Parallelization for Fast Image Reconstruction using the Stochastic Origin Ensemble Method for Proton Beam Therapy

REU Site: Interdisciplinary Program in High Performance Computing

Fernando X. Avila-Soto[1], Alec N. Beri[2], Eric Valenzuela[3], Abenezer Wudenhe[4],
Graduate assistants: Ari Rapkin Blenkhorn[4], Jonathan S. Graf[5], Samuel Khuvis[5],
Faculty mentor: Matthias K. Gobbert[5], Client: Jerimy Polf[6]

[1]Department of Computer Science and Mathematics, Muskingum University,
[2]Department of Computer Science, University of Maryland, College Park,
[3]Department of Computer Science, California State University, Channel Islands,
[4]Department of Computer Science and Electrical Engineering, UMBC,
[5]Department of Mathematics and Statistics, UMBC,
[6]Department of Radiation Oncology, University of Maryland School of Medicine

Technical Report HPCF–2015–27, hpcf.umbc.edu > Publications

Abstract

Proton beam therapy is becoming increasingly common in the field of cancer treatment because of the advantages over other forms of radiation therapy. These advantages arise from the finite range of the proton beams, the relatively low dosage of radiation upon entering a patient, and the large spike in dose at the end of the beam range known as the Bragg peak. A new computer code has been developed that uses the stochastic origin ensemble method to reconstruct an image of the gamma radiation produced by the proton beam. The objective of this research is to significantly improve the run time of the given computer code. For the reconstruction algorithm to be useful in medicine, it must be fast and precise, since it is impractical to ask that a patient lie completely still for several minutes. The original C++ code using OpenMP multi-threading on CPUs was ported to hybrid CPU/GPU code using CUDA. It shows very good speedup on the GPU up to the maximum possible number of threads, achieving a 5x speedup over the serial CPU run.

Key words. OpenMP, MPI, GPU, stochastic origin ensemble, proton beam therapy.

AMS subject classifications (2010). 78M31, 90C15, 97M60, 97R60.

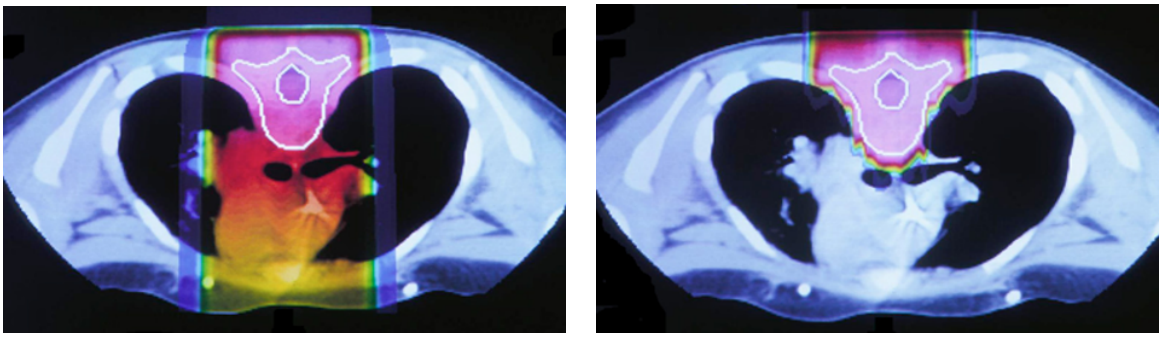
1 Introduction

Radiation therapy is a common technique used in medicine to treat cancerous tumors. It uses high energy X-rays to damage the DNA of cancer cells and stop the cells from reproducing. It is important to note that radiation also damages normal cells; however, normal cells are more capable of repairing themselves [2].

For our research purposes, we are interested in external radiation therapy, in which the radiation beam is administered from outside the body. Conventional forms of radiation treatment may include using X-ray beams. X-rays are a form of photon radiation that use high-energy rays to disrupt cancer cells [2]. Figure 1.1 demonstrates a comparison between X-ray beams and proton beams. X-ray beams have the potential to penetrate through the entire body, damaging critical organs such as the heart highlighted in orange/red in Figure 1.1 (a).

The main difference between X-ray beams and proton beams is the ability to avoid healthy tissue and/or critical organs. When treated tissue is irradiated with proton beams, the beam deposits a relatively low dose upon entering the patient, and then the dose rises to sharp maximum, known as the Bragg Peak [2]. Doctors have better control over irradiation of cancerous tissue when using proton beams because they can control the depth of the Bragg Peak based on the energy emitted from the proton beam. This capability provides greater precision and thus greater advantages over other forms of radiation treatment.

In order to fully exploit the advantages of proton radiation therapy, there is a need for a method of verifying the (in-vivo) beam range. In particular, the beam range verification must be fast and precise so that doctors can control the beam in real time [3].



