Automated Domain Decomposition for Multi-GPU Monte Carlo Electron-Photon Radiation Transport

By

Vanessa Goss

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Nuclear Engineering

in the

Graduate Division

of the

University of California, Berkeley

Fall 2023

Automated Domain Decomposition for Multi-GPU Monte Carlo Electron-Photon Radiation Transport

Abstract

Automated Domain Decomposition for Multi-GPU Monte Carlo Electron-Photon Radiation Transport

by

Vanessa Goss

Doctor of Philosophy in Engineering - Nuclear Engineering

University of California, Berkeley

Professor Raluca Scarlat, Chair

A Convolutional Neural Net (CNN) was trained to determine the optimal decomposition of a voxel domain for Monte Carlo electron-photon radiation transport. The training database was developed by collecting photon flux and surface current tally data for a simple shielding problem and a simplified human phantom brain model. The voxel matrix inputs were mapped to the tally outputs by the CNN, allowing the CNN to accurately predict tally results from material and source data. The predicted flux was then used to determine the shape and size of the subdomains, and to calculate the size of the ghost zones. The intent for the CNN is to reduce the amount of trial and error that is often necessary to decompose domains manually by a user. Poor domain decomposition can lead to longer run times. Furthermore, this work is meant to serve as a proof of concept for more complex geometry types and applications. The CNN predicted decomposition performed at best 1.6x faster than other decomposition schemes for the shielding problem and 1.3x faster for the human phantom problem. This process was done to demonstrate the power of intelligently choosing subdomain boundaries rather than using arbitrarily chosen boundaries.

To my friends, family, and UAW siblings.

Contents

List of Figures

List of Tables

Acknowledgments

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DENA0003525. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Chapter 1

Introduction

Predicting the effects of radiation as it moves through and interacts with matter is essential to assess the safety and security of radioactive sources and experiments. Generating energy deposition and dose data from these sources can be done with handheld survey meters and other radiation detectors. However, this introduces unnecessary risk and is limited by radiation safety and economic considerations. Computational methods serve as a safer, more cost effective alternative. Deterministic, Monte Carlo, and hybrid, which combines deterministic and Monte Carlo, are the categories of methods available to calculate solutions to the Boltzmann transport equation, the equation that mathematically describes radiation transport. Each method has different features and errors making them attractive for various applications. The work presented in this dissertation is focused on Monte Carlo methods for continuous-energy, coupled electron-photon transport.

Monte Carlo is a unique computational method that directly simulates individual electron and photon lifetimes or *histories*. Each particle's lifetime is tracked by a series of interactions with the surrounding material, these interactions produce secondary particles and deposit energy until the initial particle is absorbed or leaves the problem's domain. With a large enough number of histories, macroscopic behavior can be determined by averaging the individual interactions, called *tallies*. Monte Carlo is typically treated as the gold standard of computational methods for regulatory purposes because it is continuous in energy, space, and angle; additionally, MC can be used with general, more complex geometries. However, Monte Carlo is computationally expensive because it converges sub-linearly and requires a large amount of memory to store the nuclear data and domain necessary to perform the simulation. Typically, Monte Carlo simulations need to be parallelized and run on supercomputers to produce results in a reasonable amount of time. Demand for larger, more detailed problem is expected to increase as more computational power becomes available.

Recent supercomputing trends indicate that the future architectures will become increasingly parallel and dependent on general purpose graphic processing units (GPUs). GPUs have higher number of floating point operations per second (FLOPs), higher memory bandwidth, and lower energy required per FLOP, compared to existing CPUs. GPUs are inherently massively parallel and are capable of executing thousands of threads at once. These threads are also very lightweight, meaning that the cost of transferring memory between two threads is less costly than a CPU data transfer. Current and planned supercomputers by the Department of Energy (DOE) suggest that supercomputers will continue to be built wiht a set of compute nodes connected by a high speed network, and that whose compute nodes will increasingly rely on GPU technology. Monte Carlo software must either be ported to the GPU or rewritten entirely to optimize performance. The code in this dissertation is a new Monte Carlo code that seeks to mitigate the limitations of GPU processing.

Monte Carlo is embarrassingly parallel because each electron/photon history and interaction is independent from one another, making parallelization trivial on a CPU. Monte Carlo threads are task based; each thread can simulate a single particle's history without interacting with other threads. However, it is necessary that each thread can execute their own set of instructions because not every particle lifetime experiences the same set of interactions in the same order. GPU threads are required to execute the same instructions in the same order, concurrently. If the GPU threads cannot execute the instructions concurrently, the instructions will become serialized, creating slowdowns in runtime. Individual electron and photon histories have the potential to diverge dramatically due to the differences in the fundamental physical processes that drive interactions. Traditionally, Monte Carlo codes use an algorithm that follows a particle from birth to death and simulates each individual particle interaction, this is called history-based Monte Carlo. Therefore to get optimal performance on the GPU, electron-photon Monte Carlo must move away from traditional history-based algorithms and instead write event-based algorithms which would group compute nodes based on instruction performed. For event-based Monte Carlo, particles are transferred to the compute node responsible for simulating the event that is experienced by the particle. However, data transfers are also expensive; data transfer and thread divergence are two processes that govern Monte Carlo runtime on modern computing architectures.

Another key difference between a GPU and CPU is memory; in general, the GPU has a lower memory capacity and a higher memory bandwidth, but also higher latency than a CPU due to lack of multiple levels of caches. GPU memory is stored either on or off-chip. The location of the memory determines the capacity and latency, off-chip memory has a higher latency and capacity and on-chip memory has a lower latency and capacity. Monte Carlo is a memory intensive process. The nuclear data needed to simulate particle interactions and the problems domain both require large amounts of memory. A solution to this is to break up the domain into smaller, more homogeneous, subdomains that are then dispersed across compute nodes. Decomposing the domain both reduces the memory burden, making it possible to run larger problems.

Ideally, a domain decomposition scheme would break up the domain in a manner that would minimize the number of data transfers necessary to complete a MC run, as particles move from one domain to another. It should also minimize load imbalance by placing an equal amount of work on each processor, but this can difficult to determine in advance as the work per particle can very based on its trajectory. However, writing a function to perform this task optimally on a general geometry is nearly impossible due to coupled electron-photon physics. Existing domain decomposition techniques, in neutron Monte Carlo codes, use mean free paths to determine the size and shape of subdomains. This approach cannot be used for electron-photon Monte Carlo because the mean free path of electrons and photon can differ by an order of magnitude. Another notable phenomena of electron-photon physics is electron cascades. Electron cascades that occur near a subdomain boundary can cause massive runtime slowdowns.

A potential solution to this dilemma is to use Machine Learning to predict locations of minimal boundary crossing and parts of the domain where large amounts of computational work is performed. This map of pertinent features can then be used to generate subdomains that are not overburdened by data transfers. The results from these attempts will be explored and analyzed in future chapters of this dissertation. Features, such as domain decomposition, make the new continuous-energy electron-photon coupled Monte Carlo code competitive with existing Monte Carlo codes.

1.1 Previous Work: Monte Carlo Methods and the GPU

The basis for GPU Monte Carlo was research done by Martin and Brown in 1984 [10]. This paper developed a method for mapping the traditional neutron Monte Carlo problem onto SIMD vector computers. Vector computers share a similar processing model with GPUs. Neutrons are banked into vectors based on the calculations required to simulate whatever given interaction the neutron is undergoing. For example, if the particle is causing a fission interaction, it is placed into the fission buffer. As the buffers fill, they are executed by the vector computer. The buffers are then ”shuffled” to reflect the next interaction experienced by each particle and the data is moved back into contiguous blocks based on the reaction. The memory is grouped in such a way to optimize computational efficiency. This new method was coined ”event-based” Monte Carlo, as opposed to ”history-based” Monte Carlo [10].

Various universities and national labs have begun the process of porting or rewriting their existing Monte Carlo codes for the GPU. The WARP code was the first to explore continuous energy neutron transport on GPUs [5]. WARP was developed at UC Berkeley by Ryan Bergmann and had limited features due to its status as a research code. It had limited cross sections, no variance reduction, and limited tally options. WARP also runs only on a single GPU. Bergmann based their approach to event-based Monte Carlo on the work done by Brown and Martin and Martin and Vujic [39].

SHIFT was the first production level continuous-energy neutron code to adapt CPU transport routines for the GPU [19]. Their implementation includes most of the existing tally features and all of the physics capabilities of the CPU version of SHIFT. They also developed an event-based Monte Carlo algorithm and compared it to the history based approach. It was tested on an idealized and simplified small modular reactor core. Additionally, it was tested on the depleted core model. All testing was done on the Titan Cray XK7 and Summit, two supercomputers located at Oak Ridge National Lab. They found a speed up of 10.7x on the V100, indicating that the floating point performance is better on the GPU. Domain decomposition is implemented in the CPU and GPU version of SHIFT to mitigate memory issues with large problems [19].

Lawrence Livermore National Laboratory has also been porting their production level codes, Mercury and Imp, to the GPU and testing them on NVIDIA Tesla V100s for the Sierra supercomputer [35]. Mercury is a Monte Carlo code that can simulate neutrons, gammas, and light charged particles in a mesh or 3D combinatorial geometry. Imp is a newer implicit Monte Carlo code that simulates thermal x-rays. They share similar infrastructure which has made porting to the GPU easier. They implemented a version of event-based transport that executes 9 separate kernels to calculate the distance to event and sample the events. They have achieved a speedup of about 7.61x for neutrons and 5.81x for thermal photons in GPU-to-CPU comparisons on ATS-Sierra, an existing supercomputer with NVIDIA GPUs. They tested their methods on Godiva in water and Crooked Pipe, two standard benchmarks problems in neutronics and thermal photonics. They are currently working on porting their codes to El Capitan, a future supercomputer that uses AMD processors and GPUs [35].

Bleile ported a research code developed at Lawrence Livermore National Lab to the GPU to compare the difference in performance of history-based and event-based Monte Carlo [6]. They found that the event-based methodology performs better on the GPU, concluding that, in general, reducing kernel complexity has significant impact on final run time. They also recommend implementing a tally server to handle tallies, which typically require larger amounts of memory, a scarce resource on the GPU.

PRAGMA is a GPU-based continuous-energy neutronics Monte Carlo Code [12]. They implemented an event-based tracking algorithm and compared it to the history based implementation. Additionally, they explored GPU-specific techniques such as the use of built in vector types and atomic operations. Using a combination of these techniques, they were able to overcome the previous drop in performance in depleted fuel calculations. They concluded that it is possible to improve the performance of the history-based approach with minimal algorithmic modifications and performs better than event-based methods on fresh fuel. History-based Monte Carlo overall performs better on the GPU than the CPU.

Currently, all of the attempts to rewrite Monte Carlo codes have been focused on neutrons or thermal photons, except CHEETAH-MC [7], a production level code in early stages of development at Sandia National Laboratories for coupled electron-photon transport. It is being built from the ground up to provide similar capabilities to the Integrated Tiger Series (ITS) [29]. The work presented in this dissertation was developed within the CHEETAH-MC framework on a separate research fork.

Machine Learning applications to Monte Carlo are being explored by various research groups. However, there are no current publications that apply Machine Learning to electron-photon domain decomposition. Mote, from NC State, has some preliminary results for applying Machine Learning to the domain decomposition scheme in SHIFT, specifically for neutrons [30].

Outline

- **Chapter 2** will cover the Monte Carlo algorithm along with the statistics and sampling necessary to gather tally data. Monte Carlo geometries and domain decomposition are also discussed in this chapter. Finally, coupled electron-photon physics is described.
- **Chapter 3** analyzes the GPU's architecture and its limitations and strengths in comparison to a CPU. It will also review Machine Learning and describe the neural net that was chosen for the work in this dissertation.
- **Chapter 4** discusses the development of the domain decomposition methodology and how each stage of the algorithm improvements was developed and implemented.
- **Chapter 5** presents the results of the different domain decomposition techniques attempted on different GPU architectures, highlighting the memory usage and time spent transferring data between the nodes.
- **Chapter 6** draws conclusions about the results from the previous chapters along with recommendations for future studies.

Chapter 2

Monte Carlo Transport

This chapter explains coupled electron-photon theory and statistics that is goes into Monte Carlo methods for radiation transport. It covers the basic Monte Carlo algorithm, including the underlying sampling schemes. Next, different Monte Carlo geometries are described along with domain decomposition. Then, basic electron-photon interactions are diagrammed and potential challenges that the physics introduces when implementing domain decomposition.

2.1 The Monte Carlo Method

There are two general approaches to solving the Boltzmann transport equation, deterministic methods and Monte Carlo. Deterministic methods solve the equation directly, by first discretizing the problem's time, space, angle, and energy. Then the electron or photon flux is calculated at each point. The Monte Carlo method is distinct from the deterministic approach because it simulates individual particle tracks. The algorithm uses experimental cross sections to simulate particle interactions as individual source particles stream through the problem space. The sum total of these interactions is called a particle history.

Each particle history is a fully independent random walk through the geometry and can be mapped to a single computer thread. The thread, or particle history, uses a random number generator to sample probability distributions to determine the particle interactions that occur as well as the particle's state following the interaction. A few assumptions must be made in the Monte Carlo algorithm [36]. First, it is assumed that all particles travel in straight lines between interactions. Second, all particles are treated as points, meaning they do not occupy physical space within the geometry. Finally, all particle interactions occur instantaneously in time and space. Time is not a factor in time-independent Monte Carlo simulations.

Figure 2.1 illustrates a simple particle history in a 3-dimensional Cartesian space. The X's are particle interactions, some producing secondary particles, and the straight lines between the interactions indicate the path the particle moves along. When a particle crosses a material boundary or undergoes an interaction with a material, the distance to the next material boundary or particle interaction is sampled again. The details of the sampling schemes are discussed in depth in the Statistics subsection.

The main advantages of using a Monte Carlo algorithm instead of a deterministic method to predict electron-photon flux, dose, and energy deposition is that there very few assumptions that need to be made. Therefore, the problem's geometry and particles energy can be treated as fully continuous. This allows for a more accurate representation of the cross sections of the materials. Additionally, because the geometries used in Monte Carlo methods are continuous, they are able to be more complex and reflective of experimental setups. Monte Carlo geometries are discussed in greater detail in Section 2.2.

However, the drawbacks of the Monte Carlo algorithm cannot be ignored. Monte Carlo methods converge sublinerarly, $1/\sqrt{N}$, where N is the number of particle histories, can lead to long computational runtimes. The convergence rate is discussed further in the Statistics subsection. The long runtimes can be mitigated by high performance computing. As mentioned earlier, particle histories are fully independent from one another. Particle-particle interactions are so exceedingly rare, they can be ignored at most energy regimes. This makes the algorithm embarrassingly parallel, meaning that parallelizing the algorithm is trivial on the CPU. Each history is assigned to a computational thread and the number of threads depends on the computer architecture available. This scaling can be used to reduce the variance on a more acceptable timescale.

Another concern is that often times in Monte Carlo applications work, the region of interest is small and particles can miss the region entirely if the number of histories is not large enough to cover the entire space. This is very common in detector and shielding applications. Radiation detectors can have a region of interest on the scale of millimeters and can be used to survey entire buildings.

The Algorithm

History-based Monte Carlo requires tracking individual particles as they move and interact with a predetermined geometry and materials. To simulate these interactions, first the distance to interaction must be sampled using techniques described in Section 2.1. Then, the distance to the nearest material boundary is calculated using the distance formula in the appropriate coordinates. These two values are compared and the shortest distance is chosen, the particle is then moved to that location and the reaction type and post interaction is state is determined through additional statistical sampling [9]. This process is repeated N times, where N is established by the user at the beginning of the loop. Particle interactions may also produce secondary particles, that must be stored and transported after the initial N source particles. After all primary and secondary particles are transported, the results of interest are combined using statistical methods explained in Statistics subsection. These results are called tallies and are discussed in greater detail in the Tallies subsection. Algorithms 1 describes the lifetime of an individual particle in pseudo code and Algorithm 2 describes the outer most transport loop that combines individual particle histories.

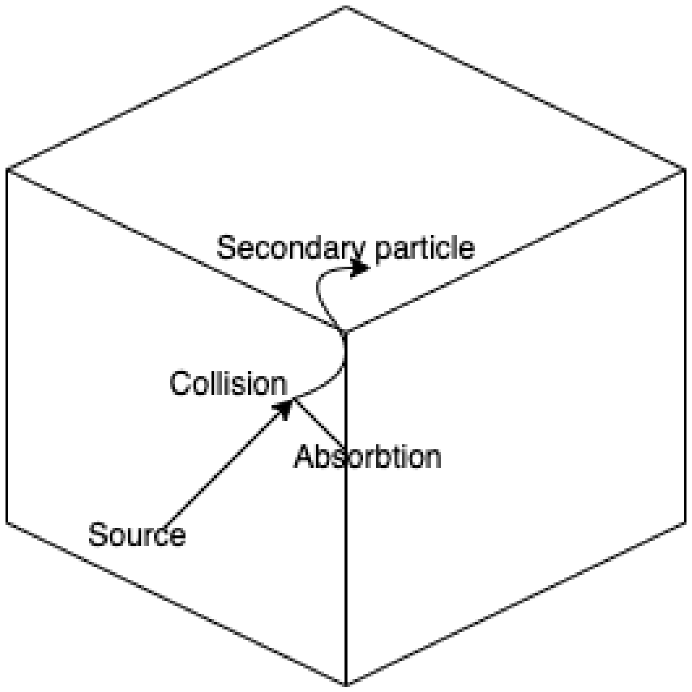

Figure 2.1: Sample Monte Carlo history in 3D geometry.

The combination of the two algorithms demonstrates a basic history based approach to Monte Carlo. The work presented in Section 4.2 explores different modifications made to this algorithm to improve performance.

Sampling

To accurately predict particle interactions in a Monte Carlo simulations, probability distribution functions (PDF) of the reactions must be sampled using experimentally determined cross-section data. One method to sample PDFs is the direct inversion method. By definition, the PDF must be positive and normalized.

To use the direct inversion method, the PDF must be invertible and integrable to create a cumulative distribution function (CDF). The resulting CDF must also be invertible. To generate a CDF, the PDF must first be integrated:

$$CDF(x) = \int_{-\infty}^{x} PDF(x')dx' \tag{2.1}$$

The resulting CDF can be inverted and used to determine the probability that a random number, ξ, following the PDF will be less than or equal to x. For example, to calculate the distance to a collision in 1D is determined using this method. Starting with the probability

Algorithm 1 An individual particle history

```
1: while particle is alive do
2:     sample distance to collision (dtc)
3:     calculate distance to boundary (dtb)
4:     calculate distance to escape (dte)
5:     if $dtc < dtb$ then
6:         determine reaction type
7:         if reaction = absorption then
8:             tally absorption
9:             kill particle
10:        end if
11:        move particle to collision site
12:        calculate post-collision energy and direction
13:        tally collision
14:        add secondary particles to particle bank
15:    else if $dtb < dte$ then
16:        move particle to boundary
17:        determine new material cross section data
18:    else
19:        tally escape
20:        kill particle
21:    end if
22: end while
```

Algorithm 2 Outer Transport Loop

```
1: input parsing
2: initialize source
3: initialize geometry
4: initialize materials
5: initialize tallies
6: while $i < N$ and particle bank is not empty  do
7:     sample particle's initial state
8:     track particle                                  ▷ Algorithm 1
9: end while
10: combine tallies
11: write to output
```

that a collision occurs at some distance x:

$$P(x) = \Sigma_t e^{-\Sigma_t x} \tag{2.2}$$

Where Σ_t is the total macroscopic cross section. The probability must then be integrated:

$$\begin{aligned} CDF(x) &= \int_{-\infty}^{x} \Sigma_t e^{-\Sigma_t x'} dx' \\ &= 1 - e^{-\Sigma_t x} \end{aligned} \tag{2.3}$$

Then to determine the distance to the next interaction, the CDF is set equal to a random variable ξ:

$$\xi = 1 - e^{-\Sigma_t x} \tag{2.4}$$

solving for the distance, x

$$x = \frac{ln(\xi)}{\Sigma_t} \tag{2.5}$$

After a particle interaction is determined, the type and material it interacts with must be sampled as well. Each of these processes can be modeled using a discrete PDF. The PDF for the isotope type is shown in Eq. 2.6. The distribution can be sampled by generating a random number on $[0, 1]$ and comparing it to a cumulative sum of the individual isotope's total macroscopic cross section [8]. When $CDF_{isotope} > \psi_1$, the particle interacts with isotope i. Figure **??** shows this process graphically.

$$PDF = \frac{\Sigma_{t,i}}{\Sigma_t} \tag{2.6}$$

$$CDF_i = \frac{1}{\Sigma_t} \sum \Sigma_{t,n} \tag{2.7}$$

The reaction type can be determined similarly, except the number density of the material no longer needs to be carried.

$$PDF = \frac{\sigma_{k,i}}{\sigma_t} \tag{2.8}$$

$$CDF_i = \frac{1}{\sigma_k} \sum \sigma_{k,n} \tag{2.9}$$

Statistics

Monte Carlo methods use a large number of individual particle histories to determine bulk behavior, combining these histories in a statistically significant way requires an outline of basic statistics of large numbers. For any given continuous random variable, x, the mean of that random variable can be calculated by taking the first moment of the $PDF(x)$. PDFs must always be positive and they must integrate to one over the space of the random variable.

To calculate a mean from a discrete set of N samples of quantity x, the simple arithmetic mean must be taken. This value is called the sample mean $\bar{X}$, rather than the true mean of a continuous random variable. It is called the sample mean because it is computed by sampling from the underlying PDF rather than calculating it directly.

The definitions of the two means are presented below:

$$\mu = \int_{-\infty}^{\infty} xP(x)dx \tag{2.10}$$

Where μ is the true mean of the function.

$$\bar{X} = \frac{1}{N}\sum_{i}^{N} x_i \tag{2.11}$$

According to the law of large numbers:

$$\lim_{N\to\infty} \bar{X} = \mu \tag{2.12}$$

In words, as the number of samples approaches infinity, the sample mean becomes the true mean of the PDF being sampled [3].

