Niels Bohr Institute

GPU Simulation of Radiation in Matter

Bachelor project
July 2, 2013

Johannes Christof de Fine Licht
08.11.1991
definelicht@nbi.dk

Supervisors:
Børge Svane Nielsen (borge@nbi.dk)
Morten Dam Jørgensen (mdj@nbi.dk)

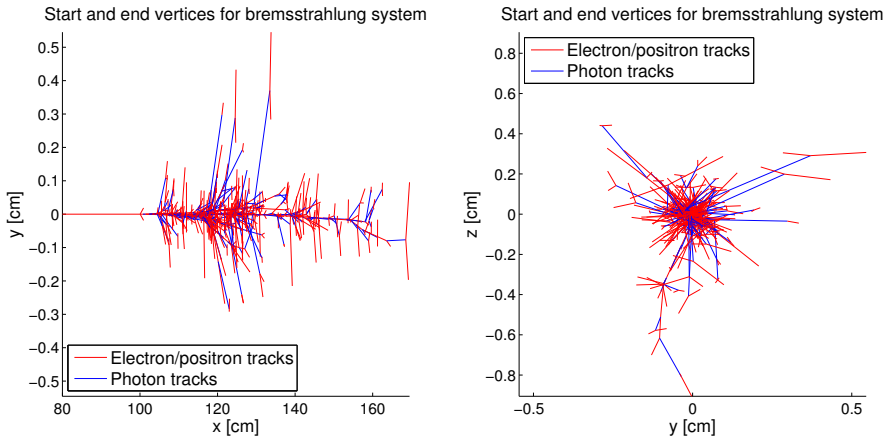

Figure 1: Connected start and end vertices for electrons and photons produced by bremsstrahlung by a single 10 GeV electron sent from $(0, 0)$ in the x-direction into an iron block starting at $x = 100cm$.

Abstract

Parallel programming on GPUs is introduced in the context of simulating collision energy loss and bremsstrahlung for charged particles propagating in matter. The employed Monte Carlo methods and the involved physics are presented, followed by an introduction to the concepts of GPU parallel programming for the Nvidia CUDA architecture. The simulations implemented in C++ and CUDA are described. The physics results of the implemented simulations are presented, and the simulations are benchmarked for various configurations between the CPU and GPU implementations, comparing and discussing the results. The GPU implementation is shown to gain a significant advantage at high particle multiplicity for collision energy loss, while the CPU implementation performs and scales better with the amount of secondary particles produced by bremsstrahlung.

Contents

List of Figures

1 Introduction

Simulation is an important tool in high energy physics. By recreating phenomena found in the labs, they can be used to build and verify the theoretical descriptions, as well as predict and act as a foundation for new experiments. A good simulation will mimic the output of actual experiments, allowing for direct comparison and verification of one by the other. When the experiments and energies become large enough, however, the scale and complexity of the simulation can quickly bring the runtime up to levels that require detail to be sacrificed to keep it practical. One such example is particle showers in a calorimeter, where a high energy particle cascades into a high number of particles products. Figure 1 shows a 10 GeV electron generating a shower of photons, electrons and positrons by bremsstrahlung when propagating through a block of material. The pictured simulation result contains 971 individual particles that have been produced and propagated, a number that scales fast with energy, as shown in figure 2.

To address performance issues arising in simulation, this paper introduces graphic processing units (GPUs) as an alternative (or potentially an aid) to traditional CPU programming. The distinct architecture of GPUs presented in section 4 allows for a high throughput of data through parallel computations, and it will be investigated if and how simulation of particles propagating in matter can be represented in a way that accomodates this architecture.

Before the implementation is presented, the basics of employed Monte Carlo techniques will be described, followed by the physics processes that are to be simulated and the most relevant aspects of GPU architecture and parallel programming. The physics output of the simulation will then be presented before extensive benchmarking between the GPU and CPU implementations is run and analyzed.

2 Monte Carlo simulation

At the core of this simulation stands the Monte Carlo method. Embracing the statistical nature of physics, using Monte Carlo simulation is to reverse

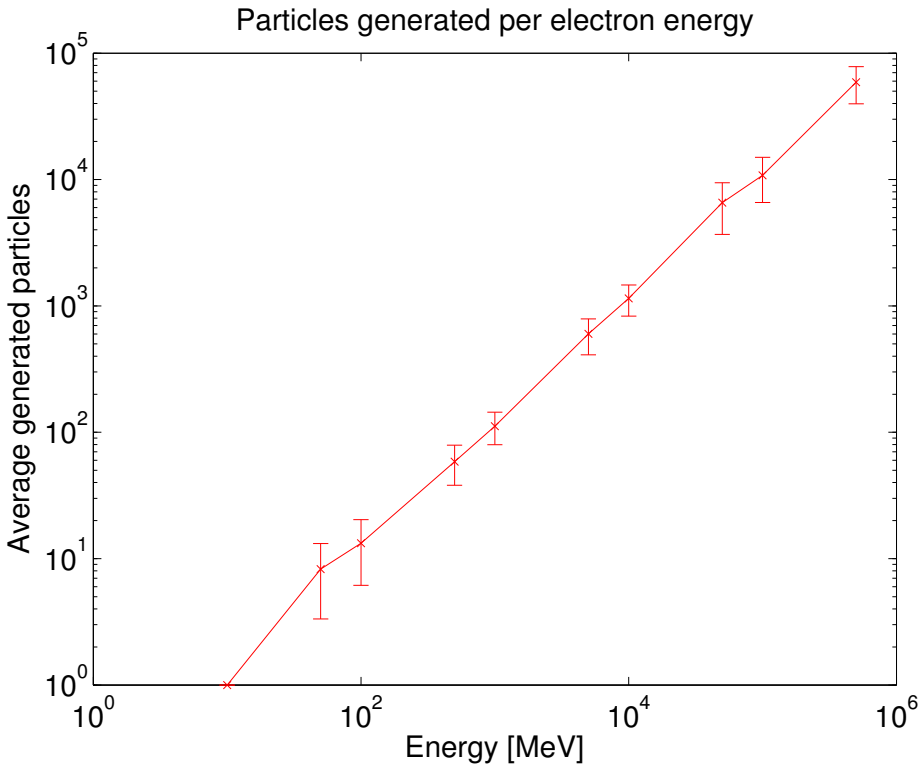

Figure 2: Average number of particles generated by the simulation for different energies of an incident electron propagating through solid iron. Each energy is sampled 64 times. The number includes electrons that are reinitiated after emitting a high energy photon.

engineer the distributions found empirically or theoretically in nature, recreating them from the bottom up through appropriately generated random numbers. This allows easy simulation of high-dimensional models, because each parameter can be generated according to its individual distribution, outputting constituted samples in the total phase space. Monte Carlo simulation is based in statistics and is indeterministic in nature, relying on a high number of replicates to obtain the desired results. This means that the total computational effort required scales drastically with the required effort for a single replicate, resulting in a high importance of algorithms and optimization compared to other scientific computing.

2.1 Random sampling

The goal of random sampling is to generate random variables that are distributed according to a distribution function F_x. The basic method for achieving this is by *inversion*. By generating a uniformly distributed random number u corresponding to a probability of an outcome x, that outcome can be calculated by solving for x in the distribution function, so

$$u = F_x(x) \Rightarrow x = F_x^{-1}(u) \tag{1}$$

Unfortunately, most commonly used distributions can not be inverted analytically, requiring other methods to be employed to properly distribute random numbers. Two such techniques, both of which will be used in this simulation, are described below.

2.1.1 Inversion approximation

A common approach to sampling from an uninvertible distribution is to numerically build a function that follows the distribution, using a combination of tabulation, extrapolation and parametrization to minimize the deviation from the original distribution. Once such a function has been made, sampling is done relatively quickly and in constant time, ensuring roughly the same sampling time across accesses. In order to obtain accurate results for more complicate distributions, however, large amounts of data points must potentially be stored in memory, and a lot of memory accesses might be necessary to generate samples. This can have performance implications in a GPU implementation in particular (section 4.4). Some approximations

may also additionally implement rejection sampling to generate a result, as described below.

