UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
The Development of WARP - A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs

Permalink
https://escholarship.org/uc/item/4s47p41g

Author
Bergmann, Ryan Mitchell

Publication Date
2014

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

The Development of WARP - A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs

by

Ryan Bergmann

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

Committee in charge:

Professor Jasmina Vujić, Chair
Professor Ehud Greenspan
Professor Per Peterson
Professor Per-Olof Persson
Assistant Professor Rachel Slaybaugh

Spring 2014

The Development of WARP - A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs

Copyright 2014
by
Ryan Bergmann

Abstract

The Development of WARP - A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs

by

Ryan Bergmann

Doctor of Philosophy in Engineering - Nuclear Engineering

University of California, Berkeley

Professor Jasmina Vujić, Chair

Graphics processing units, or GPUs, have gradually increased in computational power from the small, job-specific boards of the early 1990s to the programmable powerhouses of today. Compared to more common central processing units, or CPUs, GPUs have a higher aggregate memory bandwidth, much higher floating-point operations per second (FLOPS), and lower energy consumption per FLOP. Because one of the main obstacles in exascale computing is power consumption, many new supercomputing platforms are gaining much of their computational capacity by incorporating GPUs into their compute nodes. Since CPU-optimized parallel algorithms are not directly portable to GPU architectures (or at least not without losing substantial performance), transport codes need to be rewritten to execute efficiently on GPUs. Unless this is done, reactor simulations cannot take full advantage of these new supercomputers.

WARP, which can stand for "Weaving All the Random Particles," is a three-dimensional (3D) continuous energy Monte Carlo neutron transport code developed in this work as to efficiently implement a continuous energy Monte Carlo neutron transport algorithm on a GPU. WARP accelerates Monte Carlo simulations while preserving the benefits of using the Monte Carlo Method, namely, very few physical and geometrical simplifications. WARP is able to calculate multiplication factors, flux tallies, and fission source distributions for time-independent problems, and can run in both criticality or fixed source modes. WARP can transport neutrons in unrestricted arrangements of parallelepipeds, hexagonal prisms, cylinders, and spheres.

WARP uses an event-based algorithm, but with some important differences. Moving data is expensive, so WARP uses a remapping vector of pointer/index pairs to direct GPU threads to the data they need to access. The remapping vector is sorted by reaction type after every transport iteration using a high-efficiency parallel radix sort, which serves to keep the reaction types as contiguous as possible and removes completed histories from the transport cycle. The sort reduces the amount of divergence in GPU "thread blocks," keeps the SIMD units as full as possible, and eliminates using memory bandwidth to check if a

neutron in the batch has been terminated or not. Using a remapping vector means the data access pattern is irregular, but this is mitigated by using large batch sizes where the GPU can effectively eliminate the high cost of irregular global memory access.

WARP modifies the standard unionized energy grid implementation to reduce memory traffic. Instead of storing a matrix of pointers indexed by reaction type and energy, WARP stores three matrices. The first contains cross section values, the second contains pointers to angular distributions, and a third contains pointers to energy distributions. This linked list type of layout increases memory usage, but lowers the number of data loads that are needed to determine a reaction by eliminating a pointer load to find a cross section value.

Optimized, high-performance GPU code libraries are also used by WARP wherever possible. The CUDA performance primitives (CUDPP) library is used to perform the parallel reductions, sorts and sums, the CURAND library is used to seed the linear congruential random number generators, and the OptiX ray tracing framework is used for geometry representation. OptiX is a highly-optimized library developed by NVIDIA that automatically builds hierarchical acceleration structures around user-input geometry so only surfaces along a ray line need to be queried in ray tracing. WARP also performs material and cell number queries with OptiX by using a point-in-polygon like algorithm.

In the initial testing where 10^6 source neutrons per criticality batch are used, WARP is capable of delivering results that are anywhere from 4 to 800 pcm away from MCNP 6.1 and Serpent 2.1.18, but with run times that are 11-82 times lower, depending on problem geometry and materials. On average, WARP's performance on a NIVIDIA K20 is equivalent to approximately 45 AMD Opteron 6172 CPU cores. Larger batches are typically perform better on the GPU, but memory limitations of the K20 card restricted batch size to 10^6 source neutrons.

WARP has shown that GPUs are an effective platform for performing Monte Carlo neutron transport with continuous energy cross sections. Currently, WARP is the most detailed and feature-rich program in existence for performing continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs, but compared to production codes like Serpent and MCNP, WARP has limited capabilities. Despite WARP's lack of features, its novel algorithm implementations show that high performance can be achieved on a GPU despite the inherently divergent program flow and sparse data access patterns. WARP is not ready for everyday nuclear reactor calculations, but is a good platform for further development of GPU-accelerated Monte Carlo neutron transport. In it's current state, it may be a useful tool for multiplication factor searches, i.e. determining reactivity coefficients by perturbing material densities or temperatures, since these types of calculations typically do not require many flux tallies.

To my brother Mike, the original Dr. B.

Contents

Contents ii

List of Figures v

List of Tables viii

1 Introduction 1

1.1 Why Monte Carlo? . . . 3
1.2 Why GPUs? . . . 3
1.3 Goals and Impacts . . . 6
1.4 Reference Monte Carlo Neutron Transport Codes . . . 8
MCNP . . . 8
Serpent . . . 9
1.5 Outline . . . 10

2 Background 11

2.1 Nuclear Reactor Analysis . . . 11
2.2 Nuclear Interactions . . . 13
Elastic Scattering . . . 16
Inelastic Level Scattering . . . 19
Inelastic Continuum Scattering . . . 20
Fission . . . 20
Capture Reactions . . . 24
Other Secondary-Producing Absorption Reactions . . . 24
Bound Nuclei and Unresolved Resonances . . . 24
2.3 Temperature Effects . . . 25
Doppler Effect . . . 25
2.4 Nuclear Data . . . 28
2.5 Neutron Transport . . . 29
Neutron Balance Equation . . . 29
Neutron Distribution Function . . . 30
Reaction Rates . . . 31

Angular and Scalar Flux 32
Scattering 33
Fission Source 34
Streaming 36
Neutron Transport Equation 36
Time Independent Neutron Transport Equation 37
2.6 Monte Carlo 38
Statistics 40
Sampling Schemes 42
Cross Section Interpolation 44
Distance To Interaction 45
Isotope and Reaction Selection 45
Tabular Distribution Interpolation 46
Free Gas Treatment 47
Stochastic Mixing 50
Elastic Scattering 50
Tabular Energy Distributions 52
Inelastic Reactions 53
Flux Estimation and Tallies 54
Neutron Sources 55
External Source 56
Fission Source, k-Eigenvalue Method 56
2.7 GPUs 59
Architecture 61
CUDA 63
Memory 64
2.8 OptiX 68
OptiX Programs 68
Geometry and Intersection Programs 69
Acceleration Structures 72
2.9 Previous Works 73

3 GPU Implementations **78**
3.1 Preliminary Studies 78
2D Scattering Game 78
Ray Tracing with OptiX 87
Point-in-Polygon / Material Query 88
Instancing 90
Test Geometries 91
Results 93
3.2 WARP in Detail 95
Data Layout 96

Unionized Cross Sections . . . 96
Distribution Data as a Linked List . . . 99
Embedded Python . . . 101
Python Wrapper . . . 102
CUDA Kernels and Data-Parallel Tasks . . . 103
Grid Search Kernel . . . 105
Pseudo-Random Numbers . . . 106
Macroscopic Cross Section Kernel . . . 107
Flux Tally Kernel . . . 108
Microscopic Cross Section Kernel . . . 109
Interaction Kernels . . . 109
Concurrent Kernels . . . 109
Parallel Operations . . . 110
Remapping with Radix Sort . . . 110
Criticality Source . . . 112
Fixed Source & Subcritical Multiplication . . . 113

4 Results **114**
4.1 Criticality Tests - Multiplication Factors and Runtimes . . . 117
4.2 Criticality Tests - Flux Spectra and Fission Distributions . . . 120
4.3 Fixed-Source Tests . . . 127
4.4 Comparison to Non-Remapping . . . 131

5 Conclusions and Future Work **136**
5.1 General Conclusions . . . 136
5.2 Specific Conclusions . . . 137
5.3 Future Work . . . 138