(a) (b)

Figure 1.1: (a) X-ray treatment vs. (b) proton beam treatment.

The remainder of this report is organized as follows: Section 2 describes the importance about using the Stochastic Origin Ensemble (SOE) Algorithm as an effective method for reconstructing images of a proton pencil beam from the data collected by an ideal Compton camera, and it also describes in detail how the SOE algorithm calculates the best estimate of the new gamma origins. Section 3 describes the numerical method, its parallel implementation, the computational environment, and software used. Section 4 contains all results. It starts with examples that show how SOE works. Then uses performance analysis tools on the original implementation of the algorithm to attempt to pinpoint bottlenecks and identify opportunities for improvement in performance. Section 5 explains the best implementation of the code and what further improvements can be made in the future for more accuracy, precision, and speed.

2 Background

2.1 Compton Camera Imaging

The Compton camera has been proposed as a potential imaging tool in astronomy, industry, and medical imaging. A typical Compton camera includes two detectors, each with a different purpose. The first stage scatters the gamma rays, and the second stage detects them. Recently, Compton camera technologies have been achieved using semiconductors and gaseous detectors [1].

This research focuses on using an ideal three-stage Compton camera positioned above and orthogonal to a tissue phantom that is being irradiated by a proton pencil beam. A tissue phantom is any object that is used to emulate a real-life tissue for the purpose of testing the effectiveness of the Compton camera's ability to detect prompt gamma emmisions and to test the effectiveness of the SOE image reconstruction algorithm. Thus a tissue phantom must emit secondary gamma rays when irradiated with a proton beam. These gamma rays reach each stage of the Compton camera, where their positions are detected by the camera and subsequently used as input for the SOE algorithm [1].

2.2 Stochastic Origin Ensemble Algorithm

For this particular problem, image reconstruction is done with a stochastic origin ensemble algorithm (SOE). Gamma radiation originating from the nuclei treated by proton radiation passes through a three-stage Compton camera (CC), depositing energy at each stage. In the first stage, the energy deposit of the scattering gamma ray ΔE_1 is calculated, followed by the energy deposit ΔE_2 at stage two, and finally the scattering angle θ_2 at stage three. These parameters are used to calculate the scattering angle θ_1 at stage 1 and the initial energy of the gamma ray E_0. The calculation is done by formulas outlined by Mackin, Peterson, Beddar, and Polf [3]. At each stage i the CC records the point (x_i, y_i, z_i) where the gamma ray scatters.

This information alone is not enough to achieve an accurate estimate of the origin of a gamma ray originating from the tissue. Rather, given $p = (x_1, y_1, z_1)$, i.e., the scattering point at stage one of the CC, and the scattering angle θ_1, a cone can be formed. We refer to such cones as origin cones. The surface of an origin cone represents the set of all possible paths of the gamma ray that scattered at point p. Furthermore,

the intersection of the cone's surface and the tissue phantom represents the set of all possible origin points of the gamma ray that scattered at p. That is to say, the true origin of a gamma *must be* somewhere on its origin cone.

At the beginning of the algorithm, a 3D tissue phantom is created to represent the tissue being irradiated. This 3D tissue phantom is split up into a collection of bins, each in the shape of a rectangular prism. We refer to the resulting split up histogram as a voxelized histogram. Each of these rectangular prisms is referred to as a voxel and each takes up a certain portion of the tissue phantoms volume. For example, a $4 \times 4 \times 4$ mm^3 cube can be broken up into 8 voxels, each of size $2 \times 2 \times 2$ mm^3. In this scenario, if we let the histogram span the set $\{(x, y, z) \mid x, y, z \in [0, 8)\}$, then we can map the histogram coordinates to $\mathbb{Z}_8$ with the onto function

$$V(x, y, z) = 4 * \lfloor z/4 \rfloor + 2 * \lfloor y/4 \rfloor + \lfloor x/4 \rfloor$$

Each voxel can be "numbered", i.e., assigned a unique integer value between 0 and 7, with $V(x, y, z)$ given any coordinate (x, y, z) within its space. In a sense, a voxel can be thought of as the 3D equivalent of a pixel.

Then, the algorithm proceeds by iterating through the set of cones and selecting a point at random on each individual cone that is both on that cone's surface and within the tissue phantom. Given this set of randomly picked points, an initial probability density estimate (PDE) is formed based on the number of randomly selected points in each voxel of the voxelized histogram of the tissue phantom. The density of a voxel is defined as the number of random points located within the voxel divided by the total number of points located within the entire histogram and is used to provide an estimate of the probability that the true origin of a detected gamma ray picked at random resides within the voxel. Each cone's randomly selected point is simply a guess as to the true origin of the detected gamma from which then cone was formed.