2.1.2 Rejection sampling

For uninvertible distributions that can have their non-inverted form parameterized, rejection sampling provides results by *rejecting* self-generated samples until they comply to a distribution. The art consists of drawing the potential samples from a function approximating the inversion as well as possible in order to minimize the number of rejects before a valid sample is determined, while still maintaining speed - sampling many times from a simple function might outperform few samples from a computationally heavy function. Although it is desirable from a performance perspective to draw from a sampling function that mimics the actual distribution as closely as possible, the only restriction maintaining correctness is that $f(x) < k \cdot g(x)$ for a distribution $f(x)$ and a sampling function $g(x)$ with $k > 1$. The algorithm is presented in pseudocode below:

- Draw random number x distributed by $g(x)$
- Draw random number u distributed uniformly in $]0, 1[$
- Compute $f(x)$
- If $u < f(x)/k$ accept (return), otherwise repeat

To minimize the fraction of rejected samples, an approximation to an inverted function might be used, as described above. This can be particularly relevant in the case of GPU simulation, because indeterministic exit from the rejection procedure can be a performance implication, as described in section 4.4.

3 Radiation in matter

Charged particles propagating in matter can be put into two major categories: electron and positrons in one, and heavier particles such as muons, pions, protons and other small nuclei in the other. This simulation will treat the dominating sources of energy loss for both, in particular energy lost by elastic collisions with the nuclei of the material and inelastic collisions with the electrons and nuclei, as well as bremsstrahlung, which dominates

the energy loss by electrons and positrons above a few tens of MeV. These two processes will serve to show two distinct types of computations: homogeneously computated collision energy loss, in which only the amount of energy lost is drawn randomly, and bremsstrahlung, in which the nature itself of the required computations is determined by randomly drawn numbers. The fundamental difference between these two types in a GPU context will become apparent throughout this paper.

3.1 Landau distribution sampling

The amount of energy lost by charged particles in a thin absorber has been found to comply with a Landau distribution, which is an asymmetric probability function with a long, high energy tail that it owes to the small but finite probability of a large energy transfer happening in a single collision. Because of its asymmetry, the Landau is described by a *most probable value* rather than a mean, which can be calculated according to eq. 2 along with the width $w = 4\xi$ (with ξ as given in 3.2). Because propagation of the particles is done stochastically, the notion of *thin absorber* is interpreted as a sufficiently small spatial step through the geometry for the purpose of this simulation. The implementation uses an approximation to an inverted Landau (described in 2.1) to generate samples for the stochastic propagation, ported from the Fortran version in CERN Program Library [6]. The code uses a combination of tabulation and extrapolation to produce a sample from a uniformly drawn random number. A histogram of 2^{26} entries generated on the GPU can be seen in figure 19 of appendix B, fitted to a Landau distribution as implemented by ROOT's TH1 [9]. The location of the mean is indicated, showing how the long tail pushes the mean towards a higher value than the most probable value, which is at the peak of the distribution.

3.2 Collision energy loss

For a charged particle with mass m and charge z moving with speed $\beta = v/c$ and $\gamma = (1 - \beta^2)^{-1/2}$, the most probable energy loss by collision is computed as [1, p. 289]:

$$\Delta_p = \xi \left[\ln \frac{2mc^2\beta^2\gamma^2}{I} + \ln \frac{\xi}{I} + j - \beta^2 \right] \tag{2}$$

where $\xi = K \cdot Z\rho/A \cdot (x/\beta^2)$, Z, ρ, A and I are the atomic number, density, atomic weight and the mean excitation potential of the material, respectively, $j = 0.200$, and the constants are as given in appendix A. An additional density correction exists, which has not been included. Δ_p and 4ξ will act as the most probable value and width parameters to the Landau distribution. For comparison to the energy loss distribution that will be output by eq. 2, the mean energy loss for **heavy** charged particles due to collisions is calculated by the Bethe equation, valid in vicinity of $0.1 < \beta\gamma < 100$:

$$-\frac{dE}{dx} = K\rho\frac{Z}{A}\frac{z^2}{\beta^2}\left[\ln\left(\frac{2m_e c^2\beta^2\gamma^2 W_{\max}}{I^2}\right) - 2\beta^2\right] \tag{3}$$

where z is the elementary charge of the incident particle, ρ is the density of the material, the constants are as given in appendix A, and

$$W_{\max} = \frac{2m_e c^2\beta^2\gamma^2}{1 + 2s\sqrt{1+\beta^2\gamma^2} + s^2} \tag{4}$$

with $s = m_e/M$. For electrons and positrons, the incident particle's small mass and indistinguishability from the electrons in the matter it is colliding with give rise to some significant changes to the Bethe equation. These will not be described here, as only the most probable energy loss is required for stochastic simulation.

3.3 Bremsstrahlung

The Bremsstrahlung implementation essentially consists of two processes: photons radiated by electrons and positrons and pair production of electrons and positrons by photons. Common to both processes is that they are not isolated processes, and both depend on interaction with a surrounding medium in order to happen. The implementation used is adapted from the Geant4 toolkit source code [12], converted to C-like functions for the GPU case.

3.3.1 Photon radiation

Although common to all charged particles that are deaccelerated in matter, radiation of photons becomes the most important source of energy loss at higher energies for electrons and positrons because of their low mass. The radiated photon is essentially a discrete amount of energy lost by deflection in

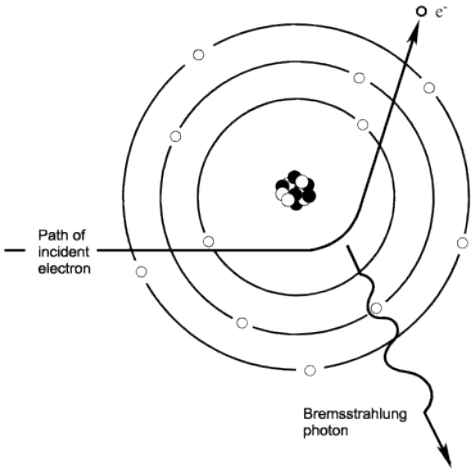

Figure 3: An electron interacts with the electric field of an atomic nucleus, radiating a photon. [7]

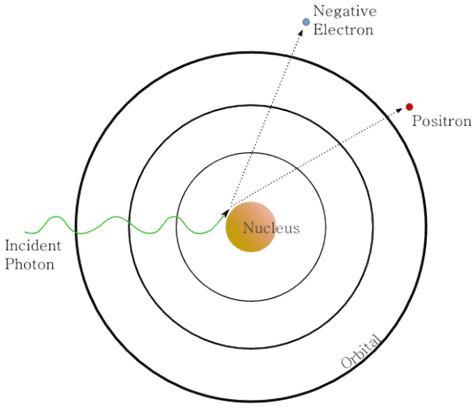

Figure 4: A photon with sufficient energy interacts with matter and produces an electron/positron pair. [8]

the electric field of an atomic nucleus. The chance of an interaction happening during a spatial step Δx is computed as

$$p(\Delta x) = 1 - e^{-\frac{\Delta x}{X_0}} \tag{5}$$

Where X_0 $[cm]$ is the radiation length for the medium that the particle is being propagated in. Using this probability stochastically assumes $\Delta x \ll X_0$ to obtain a correct distribution, as values of Δx comparable to X_0 can allow several interactions to happen during Δx and can overshoot the point of interaction by a significant amount. A fitted plot of 10^6 samples is shown in figure 20 of appendix C, showing how the mean of the output distribution becomes the radiation length. The photon energy is sampled by rejection sampling from a parameterization of tabulated results [13]. The direction of the gamma is sampled along the axis of the parent particle is distributed based on the work of Tsai [15].

3.3.2 Pair production

At suffiently high energies, an electron-positron pair can be generated from a photon interacting with matter, converting the photon energy into the rest

mass and momentum of the two leptons. This sets the threshold energy of $2m_e c^2$ required for the interaction to happen. The probability of pair production happening during a spatial step Δx is computed as eq. 5. The energy will be distributed equally, and the direction of the electron and positron is calculated symmetrically with respect to the entering photon's axis of movement [14].

4 GPU programming

Following an accelerating many-year development to match the evolution of computer graphics, graphics processing units (GPUs) are built for processing large amounts of data in parallel. In contrast to convential central processing units (CPUs), GPUs employ a much larger amount of processing cores with lower clock speeds, and have separate onboard memory. The parallel nature of GPUs arises as instructions are performed on the hardware as *single instructions, multiple data* (SIMD), exploiting parallelism in the input data to perform large amounts of simultaneous computations in parallel. For the purpose of this simulation, Nvidia's *Compute Unified Device Architecture* [16] (CUDA) platform is used to access memory and computations on the GPU.

