number of layers, currently 100. More info on the `LayerStruct` can be found in appendix B.3.

Note that all pointers to the constant memory space uses the suffix `_dc` (device constant). This naming convention is used to clarify the code, since the pointers to the constant memory space are global pointers by default. All the above described data is copied from the host `SimulationStruct`. See appendix B.1 for more info on this structure.

3.3.3 Multiprocessor registers

The multiprocessor registers are the precious resources which determine how many threads each multiprocessor may have running simultaneously. Registers are by definition used when doing calculations, and this aspect of register usage cannot be directly influenced. One can however determine the amount of data actually stored in the registers. Storing data in the registers is advantageous as no memory operation has to be performed before executing any operations on the data. The data stored in the registers should hence be data that is used frequently, such as the RNG data (`a` and `x`), all the the photon data (`PhotonStruct`) etc. The lifetime of the register storage is however limited to the kernel execution time and to transfer data between kernel runs the data has to be copied to global memory space.

3.4 Handling layers

The MCML-way of handling layer transitions and photon transport in clear layers, is a very divergent solution, and hence not suitable for GPGPU. In CUDAMCML these calculations have been re-written to be minimally divergent and to minimize the use of resources (such as registers).

The `MCd()`-kernel threats all photon propagation steps in the same way, instead of using a flora of different sub-functions. The "Hop-Drop-Spin" procedure in `MCd()` is briefly explained below:

First the step length is randomized and stored in the variable `s`. In case the current layer is glass (no scattering) the step length is set to 100 cm. Next the new z-position ($z + dz \times s$) is checked, whether this new position causes a layer interaction. If that is the case, the steplength to reach the boundary is calculated and stored in `s`. Next the "Hop" is performed using the step `s`. If there was an layer interaction, the reflection calculations are performed using the `Reflect()`-function which also updates the current layer and the propagation direction of the photon. In case the photon is transmitted through the boundary checks are made if the photon exits the slab and the transmittance and reflectance are recorded accordingly. If the photons exit the slab, the remaining photon weight is set to 0, indicating the termination of the photon. Also, in case of a layer interaction, the step length, `s`, is set to 0 to indicate that there was a layer interaction. Next, a check is made whether there was a layer interaction or not, i.e. of `s` is greater than 0. In this case the "Drop" and "Spin" procedures are performed. This method has a few advantages but also

introduces two deviations from the MCML-way of doing Monte Carlo simulations. The advantages are:

- The code does not have to go into divergent branches when there are photon-boundary interactions.
- A photon-boundary interaction does not require a special flag to indicate, to the next loop iteration, that there was an interaction. The photon state is independent of the photon history, which is crucial for efficient use of the methods descried in section 3.2, where the photon simulation is divided into several kernel calls.
- Layers of clear material can be treated as any other material, as the next step will be a new layer interaction due to the very long step length in clear media.

Deviation from the MCML standard:

- This algorithm does not store the remaining step length after a boundary interaction as MCML does. However, as the photon propagation is independent of its history, this is valid from a photon propagation point of view. Hence the CUDAMCML way of handling is equivalent to the MCML way.

3.5 Massively parallel random number generation

The first problem one runs into when writing Monte Carlo code for parallel execution is the heart of all Monte Carlo based methods, the random number generation. If one uses the same (pseudo) random number generator RNG with the same seed, all parallel instances of the code will perform *exactly* the same calculations, voiding the advantage of the multiple of processing units. A naive solution to this problem would be to seed the RNG's for the different instances differently. This would appear as a solution to the ordinary user, as the different instances would provide results that were different. However, how would one make sure that the results are truly independent? One would have to put great effort into seeding the different instances with seeds that made sure that the sequences of random numbers didn't overlap during the simulations. This solution also requires an RNG algorithm with very long period to ensure this. For GPGPU, this solution has one major disadvantage (assuming we could somehow calculate the different seeds needed), RNG's with long periods often requires quite a bit of memory to store the state of the generator (see for example the Mersenne Prime Twister [6]). For GPGPU one would like to avoid having to use the slow global memory to store the RNG state and the number of registers per thread is limited. Hence we are limited to rather simple RNG's in order to avoid global memory-based solutions.

The solution used in CUDAMCML, is based on the lightweight and simple RNG by George Marsaglia called Multiply-With-Carry (MWC) [7]. The entire state of the MWC RNG can be stored in 64 bits (8 Bytes) or two 32-bit registers and it features a period of $> 2^{60}$. The main advantage is that the different instances of the RNG can be set up to use multipliers, a number defining the sequence of random numbers generated by the MWC algorithm. Hence the different RNG's will generate completely independent sequences of random numbers, with no risk of overlapping sequences. More details on the MWC RNG and the CUDA implementation can be found in Appendix C.