During each iteration of the code, for each cone, the algorithm picks a new random point on the cone's surface and within the tissue phantom. By comparing the density of the voxel containing the current likely origin and the density of the voxel containing the newly picked random point, the algorithm determines whether the newly picked point is more likely to be the true origin of the gamma corresponding to the current cone. If the density of the voxel containing the newly picked point is greater than that of the current likely origin, the algorithm discards the cone's current likely origin and uses the newly picked random point as the cone's likely origin instead. In some instances, the algorithm may use the newly picked point even if the density of the new point's voxel is lower than that of the current point based on a random factor between 0 and 1.

After a preset number of iterations, the algorithm is complete and the set of likely origins $\{r_1, \ldots, r_N\}$ (N being the total number of cones or scattered gammas detected by the CC) is used as an estimate of the set of origins of the detected gamma rays. This 3D set of points is saved as a `.root` file for viewing. CERN's data analysis package ROOT allows the user to view a 3D grid from a variety of angles, among other things.

There are three main input parameters that can affect the results. The first is the voxel probability parameter (VPP), or the size of each voxel of the histogram of the tissue. The tissue in consideration is of volume $62.5 \times 500 \times 500$ mm^3, so the VPP can be any size that divides the volume of the histogram. Smaller VPP sizes should affect the amount of memory required to run the algorithm, as each voxel will consume a fixed amount of memory. The second parameter is the number of iterations. This study will not attempt to find an optimal number of iterations; instead, we assume that this has been evaluated by our client's research team. Finally, the third parameter involves the total number of detected gammas N.

3 Parallel Programming

3.1 Parallelizable Sections

A `pragma omp for` declaration precedes the section of the code that is run on multiple threads. This indicates to the compiler that the upcoming `for` loop consists of computation that can be distributed and performed concurrently i.e., the results of each computation are independent from one another. Another part of the compiler declaration tells the compiler how many threads to use. Within the SOE algorithm implementation given by Dr. Mackin, this number is specified by the runtime environment variable `NUMBER_OF_THREADS`, which is located within a configuration file. The location of this configuration file is specified at run time and contains all the necessary runtime environment variables necessary for this implementation of SOE.

The `for` loop of interest is the section of the code that updates every cone's likely origin/likely origin in a given iteration. After these calculations have taken place in a single iteration, a thread-synchronized critical section follows which updates the PDE.

3.2 Multithreading on a Single Node Using OpenMP

The implementation provided by the client uses the OpenMP framework to improve the performance of the algorithm. OpenMP is a framework that provides an interface to run sections of code in multiple threads on a single node. Multiple threads on the same node share memory on the heap. All threads run within the same process and address space, thus there is no need for communication between nodes as there is when using multiple nodes where each node has its own distinct process and each process has a distinct address space in memory.

OpenMP is implemented using `pragma` directives throughout the code which specify the number of threads to use and to create thread-private instances of certain variables. Each thread receives a distinct copy of these variables to avoid the problems associated with data races. The OpenMP `pragma` statements are placed around the critical sections of the algorithm (initialization, i.e., the non-critical sections, is done on a single thread).

3.3 The Cluster Maya

The UMBC High Performance Computing Facility (`hpcf.umbc.edu`) provides the 240-node cluster maya to the campus community and beyond. The 2013 components of the cluster maya are the 72 nodes with two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory, connected by a quad-data rate (QDR) InfiniBand interconnect. These nodes include 19 hybrid nodes, each with two 60-core Intel Xeon Phi 5510P accelerators with 8 GB of memory. Another 19 hybrid nodes contain two NVIDIA Tesla K20 GPUs each. Each GPU is equipped with 2496 cores and 5 GB of memory.

3.4 Intel Phi

There are three different modes in which a user can run programs with an Intel Phi accelerator: native, offload and symmetric mode. In native mode, the entire program is run directly on the Intel Phi. In offload mode, certain parts of the program are transferred from the CPU to the Phi during run time. In symmetric mode, the program is broken up into portions that run in parallel on the CPU and Phi.

While attempting to compile the SOE code on the Intel Phi, we found that the ROOT libraries were not compatible with the MIC architechture. Initially, we attempted to fix the problem by updating to the latest version of ROOT (6.04), but the problem persisted. As a result here are no results using the Intel Phi accelerator at this time.

3.5 GPU/CUDA

Utilizing the GPUs on the cluster, the CUDA API and runtime environment, we parallelized the portion of the SOE algorithm that performs density estimation and likely origin selection. The 2496 cores on the Tesla K20 GPU allow us to run a massive number of threads concurrently. When a function is called from the CPU to be run on the GPU, the threads on which the function runs are grouped into a grid of thread blocks. Each thread block contains the same number of threads as every other thread block, and each thread block can allocate memory that is shared among all of its threads. In addition, all the threads in a block can be synchronized (all threads will stop at a specific point in the code and wait until all other threads have reached the same point). The number of blocks and threads per block are parameters specified by the user in the function call.