4.1 Sequential versus parallel approach

Because GPUs are built to assume a high level of parallelism in the input data, the hardware relies on heavy pipelining of instructions on a high number of arithmetic logical units (ALUs). No branch prediction or advanced control mechanics are utilized; **all** instructions will be performed by the pipeline. Efficiency is achieved through threading by the *thread scheduler*, building a pipeline that makes up for memory access latencies by performing arithmetic instructions while data is being loaded. No cache is maintained; instead, memory can be loaded into the registers of the processing units (threads) very efficiently by exploiting coalescence, utilizing the full amount of data loaded in a memory burst (described in 4.4), which in the ideal case means that the throughput of the GPU will correspond to the throughput of its internal memory bus. In contrast, a CPU relies on its cache and data forwarding to combat the high latency of RAM access, and because no nature of the input data is assumed, branch prediction is used to pipeline conditional code. Because of this, computations that are highly sequential in nature (heavy

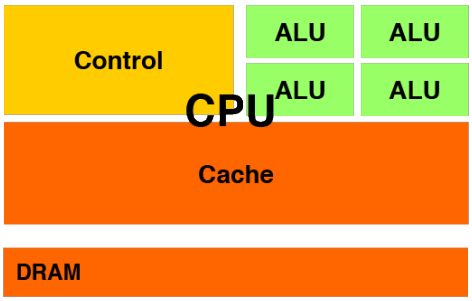

Figure 5: CPU architecture philosophy. A low number of ALUs perform instructions at a very high rate, relying on the cache to make up for high latency when loading from DRAM. The control block optimizes the pipeline with branch prediction and data forwarding.

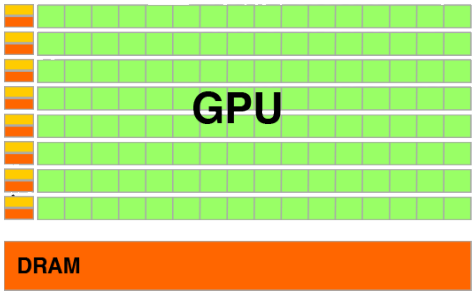

Figure 6: GPU architecture philosophy. Instructions are performed at a lower rate, but are performed concurrently on a high number of ALUs. No cache is maintained. There is limited control, instead efficiency is achieved by the thread scheduler.

branching, mutually dependent data) can execute considerably slower on a GPU, because the higher clock speed and efficient handling of branching on the CPU does not rely on a specific format of the input data.

4.2 Anatomy

Execution a CUDA program means launching one or more kernels that will be executed on the GPU. The GPU can only execute a single kernel function on all threads at a given time, leaving a thread's index as its only distinction from other threads. The order in which memory is accessed is indeterministic, leaving it to the programmer to prevent race conditions in the code. Although atomic access is possible, it is highly sequential in nature, and it is usually more desirable to restructure the code in a way so that it isn't needed. Nvidia GPUs can be identified as having a number of *streaming multiprocessors* (SMs), each containing a number of processing units, which in turn each run a number of threads. One SM can perform a single instruction at a time, meaning all concurrently running threads managed by that SM must either perform that instruction or do nothing (idle). Instruction sets are performed in *warps*, denoting the pipeline that is being executed on a SM. In order to achieve maximum concurrency, each instruction must be performed on a

number of data units divisible by the number of threads that can be run in a warp. A mid-to-high end Kepler architecture GPU can contain 7 SMs, each running 8 processing units that handle 4 threads each, resulting in a warp size of 32 concurrent threads for each SM to a total of 224 concurrently running computations (assuming aligned, branchless input). To truely take advantage of the thread scheduler, a significantly higher amount of threads should be submitted in a kernel launch, however. This allows the scheduler to hide latency from memory accesses by building the pipeline from as many threads as possible, as accessing global memory takes ∼500 clock cycles versus the 1 clock cycle reading from a register [4, 4-1].

4.3 Scheduling and memory hierarchy

Although the number of concurrently running threads is limited by the amount of SMs, the GPU scheduler relies on a large amount of threads being queued to maximize throughput. When assigning threads, the jobs submitted to each SM is divided into thread *blocks*, which are in turn divided into block *grids*. A SM has a limit to the number of threads that can be submitted and a limit to the number of thread blocks it can manage. Blocks can have between one and three dimensions of threads (x, y and z), intended for memory alignment purposes. For instance, performing operations on a large matrix might fittingly be divided into submatrices handled by two dimensional blocks of an appropriate size. Memory transfer to and from the device memory must be done explicitly by allocating and copying data from the host code, and no direct access is allowed between host and device memory without performing the necessary manual transfers.
For general computation purposes, the GPU memory model consists of the following levels.

Device global memory The bulk of the onboard memory, and the only memory that interacts with the host, meaning all data transfer to and from the GPU writes to or reads from here. Can be accessed by all threads in all blocks, but is high latency, causing alignment and coalescence to play a large role when loading from here.

Shared memory Associated with a given block, shared memory is the only memory that is shared among threads during execution, although only

threads of that block can access it. It must be utilized manually and is very low latency, allowing an appropriate data structure to avoid loading the same data from global memory if it is needed in multiple threads of the same block.

Local memory While bound to a specific thread, local memory resides in the high latency global dynamic random-access memory (DRAM). Registers and shared memory should be used whenever possible to avoid lookups in global or local memory.

4.4 Parallelizability

While a GPU is capable of heavy throughput, it puts strict requirement on the input data and the nature of the computations in order to achieve a performance increase. The basic requirements to an algorithm that is to be run in parallel on a GPU include:

1. **Homogeneity** Data should be expressed homogeneously, differing in value and not in form.

2. **Mutual independence** Computations should be mutually independent, as the order of memory accesses is indeterministic and exchange of information between threads is highly restricted.

3. **Branching** A minimum of branching should occur; any difference between instruction sets within a warp will be run sequentially, in worst case reducing the throughput to the reciprocal warp size.

In addition to these basic requirements, other factors with a high impact on performance are:

1. **Coalescence** Whenever data is fetched from global memory, the DRAM will transfer a burst of memory from the target address. Adjacent threads can exploit this by requesting memory that is coalescent in global memory, greatly reducing the amount of memory bursts required to populate the threads.

2. **Alignment** In order to optimally exploit coalescence, the loaded data should be aligned to a number of bytes that the memory burst size is divisable into in order to allow overlaps resulting in additional memory bursts being necessary.

5 Implementation

For the purpose of comparison, the simulation engine has been implemented both in convential object oriented C++ under the 2011 standard for host (CPU) execution, and as CUDA device code, which is C++ with CUDA extensions compiled with Nvidia's CUDA specific compiler *nvcc* for use on the GPU. Although the code executing actual physics is very similar on the two devices, the sorrounding engine and representation are necessarily very different in implementation, and as such, comparisons between runtime cross-device is somewhat heuristic. CPU simulations are performed by launching 8 concurrent threads (using the standard library interface [18]) handling a track each, mimicing the exploitation of mutual independence in the data. However, because the simulator and geometry are built according to an object oriented design, memory accesses to their content will be sequential to prevent concurrent writing to the same memory. Random numbers are drawn using ROOT's TRandom3 [10] on the CPU, while Nvidia's CURAND kernel functions are used on the GPU [11]. Both libraries produce high-quality low-periodicity pseudo-random numbers.
An example of a simulation set up in C++ using the framework is included in appendix D along with the produced output.

5.1 Representation

In order to comply to the first requirement of section 4.4, data required for the simulation must be represented in a homogenous fashion. In order to accomodate the second performance factor, each representation has its memory aligned to the smallest possible power of two to align to memory bursts. The instances that are represented in the simulation include:

Volume A set of parameters that can extend a physical volume in space once the type of geometrical object that the volume represents is known. The volume includes an index to the type of volume it represents, an index to the material it is composed of, and a block of bytes that can house the parameters specific to a volume type (position of origin, radius for a sphere or normal vectors to each dimensional side of a box).

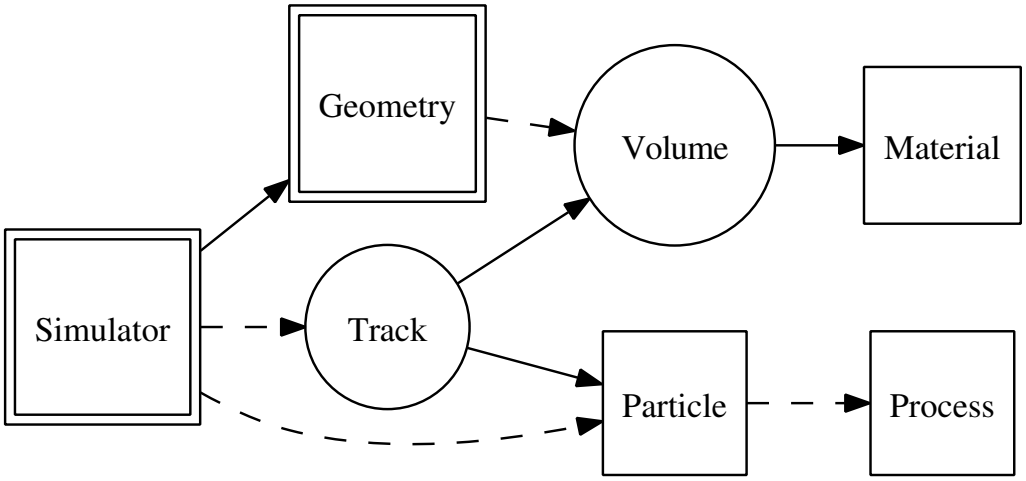

Figure 7: Structure of the implementation. Square nodes have distinct properties, circles are indistinct. Double bordered nodes are singletons in a single simulation. Dashed arrow lines indicate a one to many relationship.