A Supplementary Results **142**
A.1 2D Scattering on an NVIDIA Tesla C2075 . . . 142
A.2 Serpent Fission Source Distributions . . . 144

B Additional Tables **146**

Bibliography **148**

List of Figures

2.1 Schematic of the fission chain reaction and its relation to the multiplication factor. 13
2.2 The energy dependence of some reactions in ^{6}Li. 16
2.3 The energy dependence of some reactions in ^{235}U. 17
2.4 The energy levels of ^{177}Hf. 20
2.5 Average binding energy per nucleon. 21
2.6 The average total number of neutrons per fission, ν_{T}, for ^{235}U, ^{238}U, and ^{239}Pu. 23
2.7 Fission cross sections for ^{235}U, ^{238}U, and ^{239}Pu. 23
2.8 Maxwell-Boltzmann distribution for the speed of heavy and light nuclides at different temperatures. 26
2.9 Doppler broadening of the 1 eV fission resonance in ^{155}Eu. 27
2.10 Hierarchy in ACE formatted neutron data libraries. 28
2.11 The differential volume of the neutron balance equation. 30
2.12 The Cartesian projections of the directional vector. 31
2.13 Prompt fission neutron spectra for ^{235}U and ^{239}Pu. 35
2.14 The Monte Carlo random walk process. 39
2.15 Sampling $PDF(x) = \frac{1}{2\pi^2}(x \cos x - x)$ by the direct inversion method on $[0, 2\pi]$. 43
2.16 Sampling $PDF(x) = \frac{1}{2\pi^2}(x \cos x - x)$ by the rejection method on $[0, 2\pi]$. The auxiliary function is $f(x) = x/\pi$ and the random numbers used to sample from it are uniformly distributed on $[0, 2\pi]$ instead of $[0, 1]$. 44
2.17 Elastic scattering anisotropy at high energies in ^{16}O. 51
2.18 An inaccurate flux spectrum produced by WARP compared to a correct one from Serpent highlighting the errors caused by treating elastic scattering as isotropic for all energies. 51
2.19 The maximum theoretical GigaFLOPs of various NVIDIA GPUs vs. flagship Intel processors. 61
2.20 The architecture of a Fermi multiprocessor. 62
2.21 Relative transistor space use in GPUs and CPUs. 62
2.22 Host-device execution in CUDA and the organization of threads into blocks and blocks into the grid. 64
2.23 The total global memory bandwidth of NVIDIA GPUs vs. flagship Intel processors. 65
2.24 Memory transactions of CUDA threads. 65
2.25 GPUs Pipeline many threads to hide high memory latency. 66

2.26 The memory spaces in CUDA and NVIDIA GPUs. 67
2.27 The program flow in a launched OptiX context. 68
2.28 Ray-surface intersection: a sphere and a parameterized ray. 69
2.29 Node graph used for geometry representation in OptiX. 70
2.30 Spatial partitioning schemes with the resulting trees. 72

3.1 The 2D geometry of the scattering test. 79
3.2 The task-based algorithm used in the 2D scattering study. 80
3.3 Ray tracing done in a data-parallel way. 81
3.4 The event-based algorithm used in the 2D scattering study. 81
3.5 Mapping thread IDs to active data through a remapping vector 82
3.6 Speedup factors of the GPU implementations over the CPU implementation on a Tesla K20. 84
3.7 Speedup factors of the GPU implementations over the CPU implementation on a Tesla K20 with Σ_a increased to 0.1 in cell 0. 85
3.8 The number of active threads for the event-based GPU implementations. . . . 86
3.9 The percent of control flow divergence in blocks of the task- and event-based GPU implementations. 86
3.10 The point-in-polygon-like algorithm for determining the entering cell/material number by using ray tracing 88
3.11 The OptiX node graph using transform instancing 90
3.12 The OptiX node graph using mesh primitive instancing. 91
3.13 x-y cross sections of the geometry created by using the PIP cell/material query algorithm 92
3.14 The geometries used in the scaling study from least number of objects on the left to most on the right. 92
3.15 Trace rates of an NVIDIA Tesla K20 performing cell queries with the PIP algorithm. 93
3.16 Trace rate scaling on an NVIDIA Tesla K20 performing cell queries with the PIP algorithm. 94
3.17 Trace rates of an NVIDIA GeForce GT 650M performing cell queries with the PIP algorithm. 95
3.18 Unionizing two cross section vectors. 98
3.19 The unionized cross section layout used in WARP. 99
3.20 Making a link to distribution data in the unionized dataset. 100
3.21 WARP inner transport loop that is executed until all neutrons in a batch are completed. 104
3.22 WARP outer transport loop that is executed in between neutron batches for criticality source runs. 105
3.23 A radix key-value sort creating a remapping vector. 111

4.1 Spectrum comparison in a "Jezebel" bare ^{239}Pu sphere. 120

4.2 Fission source distribution of a "Jezebel" bare ^{239}Pu sphere calculated by WARP. 121
4.3 Relative difference of the WARP fission source distribution compared to Serpent's for the Jezebel bare ^{239}Pu sphere. 121
4.4 Spectrum comparison in a homogenized block of UO_2 and water. 122
4.5 Fission source distribution of a homogenized block of UO_2 and water calculated by WARP. 123
4.6 Relative difference of the WARP fission source distribution compared to Serpent's for the homogenized block of UO_2 and water. 123
4.7 Spectrum comparison in a single UO_2 pin surrounded by a block of water. . . . 124
4.8 Fission source distribution of a single UO_2 pin surrounded by a block of water calculated by WARP. 125
4.9 Relative difference of the WARP fission source distribution compared to Serpent's for the single UO_2 pin surrounded by water. 126
4.10 Spectrum comparison in the center UO_2 pin of a hexagonal pin array in water. 127
4.11 Fission source distribution of a hexagonal array of UO_2 pins in water calculated by WARP. 128
4.12 Relative difference of WARP fission source distribution compared to Serpent's for the hexagonal array of UO_2 pins in water. 128
4.13 Volume-average flux spectra of a fixed-source simulation with a 1 eV point source in a homogenous block of fuel, water, and boron. 129
4.14 Volume-average flux spectra of a fixed-source simulation with a 2 MeV point source in a block of water. 130
4.15 Neutron processing rate of WARP in the homogenized block and assembly test cases. 132
4.16 Active neutrons per iteration in WARP in the homogenized block and assembly test cases. 133

A.1 Speedup factors of the GPU implementations over the CPU implementation on a Tesla C2075. 143
A.2 Fission source distribution from Serpent 2.1.18 of a "Jezebel" bare Pu-239 sphere. 144
A.3 Fission source distribution from Serpent 2.1.18 of a homogenized block of UO_2 and water. 144
A.4 Fission source distribution from Serpent 2.1.18 of a single UO_2 pin surrounded by a block of water 145
A.5 Fission source distribution from Serpent 2.1.18 of a hexagonal array of UO_2 pins in water. 145

List of Tables

1.1 A comparison of an NVIDIA GPU and an Intel CPU 5
1.2 Breakdown of the cost benefits of GPUs assuming maximum performance and linear CPU scaling 6

2.1 Average distribution of of ^{235}U fission energy. 22

4.1 Geometry and materials used in the six test cases. 117
4.2 Summary of k_{eff} single-run results of the WARP criticality tests with 20/40 discarded/active criticality cycles and 10^5 histories per cycle. 118
4.3 Summary of k_{eff} single-run results of the WARP criticality tests with 20/40 discarded/active criticality cycles and 10^6 histories per cycle. 118
4.4 Summary of runtimes of the fixed-source tests with 4×10^7 total histories. . . . 129
4.5 Comparison of the non-remapping and remapping versions of WARP for the four criticality test cases. 20/40 discarded/active criticality cycles and 10^6 histories per cycle. 131
4.6 Summary table of the fraction of total kernel time spent in each WARP subroutine in criticality mode for each test case. 134

B.1 Reaction number encodings in WARP. 146
B.2 Geometric primitive number encodings in WARP. 146
B.3 ENDF MT numbers and the reactions they stand for. 147

Acknowledgments

I would like to thank all my committee members for the time and effort they put into my progression through candidacy. I would especially like to thank my advisor, Jasmina Vujić, for the opportunities she has given me as a student, and whose generosity and inclusiveness has made my experience at UC Berkeley the most personally and intellectually enriching period in my life.