3.6 Simulating an exact number of photons

Simulating an exact number of photons is a trivial matter when doing sequential simulations as MCML does. For parallel environments, this is not a trivial problem since it requires communication between all the treads running on the GPU. The only medium provided by CUDA to perform this task is via global memory and by synchronizing all threads by dividing the computation into several kernel calls. While both mehods are used in CUDAMCML, only the first one is used for the task to simulate an exact number of photons. Whenever a photon is killed by the `PhotonSurvive()`-function a memory position in global memory space (`num_terminated_photons`, see Appendix B.5) is atomically incremented by one. If the resulting number is larger or equal to the total number of photons to be simulated (minus the active photons in all the other treads) no more photons should be launched. When this condition is reached the thread indicates that it is done by flagging its corresponding position in the `thread_active`-array with a 0 (as opposed to a 1, which means the thread is active). The thread also exits the main for-loop of `MCd()` by increasing the loop counter `ii` to `NUMSTEPS_GPU`. These two actions has two implications. 1. The next time the program returns to the host, and the number of active threads are counted, the main loop in `DoOneSimulation()` can tell if there are any active threads left. If there are active treads left, `MCd()` will be called again as usual. At this stage, we would like to avoid starting to simulate photons with the inactive thread again. This is ensured by having each thread check its corresponding position in the `thread_active`-array. If the thread is flagged as inactive, the for-loop counter is once again set to `NUM_STEPS_GPU`, making sure the thread skips the main loop.

This solution ensures that an exact number of photons are simulated. However, the solution has two side-effects. First, it's very inefficient for the last photons, as the warps become less and less populated by active threads, instead of sorting the threads to ensure full warps. The second, and perhaps more severe effect is that the possibility of deterministic simulations is lost. Since the order of execution of the different threads (or warps) is undefined, and the solution relies on an atomic update of a position in global memory it is not guaranteed that the same thread will get to simulate the same number of photons, even if the sequence of random numbers for all threads are exactly the same.

3.7 Photon detection

3.7.1 Basic problem

The problem of efficiently implement a parallel code for photon detection calculations is the major bottleneck in CUDAMCML. The problem of generating the map of the internal photon absorption probability distribution is unfortunately a fundamental problem in GPGPU computing. The problem can be seen as a problem of generating a weighted histogram, where each individual photon step generates a histogram index and a weight to be added to the histogram bin corresponding to the index position. Looking at current solutions for simple histogram generation [8, 9] it is evident that current parallel histogram generation algorithms are limited to histograms with a very small number of bins ($\leq$256 bins) for superior performance. In a typical MCML simulation the A_{rz} matrix is sized hundeds times hundreds of elements, i.e. tens or hundreds of thousands of bins.

3.7.2 Current solution

The current solution to the photon detection problem is described in the section **??** below. It is an improved scheme to update the dection matrices (`Rd_ra`, `A_rz` and `Tt_ra`) in a way that is

safe in a massively parallel environment. It replaces the naive solution to atomically update the detection matrices every step of each photon, which understandably yields very poor performance, as the bandwidth for atomic memory access is very poor. Using the naive solution, the photon detection of the `A_rz` matrix alone takes up to 10x the time of the rest of the simulation, i.e. all the calculations and the detection of reflected and transmitted photons. This is the motivation for providing the `-A` option when running CUDAMCML.

An alternative solution that was tested was to write the index and weight data coalesced to global memory and then passing all this data to the CPU, to do the histogram calculation while the GPU did the simulation calculations. This solution would work very well for a simple GPU as a standard CPU would be able to handle the stream of data. The major problem with this solution is scalability. Such a solution would once again be limited by the histogram generation when using a higher end GPU (or several GPU's). Also, the PCI-express buss bandwidth would be a limiting factor as a moderately advanced GPU can generate data much faster than the PCI buss can transfer this data to a host.

3.7.3 Atomic float operations

The solution of using atomics has several problems and implications itself. First, none of the current generation Nvidia GPU's support the necessary atomic floating point operations natively. An option is to emulate atomic floating point add using atomic Exchange and Compare-and-Swap functions. This is an inefficient solution, as it requires more than one atomic operation per emulated atomic floating point operation. Instead, a solution where the weight is stored as unsigned integers have been used. The conversion to unsigned integers from float has a drawback in itself. If the initial weight of the photon is set to 0xFFFFFFFF (the largest number representable using a 32-bit unsigned integer), adding weight from millions of photons, will cause some bins to overflow as the deposited weight in the bin may exceed the weight of a single photon. Reducing the starting weight of a photon is not a good solution to this problem, as it reduces the precision of the calculations. The solution is to store the bins as 64-bit unsigned integers, and limiting the number of photons to 2^{32}-1. Using this solution all photons can deposit their entire initial weight (0xFFFFFFFF) in the same bin without the risk of overflow. However, not all GPU's support 64-bit atomic operations. The good news is that atomic 64-bit unsigned integer add can be emulated rather efficiently. See appendix A.2.6 for implementation details.