CUDA specifies three different types of functions to accommodate for the different ways of running a function when a user has access to both a CPU (commonly referred to as the host) and a GPU (commonly referred to as the device).

The first type is the `__host__` function. This is any standard C/C++ function that runs on the host. Functions written the same way as normal C/C++ functions within CUDA code are implicitly considered

`__host__` functions. In addition, the user can explicitly specify a function to be runnable only on the host by using the `__host__` keyword.

The second type is the `__global__` function. These functions are commonly referred to as kernels (not to be confused with the component of an operating system). Kernels are called from the host and run on the device, usually on more than one thread. Whenever a kernel is called, it takes two additional parameters: the number of threads in each thread block, and the number of thread blocks on which to run the function. On the current architecture of maya, only a maximum of 1024 threads can be run on a single block, but multiple blocks can be run up to a certain limit. The NVIDIA K20 GPU can run up to 13 thread blocks at a time. Within the kernel, CUDA runtime variables can be accessed such as the thread ID and block ID of the thread in which the function is being run and the number of threads as well as the number of blocks.

The third type is the `__device__` function. Device functions are called from the device and run on the device. For our purposes, `__device__` functions are virtually identical to `__host__` functions other than the location from which they are called and the location on which they are run. This lends itself to a fourth type of function, `__host__ __device__`. CUDA allows functions to be specified as being able to run directly from both the host and the device provided that every function called within the `__host__ __device__` function is also a `__host__ __device__`. Using the `__host__ __device__` specification allowed us to easily revamp much of the code (which was completely `__host__` code by default) to be run on the device as we saw fit.

To port the code to CUDA, we first copied each `.cc` file to a `.cu` so that we could rewrite any of the source code using the CUDA API as well as compile the code using NVIDIA's nvcc compiler. Then, we used the CUDA API to perform all of the memory management necessary to load the problem onto the GPU. The CUDA API provides many C-like functions for these purposes, such as `cudaMalloc`, `cudaMemcpy`, `cudaMemset`, and `cudaFree`. Once we were able to load device-versions of the density matrix and origin cone data onto the device, we converted the critical section of the algorithm to a CUDA kernel (we use the term critical section here to refer to the portion of the algorithm's code that takes up the most time. However, as a `pragma omp` declaration keyword, it refers to a section of the multi-threaded code that cannot be run concurrently, i.e., only one thread can run the section of code at any given time. These two are not to be confused).

To allow the entire critical section to be run on the device, we had to eliminate the usage of all C++ STL classes in that portion of the code. This is because all such classes are considered `__host__` code by CUDA and thus will not be run on the device. In addition, any source code provided to us that was used in the critical section had to be rewritten as code runnable on the device. For the most part, this entailed labelling the necessary functions as `__host__ __device__` functions. To eliminate the use of C++ STL classes, we rewrote the STL's `valarray` class with a limited subset of the functionality necessary to perform the same functions used by the `valarray` in the code given to us. All of the code in our version of the `valarray` is `__host__ __device__` code.

Once we had successfully revamped the code to be runnable on the device, we had to map individual threads to a certain portion of tasks. In particular, we mapped each thread to N/t cones with N as the number of cones and t as the number of threads. Thus, in theory, the code should optimally be t times faster than a serial device run when run with t threads.

3.6 MPI

To run an algorithm on multiple nodes that produces similar results as the implementation provided by Mackin, Peterson, Beddar, and Polf, each node would have to store a separate, complete copy of the entire histogram that contains all the likely origins of gammas and from which a PDE is formed. This is because the algorithm must have access to the PDE in its entirety to be able to decide whether a newly-picked random point should replace a cone's current likely origin.

Suppose the algorithm were run on two separate nodes and the histogram was "split" down the plane $x = 0$. That is, the first node stores all likely origins with $x < 0$ and the second node stores all likely origins with $x \geq 0$. Let there be a cone with its current representation point/likely origin being stored in the first node, but some of its surface contains points with $x \geq 0$. Furthermore, suppose the algorithm picks a random point on the cone's surface and within the histogram with $x \geq 0$. Then the first node does not have access to the likely origins within the voxel containing the newly picked random point and thus it cannot

calculate the density of that voxel (the density is simply the number of points in that voxel divided by the total number of points). Without this information, the algorithm cannot decide whether this newly picked point should replace the current likely origin.

Thus we cannot split the histogram or the set of cones across multiple nodes, which consume the majority of the memory that the algorithm needs, so we cannot use MPI to solve problems that consume any more memory than fits on a single node. However, we can use MPI to potentially decrease run time similar to the way mulithreading has already been used to speedup the code. This requires that a separate, complete copy of the histogram is loaded onto each node and at the end of each parallelized iteration, every update of the histogram that has been made in each node is "sent" to every other node and performed on each node-specific copy of the histogram.