I would also like to thank Jaakko Leppänen for his willingness to share his wealth of knowledge and helping me when I was stumped, Katy Huff for her lightning fast emails while helping me with programming issues, PyNE and its development team for making my life easier, and OpenMC and its development team for trailblazing open source Monte Carlo neutron transport and their clearly written source code and documentation.

I couldn't have survived without Maya Mathura and I am grateful for all the love, patience, and support she continually shows me. To my family, you are the reason I succeed, thank you for your unwavering support of my endeavors.

This work was supported by the Department of Energy National Nuclear Security Administration under Award Number(s) DE-NA0000979: The National Science and Security Consortium (NSSC), http://nssc.berkeley.edu/

Tesla K20 provided by NVIDIA through their academic outreach program.

Chapter 1

Introduction

Nuclear reactors have the highest energy density of any energy-producing technology currently available [1]. This is because of their ultimate source of energy - the atomic nucleus. Compared to the atom as a whole, the nucleus is a very small, dense region that contains most of the atom's mass. The nucleus is made of protons and neutrons and is held together by strongest elementary force known in nature - the strong nuclear force. The strong force binds atomic nuclei together and keeps matter stable.

There are certain reactions that can happen to a nucleus that will cause it to become unstable, however. When a nucleus of ^{235}U absorbs a free neutron, there is a high probability that the nucleus will rapidly become unstable and split. This splitting is called "fission," and it produces a large amount of energy. When the masses of the incident neutron, the fragments, and neutrons that result from fission are measured, they sum to less mass than that of the parent ^{235}U atom and the incident neutron. Mass has actually been converted into energy [2, 3]. This conversion also occurs in chemical reactions. If the masses of two atoms are measured before they chemically bond and after, the bonded compound will have a slightly smaller mass than the initial reactants. This "mass defect" is the amount of mass that has been converted to energy in creating the new state, whether nuclear or atomic [4]. The electron has about 2×10^3 times less mass than a proton or neutron, and the forces involved in atomic bonds are weaker than those in nuclear bonds, so the energetics of atomic reactions are about 10^6 times less than that of nuclear reactions [2]. Since a nucleus must be present in both atomic and nuclear reactions, the reactant mass is always dictated by the nucleus, but nuclear reactions release much more energy than atomic reactions. This is the reason nuclear fission produces so much energy from such a small mass compared to chemical energy sources like burning natural gas or coal. ^{235}U, for example, releases a total of 192.9$\pm$0.5 MeV per fission [3]. In chemical energy sources, the energy released per reaction is on the order of 1 eV. The energy yielded by fission is 8 orders of magnitude larger.

Most of the energy released from fission ends up as kinetic energy of the two smaller nuclei formed from fission, or fission fragments, and is quickly converted to heat in the immediate vicinity of the fission site. This heat can be used to perform many tasks, but arguably the most useful task is to drive a thermodynamic cycle to convert much of the heat

into electricity. Since the specific energy of nuclear fuel is around six orders of magnitude larger than chemical sources, the mass and volume of fuel needed to run a nuclear is orders of magnitude smaller, and the power cycle produces much less waste mass, even though its waste is very radioactive [5].

As with anything very powerful, nuclear technology must be handled with great responsibility. The reactor's behavior must be accurately predictable by designers and operators to prevent accidents that could release radioactivity into the open environment and make sure the power plant is a reliable source of clean, affordable electricity. To ensure this required reliability and safety, accurate simulations are needed to predict what will happen to the reactor if conditions within it change. Since reactors are expensive machines [6], accurate simulations are needed in the design phase as well; accurate enough to provide confidence to designers, regulators, and the public at large that a reactor will be safe before constructing a demonstration plant.

A common and accurate way to conduct reactor simulations is to solve the neutron transport equation using Monte Carlo methods. The Monte Carlo method requires few approximations to be made in the simulation model, and therefore can produce physically-accurate results. However, using the Monte Carlo method is much more computationally expensive than other methods. Typically, Monte Carlo simulations need to be run on supercomputers to produce results in a reasonable amount of time for problems relevant to engineering.

General purpose graphical processing units (GPGPUs, referred to as GPUs henceforth) are an emerging computational tool, sporting higher memory bandwidth and computational throughput as well as lower power consumption per operation compared to central processing units (CPUs), the standard type of computer processor. GPUs are touted as being "massively parallel," home to thousands of computational "cores," and capable of turning a desktop into a "personal supercomputer." Some applications can see upwards of a hundredfold speedup by running on GPUs [7]. These speedup factors make GPUs very attractive to use in extremely parallel, computationally-intensive simulations like Monte Carlo neutron transport, where trillions or more independent neutron histories are tracked.

Some argue that the speedup gains are an illusion, and multicore CPUs are more than capable of similar performance if enough optimization is done [8]. Some think that adding another programming paradigm is the wrong direction for computer science, and that it would be better if more resources were invested into existing technologies instead of spreading resources more thinly on new ones [9].

A key argument against both of these concerns is that learning a new language is not required to program GPUs. CUDA (Compute Unified Device Architecture, NVIDIA's parallel computing platform [10]) adds minimal extensions to the C programming language, and is therefore easy to program in if a developer already knows C. Another argument is that substantial performance gains can be seen without much detailed optimization. Getting speedups of 100x *will* require thorough optimization, but getting 10x speedup of a serial code is commonly seen with little to no algorithmic changes of CPU code. This is because GPUs were developed to be parallel from their inception, unlike CPUs, and GPU hardware is able to hide much of the details about how the parallel computations actually execute. CUDA

allows developers to write scalar code (as opposed to vector code), and CUDA maps this to vectors on the hardware [10]. In addition, an empirical bandwagon argument can be made for at least attempting to port codes to the GPU. Many developers are porting and seeing reasonable speedups [7], which is more than enough reason to at least attempt using CUDA for Monte Carlo neutron transport.

1.1 Why Monte Carlo?

When applied to neutron transport, the central concept of the Monte Carlo method is to directly simulate what microscopically happens to neutrons in nature by tracking every interaction they undergo, from birth to death. Once a sufficiently-large number of these “histories” are completed, sums and/or averages are taken over certain attributes to determine aggregate, macroscopic behavior [11]. Directly simulating what every individual neutron is doing is a rather brute-force way of implementing a simulation since the macroscopic behavior is what matters in the end. The benefit of the brute-force strategy is that very few assumptions have to be made, giving Monte Carlo the potential to be the most accurate way to simulate nuclear reactors.

The main drawback in using the Monte Carlo method, however, is that its convergence is governed by the central limit theorem. For many problems, obtaining sufficiently-low statistical error is slow compared to other approaches [12, 13]. This is why any way of accelerating Monte Carlo methods is of interest to the nuclear engineering community and why GPUs are being studied in this work. In other words, it is a simulation method that can greatly benefit from acceleration.

1.2 Why GPUs?

GPUs have gradually increased in computational power from the small, job-specific boards of the early 1990s to the programmable powerhouses of today. Compared to CPUs, they have a higher aggregate memory bandwidth, much higher floating-point operations per second (FLOPS), and lower energy consumption per FLOP [10]. Because one of the main obstacles in exascale computing is power consumption [14], many new supercomputing platforms are gaining much of their computational capacity by incorporating GPUs into their compute nodes. In the November 2013 Top 500 list, there are 41 GPU-accelerated supercomputers, some of which gain 50% of their computational capacity from GPU coprocessor cards [15]. Supercomputers in the number two and six spots use GPUs as well. Since CPU-optimized parallel algorithms are usually not directly portable to GPU architectures (or at least not without losing substantial performance), transport codes may need to be rewritten to execute efficiently on GPUs. Unless this is done, nuclear engineers cannot take full advantage of these new supercomputers for reactor simulations.

Table 1.1 shows a breakdown of features of both an Intel i7 (Westmere-EP) CPU and an NVIDIA Tesla C2075 (Fermi) GPU [16, 17]. An AMD CPU was used in this work, but the i7 has similar enough characteristics to be representative, and the data of interest was more readily available. The AMD processor has similar values apart from the frequency, therefore comparisons made here are still valid.