3.7.4 Fast memory caching

A rather straightforward way of speeding up the photon detection (weighted histogram) problem is to cache the most used part of the detection matrix(/histogram) in a faster (but smaller) memory space, such as the the shared memory space, in a per thread or per block fashion. The solution of per block caching, however, requires atomic operations on shared memory, a feature currently only available in GP's with CUDA Compute Capability 1.3 or better. Hence, to keep compatibility with the most common GPU's on the market, and to leave the shared memory free for the user to use for custom applications, this scheme is not used in CUDAMCML, albeit it's significant speedup potential. However, using a per-tread single bin caching scheme is possible at the cost of just one register per thread and a slight loss in code clarity. This is potentially very beneficial as each tread often will write to the same bin many steps in a row, especially if the grid resolution is not to fine. The algorithm simplistically goes like this: Take a step and calculate the index and weight to be dropped. Compare the index to the cached index. If they are equal increase the cached weight by

the calculated weight. If they are not equal deposit the cached weight atomically into the global memory bin at the position corresponding to the cached index and store the calculated index and weight in the corresponding caches. Also, after each run (1000 photon steps), deposit the cached weight into global memory so that the cache state does not have to be transferred back and forth to the global memory in between runs.

3.7.5 Implication of the current solution

The solution to the photon weight detection problem has several very important implications:

- The performance of the current solution is not very good. The problem lies with the nature of GPGPU, i.e. the poor performance of random memory access, and with the fact that all threads may want to write data to the same position in memory at the same time, forcing us to use atomic memory write functions. Currently the photon detection alone takes the same time (or more than) the time of the actual Monte Carlo simulation calculations, as shown in section 2.5, depending on the grid coarseness.
- Due to the single-bin caching, the photon weight detection solution is strongly dependent on the detection grid size and resolution. As seen in section 2.5 the performance of CUDAMCML increases with a coarser detection grid as well as a smaller detection window.
- The use of 32-bit unsigned integers for photon weight and 64-bit integers for the detection matrices can be seen throughout the source code of CUDAMCML. One of the most striking differences from MCML when using this approach, is the definition of `WEIGHT`, which in MCML typically was set to 0.0001. In CUDAMCML, this define is renamed to `WEIGHTI` to indicate the (unsigned) integer value. The value of `WEIGHTI` is set to 0.0001×0xFFFFFFFF=429497u

4 Licence

CUDAMCML is released as open source software and is licensed using the GNU General Pubic Licence version 3, a free software licence. Briefly, this gives the user the freedom to use the program and code for any purpose and to change the program to suit their needs. It also gives the user the freedom to share the software with others and to share modified versions of the software to others. It also states than there is no warranty for the software. Please see the file `copying.txt`, distributed along with CUDAMCML for the full licence.

In general we encourage the use, and modification of this code, and hope it will help users/ programmers to utilize the power of GPGPU for their simulation needs. In order to rapidly and efficiently communicate and spread our work on GPU Monte Carlo, we have chosen to publish our work on our web page rather than in a scientific journal. We would, however, appreciate if you cite our original GPGPU Monte Carlo Letter if you use this code or derivations thereof for your own scientific work:
E. Alerstam, T. Svensson and S. Andersson-Engels, "Parallel computing with graphics processing units for high-speed Monte Carlo simulations of photon migration", Journal of Biomedical Optics Letters, **13**(6) 060504 (2008).

A CUDAMCML functions

This appendix describes the function of many of the functions used in CUDAMCML. All the device- and global device functions are covered. The appendix intends to briefly explain the functions, and in the case of the device functions, highlight any deviations from their MCML equivalents.

A.1 Global device functions

Global device functions, or *kernels*, are functions executed on the device, callable from the host code. CUDAMCML uses two such global device functions. One to initiate all photons in a simuation, and one to perform a predefined number of step of photon migration Monte Carlo propagation.

A.1.1 LaunchPhoton_Global()

Definition: `__global__ void LaunchPhoton_Global(MemStruct DeviceMem)`

This function is used to initiate all photons in the simulation, prior to starting the simulation using the `MCd()`-function. The function simply creates a `PhotonStruct` per thread (in registers) and calls the device function `LaunchPhoton()` to launch each photon (done in parallel). This has two advantages. It eliminates the need for a divergent branch in the `MCd()`-function, checking if the photons have been launched or not. Also, it makes sure all photon launching is done using the `Launch_Photon()` device function. Hence, if one would like to modify the way photons are launched, this function would be the only place where one has to apply changes.

A.1.2 MCd()

Definition: `__global__ void MCd(MemStruct DeviceMem)`

This is the main kernel, doing all the Monte Carlo calculations. It only takes a single argument, a `MemStruct`-structure containing pointers to all the data of interest in the global device memory. A brief overview of the `MCd()`-function and it's role in CUDAMCML is given in the Program Overview, section 3.2.