4 Numerical Results

4.1 Experimental Design

In this subsection, we demonstrate the application results of various tests using the original version of the code.

During initialization, the application prints various parameters describing the histogram including the voxel size, matrix dimensions, and number of prompt gammas to be used in reconstruction. The program outputs the run time at various iteration increments (initially this happens every 100 iterations, but this interval increases after a certain number of iterations) and saves intermediate snapshots of the reconstructed image in `.root` files at these increments based on the set of likely origins at that point in time. At the end of each run, the program saves the final `.root` file of the reconstructed image and outputs the total run time.

We ran the original code with two different `.root` input files that each represent the setup of a particular application of proton beam therapy and a fixed set of parameters for the SOE reconstruction algorithm contained within a configuration file. The first input file contains $\approx$ 15,000 gammas, while the other contains $\approx$ 380,000 gammas. Both problems are set to perform 1,000 iterations.

Each of our runs only considers gammas within the volume of a 500 mm $\times$ 500 mm $\times$ 62.5 mm voxelized histogram that represents the tissue phantom. Voxels are simply the three-dimensional equivalent of pixels. They can be described as tiny boxes which together form a discretization of the histogram. While the actual size of the phantom is fixed among all runs, we vary the number of voxels (and by extension the voxel size itself) that make up the histogram to generate different image resolutions. We consider here a standard definition (SD) case with voxels of a $\approx$ 40 mm^3 volume and a high definition (HD) case with voxels of a $\approx$ 1 mm^3 volume.

Figure 4.1 and Figure 4.2 report application results for the large large file with $\approx$ 380,000 gammas. Figure 4.1 uses the standard definition (SD) with voxels $\approx$ 40 mm^3 in volume, while Figure 4.2 uses high definition (HD) with voxels $\approx$ 1 mm^3 in volume. The individual plots in Figure 4.1 and Figure 4.2 show the reconstructed image after 0, 100, ..., 900 iterations of the SOE algorithm. Every run of the SOE algorithm starts with a mass of points in space scattered over the domain. These points are the initial, random guesses of the true origins of each gamma detected by the CC. We refer to each of these points as a likely origin. During each iteration, the algorithm decides whether to relocate each likely origin to a location where it is more likely to have originated. In this way, with each iteration, each gamma's likely origin moves closer and closer to its true origin.

Each `.root` file also comes with a separate `.root` file containing what is considered the true origins of each detected gamma. These true origin files generated by a Monte Carlo simulation and are used to compare the SOE reconstructed images. Figure 4.3 shows a Monte Carlo generated true origins image Figure 4.3 (a) side by side with SOE recontructed images Figure 4.3 (b) and Figure 4.3 (c). In particular Figure 4.3 (b) shows results for the SD resolution and Figure 4.3 (c) shows results for the HD resolution applied to the 380,000 gammas case. This figure shows a much sharper image with the HD resolution as compared to the SD resolution. The SOE reconstructed HD image displays very similar structure to the true origins.

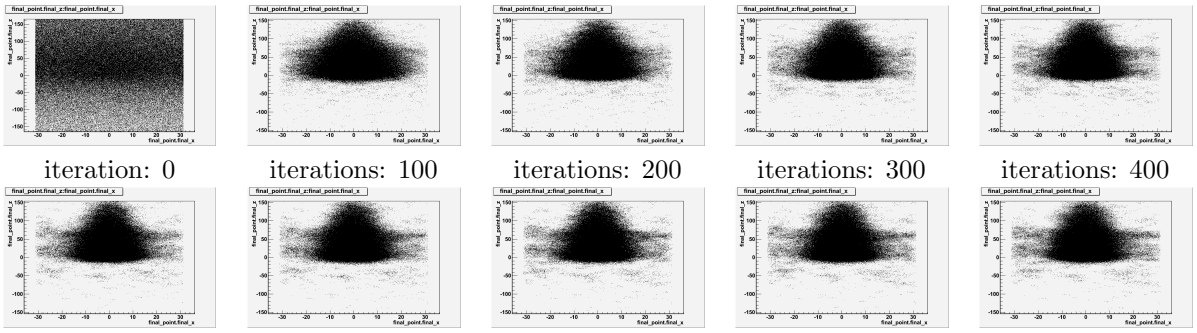


Figure 4.1: SOE image reconstruction of 380,000 gammas (SD) in increments of 100 iterations.