At first glance, it may appear that the GPU completely outstrips the CPU. The GPU has higher single precision FLOPs, which indicates that the GPU can do work faster than the CPU. The GPU also has a higher memory bandwidth, which implies that data can be accessed by the processors at a higher rate. Data is necessary to do calculations, and the higher arithmetic rate of the GPU would be wasted if the processors couldn't be fed with data at a high rate. GPUs also have a higher concurrent thread capability. "Thread" is the term used for the sequence of instructions given to a processor to be complete a specified task.

Typical Monte Carlo neutron transport algorithms are "task-based." A thread will transport the neutron until it is terminated through absorption or by leaking out of the system. The sequence of events the neutron undergoes from start to finish is called a neutron "history," and is the basic unit of work for a thread in a task-based algorithm. Neutron histories are independent of each other and the algorithm is parallelized by running many histories in independent, parallel threads. Thus, having as many parallel threads as possible should provide the greatest performance. This parallelization method requires that the threads can execute completely independently of each other, i.e. the actions of one thread do not affect the other parallel threads in any way. This is a MIMD (multiple instruction multiple data) way of executing threads. In MIMD execution model, different threads can execute different instructions on their data at a point in time. Threads appear to completely independent, and having threads at different points in the transport algorithm is not a problem since they are allowed to execute their own instructions.

It seems that GPU cards would be perfect for running Monte Carlo neutron transport because they can run large numbers of concurrent threads. The concurrent thread number is based on the width of the processor's SIMD (single instruction multiple data) units, however. SIMD is an execution model some processors use in order to lower the number of instructions needed per amount of computation done, which increases both power and computational efficiency [18]. SIMD requires the same instructions to be carried out over every element in a concurrently-processed data vector. From a thread standpoint, SIMD requires threads to execute the same instruction at a point in time. If a set of threads does not execute the same instruction at the same time (the data they act on can still be different), the GPU will serialize them. The subset of threads executing the first instruction will all execute together, then the subset executing the second instruction will execute after them. Monte Carlo typically breaks instruction regularity because of its conditional statements based on random numbers. Therefore, if Monte Carlo algorithms are to be used on GPUs, they must be implemented in a manner that carefully takes into account the limitations of the GPU.

The memory subsystems of GPUs also function in a SIMD-like way. In order to use the full memory bandwidth of the device, more than one piece of data must be loaded and

Table 1.1: A comparison of an NVIDIA GPU and an Intel CPU [16, 17, 19].

Processor	Intel i7 (Westmere-EP)	NVIDIA Tesla C2075 (Fermi)
Processing Elements	6 cores, 2 issue, 4-way SIMD	14 cores, 2 issue, 16-way SIMD
Frequency	3.46GHz	1.15 Ghz
Resident Strands / Threads (max)	48	21,504
SP GFLOP/s	166	1030
Mem. Bandwidth	32 GB/s	144 GB/s
Global Latency	∼50 clocks	200-800 clocks
FLOPs / byte	5.2	7.2
Register File	6kB	2MB
Local Storage / L1 Cache	192 kB	896 kB
L2 Cache	1536 kB	0.75 MB
L3 Cache	12 MB	-

used per transaction, and the only way multiple pieces of data can be loaded simultaneously is if they are adjacent in memory. In other words, if a program requests a single piece of data at location i, then requests a piece at location $i + 10$, these requests will be split into two separate transactions that yield only one data element each. Two data elements in two transactions produces an effective bandwidth of one element per transaction time. On the other hand, if the program requests data at i, $i + 1$, $i + 2$, and $i + 3$, the entirety of the requested data can be retrieved in a single transaction. Four data elements in a single transaction yields and effective memory bandwidth of four elements per transaction time, four times higher than the previous scenario. Having a memory subsystem that handles requests in this way is not ideal for Monte Carlo methods, however. Data is accessed in a very random way because of the random nature of the simulation, and requested data is unlikely to be adjacent. This means the full memory bandwidth of the GPU will not be used unless this problem is mitigated in some way.

Another undesirable feature of the GPU is that they have very high global memory latency compared to a CPU. Memory latency is the number of clock cycles, or amount of time, it takes for a data request to be fulfilled. As Table 1.1 shows, the GPU's global memory latency is about an order of magnitude higher than the CPU's [17, 10]. GPUs try to eliminate the effect of large global latency by pipelining memory access. Pipelining means threads that have received their data can execute as other threads are waiting for their data to load. If many requests are known, the data can be continually loaded as threads start to execute their jobs. The hope is that the jobs take longer than the memory loads, eventually all data arrives, and the later threads appear to have zero latency for their memory access. This is why it is important for GPUs to have such a large number of concurrent threads. It allows them to pipeline data access and minimize the impact of memory latency.

Another notable feature is that the GPU has a greater FLOPs/byte of memory bandwidth ratio than the CPU. This implies that GPUs could be used to turn a compute-bound problem into a bandwidth-bound problem. This may seem like a deficit, but the GPU has a higher maximum memory bandwidth, so even though a problem is bandwidth-bound on a GPU, it may still require less execution time than on a CPU.

Table 1.2: Breakdown of the cost benefits of GPUs assuming maximum performance and linear CPU scaling [20, 21, 22, 23, 19].

Processor	4x AMD Opteron 6172 @ 2.1 GHz (processors only)	3x NVIDIA Tesla C2075 (cards only)
Approximate Price (Q2 2012)	$8,000	$ 7,000
Max.TeraFLOP	0.8	3.1
Price / GigaFLOP	$10	$2.26
Price / History Power (assuming 10^3 histories/s per core and 25x GPU speedup)	$0.16	$0.11
Thermal Power	460 W	675 W
Yearly electricity cost per TeraFLOP ($.05 / kWh)	$252	$96

Table 1.2 shows a cost comparison of an AMD Opteron 6172 CPU and an NVIDIA Tesla C2075 GPU [17, 10]. The prices shown are rounded values from purchases made by the UC Berkeley Department Nuclear Engineering [20, 21]. The CPU price is for four Opteron 6172 processors, and the GPU price is for three NVIDIA Tesla C2075 cards. These numbers are shown because the department's computer cluster contains CPU and GPU nodes with these configurations. The main benefit from using GPUs is the substantially lower electric cost per FLOP, though the capital cost per amount of work can also be lower that an equivalent CPU configuration. To put this in a nuclear engineering context, it is useful to compare cost per rate of work done. This rate will be called "history power": histories run per second. Assuming a GPU speedup factor of 25 (approximately the average value of the accelerated applications reported by NVIDIA [7]) and that CPU codes scale perfectly linearly, the price per history power of the GPU configuration shown in Table 1.2 is about 31% lower than that of CPU configuration. This indicates that GPUs may have lower capital costs as well as lower power costs per history power if they are able to provide a speedup greater than 16 over serial CPU codes (this is where their cost per history powers are equal under the stated assumptions).

1.3 Goals and Impacts

Many supercomputers are incorporating GPUs into their nodes to gain efficiency, which is the main obstacle in exascale computing. Monte Carlo simulations are very computationally

intensive, but their use is often required to adequately characterize nuclear reactors. To perform Monte Carlo computations in a reasonable amount of time, supercomputers are needed. Monte Carlo codes written for use on CPUs cannot be directly run on GPUs, and no high-fidelity Monte Carlo neutron transport applications exist for the GPU. The goal of this work is to develop a program, WARP, that accelerates accurate continuous energy neutron transport simulations in general, 3D geometries by using a GPU. If the name "WARP" were an acronym, it would stand for "weaving all the random particles," with the word "weaving" referring to the lock-step way in which "all the random particles", i.e. the neutrons, are sorted into coherent bundles and transported. Using the word "warp" is also a nod to NVIDIA's terminology for a group of 32 concurrent threads.

Even though GPU computing is still in its very early stages, developing WARP hedges risk against the nuclear engineering community's computational tools becoming under powered or even obsolete. GPUs are also common in personal computers and workstations. Developing WARP will expose a substantial amount of computing power that couldn't otherwise be used and which could drastically reduce simulation times for people without access to traditional supercomputers. If WARP is able to produce accurate results around 20 times faster than a CPU code, the GPU could indeed be called a "personal supercomputer." Many supercomputers have nodes with around 20 CPU cores in them, and a 20x speedup with the WARP would make having a GPU equivalent to having a supercomputer node in a personal computer.