An important aspect of the `MCd()`-function is the use of templates to implement a separate kernel for simulations with the `-A` option. The if-statement involving the variables from the template will be evaluated at compile-time and consequently generate two separate kernels. Since the kernel ignoring the internal photon absorption probability distribution detection is simpler, the register usage is slightly lower. Also, both version benefit from this solution in terms of pure speed.

A.2 Device functions

A.2.1 rand_MWC_oc() and rand_MWC_co()

Definition: `__device__ float rand_MWC_oc(unsigned long long* x, unsigned int* a)`
Definition: `__device__ float rand_MWC_co(unsigned long long* x, unsigned int* a)`

These two functions are used to fetch (pseudo) random numbers from the MWC random number generator. The `_oc`-version provides a random number in (0,1] and the `_co`-version in [0,1). See Appendix C for more details on the MWC RNG and implementation specifics.

A.2.2 LaunchPhoton()

Definition: `__device__ void LaunchPhoton(PhotonStruct* p,`
`unsigned long long* x,`
`unsigned int* a)`

This function is used to set the initial state of a photon (by setting all the fields of the corresponding `PhotonStruct` to suitable values). The function uses the constant device memory variable `start_weight_dc` to set the initial weight of the photon, taking the specular reflection into account. The two variables used for the RNG are provided as arguments for future applications where the initial photon state is in any way randomized.

A.2.3 Spin()

Definition: `__device__ void Spin(PhotonStruct* p,`
`float g,`
`unsigned long long* x,`
`unsigned int* a)`

This function calculates a new direction for the photon, based on the anisotropy-factor `g`, provided as an argument. The function is very similar to the MCML equivalent with one minor, but potentially important addition. After the spin calculation, the direction is normalized. Since 32-bit floats has worse precision than the double precision floats used in MCML, the numerical accuracy in all calculations are reduced. When performing several operation on a set of floats, the numerical error will accumulate and after a while the length of a previously normalized vector will differ from unity. This problem isn't to bad for ordinary MCML simulations, but for future applications this might be an issue and hence the normalization is provided.

A.2.4 Reflect()

Definition: `__device__ unsigned int Reflect(PhotonStruct* p,`
`int new_layer,`
`unsigned long long* x,`
`unsigned int* a)`

This function calculates whether the photon is reflected or not. The `PhotonStruct`-pointer `p` contains the information on the current layer index (the field `layer`) and the argument `new_layer` carries the index of the second layer. The function is very similar to it's MCML equivalent but the actual calculations have been rewritten. This, less clear, implementation of the calculations is slightly faster and requires fewer registers. The function returns 1 if the photon is reflected and 0 if it is transmitted. Regardless, `p` is updated with the new photon direction and `layer`.

A.2.5 PhotonSurvive()

Definition: `__device__ unsigned int PhotonSurvive(PhotonStruct* p,`
`unsigned long long* x,`
`unsigned int* a)`

This function performs a check whether the photon survives or not. If the photon weight is larger than the defined value `WEIGHTI`, no roulette is needed and the function returns 1. On the other hand, if the weight is equal to 0, his indicates that the photon has exited the medium, meaning that the photon should be killed and the function hence returns 0. If the weight is smaller than `WEIGHTI` a roulette is performed, in the same way as described for MCML. If the photon survives the roulette, the function updates the photon weight accordingly and returns 1, else the function returns 0. To do this roulette the function relies on the the defined value `CHANCE`.

The function will handle the update of the photon weight if the photon s

A.2.6 AtomicAddULL()

Definition: `__device__ void AtomicAddULL(unsigned long long* address, unsigned int add)`

This function will emulate 64-bit unsigned integer atomic add on compute capability 1.1 devices. It will add the 32-bit unsigned integer `add` to the 64 bit unsigned integer located at `address` atomically.The need for this function is described in section 3.7

The function is implemented as:

```
if(atomicAdd((unsigned int*)address,add)+add<add)
   atomicAdd(((unsigned int*)address)+1,1u);
```

It works, by first adding `add` atomically to the least significant half of the 64-bit unsigned integer at position `address`. The atomicAdd function returns the value at the address before the add operation, lets call this value `old`. If `old`+`add` is smaller than `add` we know that there was an overflow in the addition and hence we should increment the most significant half of he 64-bit unsigned integer by one.

B Structures

B.1 SimulationStruct

The basic structure, containing all the data needed to define a single simulation is the `SimulationStruct`:

```
typedef struct
{
 unsigned long number_of_photons;
 int ignoreAdetection;
 unsigned int n_layers;
 unsigned int start_weight;
 char outp_filename[STR_LEN];
 char inp_filename[STR_LEN];
 long begin,end;
 char AorB;
 DetStruct det;
 LayerStruct* layers;
}SimulationStruct;
```

The program `main()`-function keeps an array of `SimulationStructs` and passes them one at a time to `DoOneSimulation()`, which in turn makes sure that the defined simulation is performed. The fields of a `SimulationStruct` are rather self explanatory.