Particle Has the fundamental properties of a type of particle. This can be mass, spin or lifetime.

Material Contains the physical properties necessary to evaluate physics processes that can occur for a given particle type in a material. For bremsstrahlung these are properties like atomic number, mean excitation potential and radiation length.

Track Represents the current state of a particle instance that is being propagated through the geometry. Contains an index to the particle type of the instance that is being propagated. These always have a 1:1 relationship with a thread for a given kernel launch, with one thread propagating one or zero tracks.

In addition to the master framework and the above instances, physics processes must also be represented. For the host C++ implementation, these are represented as objects added to a list of pointers for a given particle which all implement a query function that the main simulator can call. For the device CUDA implementation, however, the processes have been hardcoded, but can potentially be implemented as function pointers from a physics list that

the particles can index into. The geometry implementation allows the experimental setting to be built by placing arbitrary volumes of arbitrary materials in the desired way, which will be linked and handled by the framework.

5.2 Spawning

When generating new particles as is the case with bremsstrahlung, a procedure must be implemented to adapt this to the parallel data structure. Because memory is (and should be) very rigid on the GPU, the newly generated particles must be spawned in a way that allows them to be picked up and processed by the parallel propagation kernel, but taking indeterminism into account, so no two tracks attempt to spawn a track in the same memory location. For this, a track *pool* of a size specified prior to propagation is added. In order to allow threads to index into the pool directly, the pool is coalescent in memory, thus making potential dynamic expansion very inefficient. The pool is also placed coalescent to the active tracks, for reasons explained in section 5.3. If a track requests memory that is not available, the track must be put in a waiting state and the thread must be stopped.

5.3 Sorting

In order to propagate newly spawned particles and to address numerous other issues, track sorting is introduced. When a kernel is launched with n threads scheduled, these will index into the first $0...n-1$ track addresses. During a single kernel launch, memory indices must be static to account for indeterministic access, but between kernel launches, the track memory can be rearranged to best fit the parallel data model. For this purpose, the Thrust parallel algorithm library is utilized [17], which gives access to a host-side interface that employs highly optimized sorting algorithms run on the GPU. Because the data is already stored on the GPU, this can be performed very efficiently with no host/device transfer overhead. The tracks are then sorted according to *keys* stored in a separate and memory coalescent array. This leaves the question of how to compute these keys for the best results.

5.3.1 Determining keys to sort by

There are four primary cases to consider; living tracks that are still being propagated, dead tracks whose particle have gone out of bounds or are out

of energy, waiting tracks that are missing the required memory to spawn new tracks, and free track slots that have not yet been utilized, and do not contain any physically significant information. To make it possible for living tracks to index directly into the pool of unused tracks, the free track slots are given the highest possible key to function as a *heap*, growing downwards from the highest memory address. These will be followed by the dead tracks, as these should not be indexed into by the launched threads before all living threads have been processed. Next will be the tracks waiting for free memory, and finally the living tracks. The keys for living tracks do not affect the correctness of the procedure, but can affect the amount of branching that will occur during a kernel launch. Choosing the right keys among the living tracks can also potentially coalesce tracks that are to be treated equally or similarly, increasing hardware utilization by increasing concurrency (through reduced branching). For a bremsstrahlung simulation it might be desirable to sort by particle type as electrons and photons are treated differently, while energy loss for different heavy charged particles in matter might benefit more from sorting according to their position, as their differences only change the inputs to the same calculations. The sorting/execution procedure is visualized in figure 8. The simulation ends when the track at the first index is found dead after a sort, as this means no living or waiting tracks can remain.

5.3.2 Kernel launches versus steps per launch

Each time a kernel is launched, it must drop out after a number of iterations to perform the sorting procedure, picking up new tracks and sorting out dead ones. The frequency of these launches can affect performance in several ways. Because sorting, checking whether the propagation is done and relaunching the kernel has some amount of overhead, launching the kernel more often will directly increase runtime. On the other hand, threads whose tracks have died will remain inactive until their warp finishes processing. Choosing the right frequency of kernel launches must then be a balance between these two effects. For the purpose of benchmarking this, two test scenarios are set up: one for heavy muons propagating in a large experiment, performing only regular energy loss, and one for electrons in a small experiment, spawning large amounts of new particles by bremsstrahlung that differ in the way they are treated. The result is shown in figure 9. The runtime is shown divided by

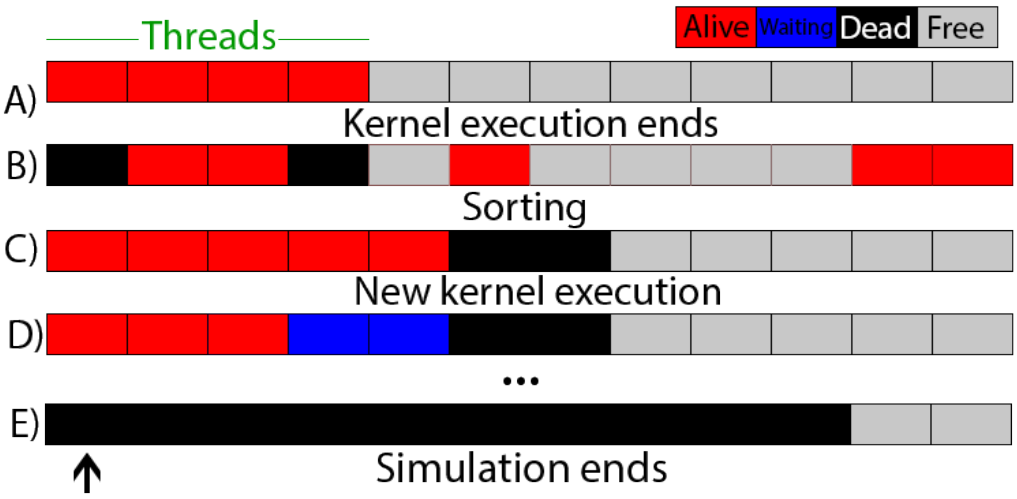

Figure 8: Visualization of the sorting/execution procedure. Only the tracks indicated by the green line are being accessed by threads. *A)* initial state, four tracks are initially active, while eight track slots are available for use. *B)* a kernel execution ends. Two threads have died, one spawning a single child and the other spawning two. *C)* the tracks are sorted before the kernel is launched again, restoring the track heap and pushing the dead tracks to the end of the list of actual tracks. *D)* a new kernel is launched. The third through sixth track request spawning children. The third requests only one and is allowed to do so, but the heap addresses for the remaining two are occupied by dead tracks. They are put in a waiting state. *E)* the final state of a simulation. The first track slot contains a dead track after the tracks have been sorted, indicating that no living tracks remain.

the highest runtime for the given scenario, and is plotted against the number of spatial propagation steps dx performed before the kernel drops and a new one is launched after the tracks have been sorted. CPU benchmarks are run for comparison. While the simple scenario where no new tracks are produced shows a trivial reduction in execution time with reduced overhead by reduced number of kernel launches, the bremsstrahlung scenario's lowest runtime is observed at a higher frequency. What is even more noticable, however, is the large difference in the relative speed between GPU and CPU simulation between the two scenarios. For the first configuration where a large amount of muons are propagated and only energy loss is calculated, the GPU runs the simulation four times faster than the CPU. In the second configuration, high energy electrons cascade into a large amount of newly spawned particles, requiring a large amount of new tracks to be spawned, and killing tracks in a fashion that is inhomogeneous memory-wise. Here the CPU performs slightly better than the GPU, finishing the simulation around 20% faster than the fastest GPU configuration.

6 Simulation output

Before more benchmarks are performed, the physics output of the simulation will be inspected. The implemented processes are centered around energy loss in matter, and are characterized by the radiation length X_0, determining the amount of photons and electron/positron pairs generated, the mean excitation potential I, as well as constant factors by the material density and atomic weight. The simulation is not *complete*, as low energy effects such as absorption of photons and annihilation of slow positrons with the electrons of the material are not included. However, the characteristics of the high energy effects should be largely unaffected.