Running Monte Carlo simulations on GPUs is not a new idea. Ever since they became programmable, people have been trying to take advantage of the GPU's highly parallel, high computational throughput features. There are several codes that perform photon transport very well on GPUs [24, 25], and there have also been studies that ran criticality simulations on a GPU using event-based algorithms [26, 27, 28]. In the most recent study, only three reactions were considered, each reaction used an artificial, single-group cross section, and only hardcoded, simple geometries were possible [26, 27]. An even earlier study attempted to use continuous-energy cross sections to perform criticality calculations, but did not use standard data, did not perform inelastic scattering according to the full ENDF laws, did not incorporate a general or scalable geometry representation, and did not implement an effective task-parallel algorithm [28]. Due to all of their various shortcomings, none of these code were able to produce results comparable to production Monte Carlo neutron transport codes.

WARP sets itself apart from any previous endeavors in its breath of scope and its novel adaption of the event-based transport algorithm. Previous codes have also either used synthetic, simplified, and/or incomplete nuclear data. WARP will loaded standard data files and accurately simulate each reaction type specified in the data. WARP will also use a flexible, scalable, and optimized geometry representation where previous studies have used simplified and restricted geometry models. Previous works have examined event-parallel algorithms, but have not parallelized them effectively and therefore did not see the benefits of adopting such an algorithm on a GPU. WARP will use highly-parallelized algorithms and slightly modify the original vision of the event-based algorithm to better suit execution on the GPU.

The previous event-based algorithms tried to implement a "shuffle" operation where neutron data was actually sorted into reaction-contiguous blocks [28], similarly to vectorized Monte Carlo Methods developed decades ago [29, 30], or used small, synthetic nuclear data and were not able to capture the effects of loading large nuclear datasets [27]. WARP also changes the unionized energy grid data format to reduce the number of data loads needed to scan cross sections. In addition, an important part of WARP's development is also to determine if existing task-based Monte Carlo algorithms can be preserved or if they need to be modified in order to take full advantage of GPUs.

These features aim to make WARP a competitive Monte Carlo neutron transport code that efficiently executes on a GPU. To validate WAPR's accuracy, it will be compared against MCNP and Serpent, two production Monte Carlo neutron transport codes. WARP uses standard nuclear data files, can run in fixed source and criticality modes, can calculate integral parameters such as the multiplication factor, reaction rates, and fission rate distribution and other distribution functions such as the neutron spectrum. Since WARP should be performing the same calculations using the same data, the runtimes of WARP compared to the reference codes can also be made. A comparison of GPU-generated results to production Monte Carlo neutron transport codes has never been done before, and WARP will be the first to do so.

1.4 Reference Monte Carlo Neutron Transport Codes

The main purpose in developing WARP is to accelerate Monte Carlo nuclear reactor simulations by using the computational power of GPUs. To determine if WARP is successful in doing this, WARP will compared against Serpent [12] and MCNP [11], two Monte Carlo neutron transport codes that are used in the nuclear engineering community. Each code has different features and strengths and weaknesses in different areas. Comparing against both codes will certify that WARP is doing the correct calculations and determine if WARP's GPU implementations are effective in accelerating them.

MCNP

Monte Carlo simulations were one of the first applications of early computers. This is reflected in the correspondence of John von Neumann and Robert Richtmyer in 1947. In his letter, von Neumann outlined how to use statistical calculations to solve the neutron diffusion equation. Shortly after, Enrico Fermi invented FERMIAC11 at Los Alamos National Laboratory to track neutrons as they travelled through fissile materials by the Monte Carlo Method [11]. With the introduction of Fortran in 1957, it became possible to write more general neutron transport code systems than writing directly in machine code, and MCNP has been developed (in various incarnations) at Los Alamos National Laboratory since 1963 [11].

Despite the age of the project, MCNP tries to include all relevant advancements in the Monte Carlo method, includes detailed and accurate physical models and data, and is the "gold standard" of Monte Carlo neutron transport codes. MCNP has become a "repository for physics knowledge" and "represents over 500 person-years of sustained effort" [11]. WARP is similar to MCNP in that MCNP reads ACE-formatted nuclear data and uses ray tracing to handle boundary crossings. Serpent is capable of reproducing the results from MCNP very well [12], but MCNP will be included in the comparisons to further validate the results.

Serpent

"Serpent is a three-dimensional continuous-energy Monte Carlo reactor physics burnup calculation code, developed at VTT Technical Research Centre of Finland since 2004. The publicly available Serpent 1 has been distributed by the OECD/NEA Data Bank and RSICC since 2009, and next version of the code, Serpent 2, is currently in a beta-testing phase and available to registered users by request" [31]. It is written in ANSI C and makes use of MPI and OpenMP for parallelization [12].

According to the Serpent website, Serpent is suggested for use in "spatial homogenization and group constant generation for deterministic reactor simulator calculations; fuel cycle studies involving detailed assembly-level burnup calculations; validation of deterministic lattice transport codes; full-core modeling of research reactors, SMR's, and other closely coupled systems; coupled multi-physics applications (Serpent 2); and educational purposes and demonstration of reactor physics phenomena" [31]. Serpent 1 has been extensively validated against standard nuclear reactor criticality benchmarks and typically compares very well with results given by MCNP and Keno-VI [31].

Like many Monte Carlo neutron transport codes (MCNP and Keno-VI included), Serpent uses a universe-based combinatorial solid geometry (CSG) model that determines which volume a neutron is in by using binary logic on second-order surfaces [11, 31]. Serpent uses a combination of ray tracing and the Woodcock delta-tracking method to track neutron movements, which is different from what many Monte Carlo codes do [31]. The Woodcock method "has proven efficient and well suited for geometries where the neutron mean-free-path is long compared to the dimensions, which is typically the case in fuel assemblies, and especially in HTGR particle fuels" [31], and limits excess geometry sampling that can occur when strong absorbers are present.

Serpent also brought the use of a unionized energy grid format for the cross section and reaction data into the mainstream. The native way nuclear data is formatted gives an isotope's cross sections a unique energy grid. It is done this way so the minimum number of data points are included in the cross section to achieve an accuracy level. Since each isotope's grid is different, a search must be done on it to determine what data points a neutron's energy lies between. When there are many isotopes present, doing many energy searches to look up cross sections can become expensive. Serpent's method is to unionize all the energy grid for all the isotopes so they all have the same energy structure and only a single grid search is needed [12]. This reformatting of the nuclear data regularizes the access of the data and

increases performance, but comes at the cost of a larger memory footprint. Serpent was not the first code to use this type of data structure [29], but is the first widely-used code to do so. Serpent can also perform cross section preprocessing at arbitrary temperatures for more accurate calculations with minimal computational cost. Serpent also does on-the-fly Doppler broadening from 0 K data, so only one set of nuclear data needs to be stored in memory (though this strategy has an additional computational cost) [31].

1.5 Outline

Chapter 2 will cover background material that was needed in the development of WARP. It covers nuclear physics, nuclear reactor analysis, and the kinematics involved in neutron transport. Then, the mathematics behind the governing equations is discussed, the solution method explained, and how the required probability distributions are sampled is detailed. After the physics and mathematics sections, the computer hardware is discussed as is the OptiX ray tracing framework, which is the major library used in WARP. The chapter concludes with an overview of the previous research of Monte Carlo neutron transport on GPUs and how WARP fits into the landscape.

Chapter 3 discusses the actual routine implementations used on the GPU in WARP. It first goes over the exploratory studies done in preparation for developing WARP. These studies show the algorithmic benefits of a very important feature of WARP – remapping the data references. It also goes over how OptiX execution is optimized for best performance in reactor-like geometries. After the preliminary studies, the data layout for cross sections is explained and its similarities and differences from Serpent pointed out. The last topic discussed is the CUDA kernels written by hand for WARP. These routines process the neutrons as they travel through the problem geometry and provide the "glue" to connect all the important tasks that WARP requires to process the neutron data.

Chapter 4 discusses the initial results produced by WARP. Four criticality tests and two fixed-source tests are shown, each highlighting different requirements of the transport routines. For criticality source runs, the flux spectra, multiplication factors, and runtimes are compared to those from Serpent and MCNP. Fission source distributions from WARP are compared with Serpent only. The two fixed source calculations are shown to illustrate WARP's capabilities, and are also compared with Serpent and MCNP. In the conclusion of the chapter, the runtimes of remapping and non-remapping versions of WARP are compared to determine the necessity of remapping the neutron data references.