- `ignoreAdetection` is a flag, describing whether or not ignore the detection of the internal photon absorption distribution, as defined by the `-A` option when running `CUDAMCML.exe` (see section 2.3). 1 means CUDAMCML will igonore the detection of the internal photon absorption distribution, i.e. the `-A` flag was provided and any other value (0) means the CUDAMCML execution will be equivalent to traditional MCML.
- `start_weight` describes the start weight of each photon (Total photon weight minus the specular reflection). This field is defined as an unsigned integer, not a float as perhaps expected. The explanation for this can be found in section 3.7.
- The field, `n_layers` is descibed below, together with the des
- The structure also contains one `DetStruct`-structure and a `LayerStruct`-pointer. Both these structures are described below.

B.2 DetStruct

The `DetStruct`-structure is a structure gathering all the data related to the photon detection grid. It is defined as:

```
typedef struct __align__(16)
{
float dr; // Detection grid resolution, r-direction [cm]
float dz; // Detection grid resolution, z-direction [cm]
int na; // Number of grid elements in angular-direction
```

```
int nr; // Number of grid elements in r-direction
int nz; // Number of grid elements in z-direction
}DetStruct;
```

The `DetStruct` is present both in host and device memory. The fields should be self explanatory to anyone familiar with traditional MCML. Since the data contained by the structure is common for all thread, the `DetStruct` is later contained in device constant memory as `det_dc` (pointer).

B.3 LayerStruct

The `LayerStruct` is used to contain all the data to describe a single layer:

```
typedef struct __align__(16)
{
float z_min; // Layer z_min [cm]
float z_max; // Layer z_max [cm]
float mutr; // Reciprocal mu_total [cm]
float mua; // Absorption coefficient [1/cm]
float g; // Anisotropy factor [-]
float n; // Refractive index [-]
}LayerStruct;
```

The fields `z_min` and `z_max` are expressed in absolute coordinates, i.e. the absolute depth at which the layer begins and ends. The total attenuation coefficient $\mu_t = \mu_s + \mu_a$ is stored as the reciprocal value $\mu_{t,r} = 1/\mu_t$. The other field are self explanatory.

The entire simulation geometry is stored as an array of `LayerStruct`'s, The first element (structure) in this array contains the data for the refractive index of the above layer only! Similarly, the last layer contains only the refractive index of the layer below, both as defined in the MCML-input file. The other fields are undefined. The `SimulationStruct`-structure provides a pointer to the array of `Layestruct`'s The number of elements in the array is given by the `SimulationStruct`-field `n_layers`. Both the `LayerStruct` array and the number of layers are information shared by all the threads and are therefore copied to the constant device memory space. The corresponding pointers are `layers_dc` and `n_layers_dc` respectively. Since dynamic allocation of constant memory is currently not allowed in CUDA, the `LayerStruct`-array is pre-defined as an array of `MAX_LAYERS` elements, where `MAX_LAYERS` is defined in CUDAMCML.h, currently as 100.

B.4 PhotonStruct

The `PhotonStruct`-structure contains all the data describing the state of a single photon i.e. the position, (normalized) direction, weight and current layer:

```
typedef struct __align__(16)
{
 float x; // Global x coordinate [cm]
 float y; // Global y coordinate [cm]
 float z; // Global z coordinate [cm]
 float dx; // (Global, normalized) x-direction
 float dy; // (Global, normalized) y-direction
```

```
 float dz; // (Global, normalized) z-direction
 unsigned int weight; // Photon weight
 int layer; // Current layer
}PhotonStruct;
```

An array of `PhotonStruct`'s is allocated in global device memory. The array has one element per thread The number of threads is defined in CUDAMCML.h as `NUM_THREADS`, (currently 17920 or 16128). When running the main Monte Carlo photon transport function `MCd()`, each thread will copy its corresponding `PhotonStruct` to its registers since a the fields of the structure are used frequently. In the end of the `MCd()`-function each tread will copy its `PhotonStruct`-data back to global memory. This means that the entire state of the simulation (of all threads) is stored in between calls of the `MCd()`-function. Hence, a long simulation can, with little penalty, be split up into several calls of the `MCd()`-function. The advantage of this feature is described in section 3.2 Note that the photon weight is stored as an unsigned integer. This is explained in section 3.7.