6.1 Stopping power

Figure 10 shows muon energy loss by collisions in copper sampled from a Landau distribution with eq. 2 and the mean energy as calculated by eq. 3. The width of the Landau distribution is used as $dx = 10\mu$m. The most dense area of the Landau samples is observed to situate around the most probable value, which is lower than the mean value. This is a consequence of the Landau's long high energy tail, shifting the mean energy towards a higher

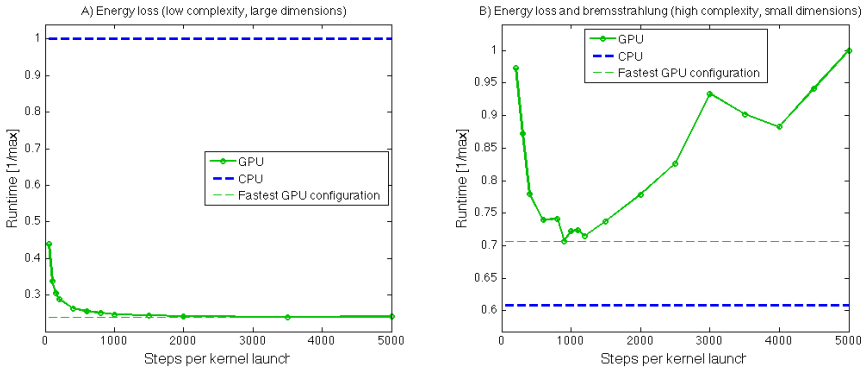

Figure 9: Benchmark of simulation time versus number of spatial propagation steps performed per kernel launch for two different levels of complexity. Lower is better. *A)* is performed in a system with large dimensions and low complexity, and with energy loss for muons only. The best GPU configuration runs at approximately four times the speed of the CPU simulation. *B)* is performed in a tight system with both bremsstrahlung (including pair production) and energy loss. The best GPU configuration runs about 20% slower than the CPU simulation (run on 8 CPU threads), and shows a non-trivial dependency on the steps performed per kernel launch.

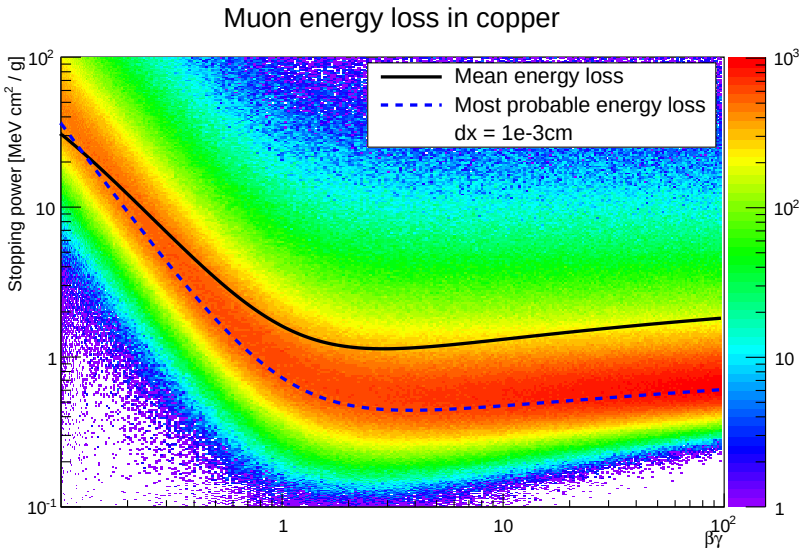

Figure 10: Stopping power for muons in copper.

value.

6.2 Bremsstrahlung

The front page features a pictured example of a single event shown in figure 1. The collision energy loss from both the incident electron and electrons and positrons generated by pair production from photons is shown in figure 11 and 12 as a distribution across the xy and yz planes for an experiment similar to the one in figure 1. The propagation is done for 64 electrons of 30 GeV incident energy entering an iron medium at $(0, 0, 0)$ with radiation length $X_0 \approx 1.76cm$ [5]. The mean energy deposit parallel to the incident beam is found at $\langle x \rangle = 35.08cm$ corresponding to $35.08/1.76 = 19.93$ radiation lengths. Appendix D demonstrates an example with alternating layers of iron and plastic, including the code used to generate the plot.

7 Benchmarking

Benchmarks are performed on a 2.3 GHz Intel Core i7 processor, run on eight concurrent threads and with 1600 MHz DDR3 memory. The GPU code is executed on an Nvidia GeForce GT 650M[1].

7.1 Energy

Benchmarks are performed for a range of energies for incident electrons for collision energy loss only and for included bremsstrahlung. The results are shown in figure 13 and 14. Again, the GPU seems to perform poorly in the bremsstrahlung simulation, while excelling at the simulation where the amount of propagating particles is fixed. This indicates a performance bottleneck in the mechanics associated with generation of new particles. With the additional track slots needing to be sorted (figure 2), the sorting procedure is a candidate. The benchmark in figure 14 is performed with a variable track heap size depending on the incident energy by $0.15 \cdot E_{\text{incident}} \cdot n$, where n is the amount of particles and 0.15 is a rounded up slope taken from a rough linear fit to the data of figure 2. To see the effect of the heap size, the same benchmark is performed where each simulation is run twice on the GPU,

[1] GeForce GT 650M, `http://www.geforce.co.uk/hardware/notebook-gpus/geforce-gt-650m`

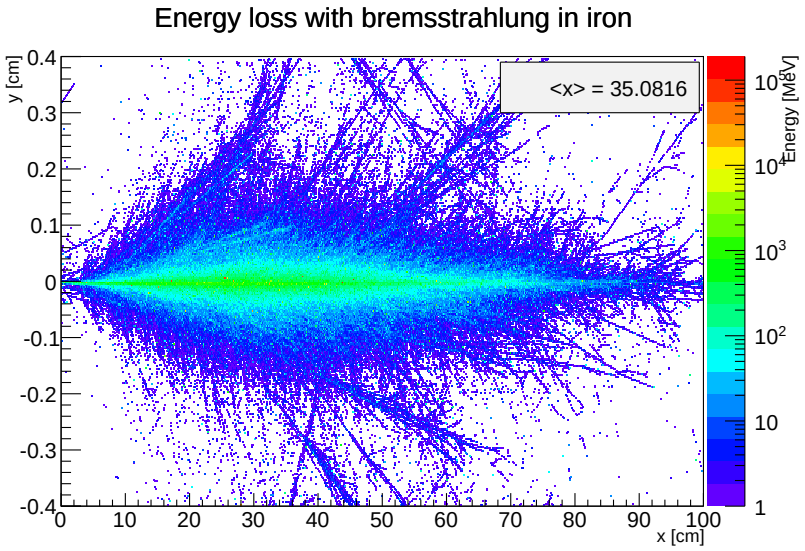

Figure 11: Energy loss in xy-plane for 64 incidents electrons at 30 GeV.

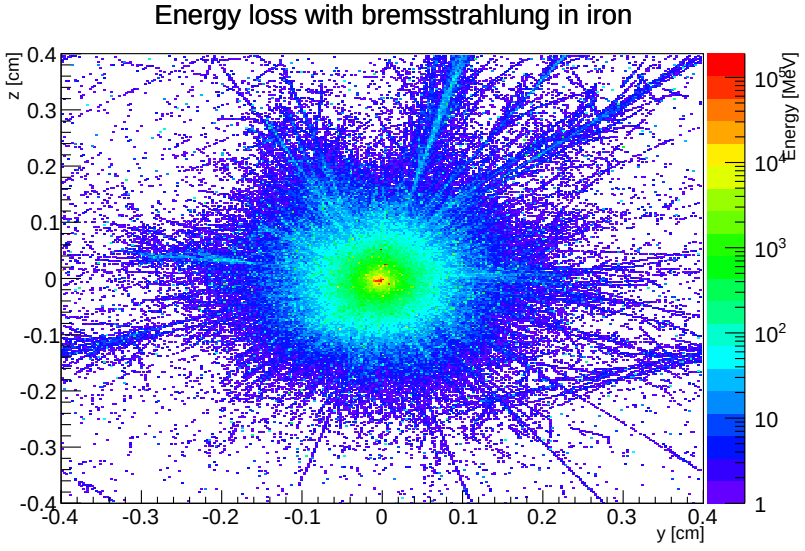

Figure 12: Energy loss in yz-plane for 64 incidents electrons at 30 GeV, shown for the same simulation run as figure 11.

once with variable and once with static heap size. The result is shown in 15. It is clear that the amount of necessary track slots that has to be managed and sorted becomes a problem at high energies, as performance scales poorly with the size of the track pool. This primarily originates from the fact that dead tracks are not cleaned up before the entire simulation is done, meaning all free and dead track slots have to be maintained even though only a fraction of them are used in a given kernel launch. How to potentially solve this problem will be discussed in section 8.