To calculate the variance, σ of the true mean, the second moment of the equation can be taken.

$$\sigma^2 = \int_{-\infty}^{\infty} x^2 P(x)dx - \mu^2 \tag{2.13}$$

Eq. 2.14 shows how to calculate the sample variance of a discrete distribution.

$$\begin{aligned} Var(X) &= \frac{1}{N-1}\sum_{i}^{N}(x_i - \bar{X})^2 \\ &= \frac{1}{N-1}\sum_{i}^{N}(x_i^2) - N\bar{X}^2 \end{aligned} \tag{2.14}$$

The Central Limit Theorem states how the uncertainty scales with the number of samples [16]. This relationship is shown in Eq 2.15

$$\sqrt{N}((\frac{1}{N}\sum_{i}^{N} x_i) - \mu) \to \mathcal{N}(0, \sigma^2) \tag{2.15}$$

The desired quantity is the variance of the mean. This can be estimated with a single measurement of the mean. The variance of the sum of a uncorrelated random variables is equal to the sum. of the variance of the variables, shown in Eq. 2.16.

$$Var(\sum_{i=0}^{N} x_i) = \sum_{i=0}^{N} Var(x_i) \tag{2.16}$$

$$Var(\bar{X_i} = Var(\frac{1}{N}\sum_{i=0}^{N} x_i) = \frac{1}{N^2}\sum_{i=0}^{N} Var(x_i) = \frac{\sigma_N^2}{N} \tag{2.17}$$

Because the central limit dictates that the sample mean is normally distributed, the confidence intervals from the normal distribution can be applied. From these intervals, an expression for the relative error can be written.

$$RelErr = \frac{\sigma}{\bar{X_N}} = \frac{1}{\bar{X_N}}\sqrt{\frac{1}{N-1}(\frac{1}{N}\sum_{i}^{N} x_i^2 - \bar{X_N^2})} \tag{2.18}$$

Tallies

Tallies are a generic term used in Monte Carlo methods to describe some physical quantity of interest that is derived by combining individual particle histories into meaningful bulk behavior. To combine histories, each history is given a score using a statistical estimator and the average and variance of the tally is calculated using techniques described in the previous section. Photon and electron flux are the two most commonly used tallies in this work.

Flux can be calculated indirectly from reaction rate. Reaction rate is determined by counting the number of collisions that take place withing the Monte Carlo simulation. This can be written as:

$$R_x = \frac{1}{W}\sum_{i}^{N} w_i \tag{2.19}$$

Where R_x is the reaction rate for some reaction x, w_i is the weight of an individual particle, and W is the total weight of the initial source particles. To get flux from reaction rate:

$$\phi = \frac{R_x}{\Sigma_x} \tag{2.20}$$

Another way to calculate flux is to use a track-length estimator. To derive the track-length tally, first the expression for the volume integrated flux must be introduced:

$$V\phi = \int d\boldsymbol{r} \int dE \int d\Omega \int dt \psi(\boldsymbol{r}, \hat{\boldsymbol{\Omega}}, E, t), \tag{2.21}$$

where V is the volume, ψ is the angular flux, $\boldsymbol{r}$ is the position, $\hat{\boldsymbol{\Omega}}$ is the direction of the particle, and E is the energy of the particle, and t is time. Flux can also be written as:

$$\psi(\boldsymbol{r}, \hat{\boldsymbol{\Omega}}, E, t) = vn(\boldsymbol{r}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.22}$$

Where n is particle density and v is velocity. Combining Eq 2.21 and Eq 2.22:

$$V\phi = \int d\boldsymbol{r} \int dE \int vdt \int d\Omega n(\boldsymbol{r}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.23}$$

Using the definition of particle density

$$N(\boldsymbol{r}, E, t) = \int d\boldsymbol{\Omega} n(\boldsymbol{r}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.24}$$

and the definition of the differential track length

$$dl = vdt \tag{2.25}$$

a new form of the volume integrated flux can be obtained:

$$V\phi = \int d\boldsymbol{r} \int dE \int dl N(\boldsymbol{r}, E, t) \tag{2.26}$$

This form of the equation reveals that the differential track length can be used to get an estimate of the flux:

$$V\phi = \frac{1}{W} \sum_{i \in T} w_i l_i \tag{2.27}$$

Where l_i is the the length of a single trajectory within a volume, T is the set of all particle tracks, w_i is the weight if an individual particle, and W is the total weight of the particles. Track-length tallies are more commonly used than reaction rate tallies because they can be used in volumes with low probability of interactions. Track length tallies were used to determine the flux for the work done in this dissertation.

2.2 Monte Carlo Geometry

Monte Carlo is a powerful computational tool. The ability to create complex, general geometries and transport individual particle histories continuously on those general geometries make it a uniquely flexible method. There are a handful of different 3D geometry types that can be represented with Monte Carlo methods including combinatorial geometry, voxel geometry, and CAD geometry. Each of these geometry types have different benefits and drawbacks. In general, the most lightweight, in terms of memory, is voxel geometry because they are relatively simple to compress. CAD geometries are the most memory intensive due to their complexity.

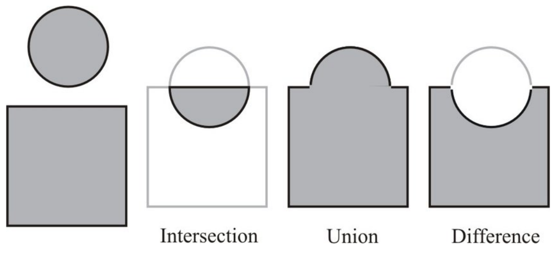

Figure 2.2: Depictions of different boolean logic for combinatorial geometry [20].

Combinatorial Geometry

Combinatorial geometry (CG) is a geometry type that can represent a large complex models using boolean operations of surfaces and basic three-dimensional shapes, called primitives. CG geometry is well suited for for models with repeating patterns or symmetries, such as nuclear reactors. Surfaces are the most fundamental building blocks that are combined to form larger three dimensional shapes, called cells. To define a cell in Monte Carlo transport codes, the region must be bound by a boolean combination of surfaces. These boolean operations are unions, intersections, and difference operations. Figure 2.2 illustrates the different operations that are used to create combinatorial Monte Carlo geometries. Each defined cell is assigned a material.

Another unique feature of CG geometry is the ability to create a universe. A universe is a group of cells that fully define a spatial domain. The universe can be repeated in a specific pattern, called lattices. Both lattices and universes can be translated and transformed; combining these two features make it easier to generate structured models, such as reactor cores. The hierarchy of cells, universes, and lattices is shown in Figure 2.3. This technique is commonly used in nuclear reactor codes because nuclear reactor cores have repeating geometries and features. This feature is not implemented in ITS and CHEETAH-MC.

Voxel Geometry

Voxels are another Monte Carlo geometry type. Voxels are essentially three-dimensional pixels; a voxel represents a value in a regular grid. In the context of Monte Carlo, the value that is represented is a material. The material can be a complex composite material. The material is linked to the cross section database. Voxels are regular in size and shape. Voxels have the advantage of being lightweight and compressible. Voxel geometries are initially defined as a 3D matrix, the 3D matrix can then be compressed similar to image compression. Figure 2.4 illustrates an example of the voxel grid. Voxel geometry are most commonly used for medical data.

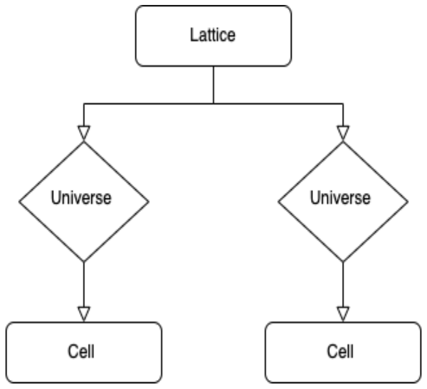

Figure 2.3: Combinatorial geometry universe hierarchy

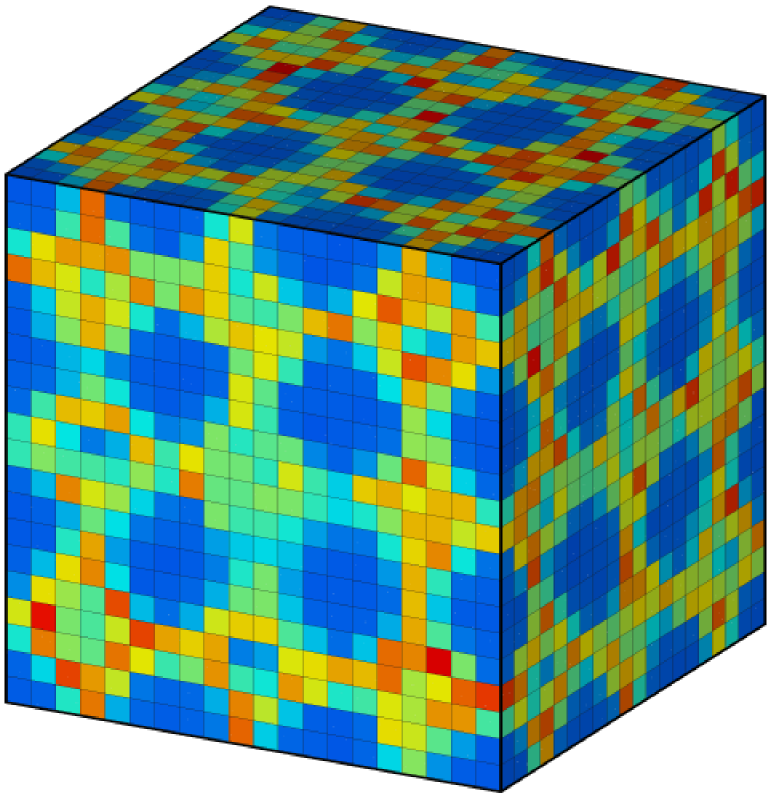

Figure 2.4: An example of voxel geometry with randomly placed materials.

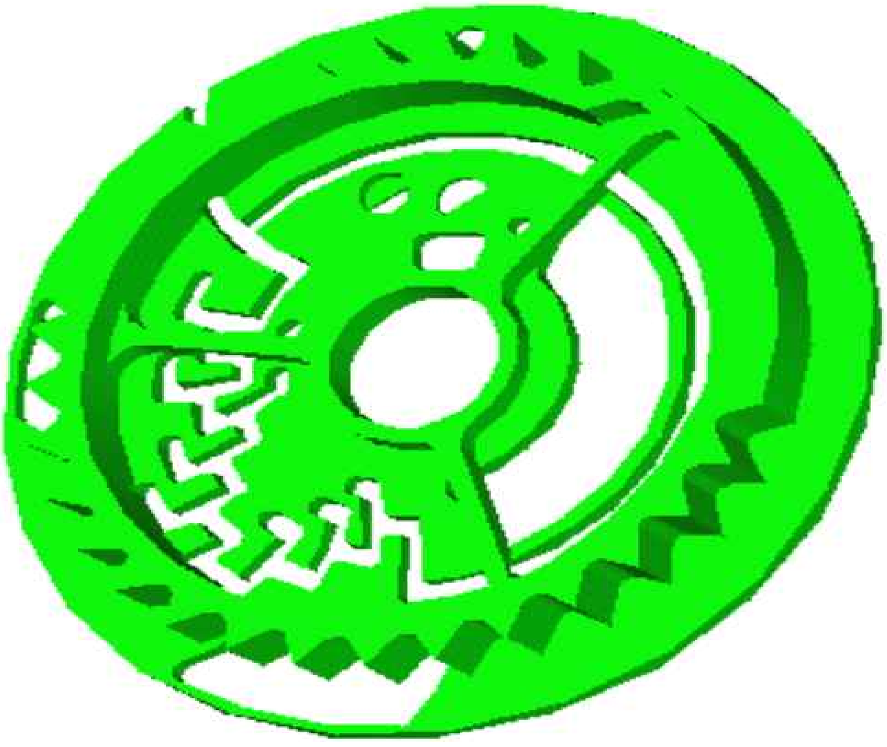

Figure 2.5: An example of a CAD drawing. Recreating this object with CG geometry would be difficult [29].

CAD

Computer Aided Drawing (CAD) is a term used to describe any mechanical or engineering designs or drawings that use computers to help optimize them. They are typically 3D images and are widely used in the field of engineering to generate high fidelity experimental designs. There are several different types of software that can be used to generate CAD geometries. Using CAD in Monte Carlo codes can greatly reduce the time it takes to generate the large, time-consuming CG models, because CAD software is ubiquitous and most are user friendly. The drawback of CAD geometries is that particle tracking directly on CAD geometries is slow, likely because looking up CAD geometry references is not optimized for non-CAD related computations. For example, DAG-MC is a geometry model that interfaces with different Monte Carlo radiation transport codes developed by the University of Wisconsin and they found that DAG-MC was slower than native geometries [40]. Although transport on CAD geometries can be slow, it can also significantly reduce the time it takes to build the geometric structure to use in the problem. An example of a CAD geometry that can be simulated in the Integrated Tiger System (ITS) is shown in Figure 2.5 [29]. CAD geometries can contain thousands detailed parts.

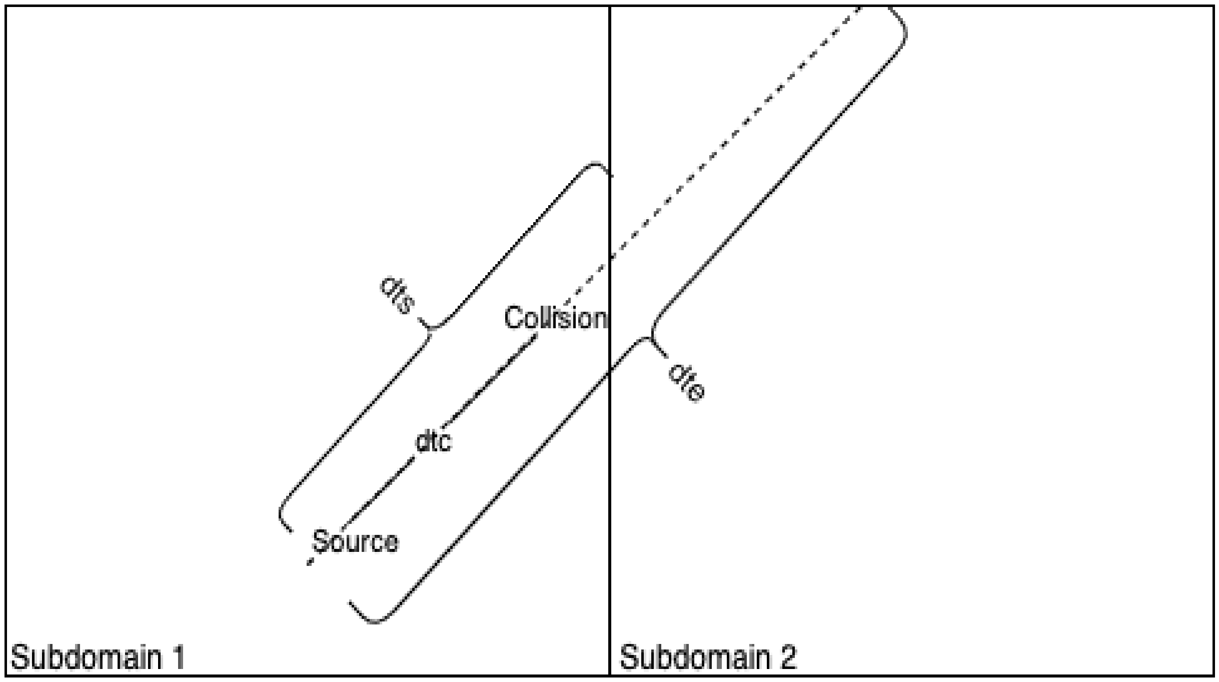

Figure 2.6: An visualization of the boundary comparisons occurring in 3

2.3 Domain Decomposition

Monte Carlo algorithms can be parallelized in two distinct ways, at the individual particle level and geometrically. Because each particle's life is independent parallelizing them is trivial, this is also referred to as embarrassingly parallel. Parallelizing the geometry, or domain, can be done through replication or decomposition. Replication is the process of duplicating the geometric information on each available processor. Decomposition splits the geometry spatially and assigns each subdomain to a unique processor, as shown in Figure 2.7. As particles leave one subdomain and enter another particles are banked and then transferred to the next subdomain. Additionally, the subdomains can and should overlap. These areas of overlap are called ghost zones. The modified algorithm, Algorithm 3 and 4, is presented below along with a visualization of the process.

Load balancing is often necessary when implementing domain decomposition. Often times the work required by the processor is spatially dependent. Different materials have different cross sections, leading to a concentration of reactions in certain parts of the geometry. To ensure that a single processor does not perform the bulk of the computational work, the geometry must be decomposed with an understanding of the underlying physics.

Memory transfer between the host and device is often the bottleneck in high performance heterogeneous computing. For example on the NVIDIA Tesla the peak bandwidth is 144

Algorithm 3 An individual particle history

```
1: while particle is alive do
2:     sample distance to collision (dtc)
3:     calculate distance to boundary (dtb)
4:     calculate distance to subdomain boundary (dts)
5:     if $dtc < dtb$ then
6:         determine reaction type
7:         if reaction = absorption then
8:             tally absorption
9:             kill particle
10:        end if
11:        move particle to collision site
12:        calculate post-collision energy and direction
13:        tally collision
14:        add secondary particles to particle bank
15:    else
16:        move particle to material boundary
17:        calculate distance to escape (dte)
18:        if escape then
19:            calculate distance to subdomain boundary (dts)
20:            if dts ¡ dte then
21:                bank particle
22:            end if
23:            tally escape
24:        end if
25:    end if
26: end while
```

GB/s while the bandwidth between the device memory and the host memory is 8 GB/s. This discrepancy in bandwidth can significantly impact the final runtime of the program. Adding particles to the particle bank and moving the particle bank to the correct device memory is the memory transfer that most greatly impacts the performance of the domain decomposition modified algorithm. There are a few ways to impact the time it takes to transfer the particles between device memory. The first way is to change the size of the particle bank. Increasing the size of the bank requires a higher amount of memory to be transferred which would take longer. However, a smaller bank would require the memory transfer to be initiated more frequently. Additionally, the location of the subdomain boundaries impacts the performance. If subdomain boundaries are placed in geometric areas with high amounts of particle streaming, the number of particles that need to be transferred will increase. Additionally, electron cascades and scattering along subdomain boundaries increase the runtime. Electrons tend to lose a small amount of energy in each scatter reaction and travel a short

Algorithm 4 Outer Transport Loop

```
 1: input parsing
 2: initialize source
 3: initialize geometry
 4: initialize materials
 5: initialize tallies
 6: while $i < N$ do
 7:     sample particle's initial state
 8:     track particle                              ▷ Algorithm 3
 9: end while
10: while particle bank != empty do
11:     Sort particle bank
12:     for size of particle bank do
13:         track particles                         ▷ Algorithm 3
14:     end for
15: end while
16: combine tallies
17: write to output
```

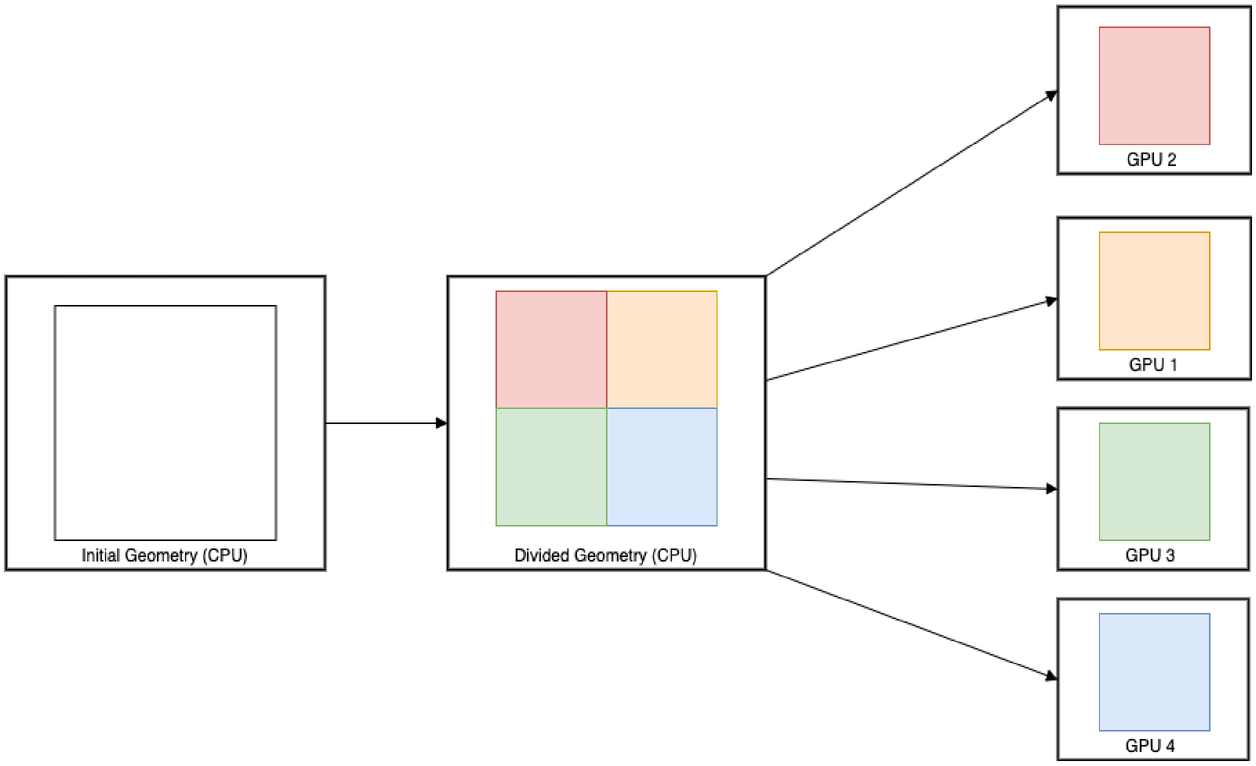

Figure 2.7: Graphic depiction of domain decomposition on advanced hybrid architectures

distance between interactions.

Finding a balance between these load balancing and memory transfer is important to fully implement domain decomposition. Other GPU radiation transport codes have implemented domain decomposition using traditional methods. For example, SHIFT, a neutron Monte Carlo radiation transport code, uses a method called Multiple Set, Overlapping Domain (MSOD) [33]. This method overlays a Cartesian mesh on the problem geometry and each mesh cell is assigned to a processor. The mesh cells do not need to be uniformly sized or spaced. The processor domains are also allowed to overlap. SHIFT also combines domain decomposition with domain replication, which allows for certain mesh cells to be replicated on different processors. This cut downs on idle processor time so fewer processors have to wait for high activity processors to transfer particles. Another key element to the MSOD approach is that particles can only be transferred to a neighboring domain. Particles are stored and communicated to neighboring domains at regular intervals. The GPU version of this method uses both device-to-device communication and device-to-host communication to transfer stored particles. The SHIFT team tested their method on Small Modular Reactor Cores and saw a parallel efficiency of about 80% compared to full domain replication [19].

PRAGMA, a neutronics GPU code developed by Seoul National University, has implemented a different domain decomposition method [11]. Instead of using a Cartesian mesh, like SHIFT, they used wedges of a wheel. This method is better for load balancing for nuclear reactors because they are radially symmetric. However, it does not scale well; as the number of subdomains increase, the wedges become optically thinner in the center which degrades the performance of the code. Domain replication is currently not supported by PRAGMA.