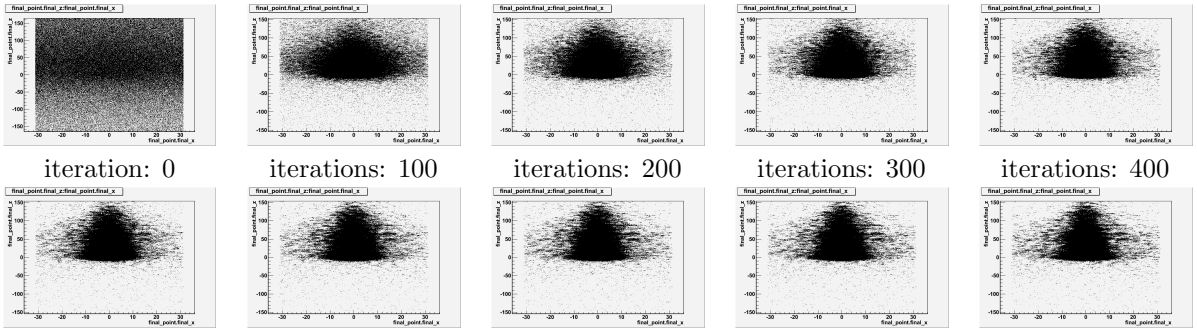


Figure 4.2: SOE image reconstruction of 380,000 gammas (HD) in increments of 100 iterations.

4.2 Performance Analysis using TAU

TAU is a profiling and tracking tool kit that conducts performance analysis of parallel programs written in Fortran, C, C++, Java, Python. We executed our program with the TAU executable wrapper.

We demonstrated the performance results of the provided C++ source code using TAU's Performance System. TAU's traditional performance measurement and analysis work flow gathers performance information through instrumentation of functions and methods. With these results, we were able to determine which parts of the code take the most time.

When the program is executed with the TAU performance analysis, a profile for the run is created. These profiles indicate the run time of specific functions and what percentage of the total run time each program takes. When run, TAU creates a plot for each thread that was used when the SOE algorithm creates the reconstructed image. In Figure 4.4, each thread has a bar containing several colors with each color representing a portion of the code. When selecting a single thread as shown in Figure 4.4, we see that,

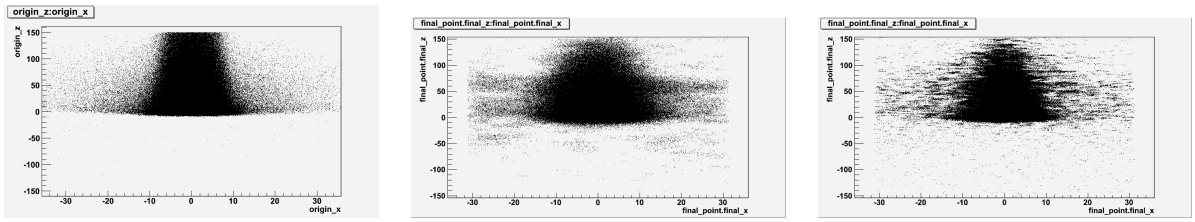


Figure 4.3: (a) Monte Carlo estimation vs. (b) SOE reconstruction SD vs. (c) SOE reconstruction HD of prompt gamma origins using 380,000 gammas. SOE reconstruction was done with 1,000 iterations.

Figure 4.4: TAU profiler showing the visual interpretation of the code and how each major function took on each thread on the same node. Each color represents a different function in the code.

along with the TAU application, the `StochasticReconstructor::calculateDensity` takes a majority of the total time for the code to run. Looking deeper into this function, we found that this function performs the bulk of the algorithm; it iterates through the set of cones relocates each cone's likely origin if appropriate, and updates the histogram according to the new locations. This was the same section of the code surrounded by `pragma omp` statements, indicating that this section of the code had been parallelized by the client and that it was most likely the section of the code taking up most of the run time. TAU confirmed this suspicion.

4.3 Timed Performance Analysis

Our base case of tests and times were done with the varying number of threads and `.root` input files provided by Dr. Polf as shown in Table 4.1. In particular, we used a small input file with $\approx$ 15,000 detected gammas and a large input file with $\approx$ 380,000 gammas and numbers of threads ranging from 1 to 32. The performance studies in this section use 1,000 iterations and standard definition (SD) with voxels ≈ 40 mm^3 volume.

One of the main metrics to gauge performance is speedup. This refers to the improvement in run time over a 1-thread run when a larger number of threads is used. We define the speedup of a run with a given number of threads as the run time of the multi-threaded run divided by the run time of the serial (single-threaded) run. That is, if $W_t(n)$ denotes the wall clock time of a run of SOE with n gammas as input using t threads, we define speedup as $S_t(n) = W_1(n)/W_t(n)$ on the CPU. Another important metric is efficiency, which tells us how well each individual thread is performing. As the number of threads increases, efficiency tends to decrease. We formally define efficiency $E_t(n) = S_t(n)/t$ on the CPU.