7.2 Number of incident particles

The bremsstrahlung energy benchmark showed bad performance for large numbers of particles generated by bremsstrahlung on the GPU. A high amount of threads running concurrent propagations is expected to run efficiently on the GPU, however, and the performance problems in 7.1 originate in the memory management associated with spawning particles on the fly, not from the multiplicity of particles itself. This is demonstrated in figure 16, where different amounts of incident particles are benchmarked between the GPU and CPU. The GPU truely starts to shine when the amount of concurrent threads reaches the thousands. Here the memory latency and lack of cache on the chip is hidden by the thread scheduler, weaving in new warps to perform calculations while others are waiting for data from DRAM, maximizing throughput by minimizing idle time.

7.3 Secondary electron threshold

When electrons emit photons by bremsstrahlung, the original electron will be killed and replaced by a secondary electron if the emitted energy is above a certain threshold. In the above, this is set low at the $2m_e$ required for pair production, but can cranked up or even turned infinite for simulations where the individual path changes along an electron's path are not important (aggregated results will remain the same), thereby reducing the amount of new particles that need to be spawned. As the spawning of new particles is the primary bottleneck of GPU performance, a benchmark is run for different secondary threshold energies, shown in figure 17. The GPU's performance compared to the CPU increases with the threshold, although it is still slower even with no secondary electrons being generated.

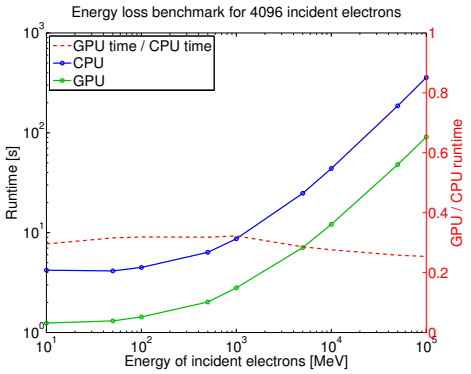

Figure 13: Runtime as a percentage of maximum for different electron incident energies with collision energy loss but no bremsstrahlung. The dashed line is the GPU runtime divided by the CPU runtime.

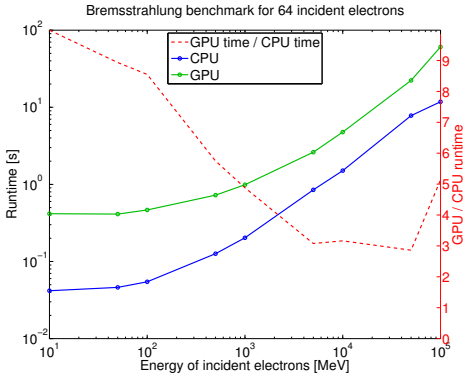

Figure 14: Runtime as a percentage of maximum for different electron incident energies with collision energy loss and bremsstrahlung. The dashed line is the GPU runtime divided by the CPU runtime.

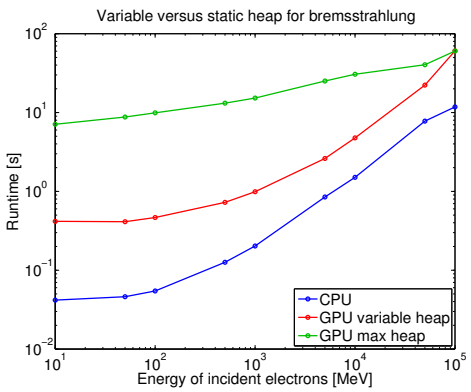

Figure 15: Energy benchmark as in figure 14, but run with variable heap size depending on the incident energy and with a static, high heap size.

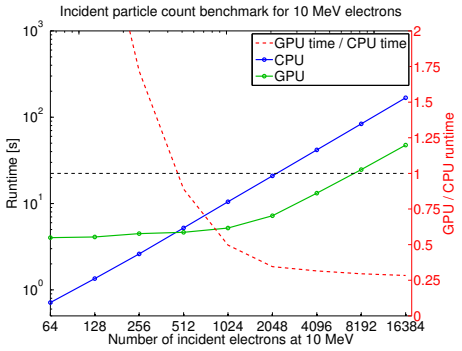

Figure 16: Benchmark of number of incident particles without bremsstrahlung. The GPU starts to outperform the CPU above 512 threads (particles).

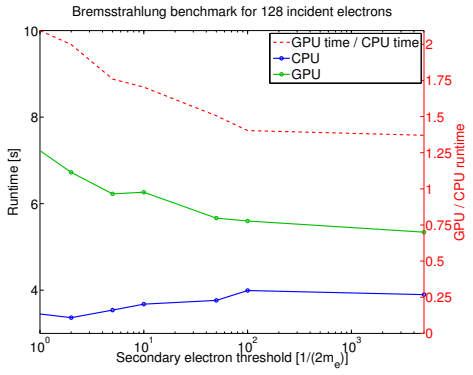

Figure 17: Benchmark of the energy threshold where electrons radiating photons are reinitiated to keep a detailed record of their path. Performed for 5 GeV electrons, where the highest threshold energy means that no electrons are reinitiated.

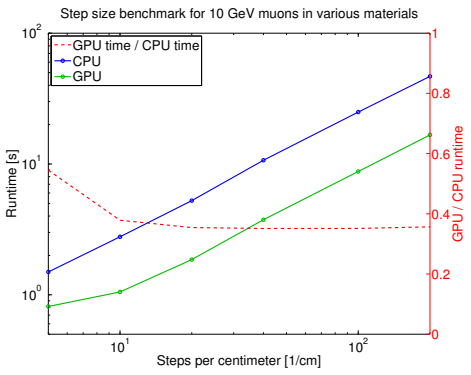

Figure 18: Benchmark of integral steps taken per centimeter during simulation. The simulation includes collision loss without bremsstrahlung for muons. Propagation done through ten sets of four different materials. Both devices scale the same.

7.4 Step size

The integral spatial step size used in the stochastic propagation of the particle is a direct factor to the number of computations that have to be done for a given simulation, determining the output resolution. Figure 18 shows a benchmark with decreasing step size (increasing steps computed per propagation length) between CPU and GPU. Although the GPU gains a slight edge at higher resolutions, both devices scale approximately the same for collision energy loss.

8 Evaluation

Through the various performed benchmarks, the strengths and weaknesses of the GPU in the context of this simulation become apparent. The bremsstrahlung energy benchmark in figure 14 and the incident particles benchmark in figure 16 stand out as the biggest contrast. For homogeneous collision energy loss computations the GPU scales very well with the number of particles, but for bremsstrahlung the memory acrobatics necessary to spawn new tracks is

a strong performance bottleneck. This is a combination of serveral architectural characteristics:

- Unused and dead tracks pile up in GPU memory, leaving a lot of records to be sorted between each kernel launch, despite most of them not being used in the next kernel launch.
- Dead particles will result in idle threads until the running kernel finishes and a new one is launched, reducing the number of concurrently running computations.
- Bremsstrahlung processes do not happen simultaneously for different threads in a warp, resulting in many inefficient memory writes that do not exploit coalescence and memory burst size.

Before these issues are solved, the GPU implementation will likely not be able to compete with the CPU code for memory heavy processes such as bremsstrahlung, as the CPU does not suffer nearly as much from the inconsistency in computations and memory writes.

8.1 Next steps

While the dynamic creation of new particles does not fit well with the GPU memory architecture, structural improvements can be made that might allow the GPU to compete. Two such improvements are suggested below.

8.1.1 Dealing with the heap

At very high energies a single electron can cascade into thousands of shower constituents, requiring a very large heap to accomodate all particles spawned throughout a simulation. While the size itself can be solved with sufficient onboard memory on the GPU, the sorting procedure becomes a major issue above some tenths of GeV for the incident particles. One approach to solving this is to sacrifice some output detail and only store relevant information. This could mean only storing energy deposited in the experiment, discarding all dying particles completely by overwriting them with new ones.
Another approach would be a two-layer heap where the size is proportional to the amount of particles expected to propagate concurrently at a given time, where one layer acts as a buffer from which dead tracks are copied

to the host memory, and the other is the live heap which tracks index into when spawning children. This is particularly interesting because CUDA allows memory transfers between host and device to be done asynchronously, sacrificing little or no performance to clean up memory on the fly if implemented correctly. The radiation length could be used to determine an efficient frequency of clean-ups and sorting. The entire buffer would have to be copied back to host for each clean-up, but since kernel launches execute asynchronously to the host code, the CPU could work on bookkeeping and storage without delaying the propagation running on the GPU. The concept is very close to that of a *framebuffer*, similar to how the GPU usually handles rendering of graphics. Some clever sorting mechanics would be be necessary for this approach to work, but an implementation could potentially shift the GPU bremsstrahlung simulation towards a competitive level of performance.