B.5 MemStruct

The `MemStruct`-structure is used to store all the pointers to global memory. The this structure is used as the only argument to the `MCd()`-function and new pointers can thus easily be added without major changes to the code.

```
typedef struct
{
PhotonStruct* p;
unsigned long long* x;
unsigned int* a;
unsigned int* thread_active;
unsigned int* num_terminated_photons;
unsigned long long* Rd_ra;
unsigned long long* A_rz;
unsigned long long* Tt_ra;
}MemStruct;
```

- The pointer `p` points to the array of `PhotonStruct`'s. The array-lenght is the same as the number of threads, `NUM_THREADS` so that each thread can store it's photon state in global memory.
- The pointers `x` and `a` points to arrays of RNG states and multipliers respectively. Since both are thread specific, both arrays are of length `NUM_THREADS`.
- `thread_active` points to an array of length `NUM_THREADS`, keeping track of whether a specific thread is active or not. For details, see section 3.6.
- `num_terminated_photons` is a pointer to a scalar unsigned int in global memory, keeping track of the total number of terminated photons at any given time. As the above field, this is related to the issue of simulating an exact number of photons, see section 3.6.
- The three pointers `Rd_ra`, `A_rz` and `Tt_ra` points to the three detection matrices, used to store reflected, absorbed and transmitted photon weights. The names were chosen in accordance with the MCML-standard. More information on this topic is available in section 3.7

C Multiply-With-Carry Random Number Generator

This appendix describes details on the (pseudo) random number generator (RNG) used in CUDAMCML, namely the Multiply-With-Carry (MWC) RNG by George Marsaglia. Marsaglia provides an excellent overview of MWC and related RNG's in [7]. A short description of the problem of massively parallel random number generation is given in section 3.5. A basic implementation of MWC for CUDA was first provided by Steven Gratton [10] and parts of the code presented here is based on his work, in particular the idea of how to seed all the generators.

The file mwc1.ps of [11] gives an early description of the MWC RNG and explains why it's particularly suitable for computer implementation.

C.1 MWC basics

The WMC pseudo random number generator is very simple. Mathematically it can be written as:

$$x_{n+1} = (x_n \times a + c) \text{mod } b,$$

where x is the sequence of random numbers, a is a multiplier, c is the carry from the previous modulus (mod) operation and b is the base for the mod operation. The sequence of random numbers is determined by the multiplier used, hence if one uses different multipliers for each thread in a parallel environment (such as CUDA) each thread will have access to a unique and independent sequence.

C.1.1 Base

On modern binary computers it is of course advantageous to use a base $b = 2^p$, where p is a positive integer. In this case the mod operation can be executed by a simple bitwise-AND operation. In practical applications one should use a base such as $b = 2^{16}$ or $b = 2^{32}$. Thinking about CUDA applications one might for example try $b = 2^{24}$ for faster integer multiplications, although the period of the generator will be shortened (see the next section).

C.1.2 Multiplier

Some restrictions apply to the multiplier a for the theory behind MWC to apply. A good multiplier should be one where $a \times b - 1$ is a safeprime, that is both $a \times b - 1$ and $(a \times b - 2)/2$ are prime (a number q is a safeprime when both q and $(q - 1)/2$ are prime). The choice of the multiplier and base determines the period of the generator; Using a suitable multiplier, the period of a MWC RNG is $a \times b - 2$ [7]. For a base $b = 2^{32}$ and multipliers a just slightly smaller than 2^{32} the period is close to 2^{64}.

C.1.3 Finding suitable multipliers

Using a base $b = 2^{32}$ and limiting ourself to multipliers $a < 2^{32}$, one must be able to test numbers $q < 2^{64}$ for primality to be able to select suitable multipliers. To create a list of suitable multipliers (remember, we need as many multipliers as we have threads in CUDA), one can use a package such as the GNU MP Bignum Library[12] designed to handle arbitrary precision arithmetics. It also has a fast implementation of the probabilistic MillerRabin primality test for large numbers. Using GMP, writing a program to find suitable multipliers is very easy.

While the Miller-Rabin primality test is probabilistic certain measures can be taken to make sure the primes found are true primes. [13] (http://www.trnicely.net/misc/mpzspsp.html) provides an analysis of the pseudo-prime problem of the Miller-Rabin algorithm, and the GNU GMP implementation in particular. It is shown that for numbers representable by a 64-bit unsigned integer ($< 2^{64}$) the problem of pseudoprimes can be completely eliminated using a Miller-Rabin test of order 13 or greater, that is, use the bases 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 and 41.

The Miller-Rabin test was used to find 150000 good multipliers $a < 2^{32}$ (base $b = 2^{32}$). The results were stored in a file, `safeprimes_base32.txt` in ASCII format, provided along with the CUDAMCML program. Each line of this file contains three numbers; first the multiplier a followed by the two primes, $a \times b - 1$ and $(a \times b - 2)/2$. The reason for this format, is compatibility with the list (and WMC implementation) by Steven Gratton [10]. Gratton also provides a list of suitable multipliers, but this list only contains 16028 entries which are exactly the 16028 first entries in `safeprimes_base32.txt`. While this makes the list file unnecessarily large, this will allow anyone to verify the multipliers and their corresponding primes.