Domain decomposition has not been implemented in coupled electron-photon Monte Carlo codes for the GPU.

2.4 Electron-Photon Physics

Electrons and photons are two fundamental particles of interest for nuclear security, medical physics, and plasma physics applications. The two particles are coupled in the sense that some photon interactions lead to the production or emission of electrons and vise versa. Because Monte Carlo simulates individual particle interactions the following section will discuss photon and electron interactions. The energy range of interest is $eV < E_{e^-,\gamma} < MeV$.

Particle collisions can cause the incident particle to change its energy and direction. They can also result in the excitation of the atom. As the atoms relax, or de-excite, the emit electrons or photons. The primary and secondary particles continue to interact with matter until they are fully absorbed into the medium or escape from the area of interest.

The probability that an interaction happens is called a cross section. Cross sections can be quantified in two separate ways: macroscopic and microscopic [15]. Macroscopic cross sections have units of inverse length and describe the probability of interaction per unit distance traveled. It is denoted by the symbol: Σ. Microscopic cross sections have units of area and are represented by the symbol: σ.

Microscopic cross sections are analogous to geometric cross sections, they represent the size of the target nucleus for a specific type of reaction. It is meant to represent the size of the reaction space of an individual atom that the incident particle "sees" as it travels through a medium [24]. Microscopic cross section is sometimes expressed in a unique unit called a barn. A barn is $10^{-24} cm^2$. Despite having units of area, the value represents the likelihood of a reaction occurring and not the physical cross section of the nuclei. Microscopic cross sections can be determined experimentally or derived from first principles depending on the interaction and incident particle.

Macroscopic cross sections are the probability of a reaction happening per unit distance that is traveled by the incident particle. The macroscopic cross section can be for the total interaction probability or for a specific type of interaction. It can be calculated from the microscopic cross section directly based on material properties. Materials of interest are often compositions. To calculate the macroscopic cross section of a composition, each individual components cross section must be calculated and summed. The material properties that are necessary to calculate the cross section are atomic fraction f_i, number n_i, density ρ, and atomic mass of the composition M_{avg}.

$$\Sigma_{composition} = \sum_{i=1}^{N_{isotopes}} n_i \sigma_i = n_{avg} \sum_{i=1}^{N_{isotopes}} f_i \sigma_i = \frac{\rho}{M_{avg}} \sum_{i=1}^{N_{isotopes}} \tag{2.28}$$

The mean free path is the inverse of the total macroscopic cross section of a material composition. It describes the average distance a particle travels before experiencing a collision. It has units of distance.

$$\lambda = \frac{1}{\Sigma_t} \tag{2.29}$$

Photons

The photon interactions that are typically modeled by a radiation transport Monte Carlo code are pair production, Compton scattering, photoelectric effect, and Rayleigh scattering [14]. The first three interactions result in the transfer of energy to electrons or positrons. Rayleigh scattering is purely elastic meaning the photon changes direction with no energy loss. The photoelectric effect dominates at lower energies, the Compton effect takes over in the intermediate ranges, and pair production is most common at higher energies, as shown in Figure 2.8.

The photoelectric effect, 2.9 is the dominant interaction at lower energies and the probability that this interaction occurs decreases dramatically at $h\nu = 0.5 MeV$. When a photon interacts with a tightly bound inner shell electron, the photon can lose all of its energy and the inner shell electron is emitted [17]. The photon must have an initial energy that is greater than the binding energy of the electron. The change in kinetic energy of the atom is effectively zero. The photon vanishes and a conservation of momentum balance can be used to calculate the exit angle of the atom and electron.

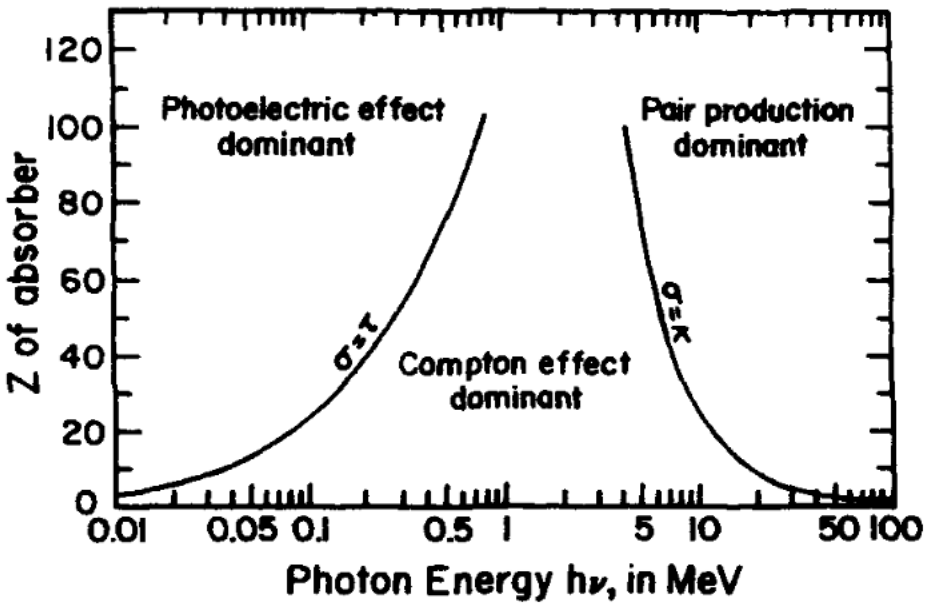

Figure 2.8: Relative cross sections of the three major photon interactions [1].

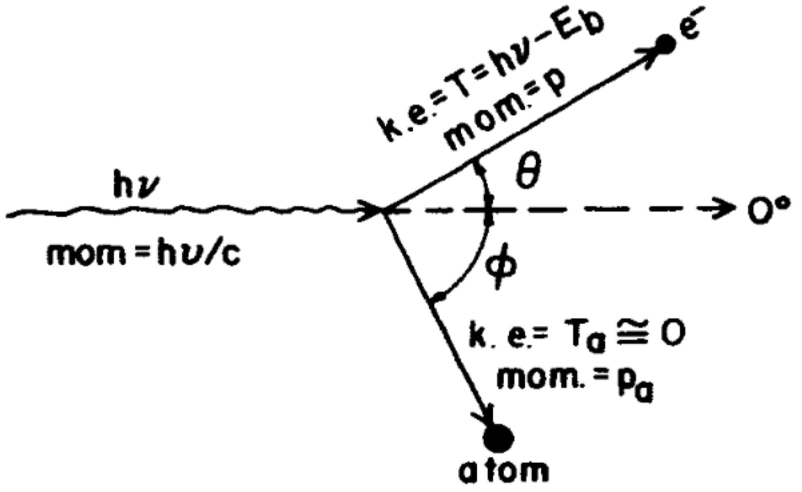

Figure 2.9: Kinematics of the photoelectric effect [1].

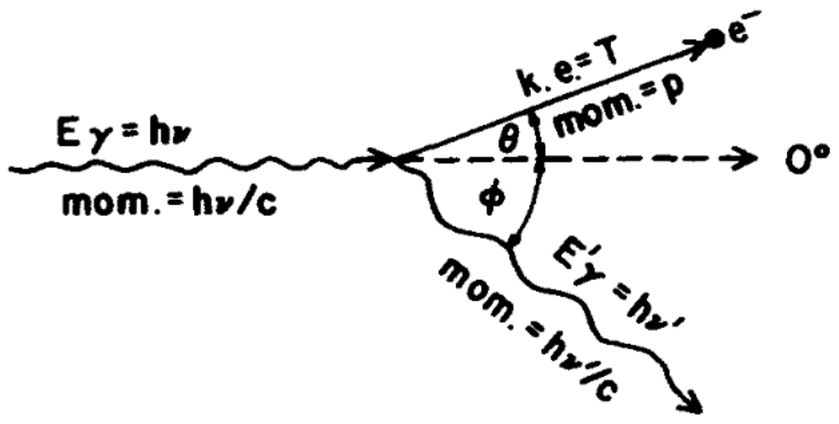

Figure 2.10: Kinematics of Compton Scattering [1].

Compton scattering is a scattering event where an incident photon of energy $h\nu_0$ scatters off an electron, of energy $h\nu_0$, off an electron. The electron is assumed to be unbound and stationary relative to the photon. These assumptions are not rigorous because electrons typically occupy an electron shell of an atom and are in motion, but they allow us to derive the Klein-Nishina cross section [22]. In this interaction, a photon is not absorbed but instead scatters in a new direction, as shown in Figure 2.10.

Pair production is an absorption interaction where a photon is absorbed into a material and two new particles, an electron and positron are emitted. Pair production can happen inside a Coulomb field, typically near a nucleus. Triplet production can also occur in the Coulomb field of an atomic electron, however this is more rare. When this interaction occurs near an atomic electron it is called "triplet production" because the host electron is also emitted in the process. Pair production requires a minimum photon energy of 1.022 MeV or the amount of energy of an electron-positron pair, $2m_ec^2$. Triplet production requires a initial photon energy of 2.044 MeV. The kinematics of pair production are shown in Figure 2.11.

Finally, Rayleigh scattering describes the process of an incident photon being scattered by the combined action of the atom. The incident photon loses no energy in the collision, however it does change directions. Therefore, Rayleigh scattering is a form of elastic scattering. The incident photon changes directions by a small angle and the effect on the atom is almost negligible, it moves just enough to conserve momentum. Rayleigh scattering has a large effect on photons of lower energies because the scatter angle is inversely proportional to incident energy.

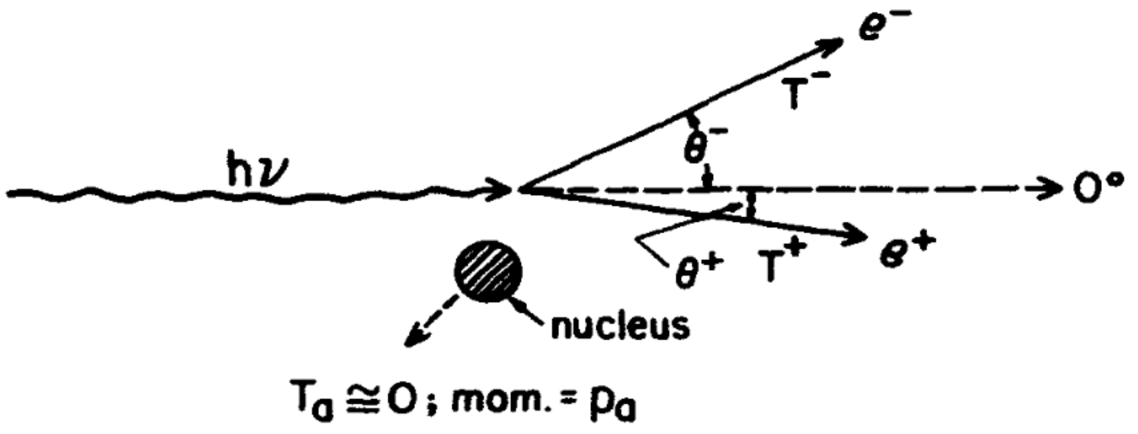

Figure 2.11: Kinematics of pair production [1].

Electrons

Charged particles lose energy in a way that is distinct from neutral particles, such as photons. Photons can travel large distances without interacting with the medium, then lose all of its energy and be absorbed after a single nuclear interaction. Charged particles interact via the Coulomb field, so every electric field passed has an impact on the incident charged particle. Most of these interactions have a small effect on the incident particle's energy, so it is often more useful to think of the particle as continuously slowing down as it moves along a path. A single MeV electron can experience 10^5 interactions before depositing all of its energy into the medium [1]. These individual scatter events can be difficult to simulate and are computationally expensive. To combat this issue, Berger developed the condensed history algorithm to combine scatter events based on the continuous slowing down method [4]. The application of the condensed history algorithm to Monte Carlo transport, along with a derivation of the method, was explored by Ed Larsen [25].

Other non-scatter events that electrons experience are Bremsstralung and atomic relaxation [23]. As electrons slow down in a high Z material, secondary photons are created, also called Bremsstrahlung radiation. This happens as electrons pass through a strong Coulomb field. Atomic relaxation also produces secondary particles. Atomic relaxation describes the process where an electron is ejected from an atom and a vacancy is left in some inner shell. Electrons from higher shells will fall and fill the vacancy, this process continues until each inner shell is filled. As the electrons move from a higher shell to a lower shell, a photon is emitted with an energy that matches the difference in the electron shells. Auger electrons can also be emitted in a non-radiative transfer process. Atomic relaxation leads to a cascade of electron and photon emissions.

Positrons are particles with the same mass as electrons and equal but opposite charge.

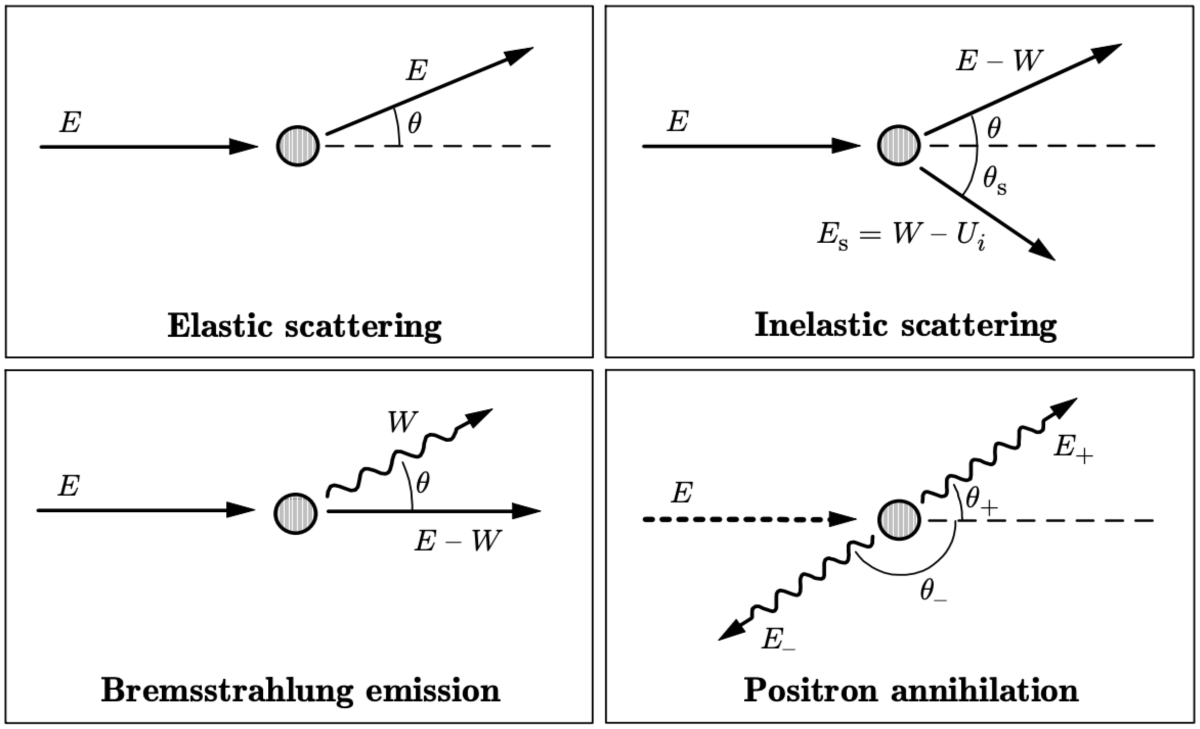

Figure 2.12: Basic interactions of electrons and positrons [34].

Positrons can be emitted via radioactive decay or pair production. They can undergo annihilation while in motion or at rest. Annihilation is when an electron-positron pair combine and a form two photons, while maintaining total momentum and energy. A depictions of the basic electron interactions are shown in Figure 2.12.

Chapter 3

GPUs and Machine Learning

This chapter first covers GPU architecture, including memory spaces, thread hierarchy, and CUDA best practices. Then it explains the basics of machine learning, including regression and Neural Nets.

3.1 GPUs

The General Purpose Graphics Processing Unit (GPGPU), more commonly referred to as GPUs, is massively parallel processors that can execute thousands of threads concurrently. GPUs were initially developed to process graphics to be displayed on a monitor. Processing graphics requires operations on two dimensional arrays of pixels and other similar operations that can often be executed in parallel. To move and manipulate these images on a computer, basic linear algebraic operations are needed. Operations on matrices and arrays of pixels often have massive fine-grained parallelism which GPUs can take advantage of and run faster than a CPU. Therefore, GPUs are optimized to compute matrix operations that are offloaded by the CPU. GPUs were initially not programmable and the commands they executed were largely invisible to the programmer. CPUs were used to process complex tasks and the GPU quietly processed the simpler tasks, but today's GPGPUS aer programmable and can be used for a wide range of problems.

CUDA is a programming language released in 2006 that is used to program NVIDIA GPUs. In the CUDA programming model, code is written for the host (CPU) and device (GPU) separately. The host directs the device to execute code and memory storage. Data must be allocated and transferred from the host to the devices explicitly. A more in-depth description of the memory spaces and allocation is discussed in Section Memory. The operations performed on the GPU are called 'kernels'. Kernels are analogous to functions in C++, except they execute N times across different CUDA threads rather than once. NVIDIA recommends that kernels should be lightweight and simple with minimal global memory access to optimize runtime. The CUDA kernels must be fully independent of each other and are independent of order. However, there can be barriers that allow threads to wait for each other to execute. These barriers or locks allow for thread synchronization to resolve race conditions, but come at the cost of increased runtime.

Threads are organized into blocks and blocks are grouped into a grid. Threads within a block must execute independently in an arbitrary order, either in parallel or serially. This allows for easier scaling because blocks can be scheduled in any order across N cores. Inside a block, threads share data through shared memory and their execution can be synchronized to coordinate when memory is accessed. This execution scheme is called SIMD (single instruction multiple data). The same series of instructions are carried out over different data sets. On the GPU, the SIMD execution scheme is abstracted into single-instruction multiple-thread. The key difference between these two concepts is that the SIMT model allows for different threads to branch off and diverge from each other. Thread divergence gives the programmer some flexibility; however the performance cost could be staggering. Each divergent thread must be serialized.

Current NVIDIA GPU architectures have a maximum of 1024 threads per block and all threads in a thread block reside on the same streaming multiprocessor. They also all share the memory available on a given multiprocessor core. Thread blocks are grouped into one-dimensional, two-dimensional, or three-dimensional grids. The size of the data being processed determines the number of blocks in a grid. Figure 3.1 illustrates an example of thread hierarchy on the GPU.

CUDA 9.0 introduced an additional, optional, level of thread hierarchy called thread block clusters. Thread blocks inside a thread block cluster are guaranteed to be scheduled on a streaming multiprocessor. Clusters are also organized into one-dimensional, two-dimensional, and three-dimensional structures, similar to threads in a grid, as shown in Figure 3.2. The threads in a cluster can access the distributed shared memory of the GPU. The size of the distributed shared memory space is the number of thread blocks per cluster multiplied by the size of shared memory per thread block. Therefore, using thread block clusters can potentially increase the memory available. Memory is a significant limitation on the GPU.

Memory

When programming with CUDA, it is required that the programmer assumes that the host and device operate separately with their own memory spaces. It is necessary to allocate and deallocate device memory explicitly. Additionally, data must be transferred from the host to the device and back again. Device memory can be allocated as CUDA arrays or as linear memory. CUDA threads have access to multiple memory spaces during their lifetime, these memory spaces are shown in Figure 3.4. All CUDA threads have access to the global memory allocated on the GPU and thread blocks, or thread block clusters, can read, write, and perform atomic operations in the shared memory space. Global memory is the largest memory space on the GPU, this is the memory space that is quoted when discussing total RAM on each GPU card.

Although global memory has the largest memory capacity, it is the slowest to access. Shared memory is typically much faster than global memory, but is limited in size. Shared

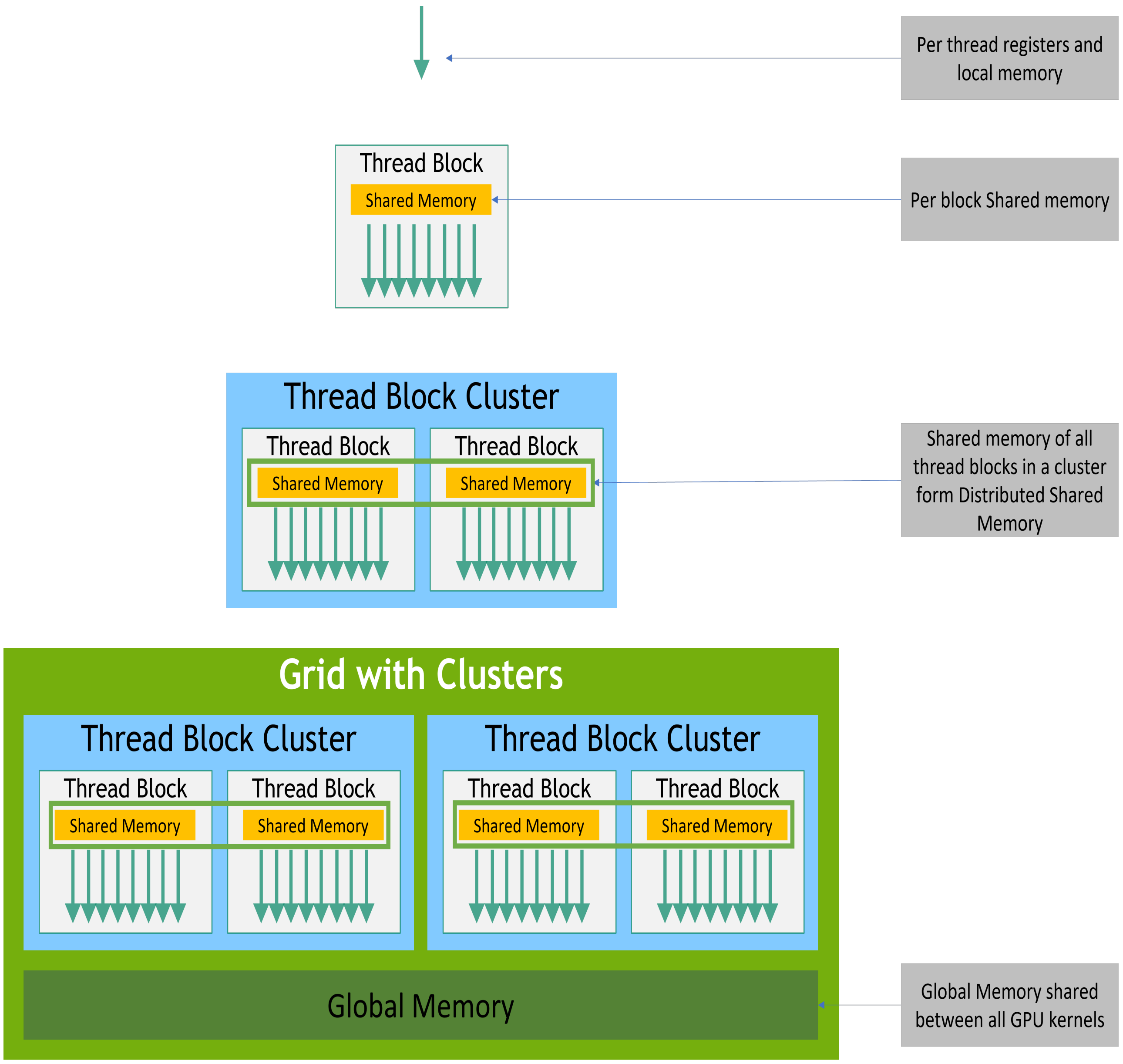
Per thread registers and local memory
Thread Block
Shared Memory
Per block Shared memory
Thread Block Cluster
Thread Block
Shared Memory
Shared memory of all thread blocks in a cluster form Distributed Shared Memory
Grid with Clusters
Thread Block Cluster
Global Memory
Global Memory shared between all GPU kernels

Figure 3.1: Memory hierarchy of an NVDIA GPU [31].