Table 4.1 shows the speedup of run time when various numbers of threads are used. For our test case,

Table 4.1: Wall clock times, speedup, and efficiency of 1,000 iterations of the SD SOE algorithm with the small (15,000 gammas) and large (380,000 gammas) files as input with various numbers of threads on a single node with two eight-core CPUs. These runs were performed using ASH (Averaged Shifted Histograms) for density estimation.

(a) Wall clock times W_t (seconds)						
	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Small	287	169	89	49	27	28
Large	415	197	104	57	33	36
(b) Observed speedup S_t						
	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Small	1.00	1.70	3.22	5.86	10.39	10.04
Large	1.00	2.10	3.99	7.26	12.25	11.45
(c) Observed efficiency E_t						
	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Small	1.00	0.85	0.80	0.73	0.65	0.31
Large	1.00	1.05	1.00	0.91	0.77	0.36

Table 4.2: Wall clock times, speedup, and efficiency of 1,000 iterations of the SD SOE algorithm with the small (15,000 gammas) and large (380,000 gammas) files as input with various numbers of threads on a single node with two eight-core CPUs. These runs were performed without ASH.

(a) Wall clock time W_t (seconds)						
	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Small	40	22	13	8	6	6
Large	955	513	286	163	101	108
(b) Observed speedup S_t						
	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Small	1.00	1.82	3.08	5.00	6.67	6.67
Large	1.00	1.86	3.34	5.86	9.45	8.84
(c) Observed efficiency E_t						
	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Small	1.00	0.91	0.77	0.63	0.42	0.21
Large	1.00	0.93	0.84	0.73	0.59	0.28

we only used powers of two for the number of threads (1, 2, 4, 8, 16, 32). Each compute node on maya contains two CPUs with 8 cores each for a total of 16 cores. Thus, for runs on the CPU, we expected to see an increase in speedup as the number of threads increased from 1 to 16. As expected, a bottleneck was reached at 32 threads and the speedup decreased from 16 to 32 threads. This is because 32 threads cannot truly be run concurrently if only 16 cores are available.

The original runs in Table 4.1 were performed using ASH (Average Shifted Histograms, which requires the usage of multiple histograms) for density estimation, and other runs were performed without ASH, i.e., only using a single histogram for density estimation. Our hybrid CPU/GPU implementation is not able to use ASH, so our hybrid CPU/GPU results always use only a single histogram and should only be compared against CPU results that do *not* use ASH either. Thus, Tables 4.2 and 4.3 provide a comparison of CPU and hybrid CPU/GPU results without the use of ASH.

Table 4.2 shows the run time of the SOE algorithm on the CPU and compares the speedup time small and large file sizes. Using 1 thread as the control and comparing other runs, we see speedup as we increase the number of threads up to 16 threads. When we hit 32 threads, as previously observed, there is a decrease in efficiency.

For our hybrid CPU/GPU runs in Table 4.3, we start with a base case of 512 threads. This is done because the maximum number of threads that can be run on the GPU that is a power of two is 8192, and using 512 as the base case gives us 5 different run times for comparison with the 5 CPU run times with the number of threads in the range from 1 to 16 (as well as being powers of 2) in Table 4.2. In addition, this gives us two run times that use more threads than total number of 2496 cores available on the GPU, allowing us to conclude whether or not hyperthreading can be effective on the GPU. Thus, we define speedup on the GPU as $S_t = W_{512}/W_t$ and efficiency on the GPU as $E_t = S_t/(t/512)$.

Table 4.3 shows that run times decrease with increasing numbers of threads. This decrease continues all the way to the largest number of 8192 threads in the table, which confirms that hyperthreading is effective on the GPU. Comparing Tables 4.2 and 4.3, we see in the first column that the GPU runs are slower than the CPU runs. This is not indicative of any problem, since the choice of 512 threads is artifical, as explained above. Rather, on a massively parallel device such as a GPU, it is natural to use as many threads as possible at all times, as confirmed to be effective above for 8192 threads. Therefore, using the CPU implementation with 1 thread as the comparison, we see a speedup of the 8192 threads GPU run over the serial CPU run of about 4x.

However, using strictly powers of 2 did not allow us to use the maximum number of threads that can be run on the GPU. Using CUDA's maximum number of threads per block (1024), we ran the algorithm using every number of blocks in the range of 8 to 15 (using 16 blocks with 1024 threads per block resulted in an error). We found that speedup increased from 8 to 13 blocks and then sharply decreased when using more than 13 blocks. This coincides with the fact that the Tesla K20 GPU is composed of 13 Streaming Multiprocessors (SM), each containing 192 cores. Using a greater number of thread blocks than SMs in the

Table 4.3: Wall clock times, speedup, and efficiency of 1,000 iterations of the SD SOE algorithm with the small (15,000 gammas) and large (380,000 gammas) files as input with various numbers of threads on one GPU of hybrid CPU/GPU node. These runs were performed without ASH.