The program for finding the multipliers is provided below (Assuming the GNU GMP libraries are properly installed):

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "gmp.h"

int main()
{
  FILE* file;
  mpz_t n1,n2,a;
  unsigned long long i=0;
  mpz_init(n1);
  mpz_init(n2);
  mpz_init(a);

  mpz_set_str(a,"4294967118",0);

  file = fopen("safeprimes_base32.txt","w");

  while(i<150000)
  {
    mpz_mul_2exp(n2,a,32);
    mpz_sub_ui(n2,n2,1lu);
    if(isprime(n2))
{
      mpz_sub_ui(n1,n2,1lu);
      mpz_fdiv_q_2exp(n1,n1,1lu);
      if(isprime(n1))
      {
        //We have found our safeprime, calculate a and print to file
        mpz_out_str (file, 10, a);
```

```
        fprintf(file," ");
        mpz_out_str (file, 10, n2);
        fprintf(file," ");
        mpz_out_str (file, 10, n1);
        fprintf(file,"\n");
        printf("%d\n",i++);
      }
    }
  mpz_sub_ui(a,a,1lu);
  }
  fclose(file);
  exit (0);
}


int isprime(mpz_t n)
{
  int i;
  int class;
  int alist[]= {2,3,5,7,11,13,17,19,23,29,31,37,41};
  for(i=0; i<13; i++)
  {
    class = mpz_probab_prime_p (n, alist[i]);
    if(class==0) return 0;
  }
  return 1;
}
```

C.1.4 Seeding the generator

While giving each thread on the GPU a unique multiplier to make sure all the threads experience unique sequences of pseudo random numbers, one still has to seed all those parallel generators properly, to enure that two consecutive simulations will not give the exact same results. Seeding the MWC RNG is done by setting initial values for x and c. For the MWC theory to be valid, some restrictions apply to the seeds: any pair $[x, c]$ is valid as long as $0 \leq c < a$, $0 \leq x < b$, excluding $[0, 0]$ and $[b-1, a-1]$. Since manually seeding thousands of generators with valid seeds is a tedious task, a trick is applied to generate proper seeds. As described in section 2.3, the CUDAMCML user is given the option to provide a 64-bit seed, via the `-S` option, or a 32-bit seed from the C `clock()`-function (providing the current time) is used. This initial seed is used to seed a single MWC RNG (using the first multiplier in the list provided in `safeprimes_base32.txt`). This the sequence of random numbers from this RNG is in turn used to provide valid seeds for all the other MWC generators.

C.2 Implementation

C.2.1 Setting up the generators

Setting up the generators is started by declaring two arrays of the same length as the number of threads used in CUDA. The first one, `x`, is ana array of 64-bit unsigned integers and the second one, `a`, an array of 32-bit unsigned integers. Each of the 64-bit unsigned integers of `x` is set up so that the upper 32-bit house the 32-bit unsigned integer representing the carry, c, and the lower 32-bits represent the random number, x. The `a`-array is filled with multipliers from the `safeprimes_base32.txt` file and the `x`-array is filled with suitable c's and x's for each corresponding multiplier (as described in C.1.4).

Prior to each simulation performed in CUDAMCML, the `a` and `x` arrays are copied to global device memory. Each kernel tread then copies the corresponding multiplier and RNG state (`x`) each kernel execution. At the end of each kernel execution, the RNG state is copied back to global memory to store the current state of the RNG. When all kernel executions are done, the `x`-array is copied back to host memory to store the state of all the RNG's. Hence, if CUDAMCML was run with an input file defining multiple simulations of the exact same problem, the results of the individual simulations would not be exactly the same.

C.2.2 CUDA MWC implementation

Below is the current CUDA implementation of MWC. For good performance the two pointers, `a` and `x`, are assumed to point at multiprocessor registers containing the multiplier and RNG state respectively.

```
__device__ float rand_MWC_co(unsigned long long* x,unsigned int* a)
{

//Generate a random number [0,1)
x=(*x&0xffffffffull)*(*a)+(*x>>32);
return __fdividef(__uint2float_rz((unsigned int)(*x)),(float)0x100000000);