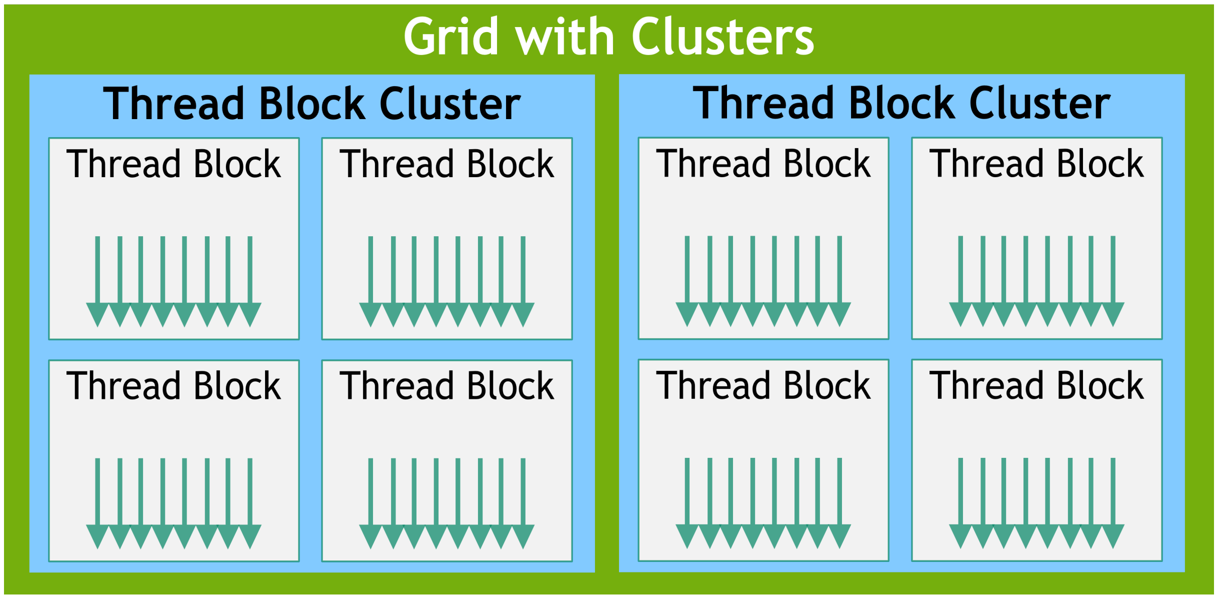

Figure 3.2: Thread cluster structure of an NVIDIA GPU [31].

memory is on the multiprocessor chip and has low memory latency; which is the time it takes for a processor to receive the information it requested. Technically, all three of these memory spaces share the same physical location on the GPU, however the way the multiprocessors accesses the memory is distinct. Constant and texture memory spaces are also accessible by all threads, however, they are static and read-only. Constant memory is the most limited in terms of size and it performs the best when all threads access the memory simultaneously. Texture memory can be allocated for the entire global memory space. It performs the best for linear interpolation for 2D arrays.

Local memory is thread-specific and is designed to hold data that cannot fit into register memory. For example, if a large structure or array cannot fit in the register. The register memory is responsible for doing real time arithmetic operations, so it typically only holds the small amount of information required to execute basic functions. In Fermi and newer NVIDIA architectures, the local memory is cached by the L1 cache on the GPU instead of being stored in the global memory. Memory caches are small intermediate storage between the register and global memory. They hold data store data that has been loaded previously to avoid redundancies in global memory fetches. Typically, there are multiple levels to memory caches, L1 loads the fastest and the time it takes to load the memory increases as the number increases. Figure 3.4 is a comparison of the GPU and CPU memory spaces.

The GPU's bandwidth describes the rate at which it transfers data to or from memory (vRAM) to the processors. To maintain a high computational rate, a high bandwidth is required. If the bandwidth is too low, thousands of GPU threads will idle until they receive

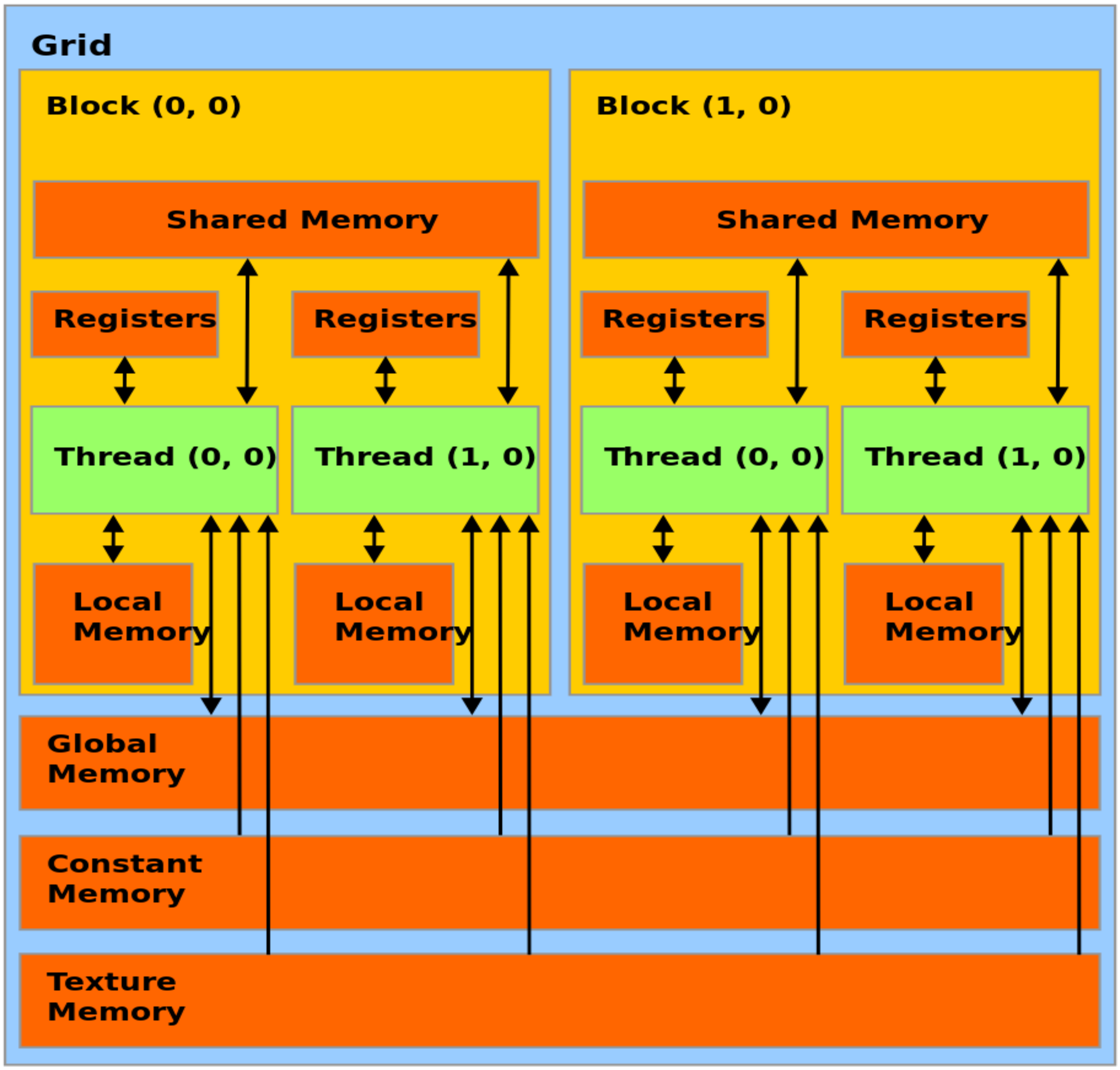

Figure 3.3: A schematic of the different memory spaces on a GPU [31].

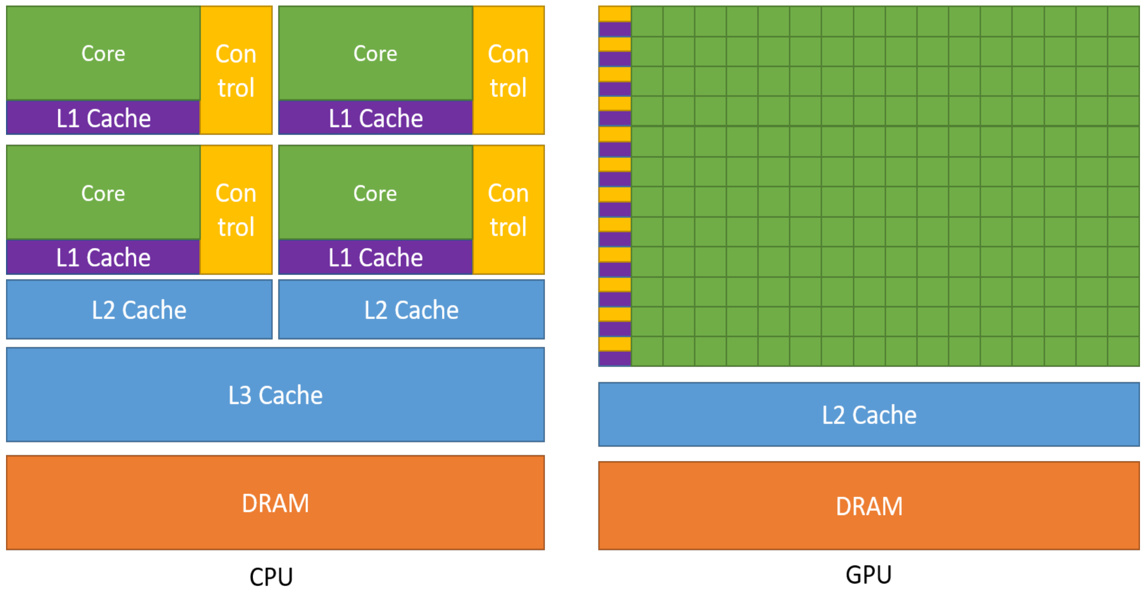

Figure 3.4: A comparison of GPU and CPU memory spaces [31].

the requested data from memory. Ideally, the memory bandwidth would match the computation rate. On the chip itself, the memory interface, which is the number of individual links on the data bus between the cores and the vRAM, determines the bandwidth. Table 3.1 lists some current NVIDIA GPU capabilities.

NVIDIA GPU Model	vRAM	Memory Interface Width	Memory Bandwidth
P4000	8GB	256-bit	243 GB/s
P5000	16GB	256-bit	288 GB/s
P6000	24GB	384-bit	432 GB/s
V100	32GB	4096-bit	900 GB/s
RTX4000	8GB	256-bit	416 GB/s
RTX5000	16GB	256-bit	448 GB/s
A4000	16GB	256-bit	448 GB/s
A5000	24GB	384-bit	768 GB/s
A6000	48GB	384-bit	768 GB/s
A100	40GB	5120-bit	1555 GB/s

Table 3.1: Memory properties of common NVDIA GPUs [31].

Thread Divergence

CUDA parallelism follows the SIMT (Single Instruction Single Thread) model. CUDA kernels describe the series of instructions to be executed on each thread. CUDA assumes that each thread will be executed simultaneously in parallel. Threads diverge when there are conditional statements in the GPU kernel and different threads take different branches of the conditional.

Thread divergence occurs when a conditional set of instructions begins to execute within the warp. Because warps must execute the same set of instructions, the GPU must execute the code one set of conditionals at a time. This work around essentially serializes the code because the threads that do not evaluate the initial set of conditionals are left idle. Sections of threads will continue to execute on set of conditionals at a time until the flow re-converges after each combination of conditionals are realized. This process greatly reduces the performance of the GPU; idle threads are still assigned to register memory and execution slots, meaning they cannot be used until the warp re-converges.

3.2 Machine Learning

Machine learning is a powerful tool that is used for many different purposes, including classification, feature extraction from large data sets, and computing models that can be used for inference. There are two major types of learning: supervised and unsupervised learning. The primary difference between supervised and unsupervised learning is that supervised learning involves a set of training data in which the answer is known. Unsupervised learning is used to organize or cluster data but without training data mapped to answers provided in advance. The work in this dissertation focuses on supervised learning.

Supervised learning requires all training data to be labeled with a numerical value, binary value, or group membership. The goal of a supervised learning model is to match an independent variable input and predict the correct output or label. There are two major types of supervised learning problems: regression and classification.

Regression describes the process of mapping numerical outputs to inputs. The training data is also used to quantify the error of the model. Predicting the numerically dependent values from the independent values requires a loss function. The loss function measures the error and can include information about the complexity of the model. A commonly used loss function is the squared-error loss function. Additional terms can be added to the loss term to minimize the magnitude of the errors and add smoothness to the loss function. Loss terms and regression will be discussed further in the next section of this report.

The two main challenges to machine learning are the data quality and data quantity. Data quality refers to the data's clarity. For example, bias can be introduced to the model if the training data set is not representative of the problem that needs to be solved. The training data may also be of low quality if it is full of noise, errors, and outliers. Data quantity refers to the amount of data included in the training sets. Training data must be sufficiently large to cover the complexities and combinations of inputs and outputs. These two major challenges can cause the model to be overfit, meaning it performs sufficiently well on the training data but does not generalize well. They can also cause the model to be underfit, meaning the model is too simplistic and returns poor results. The best way to avoid these pitfalls is to generate sufficient data and split the data set into two separate ones: the training set and the test set. This will allow for a better estimate of the error on the model.

Regression

Regression is a from of supervised learning for dependent numerical variables that are in some continuous range. Starting with collected data, J and K, where J is the independent variables, $x_1, x_2, ..., x_J$, and K is the dependent variables, y_1, y_2, ... , y_k, for I cases. Both independent variables and dependent variables are collected and stored in a database. The data sets can be written as matrices; $\mathbf{X}$ has a size of IxJ and $\mathbf{Y}$ has a size of IxK. To map the dependent variables to the independent variables, supervised learning attempts to define

$$y = f(x) + \epsilon(x, z) \tag{3.1}$$

Where $\epsilon(x, z)$ is the error. Machine learning cannot find an equation that perfectly maps the two variables which is why an error term must also be defined. The error function describes the difference between the function produced by the ML model and the true value of y. However, the error is also written a function of the dependent variable x and potential unknown variables z to fully capture the complexity of the model. Not every database has this type of data available because it relies on the ability to quantify the quality of the data. This section will focus on defining the loss function, or the difference between $f(x)$ and y [27].

The loss function used in this work is the squared-error loss function:

$$L_{se} = \sum_{i=1}^{I} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2 \tag{3.2}$$

The loss function must be smooth so it can be derived and set to zero to minimize the overall error of the model. The derivative of the loss function is taken with respect to some parameters w_p.

$$\frac{\delta L}{\delta w_p} = 0, p = 1, 2, ...P \tag{3.3}$$

The simplest version of regression is linear regression. The general linear model is

$$y = \sum_{j=1}^{J} = w_j x_j + b \tag{3.4}$$

where b is the bias in the model. In matrix form is

$$\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{y}, \tag{3.5}$$

where $\boldsymbol{y}$ contains the measurements that are to be mapped. The values of b and w_j must minimize the error loss function:

$$L = \sum_{i=1}^{I}(\hat{y} - y_i)^2 \tag{3.6}$$

where $\hat{y}$ is the predicted value for any given case i. This basic model is called a least squares model. This mapping of input to output can be visualized in Figure 3.5.

There are many ways to solve for the coefficient vector, the most used one in Machine Learning codes is to decompose the matrix, using the Singular Value Decomposition, then invert. However, the computational complexity of inverting a matrix is about $O(n^3)$. A faster, better-suited approach is Gradient Descent.

Gradient Descent is an iterative method that calculates the gradient of the error function to find the local minimum. By finding the local gradient, the method moves in the direction of steepest descent until a minimum is found. Figure 3.6 shows a visual example of this method. The step size of the descent is called the learning rate. A higher learning rate will cause divergence because the algorithm will jump around the function, never finding the minimum and a smaller learning rate will take a long time to converge, as shown in Figure 3.6.

Modifications to the step size may be made throughout the method to avoid local minima and to find only the global minimal. The step size can be gradually decreased to skip over local minima in earlier iterations of Gradient Descent.

Gradient Descent uses the whole training set to calculate the gradient at each iteration which can be computationally expensive. To reduce the computational overhead of gradient descent, a smaller subset of the data may be used rather than the entire dataset. Stochastic Gradient Descent chooses a single random data point to calculate the gradient; however, it can be significantly noisier and require more iterations to determine the minimum. Batch Gradient Descent uses a subset of the larger data set to calculate the local gradient. This method converges faster than Stochastic Gradient Descent but is computationally more expensive. The step size and batch size dramatically impact the time the method takes to converge.

Linear regression is the most basic form of regression. Other, more complicated non linear models include, exponential models, power-law models, and logistic models. It is typical for machine learning applications to use the most basic form of regression first before adding complexity to the model.

Neural Nets

A neural network is a model that uses a nonlinear transformation on top of the linear combinations introduced in the previous section. The nonlinear component, called the activation

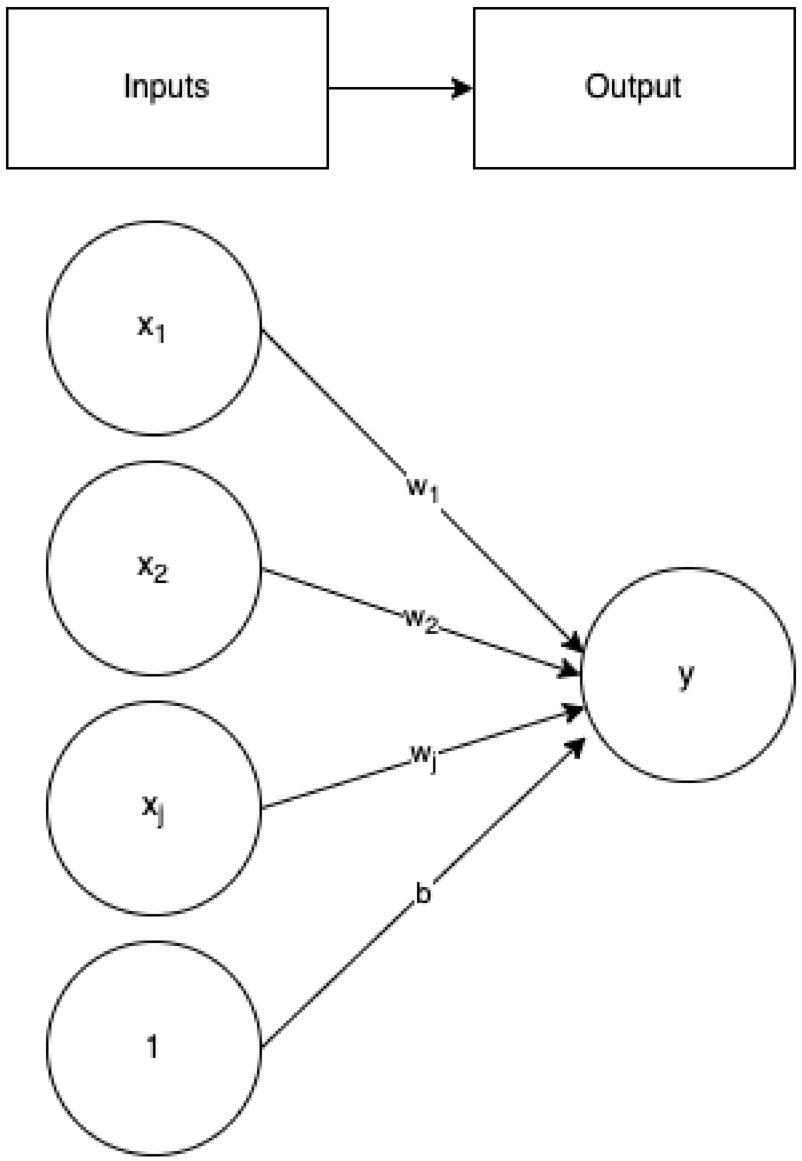

Figure 3.5: Schematic for basic linear regression.

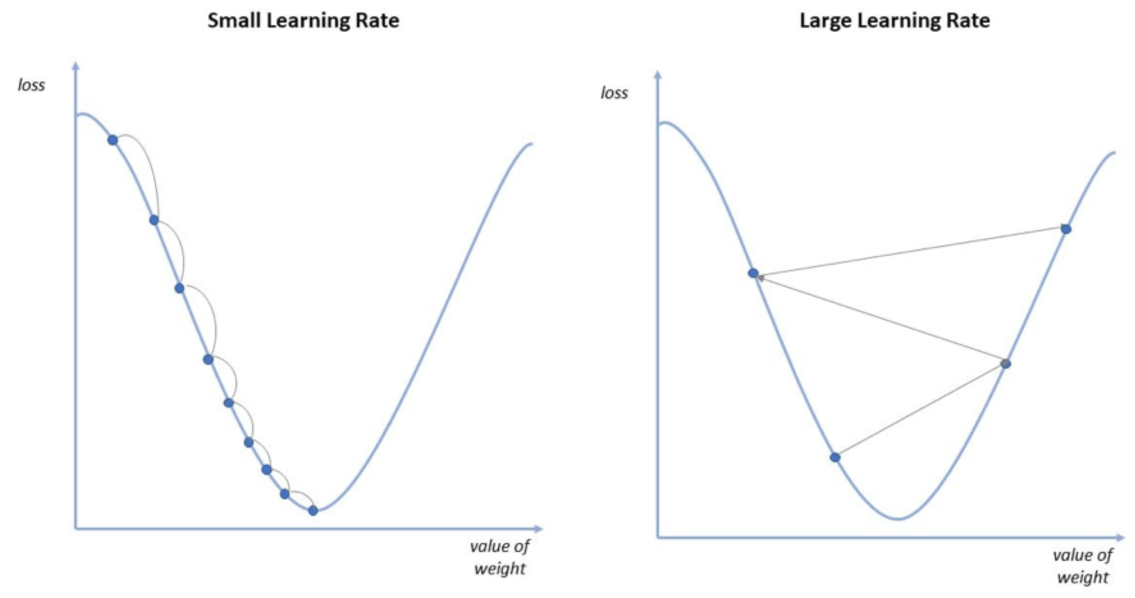

Figure 3.6: Gradient descent comparison between different step sizes [27].

function, generates intermediate variables called hidden layers. Common activation functions are stepwise, log, arctangent, and hyperbolic tangents. The number of hidden layers in a model can be changed, but most problems require a single hidden layer. The input data is fed into the network and are multiplied by the weights and combined with the activation function to form intermediate variables, called hidden units. The intermediate variables are then combined with the bias to get the final value y. The simplest form of a neural network is a feed-forward network; it has a single hidden layer. Feed-forward networks are flexible, meaning they are well equipped to handle data with large amounts of noise; they are also relatively easy to maintain.