The final chapter draws conclusions from the previous chapters. In addition to drawing conclusions about the best-performing algorithms and configurations for conducting continuous energy Monte Carlo neutron transport on GPUs, the success of WARP in completing this work's initial goals is discussed. After these concluding remarks, a future roadmap for WARP is proposed, and the amount of work needed to be done to make WARP fully featured and reliable enough to gain community acceptance is enumerated.

Chapter 2

Background

This chapter explains the fundamental theory and mathematics that went into the development of WARP. It covers nuclear physics, nuclear reactor analysis, and the kinematics involved in neutron transport. Next, the mathematics behind the governing equations is discussed, the Monte Carlo solution method explained, and how the required probability distributions are sampled is detailed. After the physics and mathematics sections, the computer hardware is discussed as is the OptiX ray tracing framework, which is the major library used in WARP. The chapter concludes with an overview of the previous research of Monte Carlo neutron transport on GPUs and how WARP fits into the landscape.

2.1 Nuclear Reactor Analysis

Nuclear reactors generate electricity from the heat produced by fissions induced in heavy nuclei. Fissions are induced in nuclear reactor cores by a free neutron population present in the core, and this free neutron population is self-sustaining if the core is maintained within certain property ranges. The neutron density in the core is much smaller than the nuclei density and it can change much more quickly. Reactor power is directly proportional to the fission reaction rate, which is derived from the neutron density, and thus can change as quickly as the neutron density. Therefore, to know a core's most basic state, one must know the neutron density distribution. Knowing how reactor power will change given changes in the environment means knowing how the neutron distribution will respond to the changes. Accurately solving the neutron transport equation is essential for characterizing reactor behavior.

When a uranium (or other fissile heavy metal) nucleus fissions, two or more neutrons are released. If, on average, exactly one of these neutrons goes on to induce another fission, the system in which the fissions are occurring is *critical*, or able to maintain a constant neutron population and fission reaction rate (and correspondingly constant power) without any external neutron sources. This is a stable fission chain reaction. The fissile nucleus rarely splits evenly, and the spectrum of lighter isotopes that are created by fission are called

fission products [3]. Fission products accumulate in the fuel material and can inhibit the fission chain reaction through absorption when they reach a high enough concentration.

A *fissionable* isotope is one that has a non-zero fission cross section, i.e. there is a probability it will fission if it absorbs a neutron. A *fissile* isotope is one that can be induced to fission by absorbing a neutron at any energy. Some isotopes, like ^{235}U, can absorb a neutron at low energies and fission. This is because the energy imparted to the nucleus through the absorption reaction is enough to destabilize it to the point where it splits [3]. For some isotopes, like ^{238}U, absorbing a low-energy neutron does not impart enough energy to the nucleus to induce fission. It only fissions when a high-energy neutron is absorbed and the neutron brings additional energy with it to destabilize the nucleus sufficiently for fission to occur. Therefore, ^{238}U is fissionable, but not fissile. ^{238}U is also considered a *fertile* isotope since when it captures a neutron, it becomes ^{239}U which decays to ^{239}Np and then to ^{239}Pu, which is fissile. When a nucleus absorbs a neutron and produces no secondary particles (only gamma rays are produced), it is called "capture" since the neutron is added to the nucleus of the previous isotope. Fertile isotopes are those that decay to fissile isotopes after capturing a neutron [3].

The *effective multiplication factor*, k_{eff}, is the ratio of the number of neutrons in subsequent neutron generations. It is an integral quantity, and describes how many secondary neutrons are induced by an average neutron in a core. In other words, it describes the change in the neutron population between generations. If $k_{\text{eff}} = 1$, then subsequent neutron generations contain, on average, the same number of neutrons and the core is critical. If it is below 1, subsequent neutron generations are smaller and the core is subcritical. If k_{eff} is greater than 1, subsequent generations are larger and the core is supercritical. Figure 2.1 shows a schematic of the neutron generations, their size, and their relation to k_{keff}. The effective multiplication factor can also describe the time rate of change of the neutron population if the average neutron lifetime is known. The multiplication factor is an important quantity in characterizing nuclear reactor behavior because the core power is directly proportional to the fission rate, and the multiplication factor therefore dictates the time rate of change of the core power.

The most common type of commercial reactors are light water reactors [32]. These reactors use enriched uranium oxide, UO_2, fuel. They are cooled and moderated with light water, which is water containing predominately ^{1}H with trace amounts of ^{2}H. In the nuclear engineering lexicon, "moderator" means a material that reduces neutron energies to thermal levels, that is, to nearly thermal equilibrium with the moderator nuclei [3]. The most effective moderators are light nuclei, like hydrogen, since they have a mass similar to that of a neutron and an elastic scattering reaction can transfer a large fraction of the neutron's energy to the moderator nucleus. Light water reactors, or LWRs, are "thermal," since most of their fissions are induced by thermal neutrons. "Fast" reactors, on the other hand, are designed to minimize the slowing-down of fission neutrons and, therefore, lack a neutron moderator and typically use liquid metal for their coolant. [3].

Many factors influence the neutron population, and therefore must be incorporated into the solution method. The neutron population has strong feedback mechanisms from the

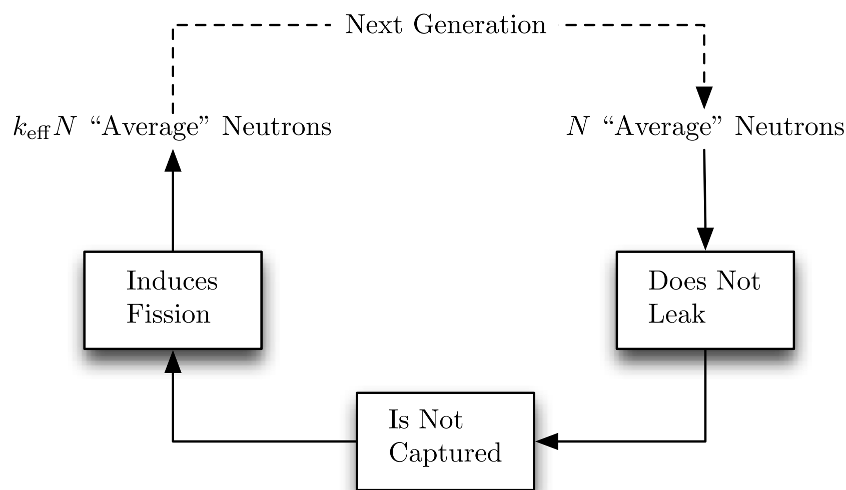


Figure 2.1: Schematic of the fission chain reaction and its relation to the multiplication factor.

materials present in the core. Each isotope in the core may undergo several different interactions with neutrons, and these reaction probabilities may have very strong dependence on the incoming neutron energy. Using the Monte Carlo method to approximate solutions to the neutron transport equation requires all of the neutron interactions to be explicitly modeled. The core geometry also affects the neutron distribution, since it describes the spatial extent of the core materials. The Monte Carlo method tracks neutrons as they move through the core materials, sampling the reactions they can undergo along the way. These reactions dictate fission chain reaction and maintain the core's effective multiplication factor. Tracking them allows the estimation of detailed reaction rates in the core, and allows engineers to better understand reactor behavior and design reactors to specifications. The next section will discuss all the possible reaction types a neutron can undergo when it collides with a nucleus and how interaction probabilities are classified and quantified.

2.2 Nuclear Interactions

The core of a Monte Carlo simulation is explicitly modeling the individual interactions the neutron undergoes as it moves through matter. At the highest level, these reactions can be broken down into two broad categories - scattering and absorption. Scattering reactions are when a neutron “bounces off” a nucleus. They change a neutron's direction and energy, but do not terminate its progression through matter. Absorption reactions are when the neutron becomes part of the nucleus. This state of a nucleus plus the incoming free neutron is called a “compound nucleus.” The neutron does not continue in a free state, and its

progress through the material is ended.