}//end __device__ rand_MWC_co
```

This function requires some explanation. First, we remember that the actual random number, x is housed in the lower 32 bits of the 64-bit variable `x` as a 32-bit unsigned integer. Hence one can access x, either by typecasting `(unsigned integer)*x` (the `*` is there as `x` is a pointer) or by doing a bitwise AND operation with a 64-bit number containing 32 zeros followed by 32 ones: `*x&0xffffffffull`. Likewise, c is stored in the upper 32 bits and can be accessed by `*x>>32`. Hence, performing the operation `x=(*x&0xffffffffull)*(*a)+(*x>>32);` updates both x and c. Now, the sought after random number, x is an unsigned integer uniformly distributed in $[0, 2^{32}-1]$. To convert this to a floating point number (32-bit) $[0, 1)$ we perform a floating point division: $x/2^{32}$. While this is mathematically correct, it is not numerically correct. This, since the typecast `(float)((unsigned int)*x)` would yield 2^{32} when $x = 2^{32} - 1$ due to the limited numerical accuracy of the 32 bit floating point standard for large exponents. Hence the built in typecasting function `__uint2float_rz()` is used to ensure towards-zero rounding, i.e. $x = 2^{32} - 1$ will be rounded down to the largest number $< 2^{32}$ that the 32 bit floating point standard allows.

The reason for this entire manoeuver is the possibility to have access to random numbers, both $[0, 1)$ and $(0, 1]$ (the latter, simply by taking 1-the former). The $(0, 1]$ interval is of particular interest when using operations taking the log of a random number, to avoid undefined answers.

D How to compile

Linux

A simple makefile is provided which should work, provided that the CUDA Toolkit and CUDA SDK is installed correctly. This has only been tested on Ubuntu 9.04 (64-bit) with CUDA 2.1.

Mac OS X

No support yet. A makefile will hopefully be provided in the future.

Windows

I recommend Microsoft Visual Studio 2005 Express (2008 should work as well but is untested) available for free on http://www.microsoft.com/express/download/. For VS2005 I recommend the CUDA VS Wizard (http://cudavswizard.sourceforge.net/) to create a VS project. If you create your own project, remember that CUDAMCML requires the 1.1 Architecture for the atomics-functions. Hence, pass `-arch=sm_11` to the compiler.
Alternatively the Visual Studio Command Prompt can be used. Go to the CUDAMCML folder and type (as a single line):
`nvcc -Xptxas -v ./src/CUDAMCMLmain.cu -I "C:\Documents and Settings\All Users\Application Data\NVIDIA Corporation\NVIDIA CUDA SDK\common\inc" -arch=sm_11 -O3 -o CUDAMCML.exe` Once again, this assumes that the CUDA Toolkit and SDK has been successfully installed in the default locations.
CUDAMCML has been successfully compiled and used on Windows XP (32-bit) with CUDA 1.1-2.2 using Visual Studio 2005 Express.

E Release Notes

July 6, 2009

Bugfixes:

- Output filenames are now allowed to start with numbers, for example: `1.mco`, Thanks to William Lo for finding this bug.
- The call to the sincos()-function in the Reflect()-function is now correct. Thanks to Terence Leung for finding this bug.
- Hopefully fixed a bug where the simulation would stall when using many ($> 2 \times 10^9$) photon packets. Thanks to Can Xu for noting this bug.

New CUDAMCML features:

- Better performance thanks to the per-tread single bin caching.
- Changed the `-A` implementation to a more elegant template-solution.
- Added a makefile for Linux.

New document features:

- Solved the minor deviation in the validation.
- Updated performance comparison with optimised MCML on both a low-end (P4) CPU and a high-end (Core i7) CPU.
- Added the Release Notes Appendix
- Added a brief description on how to compile CUDAMCML on differnt platforms

April 3, 2009

Original release.

References

[1] L.H. Wang, S.L. Jacques, and L.Q. Zheng. Mcml monte carlo modeling of light transport in multilayered tissues. *Comput. Meth. Prog. Bio.*, 47(2):131–146, July 1995.

[2] E. Alerstam, T. Svensson, and S. Andersson-Engels. Parallel computing with graphics processing units for high-speed monte carlo simulations of photon migration. *J. Biomed. Opt.*, 13(6):060504, 2008.

[3] L. Wang and S.L. Jacques. *Monte Carlo modeling of light transport in multi-layered tissues in standard C*, 1998.

[4] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide, Version 2.0, 6/7/2008.*

[5] W. Lo, K. Redmond, J. Luu, P. Chow, J. Rose, and L. Lilge. Hardware acceleration of a Monte Caro simulation for photodynamic treatment planning. *J. Biomed. Opt.*, 14(1):014019, 2009.

[6] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[7] G. Marsaglia. Random number generators. *J. Mod. Appl. Stat. Meth.*, 2(1):2–13, 2003.

[8] R. Shams and R.A. Kennedy. Efficient histogram algorithms for nvidia cuda compatible devices. *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, 2007.

[9] R. Shams and N. Barnes. Speeding up mutual information computation using nvidia cuda hardware. *Proc. Digital Image Computing: Techniques and Applications (DICTA)*, pages 555–560, 2007.

[10] S. Gratton. Graphics card computing for cosmology, http://www.ast.cam.ac.uk/ stg20/gpgpu, April 2009.

[11] G. Marsaglia. The Marsaglia Random Number CDROM, with The Diehard Battery of Tests of Randomness of randomness, http://www.stat.fsu.edu/pub/diehard/, Technical report, Florida State University, 1995.

[12] The GNU MP Bignum Library, http://gmplib.org/, April 2009.

[13] T. R. Nicely. GNU GMP mpz_probab_prime_p pseudoprimes, http://www.trnicely.net/misc/mpzspsp.html, April 2009.