Adding additional hidden layers is typically used for problems with an extremely large dataset, such as speech recognition. The number of hidden layers used in a model is a type of hyperparameter. Hyperparameters are parameters of the machine learning algorithm that are set before training the model and they must remain constant throughout the training. Hyperparameters must be tuned before training the model. Other hyperparameters include the batch size and step size mentioned in the previous section. Adjusting hyperparameters changes the convergence rate and accuracy of the neural net.

Input and output training data must be normalized when training a neural net because nonlinear functions cause large variations in the model. Unnormalized data introduces large variations into the gradient. Gradient Descent converges faster when the gradients are similar in magnitude and sign. The data is typically standardized by subtracting the mean of the data and dividing by its standard deviation.

Convolutional Neural Nets (CNN) are neural nets that use convolutions as the hidden layer, bookended by feed-forward layers. This is particularly powerful for image data, where each individual pixel is considered an input and is mapped to some output value of interest. Using a basic feed-forward network on a data set of that size would require a large amount of weights and biasing parameters, resulting in a slow training time. By using convolutions, each pixel is no longer required to be treated as an independent input, instead convolutions can be used to create hidden layer variables and highlight specific structures in the image data. Resulting in a smaller number of biasing parameters and a shorter training time. This is possible by setting a subset of the biasing parameters to the same value and applying the weights to make a pattern.

The definition of a convolution of two functions, f and g, is

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \tag{3.7}$$

The convolutional operator, $*$, is commutative and linear. Convoluting the function, f around t is meant to produce a smoothed version of the function $f(t)$. The three most commonly used functions for $g(t)$ are the Heavyside function, the triangular function, and the Gaussian function. Examples of these functions are shown in Figure 3.7. These functions can be defined on a finite scale, so the convolutional operator can be rewritten as

$$(f * g)(t) = \int_{-a}^{a} f(t-\tau)g(\tau)d\tau \tag{3.8}$$

However, image data is often not in continuous functional form. Image data is a 2D matrix of color indexes, $\boldsymbol{F}$, the convolution of image data can be written as:

$$(\boldsymbol{F} * \boldsymbol{G})_{m,n} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} F_{i,j} G_{m-i,n-j} m, n = 0, ..., N-1 \tag{3.9}$$

In the case of discrete 2D matrices, the matrix $\boldsymbol{G}$ can be written as a $3x3$ stencil to average $\boldsymbol{F}$:

$$\boldsymbol{G}_{average} = \frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{3.10}$$

Therefore, the convolution of $\boldsymbol{F}$ and $\boldsymbol{G}$ gives the average of nine values near some index $F_{m,n}$

$$(\boldsymbol{F} * \boldsymbol{G})_{m,n} = \frac{1}{9} \sum_{i=-1}^{1} \sum_{j=-1}^{1} F_{m-1,n-1} \tag{3.11}$$

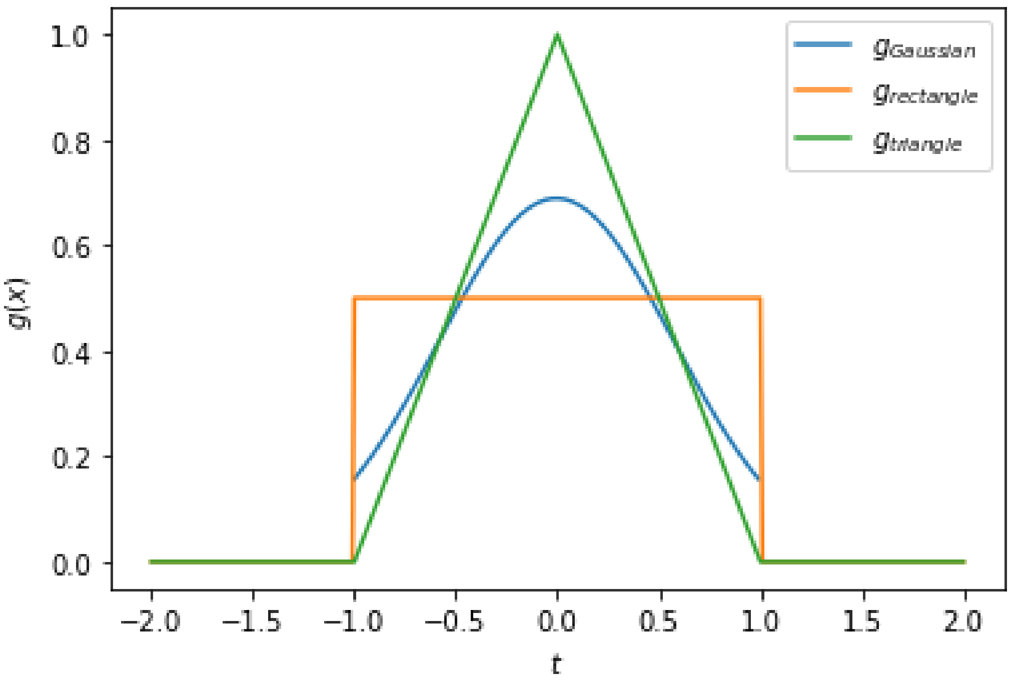

Figure 3.7: Plot of different g functions, Gaussian, rectangle, and triangle.

The filter $\boldsymbol{G}$ can be any size and form. For example, $\boldsymbol{G}$ can also be the discrete Laplacian, [2].

$$\boldsymbol{G}_L = \frac{1}{h^2}\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{3.12}$$

The Laplacian stencil is often used to find edges in an image. An example of this is in Fig **??**.

Pooling is another technique often used in Machine Learning to average part of a database input. In 2D, this is often an image. A parameter called a "stride" is the subsection of the matrix that is being averaged. For example, a matrix of size MxN with a stride S will result in a new size of $\frac{M}{S}\boldsymbol{x}\frac{N}{S}$ after pooling. The pooling operator is defined as:

$$AvePool(\boldsymbol{X})_{kl} = \frac{1}{S^2}\sum_{i=(k-1)S+1}^{kS}\sum_{j=(l-1)S+1} X_{i,j}; k = 1, ..., \frac{M}{S}; l = 1, ..., \frac{N}{S} \tag{3.13}$$

A Convolutional Neural Net combines pooling and convolutions to highlight specific features in image data. One of the benefits of a CNN is that a neural net can be trained on data with a large amount of inputs, because the pooling layers compress the data to a smaller matrix. A typical CNN combines many convolution layers and pooling to decrease the size of

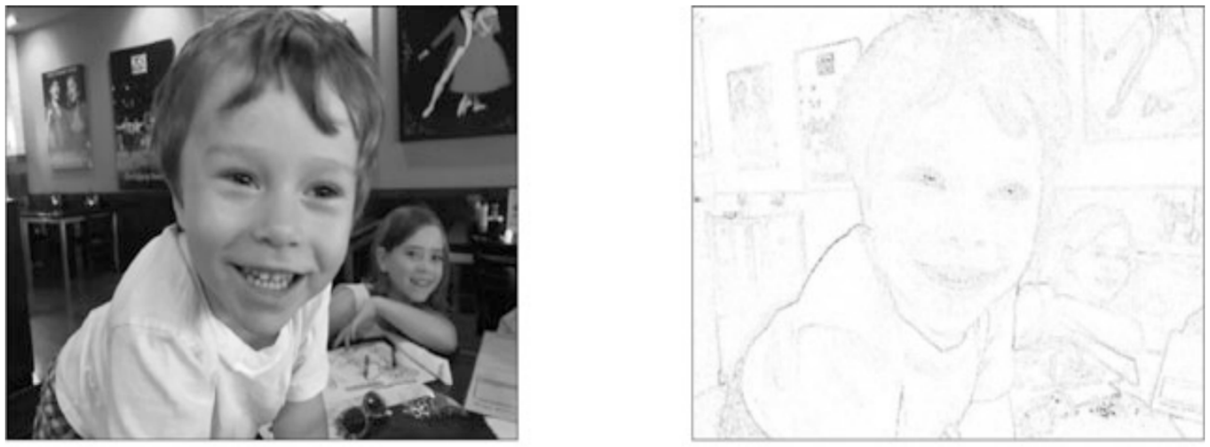

Figure 3.8: An example of 2D convolutions on an image. The one to the right is a 2D Laplacan convolution of the image of the left. [27]

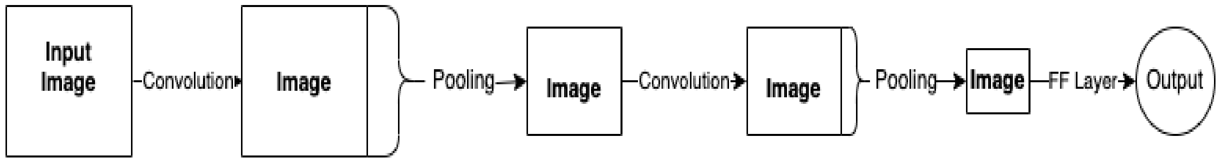

Figure 3.9: A schematic of a Convolutional Neural Net. Each pooling layer shrinks the size of the image and the convolution layers highlight specific features of the image.

the images. Each layer of convolutions and pooling is passed through an activation function. The smaller image is then passed to a feed-forward layer. The feed-forward layers predict the output images. Fig 3.9 shows an example of this process. The feed-forward layers can be used to predict a classification problem or a regression problem. This method can also be expanded to more dimensions for image prediction or 3D models. However, because this type of learning is supervised, it is not as flexible as unsupervised learning.

Chapter 4

Methodology and Implementation

The changes made to a research version of CHEETAH-MC can be broken up into two major categories. The first is the pre-processing step that has been added to predict where the optimal subdomain boundary placement occurs. This requires the CHEETAH-MC input file to be fed into the CNN and the output of the CNN to be passed back to CHEETAH-MC. The second is modifications made to the tracking algorithm itself, which includes the implementation of ghost zones and subdomain boundaries, a sorted particle bank, and swapping the sorted particle banks. This methodology has been tested and developed for history-based Monte Carlo transport through voxel geometry.

Training the CNN required a clearly defined test problem, which is described in Section 4.1. The CNN was first tested on a simple detector-shielding problem to develop a workflow for the CNN and to determine if the CNN is properly suited for this application. Best practices for neural nets is to start with the simplest neural net and hyper-parameters and add complexity if necessary. Once the accuracy and validity of the CNN was deemed suitable, the method was expanded to a voxelized human skull.

Modifications made to the algorithm are outlined in Section 4.2. These changes are highlighted in the pseudo-code, and descriptions of each change are in the following subsections. All of the code development was done in CUDA and C++ to optimize performance on the GPU. CUDA allows for more fine control over the data transfer process and memory management.

The work in this section explores the tools and features implemented in CHEETAH-MC that allow us to choose subdomain boundaries using Machine Learning.

4.1 Convolutional Neural Network Pre-processing

A CNN was developed to predict the flux tallies and internal surface particle count of two test problems using both photon and neutron input data. This predicted data was then used to provide CHEETAH-MC with subdomain boundaries that load balance the geometric domain while minimizing the amount of unnecessary data transfers. Decomposing a geometric domain requires that those two concepts be balanced. The predicted flux informs the load balancing of the Monte Carlo simulation and the internal surface crossing informs the memory transfer requirements of the Monte Carlo simulation. In Monte Carlo, flux tells us where the particles move and how they interact with the geometry, so it is an appropriate predictor for computational work. Particle count tells us how many particles we have to transfer between subdomains, or GPUs.

To avoid placing subdomain boundaries in trivial places, or places that make the subdomains trivially small, a constraint was added to the pre-processing step requiring each subdomain to be at least 25% of the original volume. The boundaries are chosen by finding the maximum predicted flux and minimum predicted internal surface crossing. Again, the flux is used as a proxy for computational work so by doing a search for the minimum internal surface crossing near the maximum flux, we can ensure that both the load balancing and memory transfer frequency are taken into account. The ghost zones are chosen in a similar way.

Similar to how the subdomains were chosen, we can also determine the ghost zones. A ghost zone is the area of overlap between two or more subdomains, these parts of the geometric domain are stored on multiple processors. The purpose of the ghost zone is to prevent unnecessary data transfers. Particles, electrons especially, scatter more as they lose energy. Particles can scatter isotropically, meaning it is equally likely to scatter backwards. If this happens along a subdomain boundary, particles will be transferred between the same two GPUs repeatedly. Moving a single particle back and forth, between two GPUs is costly. The ghost zones are determined by taking the gradient of the flux and finding where the gradient is minimized.

Shielding Problem

The first test problem that was explored was a simple source, shield, detector problem. This was chosen as a proof of concept and shielding problem because the physics is intuitive and simple to predict. It also provided enough flexibility to fill out a large enough database to train a CNN. The location and random holes can be added to a shield to produce enough reasonable randomness for a simplified shield-detector problem. This simple problem served as a test bed to develop a workflow to train a CNN. To develop this data base, a base voxel geometry was generated, shown in Figure 4.1. The base geometry is made up of two simple materials, a highly scattering material and a highly absorbing material. The cross sections for the materials are listed in Table 4.1. The voxel size was set to 1 cm and the total domain size was set to be a 3-dimensional grid of (100,100,1) voxels. Then a highly absorbing slab of material 2 was placed randomly within the geometry. Next, subtle variations were added to the base geometry by making random voxels highly absorbing. A monoenergetic and monodirectional photon source with an energy of 2.5 MeV and photon-only physics were used for this simple proof of concept problem. Finally, tally data was collected for each variation of the geometry. A total of 1000 datapoints were used in the training dataset, and another 300 were used to validate the model. The CNN was trained to map the cross-section

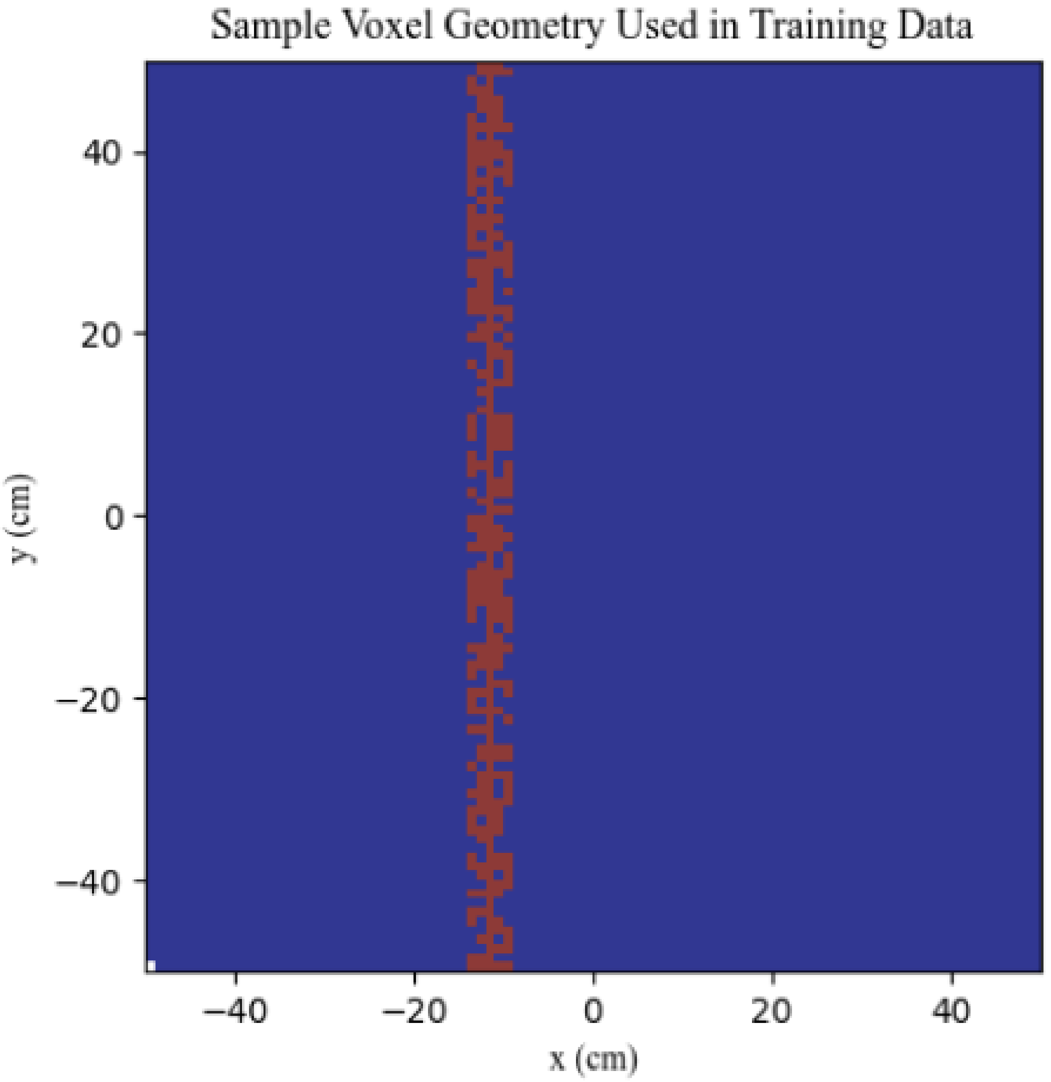

Figure 4.1: Sample voxel shield model used to train the CNN.

data inputs to the tally data that was produced by CHEETAH-MC. Essentially, the CNN was trained to identify and avoid the areas of the geometry with high amounts of scattering and secondary particle production.

Additionally, the CNN was used to determine the ghost-zone size used in the Monte Carlo model. The CNN identified the ghost zone by using the gradient of the predicted tally data to determine where the flux changes the least between some local minimum and maximum. This gives an appropriate constraint to avoid frequent memory transfers.

Material Number	Absorption Cross Section (cm^{-1})	Scatter Cross Section (cm^{-1})
1	0.005	0.01
2	0.2	0.05

Table 4.1: Training Data: Report of geometry parameters used in database.

Human Phantom

The second test problem that was developed was a voxelized human phantom. A human phantom was chosen for a variety of reasons. First, it was already in the correct geometric format. Voxels are not as commonly used as combinatorial geometry in Monte Carlo applications, but human phantoms are often represented by voxel geometry because it is lightweight and compatible with medical physics data processing. However, despite being lightweight, the medical physics community runs into memory capacity issues and the phantoms often have to be domain decomposed [41]. The need for a higher memory capacity within the medical physics community made the human phantom an attractive test problem. The human body also has natural variation that can be exploited to generate a large database for the CNN. Additionally, simplifying human phantoms and approximating complex geometric shapes within the human body is common in medical physics applications [2]. An example of this simplification is shown in Figure 4.2. Taking these assumptions and factors into account allow us to develop a large enough database full of unique human phantom inputs and outputs that are appropriately representative of the male human body.

To develop this database, a base voxel geometry was generated, shown in Figure 4.3. The base geometry is made up of a simplified voxelized skull and brain. The voxel size was arbitrarily set to 0.4 cm and the total domain size was set to be a three-dimensional grid of (50,50,50) voxels. Next, subtle variations were added to the base geometry, motivated by natural variations in the human anatomy. For example, the volume of a brain varies by 17% amongst adult human males [18]. The shape and volume of the brain and skull were varied within these parameters, forming a data base that is representative of a human male skull. The material information for the composition of the brain, bone, and air is shown in Table 4.2. The material composition data is taken from the human phantom database developed by ICRP [21]. The densities of the materials are available in Table 4.3.

A monoenergetic and monodirectional photon source with an energy of 4 MeV was used for this problem, a beam energy typically used in medical physics applications [18]. Secondary particles and electrons were enabled to more accurately model medical physics problems. Finally, photon flux and internal surface crossing tally data was collected for each variation of the geometry. A total of 1000 datapoints were used in the training dataset, and another 300 were used to validate the model. The CNN was trained to map the cross-section data inputs to the tally data that was produced by CHEETAH-MC. Like the simple shielding problem, the CNN was trained to identify and avoid the areas of the geometry with high amounts of scattering and secondary particle production. By placing subdomain boundaries

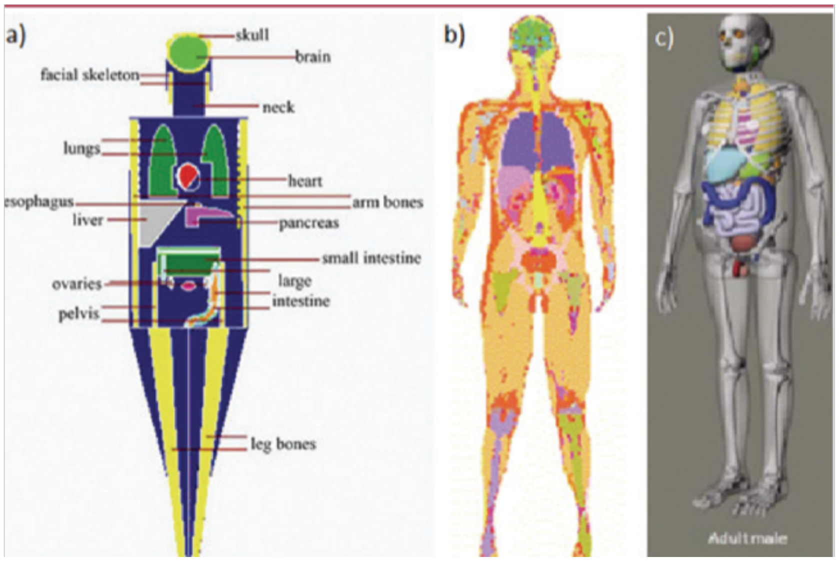

Figure 4.2: An example of human phantoms and how they are simplified. Each version of the human phantom is voxelized, but the geometry becomes simplified from left to right. [2]

Material Name	H	C	N	O	Na	Mg	P	S	Cl	K	Ca
Brain Tissue	0.107	0.144	0.022	0.731	0.713	0.0	0.002	0.004	0.002	0.003	0.0
Bone	0.036	0.159	0.042	0.448	0.003	0.002	0.094	0.003	0.0	0.0	0.213
Air	0.0	0.0	0.2	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 4.2: Training Data: Report of material properties used in database [18].