8.1.2 Track-level integral step size

The propagation in this simulation is done completely stochastically, evaluating each track at each spatial step dx, which is a static value. A general optimization would be to dynamically change the simulation dx to an appropriate size depending on the physics being performed at a given time, but even more interesting in the GPU case is to associate dx with individual tracks. This would not only allow some generic optimization, but might be used to urge coalescent tracks to perform memory heavy processes such as bremsstrahlung simultaneously, potentially gaining a great architectural advantage by having many tracks perform memory writes in a coalescent fashion, and having more control of when memory clean ups and sorting should be performed. This comes at a resolution trade-off, which could be an issue in the case of magnetic fields or fine-grained analysis of material energy deposit, but could make a significant difference to the viability of GPU simulation in the context of memory heavy physics processes.

8.2 As an auxiliary device

Making use of the GPU does not necessarily mean to run all computations on it, as it can also be employed as an auxiliary engine to the CPU. This is made possible by the asynchrony between the two processing/memory systems, allowing both to perform computations simultaneously. Making truely optimal use of the CPU/GPU system would mean maximizing utilization

across both, thus performing any task with a parallel structure that requires a high data throughput on the GPU while the CPU handles the main execution can lead to a "free" performance increase. Libraries such as Thrust make this particularly easy from a high level perspective, but with some routine small kernels can be implemented and integrated with a project relatively quickly. In the context of simulation, the GPU could be used as a simple propagator alone, or as a matrix computation engine, focusing its efforts on the most well defined application of parallel programming.

9 Wrapping up

A simulation framework that allows execution on both CPU and GPU has been developed, supporting arbitrary particle and geometry input. For the purpose of this paper, implementations of collision energy loss and bremsstrahlung for charged particles have been adapted to both architectures, producing output that correctly follow the appropriate distributions. To accomodate the generation of new particles on the GPU, a spawning/sorting procedure has been implemented, making it possible to write new tracks to GPU memory by using a track heap, without requiring tracks to be copied back to the host during simulation.
While performing very well at high particle multiplicity for the homogeneously computed collision energy loss, the presented GPU implementation shows a significant performance deficit spawning the particle products of the bremsstrahlung processes at high energies. The reasons are found in the GPU architecture, and suggestions to structural improvements that might solve this have been presented. Whether the performance of physics simulations can benefit from GPU computations is shown to rely heavily on the nature of the problem, and puts intricate restrictions and heavy dependence on the data representation and the algorithms employed to perform the simulation. Nevertheless, the GPU proves to be a powerful tool when submitted to the right conditions, and might also find its place as an asynchronous bonus engine.

References

[1] Particle Data Group, *Review of Particle Physics*. IOP Publishing, Volume 37, July 2010.

[2] Leo, W. R., *Techniques for Nuclear and Particle Physics Experiments*. Springer-Verlag, Second Revised Edition, 1994.

[3] Nvidia, *CUDA Toolkit Documentation*, offered at `http://docs.nvidia.com/cuda/index.html`.

[4] Hwu, W.M., lectures given on *Heterogeneous Parallel Programming*, offered through Coursera, by University of Illinois at Urbana-Champaign, 2012. `https://www.coursera.org/course/hetero`.

[5] Particle Data Group, *Calculation of radiation length in materials*, `http://pdg.lbl.gov/2013/AtomicNuclearProperties/`, revision December 2012.

[6] Landau Distribution from CERN Program Library, `http://wwwasdoc.web.cern.ch/wwwasdoc/shortwrupsdir/g110/top.html`.

[7] Picture of bremsstrahlung photon radiation, `http://www.radiologyschools.com/radiology-courses/radbiol/01physics/phys-03-05.html`.

[8] Picture of photon pair production, `http://ehs.unc.edu/training/self_study/xray/7.shtml`.

[9] ROOT TH1, `http://root.cern.ch/root/html/TH1.html`.

[10] TRandom3, `http://root.cern.ch/root/html/TRandom3.html`.

[11] Nvidia's CURAND for generating random numbers in the device kernel, `http://docs.nvidia.com/cuda/curand/index.html`.

[12] Geant4 source code, `http://geant4.web.cern.ch/geant4/support/download.shtml`.

[13] Geant4 User Guide, *Bremsstrahlung*. `http://geant4.cern.ch/G4UsersDocuments/UsersGuides/PhysicsReferenceManual/html/node42.html`, last updated 20.05.04.

[14] Geant4 User Guide, *Gamma Conversion*.
`http://geant4.cern.ch/G4UsersDocuments/UsersGuides/PhysicsReferenceManual/html/node58.html`, last updated 18.06.2001.

[15] Tsai Y. S., *Pair production and bremsstrahlung of charged leptons* from Review of Modern Physics, Vol. 46, No. 4, October 1974.

[16] CUDA, `https://developer.nvidia.com/category/zone/cuda-zone`.

[17] Thrust parallel algorithms library, `https://developer.nvidia.com/thrust`.

[18] STL threads, `http://en.cppreference.com/w/cpp/thread/thread`.

Appendix A

Electron mass	$m_e c^2$	0.510998910 MeV
Electron radius	r_e	2.817940325 fm
Avogadro's number	N_A	$6.0221415 \cdot 10^{23}$ mol^{-1}
$2\pi N_A r_e^2 m_e c^2$	K	0.307075 MeV g^{-1}

Appendix B

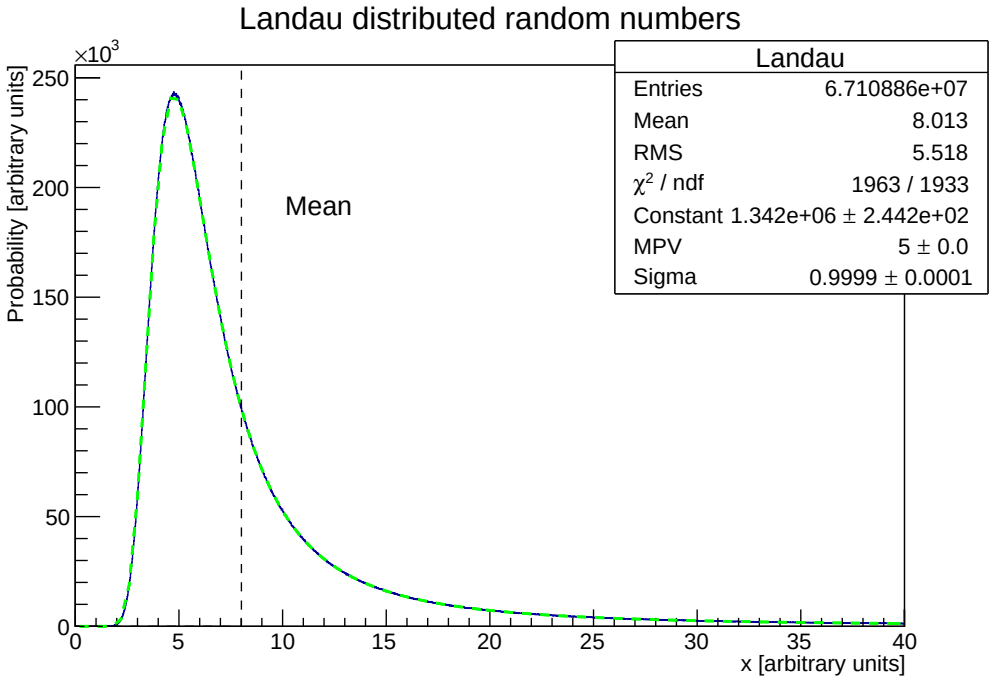

Figure 19: Landau distributed random numbers generated on the GPU with most probable value $MPV = 5.0$ and $w = 1.0$ (sigma) using an approximation to an inverted Landau approximation, fitted to a Landau function with ROOT's TH1.