Scattering reactions can be further broken down into elastic and inelastic types. Elastic reactions are those where both momentum and kinetic energy are conserved, i.e. ones where any energy the neutron loses is given to the target nucleus. Inelastic reactions are those that conserve momentum but do not conserve kinetic energy, i.e. ones where the kinetic energy of the particles can be converted to an internal mode of the target nucleus. In some cases, the inelastic scattering reactions form a compound nucleus for a short time. A single neutron is emitted from the compound nucleus, resulting in the same particles that entered the reaction, but the nucleus is left at an excited state, making the reaction inelastic. Inelastic scattering reactions are further divided by the energy given to the nucleus [3].

There are many types of different absorption reactions, which are categorized by how the compound nucleus behaves, i.e. the number and type of secondary products it produces as it de-excites. Some reactions do not produce any secondary particles, and the neutron is "captured." Others produce secondary neutrons, like (n,2n) reactions where the neutron is absorbed, but the excited compound nucleus decays to a ground state by emitting two neutrons. Fission is also classified as an absorption reaction since a compound nucleus is formed, but in this case the compound nucleus splits instead of relaxing to another state [3].

The probabilities for individual reactions occurring are expressed in terms of cross sections. Cross sections are referred to in two ways - microscopically and macroscopically. Microscopic cross sections, represented by Greek lowercase sigma (σ), have units of area and describe the individual nucleus interaction probabilities in terms of the apparent "size" of the reaction. Macroscopic cross sections, represented by Greek capital sigma (Σ), take into account the density of nuclei, and describe the interaction probability per unit length along a neutron's direction of travel.

Microscopic cross sections are like geometric cross sections – they represent the "size" of the target nucleus for a particular reaction. The classical analogy is that if neutrons and nuclei are hard spheres, and neutrons are randomly shot through a material, more neutrons will hit the larger targets than the smaller ones. Cross sections are also expressed in units of area, the "barn," which is 10^{-24} cm^2. This unit was coined by Baker and Holloway while performing scattering experiments with uranium since "a cross section of 10^{-24} cm^2 for nuclear purposes was really as big as a barn"[33]. Of course, nuclear cross sections have no literal meaning in terms of the actual sizes of the nuclei, they only represent the likelihood of a particular reaction occurring.

Working at the macroscopic scale, which is where measurements are taken, a macroscopic cross section is the probability of a reaction happening per unit distance a neutron travels. With this parameter, an equation can be written that describes the survival probability of a group, or ensemble, of particles. Describing a group is necessary since neutrons are discrete particles, but the dimension x is *continuous*. Given a particle packet containing N particles and Σ, their interaction probability per unit distance the neutrons travel, the change in the uninteracted population over the differential distance dx is the product of the population N and the interaction probability, as show in Eq. (2.1) [3]. N is the number of neutrons that have not yet interacted and does not have dimensionality. When number densities are

discussed later, they will be represented by a lowercase n, which has units of inverse volume, typically $1/\text{cm}^3$. N could also be a neutron current, or the number of neutrons crossing unit surface area perpendicular to x per unit time, which implies that the N is at steady-state and has no time dependence. Treating N as a number of neutrons implies it is a transient neutron pulse traveling through the material and position and time are directly related by the neutrons' speed.

$$\frac{dN}{dx} = -\Sigma N \tag{2.1}$$

Integrating Eq. (2.1) over an interval yields an expression for the number of *non-interacting* particles left in a packet after crossing that interval. Dividing the surviving number by the initial gives a dimensionless expression for the non-interaction probability, P_{NI}, over the interval x_1 as shown in Eq. (2.2), where N_0 is the initial number of particles [3]. The expressions for non-interaction probability will be important later when it is necessary to sample the distance to the next interaction for individual neutrons in the Monte Carlo method.

$$P_{\text{NI}} = \frac{N}{N_0} = \exp\left(-\Sigma x_1\right) \tag{2.2}$$

$$\Sigma = \frac{-\ln\left(\text{N}/\text{N}_0\right)}{x_1} \tag{2.3}$$

Macroscopic cross sections are simply the microscopic cross section multiplied by the isotope's number density [3]. When a material is composed of multiple isotopes, the total macroscopic cross section for the material is the sum of the material's individual macroscopic cross sections. Compositions are commonly given in terms of atomic fraction, f_i, of isotope i, and total material mass density, ρ. It is then necessary to compute each isotope's number density from the average atomic mass of the isotopic combination, M_{avg}. The average atomic mass is computed from the individual isotopes' fractionally-weighted atomic masses, as shown in Eq. (2.5). Once this is computed, the average number density of the mixture can be calculated with Eq. (2.6). Using this value, the material's macroscopic cross section can be calculated with Eq. (2.7). Compositions are also often given in weight percents, and the process of calculating macroscopic cross sections from them is very similar.

$$\sum_{i=1}^{N_{\text{isotopes}}} f_i = 1 \tag{2.4}$$

$$M_{\text{avg}} = \sum_{i=1}^{N_{\text{isotopes}}} f_i M_i \tag{2.5}$$

$$n_i = f_i n_{\text{avg}} = f_i \frac{\rho}{M_{\text{avg}}} \tag{2.6}$$

$$\Sigma_{\text{material}} = \sum_{i=1}^{N_{\text{isotopes}}} n_i \sigma_i = n_{\text{avg}} \sum_{i=1}^{N_{\text{isotopes}}} f_i \sigma_{\text{i}} = \frac{\rho}{M_{\text{avg}}} \sum_{i=1}^{n_{\text{isotopes}}} f_i \sigma_i \tag{2.7}$$

The above expressions do not take any energy dependence into account. The quantum mechanical effects that are present in reality can cause cross sections to depend strongly on the energy (or velocity) of the neutron and the target nucleus. Many nuclides have resonances where the interaction probability spikes to very large values. This typically happens when the incoming neutron's energy in the CM frame is near an energy level of the resultant compound nucleus [3]. Figure 2.2 shows the energy dependence of various reaction types in ^{10}B and Figure 2.3 shows the dependence of some reactions in ^{235}U.

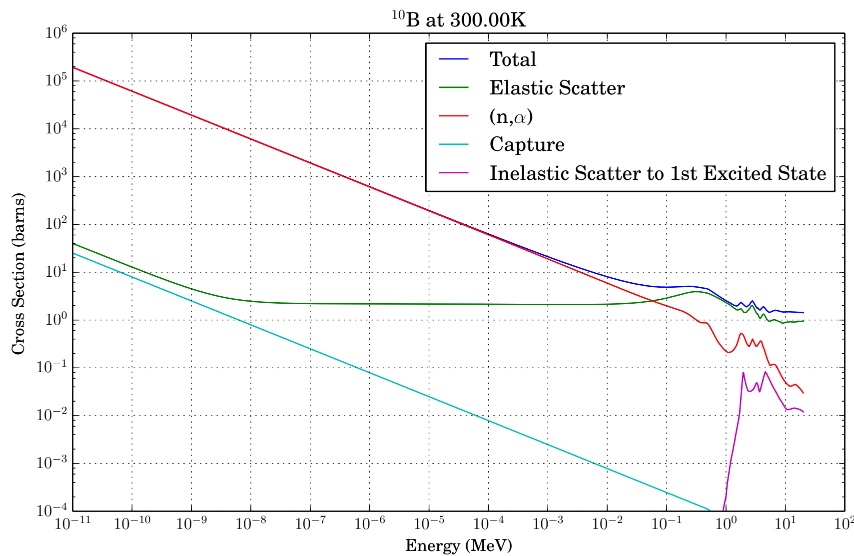


Figure 2.2: The energy dependence of some reactions in ^{6}Li.

It is important to note the complexity shown in in the cross sections for ^{235}U in the 0.1 eV to 40 keV range. This is referred to as the "resonance region" and adds much of the complexity in accurately modeling neutron transport. There are no simple functional representations available for such cross sections, so they must be represented in a point-wise tabular format.