Material Name	Density ($\frac{g}{cc}$)
Brain Tissue	1.05
Bone	1.92
Air	0.001225

Table 4.3: Training Data: Report of material properties used in database [8].

in these areas, the number of times a particle is moved between two processors is minimized.

Validation Data and Accuracy

The CNN was trained and validated using a built-in Keras package [13]. Keras is a high-level API that was developed for TensorFlow to simplify and make it easier for a user to produce ML models. Keras specifically manages the layers, biasing, loss functions, and metrics dis-

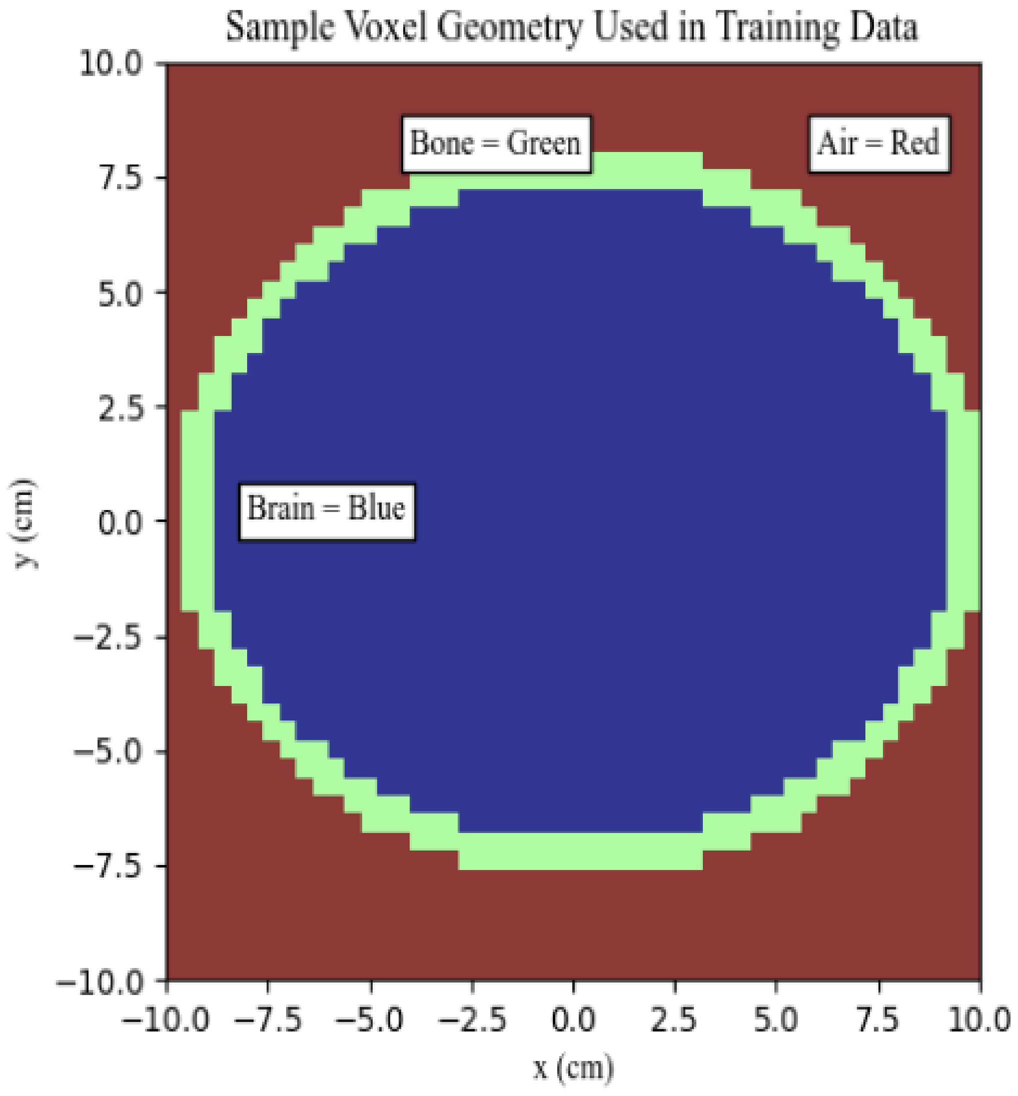

Figure 4.3: Sample simplified brain phantom used to train the CNN.

cussed in Section 3.2. There are built in layers for 3D convolution and hyperparameter tuners, which were both used a starting point for the model used to predict the flux and internal surface crossing [32].

The accuracy reported from these models is calculated using Keras's built in metric functionality. Accuracy of a neural net describes the percentage of predicted values that match the actual values. Therefore, if a predicted value matches the actual value in the database, the value is considered "correct". To compare values a subsection of the training database must be used as a validation set. Best practices for validation training sets is to randomly choose values in the database that were not used to train the neural net. Final reported accuracy is reported by dividing the number of "correct" predictions by the total number of values in the validation set. Typical values for acceptable levels of accuracy range from 70% to 90% for neural nets [26].

4.2 Modified Transport Algorithm

This section describes the modifications made to the domain decomposition algorithm, specifically how the particle bank is sorted and transferred using CUDA memory transfer.

Algorithm 5 Outer Transport Loop

```
1: input parsing
2: initialize source
3: initialize geometry
4: pass geometry to CNN
5: pass CNN subdomain boundaries to host CPU
6: initialize materials
7: initialize tallies
8: for the number of GPUs on compute node do
9:     copy particle bank from CPU to GPU
10:    while $i < N$ and particle bank is not empty do
11:        sample particle's initial state
12:        track particle                                  ▷ Algorithm 1
13:    end while
14:    sort particle bank
15: end for
16: combine tallies
17: write to output
```

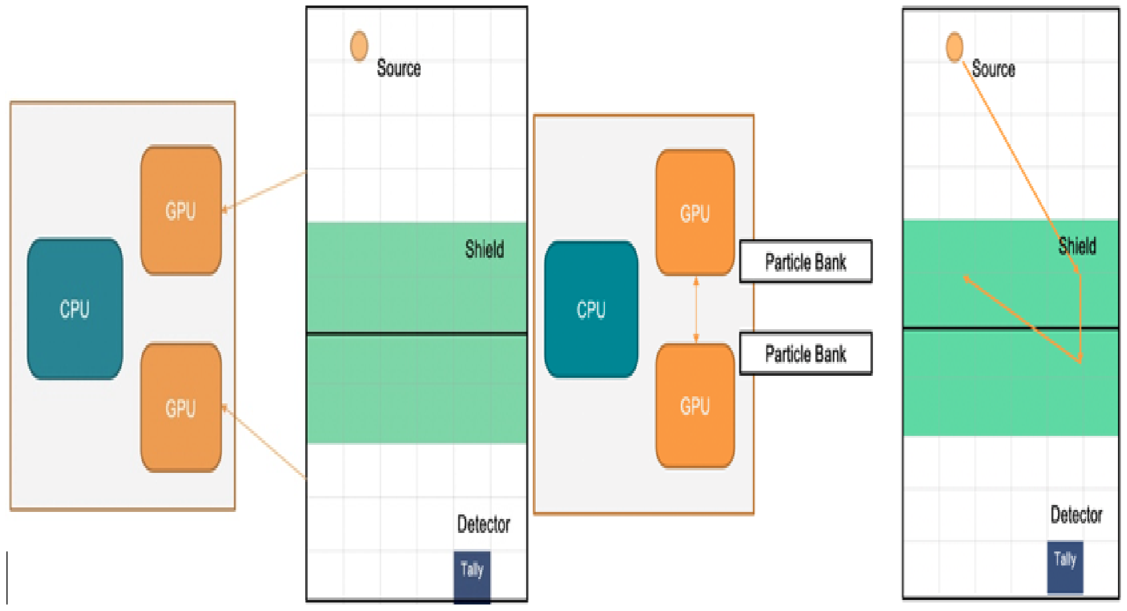

Figure 4.4: An example of a domain containing a source, shield, and detector that has been decomposed into two subdomains. Particles leave and reenter the subdomains as the simulation executes.

Particle Bank

As particles move through the voxel geometry and cross from one subdomain to another, the particles must be transferred from one GPU to another. The particles can be transferred one at a time, but that is not the best use of computational resources. As discussed in Section 3.1, memory transfer and allocation is often the bottleneck of high performance computing. Transferring particle by particle is inefficient, therefore a particle bank was implemented to store particles temporarily before being transferred. The particles are stored in a simple fixed-length array.

For the simplist case, a compute node with only two GPUs, the particle banks can be swapped directly. CUDA allows for direct GPU-to-GPU communication, which is faster than transferring the banks through the CPU and back to the GPUs. Figure 4.4 illustrates the power of device-to-device memory transfer.

For more than two GPUs, direct GPU-to-GPU communication cannot be utilized. The subdomains have more than one nearest neighbor, so the particles must be stored with additional subdomain information. Each subdomain is labeled with a unique numerical ID and this subdomain label is stored in the particle data. The subdomain label is updated as the particles approach and cross the subdomain boundaries. Once the particle bank is full, the bank must be transferred back to the CPU and sorted. The particles are then redistributed to the GPUs, as shown in Figure 4.5.

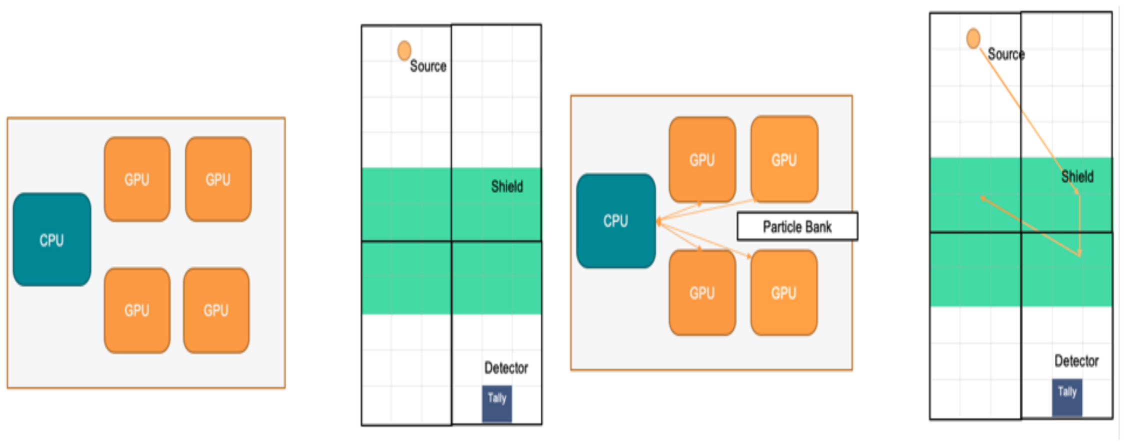

Figure 4.5: An example of a domain containing a source, shield, and detector that has been decomposed into four subdomains. Particles leave and reenter the subdomains as the simulation executes.

Sorting

As GPUs grow in popularity, sorting algorithms are being remapped to the GPU architecture. Specifically, algorithms that utilize the divide and conquer strategy. Divide and conquer describes a class of sorting algorithms that break down an original problem into smaller subproblems that mirror the original problem [28]. Each subproblem is then solved recursively and the solutions are then combined for the original problem. However, true recursion cannot be implemented on the GPU, therefore recursion is emulated by calling a kernel on a segment of the original data. Because each subproblem is independent from one another, they can be distributed across shared memory systems and distributed memory systems, making divide and conquer algorithms compatible with GPUs [37]. The subproblems are assigned to a thread block on the GPU.

The sorting algorithm implemented in CHEETAH-MC was a simple count sort algorithm [38]. The count sort algorithm is particularly powerful when the range of the input values is small compared to the number of elements that need to be sorted, like in the case where there are only 4 GPUs to sort. The count sort algorithm counts the frequency of each unique element and then places the elements in the correct order.

Chapter 5

Results

This chapter presents the results for the shielding problem, the human phantom problem with a photon source, and the human phantom problem with an electron source. The shielding problem was developed first as a proof of concept and testing ground for CNNs. Then a new CNN was trained for the human phantom problem with a photon source. For each CNN trained, the runtime and accuracy is reported in the relevant sections below. The runtimes are then compared to arbitrarily chosen boundaries and uniform boundaries.

The accuracy of the model can be improved by tuning the hyperparameters of the CNN. The subdomain boundaries are chosen by using the predicted flux and internal surface crossing tallies to estimate where the internal surface crossings are minimized. The subdomains were also limited in size to be at least 25% of the volume of the overall domain to avoid null results, such as a subdomain that does a negligible amount of computational work.

The domain decomposition scheme was extended beyond two subdomains for the human phantom model with a photon source. The results for the four subdomain problem, along with a comparison to other decomposition schemes are presented in the final section.

5.1 Shielding Problem

The shielding problem was developed to test the capabilities of a CNN. It was determined to be a relevant test because it has intuitive physics. This model was only tested on two NVIDIA Volta GPUs. The CNN had a prediction accuracy of 78.65% with a training time of 260 seconds, which is reasonable for a simple proof of concept neural net. The boundaries chosen by the CNN were compared to arbitrarily chosen domain boundaries, Figure 5.2, and uniform domain boundaries, Figure 5.3. The CNN prediction, Figure 5.1, ran 1.6x faster than arbitrarily chosen boundaries and 1.1x faster than uniformly chosen boundaries, as shown in Table 5.1.

Each figure in this section shows different domain decomposition schemes on the same base geometry. Each subdomain and ghost zone is labeled and is filled in with a unique color. The flux tally is also plotted in the same figures. The uniform boundary and the

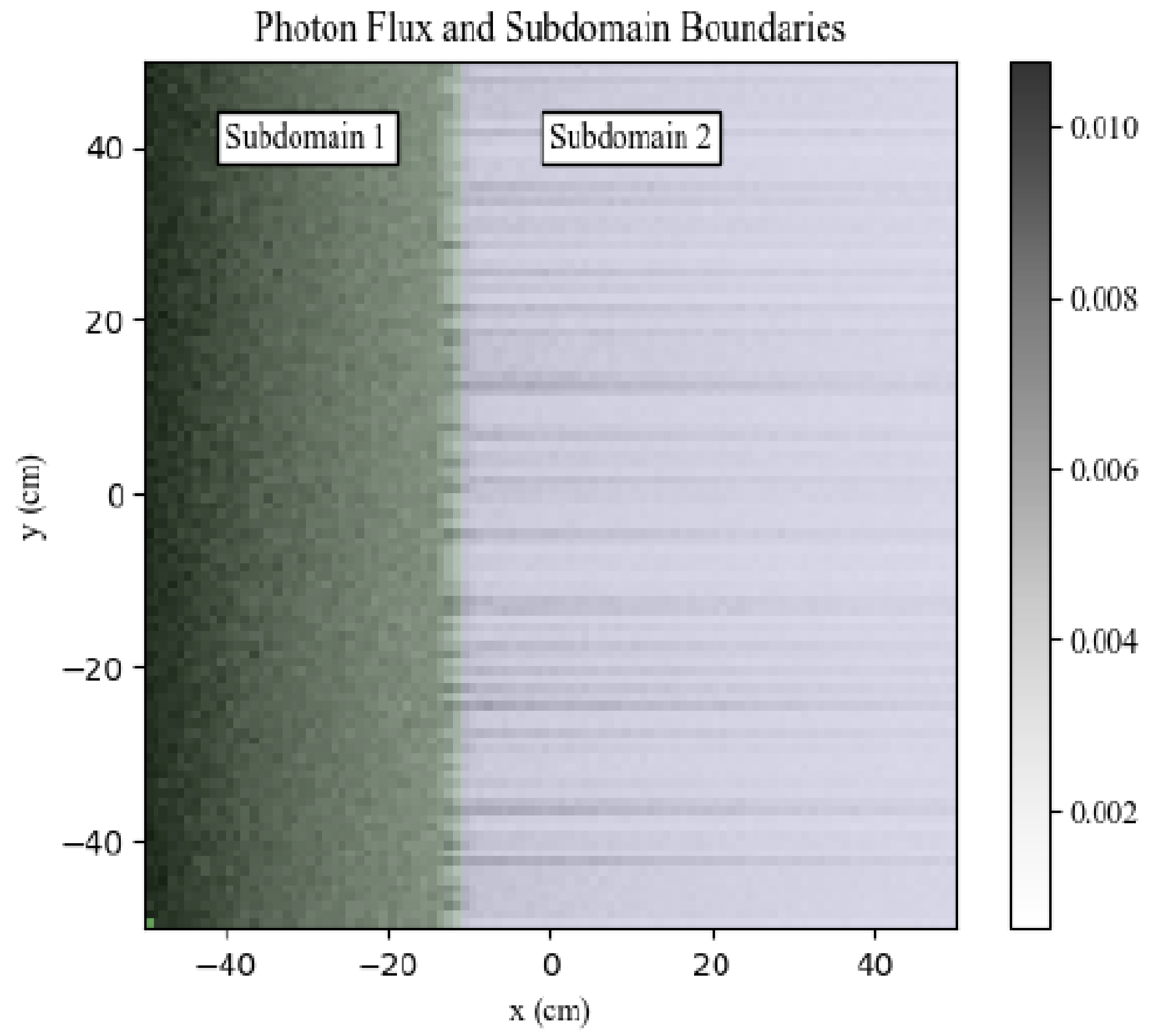

Figure 5.1: Photon flux in a CNN decomposed simple shield voxel geometry with ghost zones.

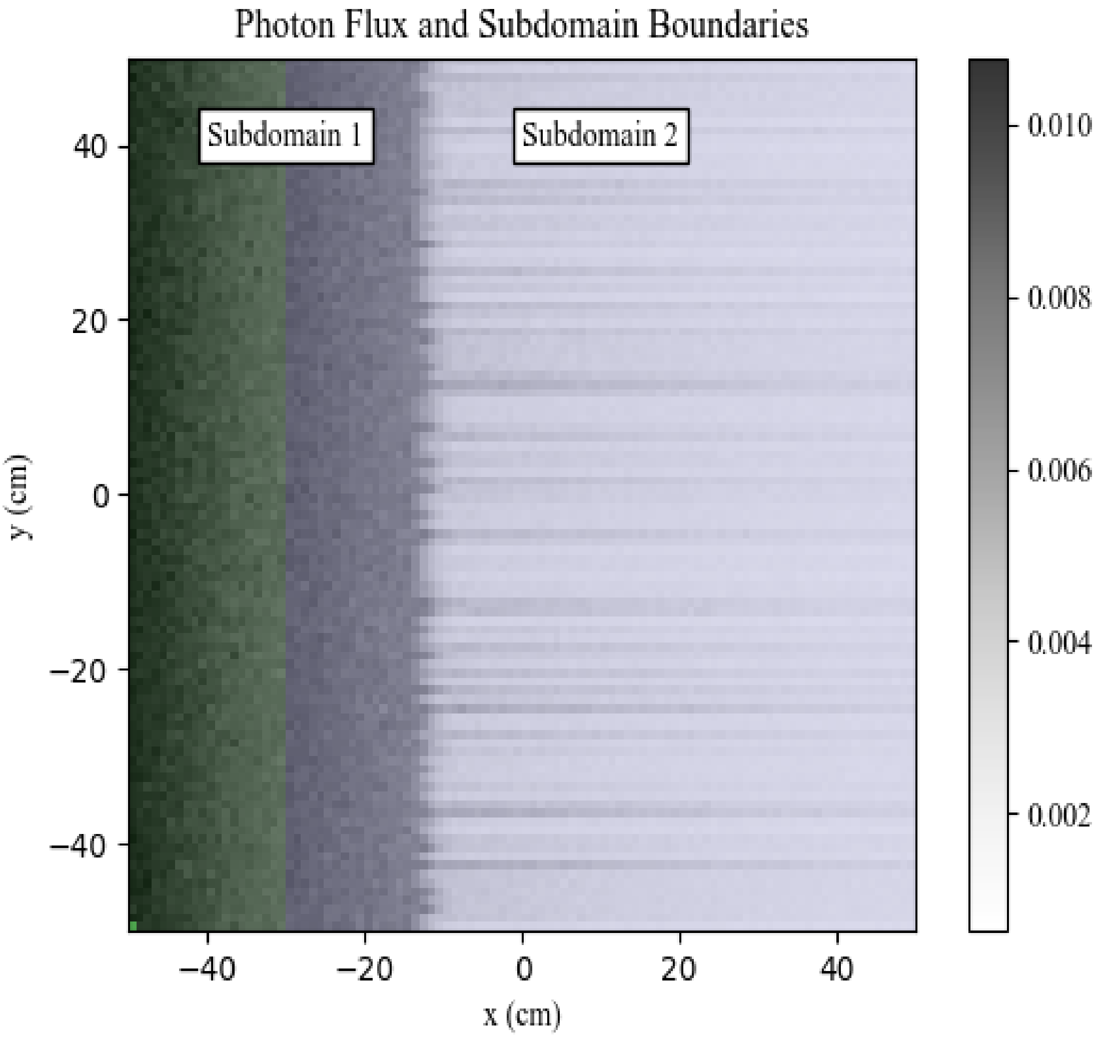

Figure 5.2: Photon flux in a arbitrarily decomposed simple shield voxel geometry with ghost zones.

CCN chosen boundaries are not significantly different in this particular randomly generated geometry because the uniform subdomain boundary is also placed after the shield. The shield stops a majority of the incident particles and results in fewer memory transfers. However, if the boundary is placed before the shield like in the arbitrarily chosen boundaries, the runtime increases.