Appendix C

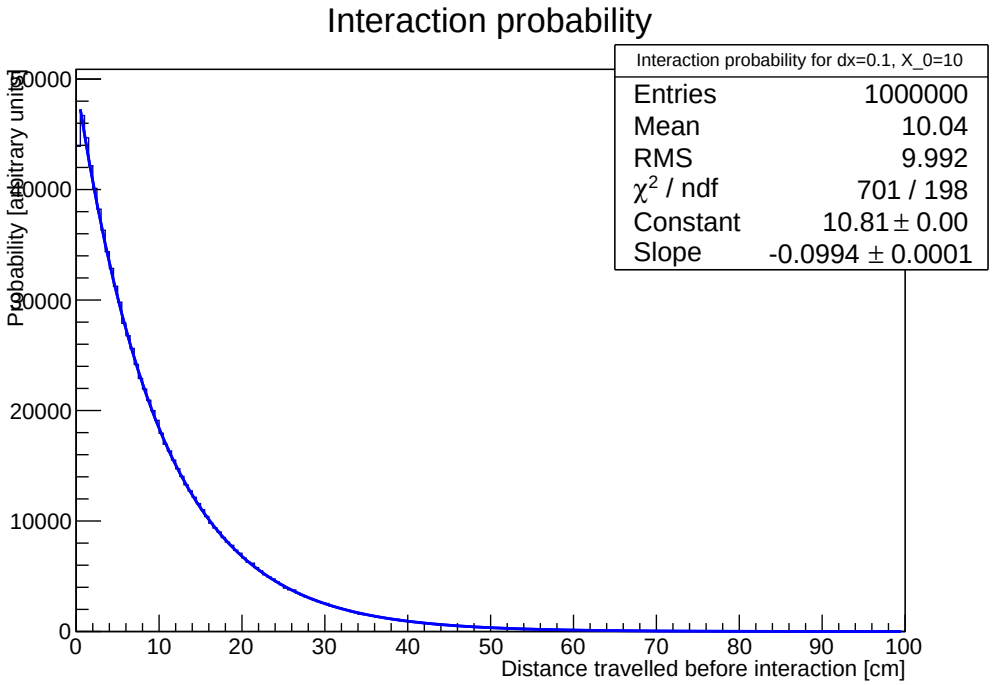

Figure 20: Distance travelled with a probability of interaction as eq. 5 for a step size of $dx = 0.1$cm and radiation length $X_0 = 10.0$cm. The blue line is a fit to an exponential function with parameters as shown.

Appendix D

Below is an example of a program running a bremsstrahlung simulation for layers of alternating iron and plastic of 5 cm each, producing the output shown in figure 21.

```
#include <sstream>

#include "Geometry/Geometry.hh"
#include "Geometry/SimpleBox.hh"
#include "Simulation/Simulator.hh"
#include "Simulation/EnergyLoss.hh"
#include "Simulation/GEANT4Bremsstrahlung.hh"
#include "Simulation/GEANT4PairProduction.hh"
#include "Simulation/BetheEnergyLoss.hh"
#include "Simulation/GetTime.hh"

#include <TApplication.h>
#include <TCanvas.h>

using namespace na63;
using namespace std;

int main(int argc, char* argv[]) {

  // Set up geometry
  Geometry geometry;
  geometry.AddMaterial(Material("vacuum",0,0,0,0,0));
  Material iron(
    "iron",
    kIronAtomicNumber,
    kIronDensity,
    kIronAtomicWeight,
    kIronMeanExcitationPotential,
    kIronRadiationLength
  );
  Material plastic(
    "plastic", // Polyvinyltoluene
    0.54141,
    1.032,
    // Use weighted mean of constituents
    (1.008 * 0.085 + 12.01 * 0.915) / 2,
    64.7,
    42.54
  );
  geometry.AddMaterial(iron);
  geometry.AddMaterial(plastic);
  SimpleBox bounds(
    "vacuum",
    ThreeVector(1*m,0,0),
    ThreeVector(2*m,1*m,1*m)
```

```
);
geometry.SetBounds(bounds);
const Float layer_size = 5 * cm;
const int layers = 20;
for (int l=0;l<layers;l++) {
  geometry.AddVolume(
    SimpleBox(
      "iron",
      ThreeVector((2.0*l + 0.5)*layer_size,0,0),
      ThreeVector(layer_size,1*m,1*m)
    )
  );
  geometry.AddVolume(
    SimpleBox(
      "plastic",
      ThreeVector((2.0*l + 1.0 + 0.5)*layer_size,0,0),
      ThreeVector(layer_size,1*m,1*m)
    )
  );
}

// Set up simulator
Simulator simulator(&geometry);
simulator.device = CPU;
simulator.cpu_threads = 8;
simulator.record_energyloss = true;

// Set up energy loss recording
EnergyLoss energy_loss(
  600,0,200, // x
  300,-5,5,  // y
  300,-5,5   // z
);
simulator.RecordEnergyLoss(&energy_loss);

// Set up particle types
Particle electron("electron",11,kElectronMass);
Particle photon("photon",22,0.0);
Bremsstrahlung bremsstrahlung(&electron);
PairProduction pair_production;
BetheEnergyLoss bethe_energy_loss;
electron.RegisterProcess(&bremsstrahlung);
electron.RegisterProcess(&bethe_energy_loss);
photon.RegisterProcess(&pair_production);
simulator.AddParticle(electron);
simulator.AddParticle(photon);

// Create initial tracks
const int n_tracks = 64;
const Float energy = 50 * GeV;
const Float mass = kElectronMass;
const Float momentum = sqrt(energy*energy - mass*mass);
vector<Track> tracks(
  // Create 64 identical electrons
  64,
```

```
    Track(
      11,
      -1,
      FourVector(),
      FourVector(momentum,0,0,energy)
    )
  );
  simulator.AddTracks(tracks);

  // Run propagation, get benchmark
  simulator.Propagate();
  double benchmark = simulator.GetBenchmark();

  // Plot xy-plane of energy loss and save to file
  TCanvas c;
  c.SetRightMargin(0.125);
  c.SetLogz();
  TH2F *hist_xy = energy_loss.BuildXYHistogram();
  stringstream title;
  title << "Bremsstrahlung in iron/plastic layers, "
        << "propagated in " << benchmark
        << " seconds.;x [cm];y [cm];Energy [MeV]";
  hist_xy->SetTitle(title.str().c_str());
  hist_xy->SetStats(0);
  hist_xy->GetXaxis()->SetRangeUser(0,200);
  hist_xy->Draw("colz");
  c.SaveAs("Plots/layer_energyloss.pdf");

  return 0;
}
```

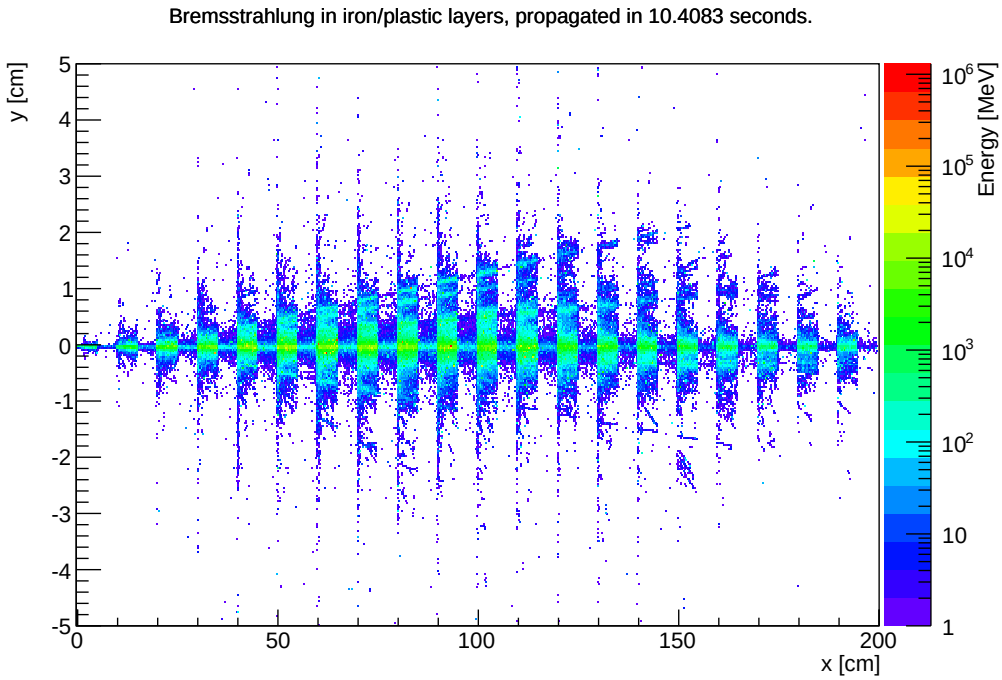

Figure 21: Energy loss in xy-plane for 64 incidents electrons at 50 GeV in alternating layers of iron and plastic.