(a) Wall clock time W_t (seconds)					
	512 threads	1024 threads	2048 threads	4096 threads	8192 threads
Small	93	60	38	25	15
Large	2061	1245	660	382	229
(b) Observed speedup S_t					
	512 threads	1024 threads	2048 threads	4096 threads	8192 threads
Small	1.00	1.55	2.45	3.72	6.20
Large	1.00	1.66	3.13	5.40	9.00
(c) Observed efficiency E_t					
	512 threads	1024 threads	2048 threads	4096 threads	8192 threads
Small	1.00	0.78	0.61	0.47	0.39
Large	1.00	0.83	0.78	0.67	0.56

Table 4.4: Run times of the hybrid CPU/GPU code to reconstruct an image with $\approx$ 380,000 gammas, performing 1,000 iterations, using CUDA thread blocks with 1,024 threads per block and various numbers of thread blocks

Wall clock times (seconds)							
8 blocks	9 blocks	10 blocks	11 blocks	12 blocks	13 blocks	14 blocks	15 blocks
251	237	226	214	204	196	265	257

GPU and the maximum number of threads per block will result in a slower run time than a run using 8 thread blocks each with the maximum number of threads. The fastest run time we achieved on a CPU/GPU run with the large file size (380,000 detected gammas) was 196 seconds using 13 thread blocks each with 1,024 threads for a total of 13,312 threads. Compared to the serial CPU run time of 955 seconds, this is a $\approx$ 5x speedup.

5 Conclusions and Future Work

By making use of the CUDA API and runtime environment, we were able to successfully port the code given to us by Mackin, Peterson, Beddar and Polf to a hybrid CPU/GPU architecture. Our implementation can run on any CUDA-compatible device (any NVIDIA GPU with the sufficient compute capabilities). We found that it is more efficient to run the entire algorithm on the GPU as a single kernel performing every single iteration than to have a separate kernel call for each iteration. This may be due to the overhead incurred during the creation and initialization of thread blocks and grids on the GPU. We did not formally verify this result.

We were able to achieve a maximum speedup (when comparing to the serial CPU run) of $\approx$ 5x by using a total of 13,312 threads on the GPU (13 thread blocks each running the maximum number of threads per block, 1024). We also found that increasing the number of threads that the hybrid code runs on results in a very good speedup. Increasing the number of threads past the number of total of available cores (2496) still results in a substantial speedup, unlike the behavior that occurs on a compute node when the number of threads is greater than the number of available cores.

Our hybrid implementation is not optimal. This code may be able to be optimized further to the point where it can be run on the cluster maya with better performance than an OpenMP run with 16 threads on a single node. Moreover, our implementation cannot use average shifted histograms (ASH) for density estimation; the user is currently restricted to using a single histogram for estimation. The use of ASH could be added to the functionality in the future.

Furthermore, future research could allow the code to be run on the Intel Phi accelerator. The key

roadblock is that the ROOT library has not yet been ported to the Phi. The Intel Phi accelerator has 60 cores and allows for hyper-threading. Through hyper-threading, each core can run 4 threads at a time, which means that a total of 240 threads can be run on the code simultaneously. One way to implement multi-threading on the Intel Phi is through the use of the OpenMP framework, which can be used in conjunction with MIC architecture.

Acknowledgments

These results were obtained as part of the REU Site: Interdisciplinary Program in High Performance Computing (hpcreu.umbc.edu) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2015. This program is funded by the National Science Foundation (NSF), the National Security Agency (NSA), and the Department of Defense (DOD), with additional support from UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). HPCF is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from UMBC. Co-author Abenezer Wudenhe was supported, in part, by the UMBC National Security Agency (NSA) Scholars Program through a contract with the NSA. Graduate assistants Ari Rapkin Blenkhorn, Jonathan Graf, and Samuel Khuvis were supported during Summer 2015 by UMBC.

References

[1] Andriy Andreyev, Arkadiusz Sitek, and Anna Celler. Fast image reconstruction for Compton camera using stochastic origin ensemble approach. *Med. Phys.*, 38(1):429–438, 2011.

[2] Ringo Doe. All About Proton Therapy, August 2009. http://www.oncolink.org/treatment/article.cfm?c=186&id=433, accessed August 6, 2015.

[3] Dennis Mackin, Steve Peterson, Sam Beddar, and Jerimy Polf. Evaluation of a stochastic reconstruction algorithm for use in Compton camera imaging and beam range verification from secondary gamma emission during proton therapy. *Phys. Med. Biol.*, 57(11):3537–3553, 2012.