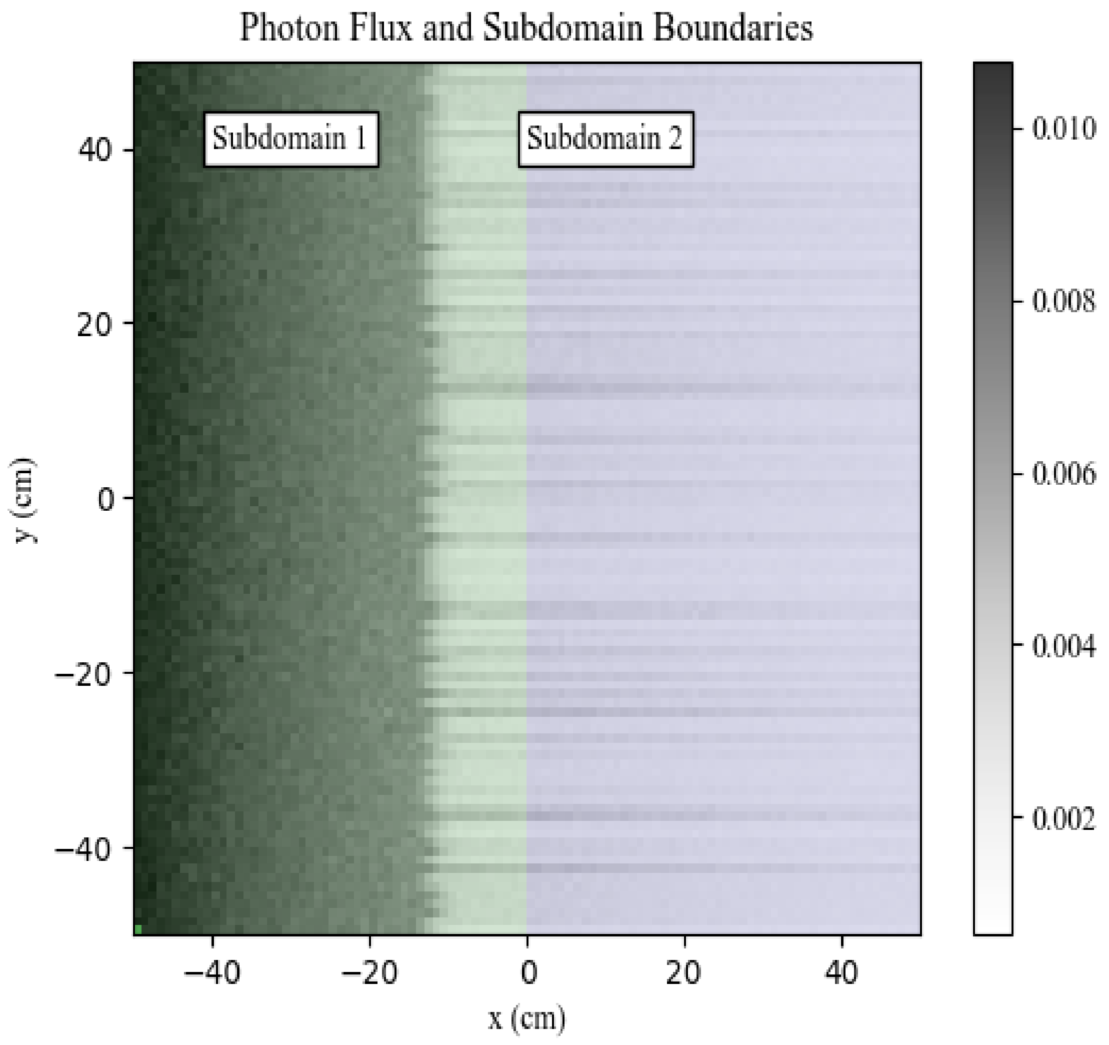

Figure 5.3: Photon flux in a uniformly decomposed simple shield voxel geometry with ghost zones.

Decomposition Scheme	Time
Uniform	165.41 seconds
Arbitrary	246.2 seconds
CNN Prediction	150.12 seconds

Table 5.1: Run times for a shielding model using 2 NVIDIA Volta GPUs.

Decomposition Scheme	Time
Uniform	682.43 seconds
Arbitrary	829.32 seconds
CNN Predicted	628.28 seconds

Table 5.2: Run times for a human phantom model using a photon source using 2 NVIDIA Volta GPUs.

5.2 Human Phantom

This section presents the results for the human phantom problem. The human phantom was used to train a CNN using a photon source. The CNN chosen boundaries were compared to other decomposition schemes, such as uniform and arbitrarily user-chosen boundaries. The runtimes of each decomposition scheme are presented below along with a depiction of the flux and subdomains. The human phantom problem was first tested on two GPUs then four. Each figure is a 2D cross section of the human skull phantom.

Photon Flux Tally: 2 GPUs

The human phantom geometry with a photon source was developed to further test the capabilities of the CNN. The model was initially tested on two NVIDIA Voltas then later extended to four. The CNN had an accuracy of 82.71% with a training time of 831 seconds. The CNN prediction ran 1.1x faster than arbitrarily chosen boundaries and 1.3x faster than uniformly chosen boundaries, as shown in Table 5.2.

Figure 5.4 shows the different subdomain boundaries chosen by the CNN. Each subdomain is labeled and the colors represents each GPU where the geometry resides. The flux is also plotted along the human phantom geometry. It is illustrated as a black and white gradient. The uniformly chosen subdomain boundaries, Figure 5.5, are similar to the ones chosen by the CNN so the total runtime is not significantly different. The arbitrarily chosen boundaries, Figure 5.6, perform the worst.

Photon Flux Tally: 4 GPUs

The human phantom model was then extended to run on 4 NVIDIA Volta GPUs. The same CNN from Section 5.2 was used to decompose this geometry. However, the constraints were changed slightly to account for the additional subdomains. Instead of limiting subdomain size to 25%, it was limited to 12.5%. The human phantom ran slower on 4 GPUs, this is likely due to the sorting step that needs to take place between memory transfers. Additionally, GPU to GPU communication was not utilized in the 4 GPU run. All particle banks were sent to the host CPU, sorted, and redistributed to the GPUs. These additional memory

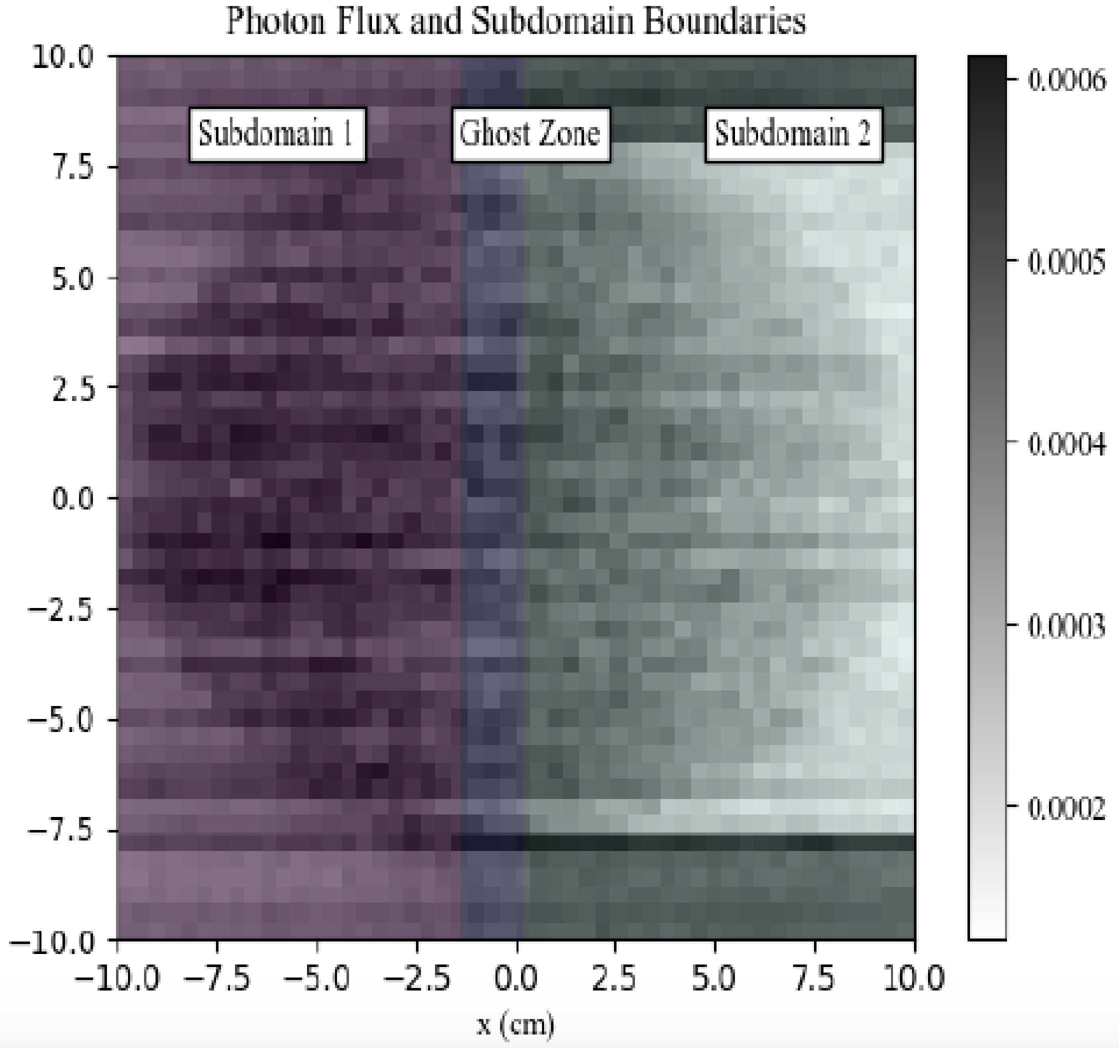

Figure 5.4: Photon flux in CNN decomposed simple brain and skull human phantom voxel geometry with ghost zones.

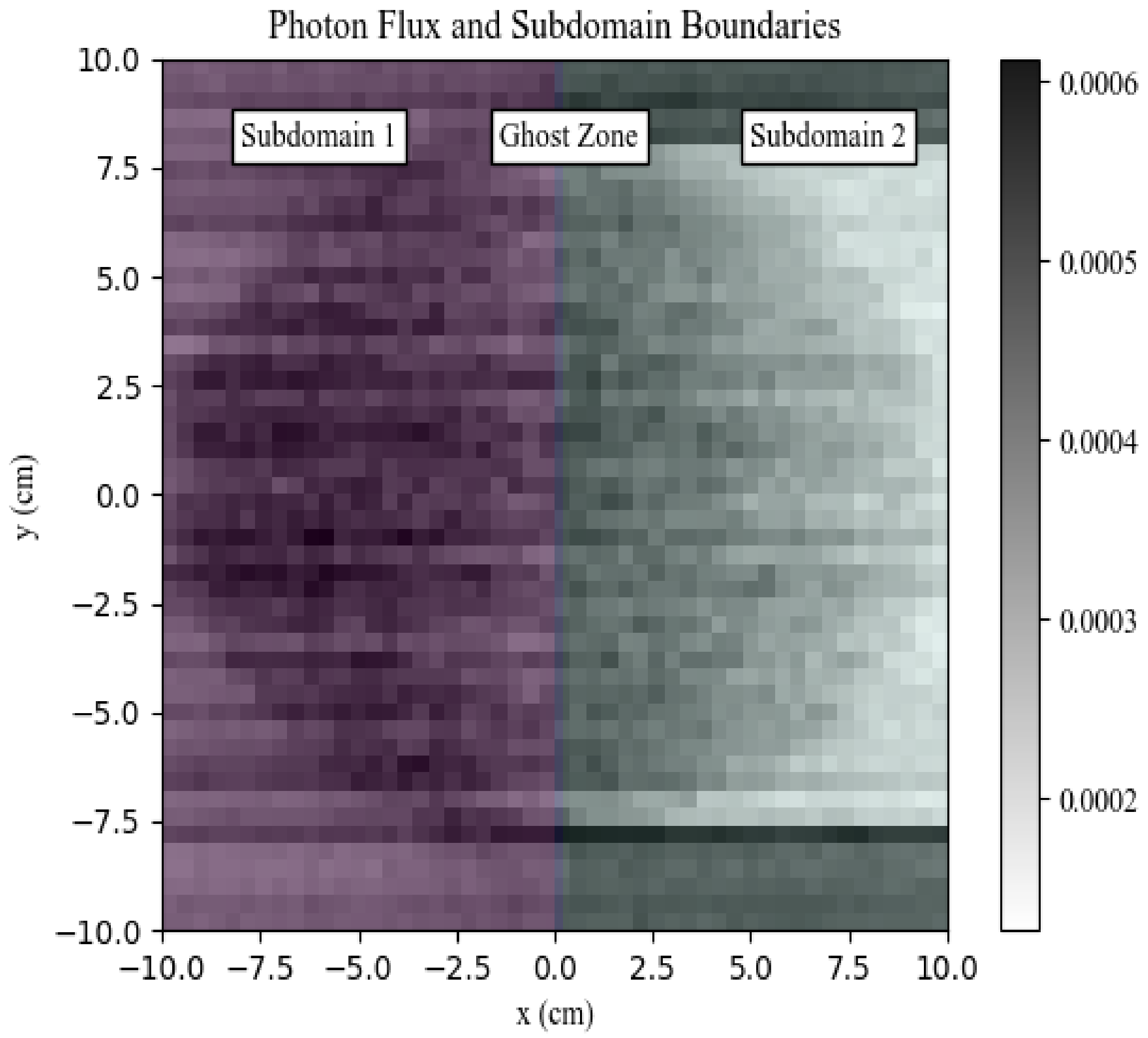

Figure 5.5: Photon flux in uniformly decomposed simple brain and skull human phantom voxel geometry with ghost zones.

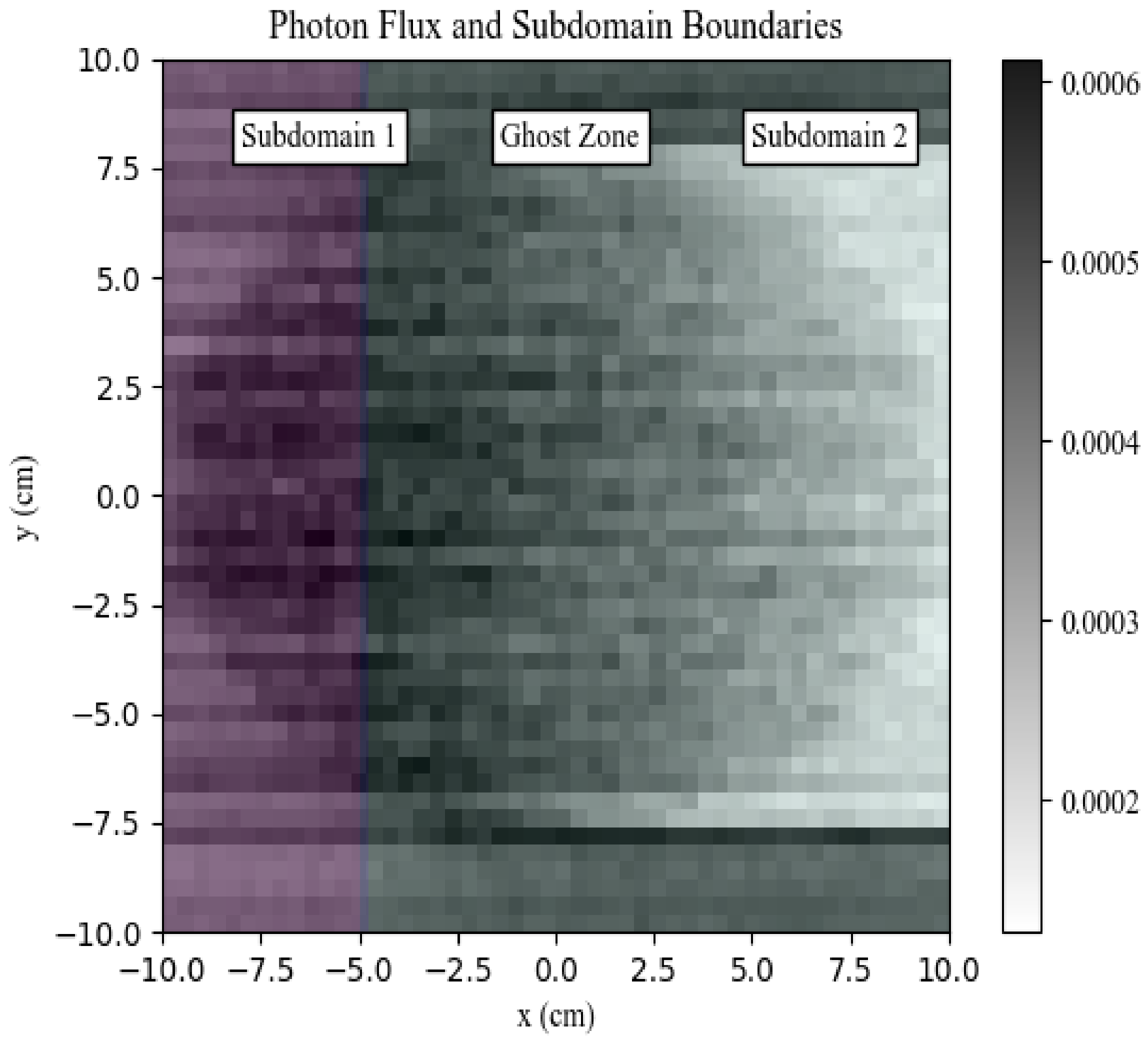

Figure 5.6: Photon flux in arbitrarily decomposed simple brain and skull human phantom voxel geometry with ghost zones.

Decomposition Scheme	Time
Uniform	1281.14 seconds
Arbitrary	1651.87 seconds
CNN Predicted	1146.32 seconds

Table 5.3: Run time for a human phantom model using a photon source using 4 NVIDIA Volta GPUs.

Problem Type	Particle Type	Time
Shield	photon	41.7 CPU hours
Human Phantom (photon source)	electron-photon	216.7 CPU hours

Table 5.4: CPU hours required to generate the databases required to train the CNN.

transfers added to the final runtime. The CNN prediction ran 1.1x faster than arbitrarily chosen boundaries and 1.4x faster than uniformly chosen boundaries, as shown in Table 5.3.

Figure 5.7 shows the different subdomain boundaries chosen by the CNN. Each subdomain is labeled and the colors represents each GPU where the geometry resides, however the ghost zones are not shown in these plots because the figures become too difficult to read. The uniformly chosen subdomain boundaries, Figure 5.8, are split evenly along the origin. The arbitrarily chosen boundaries, Figure 5.9, perform the worst but the difference is not as stark as the other two results presented in this chapter. This is likely due to the additional sorting step and more frequent memory transfers. Additional testing would need to be conducted with larger, more complex geometries to fully understand the scaling of this method. However, Monte Carlo performance is very problem dependent, so this would require further study and research into large benchmark problems in the medical physics community.

5.3 Database Development Time

To train the CNN, a total of 1000 cases were needed for the shielding problem and human voxel problem. The total time it took to generate the database is presented below in Table 5.4. The times presented are estimates and not fully accurate they are calculated based on the time it took to run a single case. The amortized time is not reflected in the CNN training time, presented earlier in this chapter.

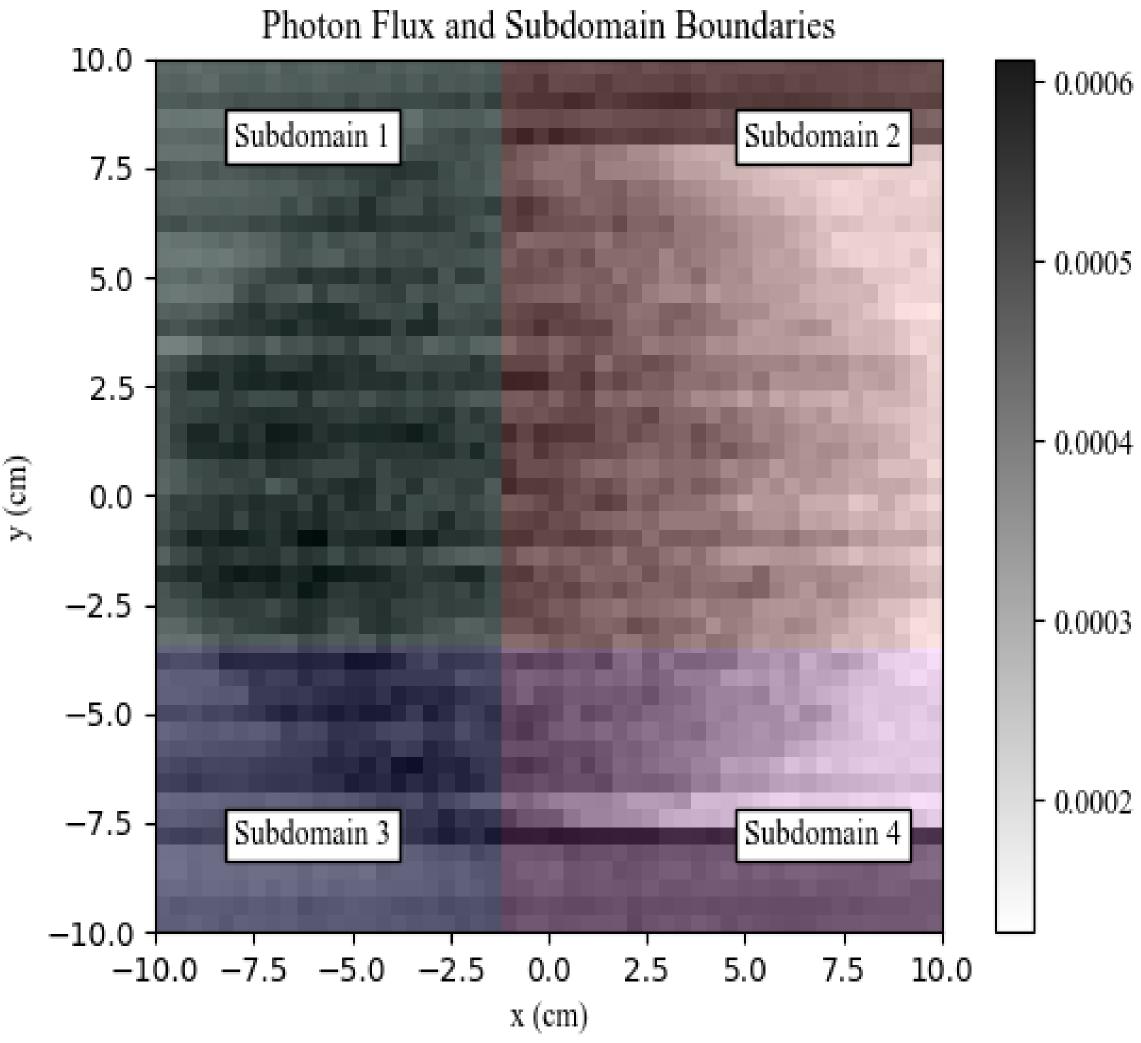

Figure 5.7: Photon flux in CNN decomposed simple brain and skull human phantom voxel geometry with ghost zones.

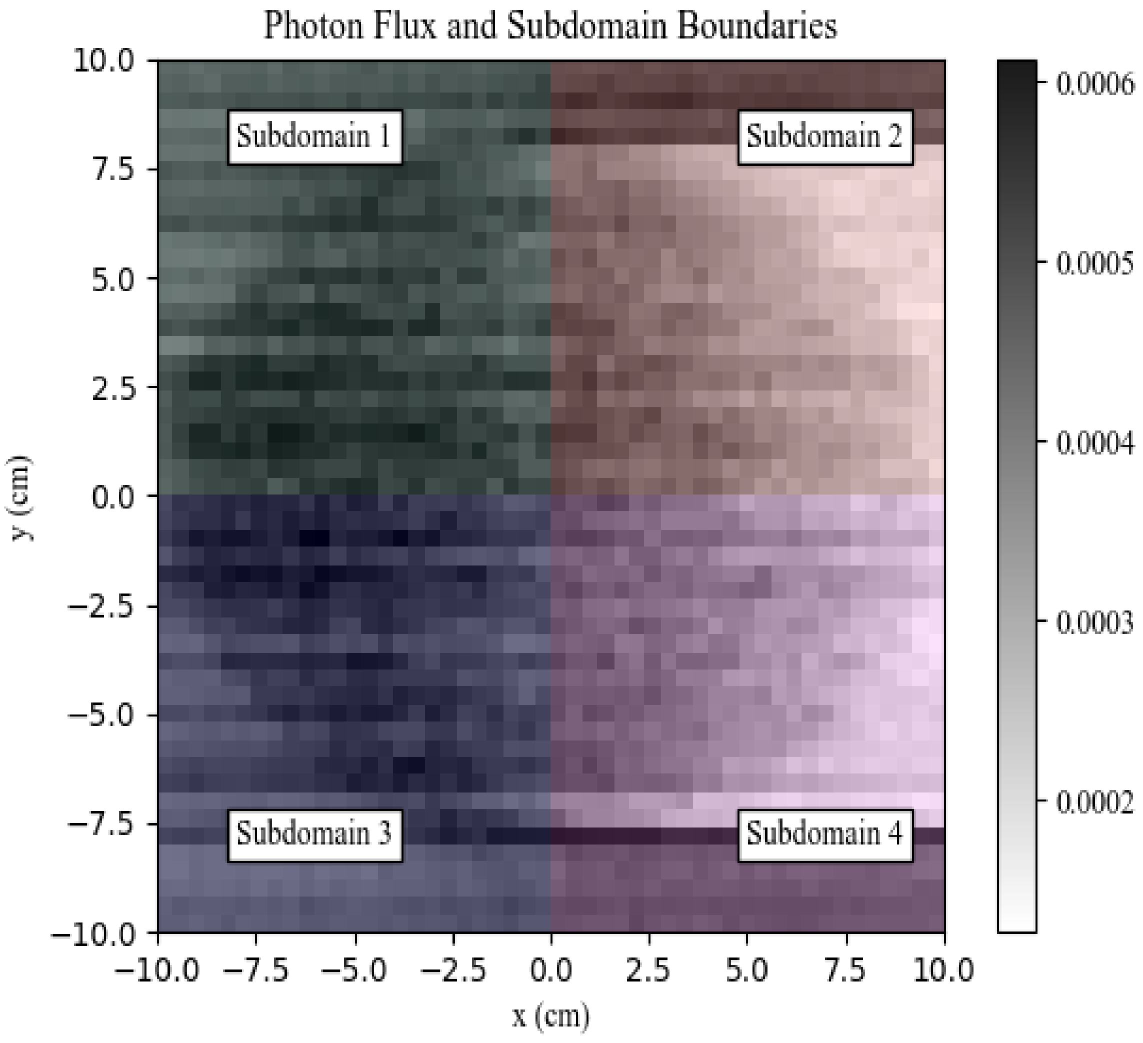

Figure 5.8: Photon flux in uniformly decomposed simple brain and skull human phantom voxel geometry with ghost zones.

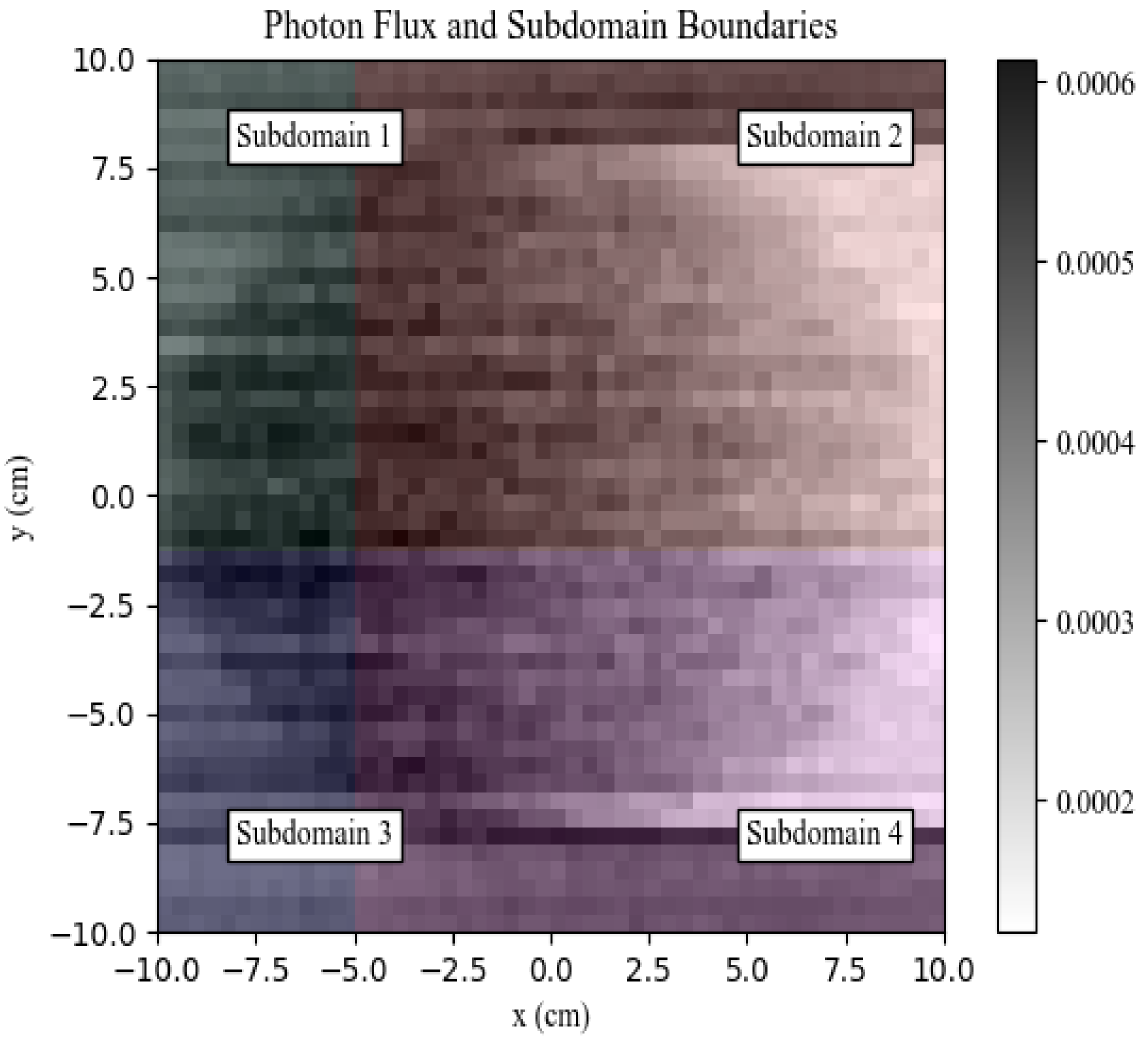

Figure 5.9: Photon flux in arbitrarily decomposed simple brain and skull human phantom voxel geometry with ghost zones.

Chapter 6

Conclusions and Future Work

Using a Convolutional Neural Net to intelligently choose subdomain boundaries performs the best from the other methods presented in this work, such as user-defined boundaries and uniformly-defined boundaries. The CNN outperforms other methods for a voxelized human skull and brain and a simple shielding problem. This methodology can be expanded to other commonly simulated electron-photon problems. However, due to limitations on CHEETAH-MC user-generated training data available, Neural Nets would benefit a code with a larger user base and readily-available data.

The runtimes also do not fully reflect the amount of time it took to train the models and generate the data. Taking this additional time into account makes it difficult to justify a full database generation.

Conclusions

Domain decomposition is commonly used in production level Monte Carlo codes. The way it is implemented varies from code to code, but most codes have the option for users to directly define subdomain boundaries, which can lead to arbitrarily chosen boundaries. User defined boundaries have the longest runtimes and present the worst case scenario for domain decomposition. The CNN chosen boundaries perform 1.6x faster than the arbitrarily chosen boundaries. The CNN outperformed arbitrarily chosen boundaries by 1.4x for the human phantom problem. Users often do not have the physics knowledge to correctly identify the optimal subdomain boundaries. These results demonstrate that at a very minimum, production level codes should not recommend user-defined subdomain boundaries and implement a version that removes that requirement from the user.

In the second case presented, the method that breaks up the domain into uniformly sized subdomains. This method is available in some, but not all, production level Monte Carlo codes. The CNN chosen boundaries ran 1.1x faster than uniformly chosen boundaries for the shielding problem and 1.4x faster for the human phantom problem. This is not significant enough to necessitate training and data base production for every potential Monte Carlo model. However, it does provide a proof of concept for other applications of Neural Nets in

Monte Carlo. This method would work better for a more standardized input model, like a nuclear reactor core. Nuclear reactor cores all have the same basic structure and features, so training a neural net to identify the features that significantly impact runtime when decomposed.

Furthermore when extended to 4 GPUs, the human phantom problem performed worse, likely due to the use of CPU-GPU communication rather than direct GPU-to-GPU communication. Extending this problem to more than 4 GPUs would reveal how the human phantom model continues to scale.

Future Work

Neural nets are powerful predictive algorithms that have many potential use cases in Monte Carlo methods. The work presented in this dissertation offers one application. This work can be expanded in a multitude of ways. Expanding, this methodology to combinatorial geometry is the first and most obvious step. However, users typically expect to create a large variety of CG models, ranging in size and complexity. Using supervised learning for these cases is rather limiting, unless there is a larger database with a sample of different user inputs and outputs. Storing data of that size would be difficult and cost prohibitive. An alternative approach for future research is using reinforcement learning to optimize the load balancing of the CG model. Reinforcement learning can be implemented to take in runtime data and reshuffle the subdomain boundaries on the fly. Using Machine Learning in this way removes any requirement to store user data long term. However, training a neural net in real time also requires long runtimes, so it may be time prohibitive or only necessary for Monte Carlo models that are slow to converge.

A potentially easier to implement use case for ML is to use a radically simpler classification neural net to chose which types of variance reduction techniques to use for the user developed Monte Carlo models. Variance reduction is a class of methods that are used to reduce the convergence time, and consequently runtime, of Monte Carlo methods. However, there are a variety of techniques available and each one has different benefits depending on the Monte Carlo model. Using a classification neural net to choose the bet variance reduction technique could prevent user pitfalls.

Bibliography

[1] Frank Herbert Attix. *Introduction to Radiological Physics and Radiation Dosimetry.* en. 1st ed. Wiley, Nov. 1986. ISBN: 978-0-471-01146-0 978-3-527-61713-5. DOI: 10.1002/9783527617135. URL: https://onlinelibrary.wiley.com/doi/book/10.1002/9783527617135 (visited on 10/16/2023).

[2] Elie Hoseinian Azghadi, Laleh Rafat Motavalli, and Hashem Miri Hakimabad. "Hybrid Phantom Applications to Nuclear Medicine". In: *Journal of biomedical physics & engineering* 2 (2011), pp. 37–41. URL: https://api.semanticscholar.org/CorpusID:14760560.

[3] M. T. L. B. In: *Journal of the Institute of Actuaries (1886-1994)* 84.2 (1958), pp. 232–234. ISSN: 00202681. URL: http://www.jstor.org/stable/41139316 (visited on 10/16/2023).

[4] MJ Berger. "Monte Carlo calculation of the penetration and diffusion of fast charged particles". In: *Methods in Computational Physics* 1 (1963).

[5] Ryan Bergmann. "The Development of WARP - A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs". In: (2014).

[6] Ryan C Bleile. "ENHANCING MONTE CARLO PARTICLE TRANSPORT FOR MODERN MANY-CORE ARCHITECTURES". In: (2021).

[7] Kerry Bossler, Martin Crawford, and Luke Kersting. "CHEETAH-MC: Next-generation Monte Carlo electron-photon transport". In: (2023).

[8] Forrest B. Brown. "Advanced Computational Methods for Monte Carlo Calculations". In: (Jan. 2018). DOI: 10.2172/1417155. URL: https://www.osti.gov/biblio/1417155.

[9] Forrest B. Brown. "Monte Carlo Techniques for Nuclear Systems - Theory Lectures". In: (Nov. 2016). DOI: 10.2172/1334102. URL: https://www.osti.gov/biblio/1334102.

[10] Forrest B. Brown and William R. Martin. "Monte Carlo methods for radiation transport analysis on vector computers". In: *Progress in Nuclear Energy* 14.3 (1984), pp. 269–299. ISSN: 0149-1970. DOI: https://doi.org/10.1016/0149-1970(84)90024-6. URL: https://www.sciencedirect.com/science/article/pii/0149197084900246.

[11] Namjae Choi, Kyung Min Kim, and Han Gyu Joo. "Initial Development of PRAGMA – A GPU-Based Continuous Energy Monte Carlo Code for Practical Applications". en. In: (2019).

[12] Namjae Choi, Kyung Min Kim, and Han Gyu Joo. "Optimization of neutron tracking algorithms for GPU-based continuous energy Monte Carlo calculation". In: *Annals of Nuclear Energy* 162 (2021), p. 108508. ISSN: 0306-4549. DOI: `https://doi.org/10.1016/j.anucene.2021.108508`. URL: `https://www.sciencedirect.com/science/article/pii/S0306454921003844`.

[13] François Chollet et al. *Keras*. `https://keras.io`. 2015.

[14] Handel Davies, H. A. Bethe, and L. C. Maximon. "Theory of Bremsstrahlung and Pair Production. II. Integral Cross Section for Pair Production". In: *Phys. Rev.* 93 (4 Feb. 1954), pp. 788–795. DOI: `10.1103/PhysRev.93.788`. URL: `https://link.aps.org/doi/10.1103/PhysRev.93.788`.

[15] James J. Duderstadt et al. "Nuclear Reactor Analysis by James J. Duderstadt and Louis J. Hamilton". In: *IEEE Transactions on Nuclear Science* 24.4 (1977), pp. 1983–1983. DOI: `10.1109/TNS.1977.4329141`.

[16] William L. Dunn and J. Kenneth Shultis. "Central Limit Theorem". en. In: *Exploring Monte Carlo Methods*. Elsevier, 2023, pp. 501–506. ISBN: 978-0-12-819739-4. DOI: `10.1016/B978-0-12-819739-4.00021-4`. URL: `https://linkinghub.elsevier.com/retrieve/pii/B9780128197394000214` (visited on 09/13/2023).

[17] U. Fano, K. W. McVoy, and James R. Albers. "Sauter Theory of the Photoelectric Effect". en. In: *Physical Review* 116.5 (Dec. 1959), pp. 1147–1156. ISSN: 0031-899X. DOI: `10.1103/PhysRev.116.1147`. URL: `https://link.aps.org/doi/10.1103/PhysRev.116.1147` (visited on 10/16/2023).

[18] Xu Feng et al. "COMPARISON OF ORGAN DOSES IN HUMAN PHANTOMS: VARIATIONS DUE TO BODY SIZE AND POSTURE". en. In: *Radiation Protection Dosimetry* (Apr. 2016), ncw081. ISSN: 0144-8420, 1742-3406. DOI: `10.1093/rpd/ncw081`. URL: `https://academic.oup.com/rpd/article-lookup/doi/10.1093/rpd/ncw081` (visited on 10/17/2023).

[19] Steven P. Hamilton et al. "Domain decomposition in the GPU-accelerated Shift Monte Carlo code". In: *Annals of Nuclear Energy* 166 (2022), p. 108687. ISSN: 0306-4549. DOI: `https://doi.org/10.1016/j.anucene.2021.108687`. URL: `https://www.sciencedirect.com/science/article/pii/S0306454921005636`.

[20] Iestyn Jowers. "Computation with Curved Shapes: Towards Freeform Shape Generation in Design". In: (Nov. 2023).

[21] Wolfgang Kainz et al. "Advances in Computational Human Phantoms and Their Applications in Biomedical Engineering—A Topical Review". In: *IEEE Transactions on Radiation and Plasma Medical Sciences* 3.1 (2019), pp. 1–23. DOI: `10.1109/TRPMS.2018.2883437`.

[22] O. Klein and Y. Nishina. “Über die Streuung von Strahlung durch freie Elektronen nach der neuen relativistischen Quantendynamik von Dirac”. de. In: *Zeitschrift für Physik* 52.11-12 (Nov. 1929), pp. 853–868. ISSN: 0044-3328. DOI: `10.1007/BF01366453`. URL: `http://link.springer.com/10.1007/BF01366453` (visited on 10/16/2023).

[23] H. W. KOCH and J. W. MOTZ. “Bremsstrahlung Cross-Section Formulas and Related Data”. In: *Rev. Mod. Phys.* 31 (4 Oct. 1959), pp. 920–955. DOI: `10.1103/RevModPhys.31.920`. URL: `https://link.aps.org/doi/10.1103/RevModPhys.31.920`.

[24] J R Lamarsh. “Introduction to nuclear engineering”. In: (Jan. 1975). URL: `https://www.osti.gov/biblio/5272097`.

[25] Edward W. Larsen. “A theoretical derivation of the Condensed History Algorithm”. In: *Annals of Nuclear Energy* 19.10 (1992). In Honour of Jacques Devooght, pp. 701–714. ISSN: 0306-4549. DOI: `https://doi.org/10.1016/0306-4549(92)90013-2`. URL: `https://www.sciencedirect.com/science/article/pii/0306454992900132`.

[26] Navin Kumar Manaswi. “Understanding and Working with Keras”. In: *Deep Learning with Applications Using Python : Chatbots and Face, Object, and Speech Recognition With TensorFlow and Keras*. Berkeley, CA: Apress, 2018, pp. 31–43. ISBN: 978-1-4842-3516-4. DOI: `10.1007/978-1-4842-3516-4_2`. URL: `https://doi.org/10.1007/978-1-4842-3516-4_2`.

[27] Ryan G. McClarren. “Convolutional Neural Networks for Scientific Images and Other Large Data Sets”. en. In: *Machine Learning for Engineers*. Cham: Springer International Publishing, 2021, pp. 149–172. ISBN: 978-3-030-70387-5 978-3-030-70388-2. DOI: `10.1007/978-3-030-70388-2_6`. URL: `https://link.springer.com/10.1007/978-3-030-70388-2_6` (visited on 10/06/2023).

[28] Gang Mei et al. “A sample implementation for parallelizing Divide-and-Conquer algorithms on the GPU”. In: *Heliyon* 4.1 (2018), e00512. ISSN: 2405-8440. DOI: `https://doi.org/10.1016/j.heliyon.2018.e00512`. URL: `https://www.sciencedirect.com/science/article/pii/S2405844016326032`.

[29] Thomas Miller. “ITS version 6.4: The Integrated TIGER Series of Monte Carlo Electron/Photon Radiation Transport Codes”. en. In: ().

[30] Alexander Mote et al. “Advancements in Shift: Time-Dependent Domain Decomposition and ML-Aided Processor Allocation”. In: (2023).

[31] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. *CUDA, release: 10.2.89*. 2020. URL: `https://developer.nvidia.com/cuda-toolkit`.

[32] Tom O’Malley et al. *KerasTuner*. `https://github.com/keras-team/keras-tuner`. 2019.

[33] Tara M. Pandya et al. “Implementation, capabilities, and benchmarking of Shift, a massively parallel Monte Carlo radiation transport code”. In: *Journal of Computational Physics* 308 (2016), pp. 239–272. ISSN: 0021-9991. DOI: `https://doi.org/10.1016/j.jcp.2015.12.037`. URL: `https://www.sciencedirect.com/science/article/pii/S0021999115008566`.

[34] “PENELOPE-2018: A Code System for Monte Carlo Simulation of Electron and Photon Transport, Workshop Proceedings, Barcelona, Spain, 28 January-1 February 2019”. en. In: (2019).

[35] M Pozulp et al. “Progress Porting LLNL Monte Carlo Transport Codes to Nvidia GPUs”. In: (2023).

[36] Paul K. Romano et al. “OpenMC: A state-of-the-art Monte Carlo code for research and development”. In: *Annals of Nuclear Energy* 82 (2015). Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013, SNA + MC 2013. Pluri- and Trans-disciplinarity, Towards New Modeling and Numerical Simulation Paradigms, pp. 90–97. ISSN: 0306-4549. DOI: `https://doi.org/10.1016/j.anucene.2014.07.048`. URL: `https://www.sciencedirect.com/science/article/pii/S030645491400379X`.

[37] Nadathur Satish, Mark Harris, and Michael Garland. “Designing efficient sorting algorithms for manycore GPUs”. In: *2009 IEEE International Symposium on Parallel Distributed Processing*. 2009, pp. 1–10. DOI: `10.1109/IPDPS.2009.5161005`.

[38] Weidong Sun and Zongmin Ma. “Count Sort for GPU Computing”. In: *2009 15th International Conference on Parallel and Distributed Systems*. 2009, pp. 919–924. DOI: `10.1109/ICPADS.2009.30`.

[39] Jasmina L. Vujic and William R. Martin. “Vectorization and parallelization of a production reactor assembly code”. In: *Progress in Nuclear Energy* 26.3 (1991), pp. 147–162. ISSN: 0149-1970. DOI: `https://doi.org/10.1016/0149-1970(91)90033-L`. URL: `https://www.sciencedirect.com/science/article/pii/014919709190033L`.

[40] Paul P.H. Wilson et al. “Acceleration techniques for the direct use of CAD-based geometry in fusion neutronics analysis”. In: *Fusion Engineering and Design* 85.10 (2010). Proceedings of the Ninth International Symposium on Fusion Nuclear Technology, pp. 1759–1765. ISSN: 0920-3796. DOI: `https://doi.org/10.1016/j.fusengdes.2010.05.030`. URL: `https://www.sciencedirect.com/science/article/pii/S0920379610002425`.

[41] Yeon Soo Yeom et al. “Computation Speeds and Memory Requirements of Mesh-Type ICRP Reference Computational Phantoms in Geant4, MCNP6, and PHITS”. en. In: *Health Physics* 116.5 (May 2019), pp. 664–676. ISSN: 1538-5159, 0017-9078. DOI: `10.1097/HP.0000000000000999`. URL: `https://journals.lww.com/00004032-201905000-00012` (visited on 10/18/2023).