Elastic Scattering

Elastic scattering conserves both the momentum and the kinetic energy of the reacting particles and occurs when the neutron does not enter the nucleus, but bounces off of its potential field. Since there is only a single exiting particle, elastic scattering is a two-body interaction and the kinematics are constrained by conservation of momentum and total

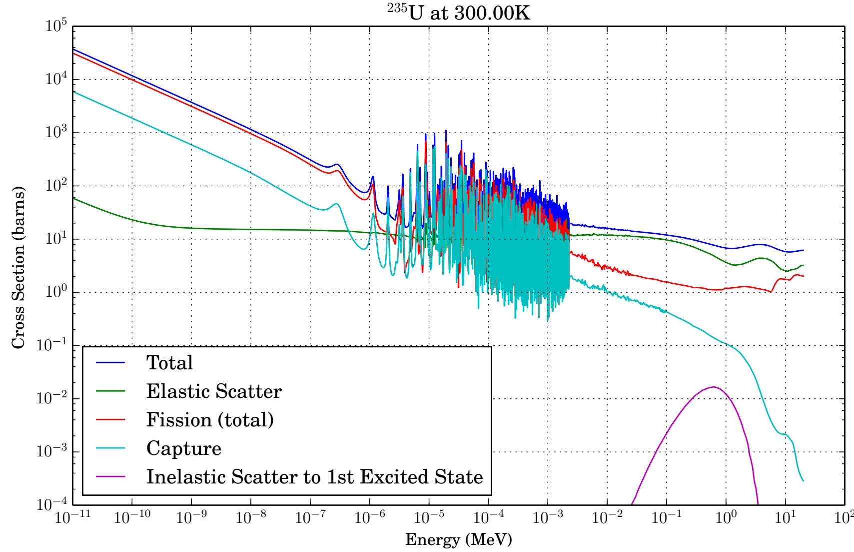


Figure 2.3: The energy dependence of some reactions in ^{235}U.

energy. The angle between the neutron's incoming and outgoing directions, or scattering angle, θ, is often represented as its cosine, $\mu = \cos\theta$. The scattering angle is the only unconstrained variable in the elastic scattering interaction, and the exiting neutron kinetic energy can be determined if the angle between the incoming and exiting particle is known.

This problem is greatly simplified if the velocities of the interacting neutron and nucleus are transformed into the center-of-mass (CM) frame, where net momentum before and after the collision is zero. The velocity of the CM is defined in Eq. (2.9) where m and M are the neutron's and the target's respective masses, v and V are their respective velocity vectors, and A is the ratio of the target's mass to the neutron's mass (also know as the "atomic mass ratio" or AMR). The derivation from here follows closely with that in [12].

$$A = \frac{M}{m} \tag{2.8}$$

$$\vec{\boldsymbol{v}}_{\mathbf{CM}} = \frac{m\vec{\boldsymbol{v}} + M\vec{\boldsymbol{V}}}{m+M} = \frac{\vec{\boldsymbol{v}} + A\vec{\boldsymbol{V}}}{1+A} \tag{2.9}$$

The CM velocities of the target and the neutron are then calculated by subtracting the CM velocity from them, as shown in Eq. (2.10). The "c" subscript will denote the CM-frame values from now on, whereas v_{CM} will denote the velocity of the CM frame relative to the stationary Lab frame.

$$\begin{aligned}\vec{\boldsymbol{v}}_{\mathbf{c}} &= \vec{\boldsymbol{v}} - \vec{\boldsymbol{v}}_{\mathbf{CM}}\\ \vec{\boldsymbol{V}}_{\mathbf{c}} &= \vec{\boldsymbol{V}} - \vec{\boldsymbol{v}}_{\mathbf{CM}}\end{aligned} \tag{2.10}$$

Once in the CM frame, the equation for conservation of momentum can be written as Eq. (2.11), where the primed values are those after the collision. Since the net momentum is zero, the neutron and the target must be traveling in exactly opposite directions, as shown in Eq. (2.12).

$$\begin{aligned} m\vec{\boldsymbol{v}}_{\mathrm{c}} + M\vec{\boldsymbol{V}}_{\mathrm{c}} &= m\vec{\boldsymbol{v}}'_{\mathrm{c}} + M\vec{\boldsymbol{V}}'_{\mathrm{c}} = 0 \\ \vec{\boldsymbol{v}}_{\mathrm{c}} + A\vec{\boldsymbol{V}}_{\mathrm{c}} &= \vec{\boldsymbol{v}}'_{\mathrm{c}} + A\vec{\boldsymbol{V}}'_{\mathrm{c}} = 0 \end{aligned} \tag{2.11}$$

$$\begin{aligned} \vec{\boldsymbol{v}}'_{\mathrm{c}} &= -A\vec{\boldsymbol{V}}'_{\mathrm{c}} \\ \vec{\boldsymbol{v}}_{\mathrm{c}} &= -A\vec{\boldsymbol{V}}_{\mathrm{c}} \end{aligned} \tag{2.12}$$

The equation for conservation of energy is shown in Eq. (2.13). Q is the amount of energy released by the reaction and is zero for elastic scattering. It is convenient to include in this derivation for use later in inelastic collision kinematics where it is nonzero.

$$\begin{aligned} mv_{\mathrm{c}}^2 + MV_{\mathrm{c}}^2 &= mv_{\mathrm{c}}'^2 + MV_{\mathrm{c}}'^2 + 2Q \\ v_{\mathrm{c}}^2 + AV_{\mathrm{c}}^2 &= v_{\mathrm{c}}'^2 + AV_{\mathrm{c}}'^2 + \frac{2Q}{m} \end{aligned} \tag{2.13}$$

There are now two unknowns (the primed values) and two equations, and the final velocities of the neutron and the target can be determined. Substituting Eq. (2.12) into Eq. (2.13) and solving for either v'_{c} or V'_{c} yields either equation in Eq. (2.14). If Q is zero, as it is in elastic scattering, the initial and final velocities are the same for both the neutron and the target, and the interaction only causes a rotation in the CM frame.

$$\begin{aligned} v'_{\mathrm{c}} &= \sqrt{v_{\mathrm{c}}^2 + \frac{2AQ}{m(A+1)}} \\ V'_{\mathrm{c}} &= \sqrt{V_{\mathrm{c}}^2 + \frac{2Q}{mA(A+1)}} \end{aligned} \tag{2.14}$$

At first glance, it seems like the interaction has been fully characterized, but Eq. (2.12) only relates the initial state of the neutron to the initial state of the target and the final state of neutron to the final state of the target. The initial state and final state of the neutron still need to be related. It has been mentioned already that the interaction is only a rotation in the CM frame, so the initial and final state of the neutron's direction can be related by a three-dimensional rotation formula.

An efficient algorithm is given by Eq. (2.15) [13]. In the formula, $\mu = \cos\theta$, and $\hat{\Omega}_x$, $\hat{\Omega}_y$, and $\hat{\Omega}_z$ are the cartesian projections of the velocity's unit vector. It is "efficient" in the sense

that to rotate a vector, a full 3x3 matrix does not need to be constructed and multiplied by the vector. In other words, matrix-vector operations are not needed and the rotation can be carried out with three scalar operations.

$$
\begin{aligned}
\hat{\Omega}'_x &= \mu\hat{\Omega}_x + \frac{\sqrt{1-\mu^2}(\hat{\Omega}_x\hat{\Omega}_z\cos\phi - \hat{\Omega}_y\sin\phi)}{\sqrt{1-\mu\hat{\Omega}_z^2}} \\
\hat{\Omega}'_y &= \mu\hat{\Omega}_y + \frac{\sqrt{1-\mu^2}(\hat{\Omega}_y\hat{\Omega}_z\cos\phi + \hat{\Omega}_x\sin\phi)}{\sqrt{1-\mu\hat{\Omega}_z^2}} \\
\hat{\Omega}'_z &= \mu\hat{\Omega}_z - \sqrt{(1-\mu^2)(1-\mu\hat{\Omega}_z^2)}\cos\phi
\end{aligned}
\tag{2.15}
$$

If the polar and azimuthal rotation angles, θ and ϕ, respectively, are determined, the initial neutron velocity vector can be rotated through these angles to its final value. After the rotation is done, the final velocities are known in the CM frame and they can be transformed back to the Lab frame to give the final velocities there, as shown in Eq. (2.16).

$$
\begin{aligned}
\vec{\boldsymbol{v}}' &= \vec{\boldsymbol{v}}'_{\mathbf{c}} + \vec{\boldsymbol{v}}_{\mathbf{CM}} \\
\vec{\boldsymbol{V}}' &= \vec{\boldsymbol{V}}'_{\mathbf{c}} + \vec{\boldsymbol{v}}_{\mathbf{CM}}
\end{aligned}
\tag{2.16}
$$

Inelastic Level Scattering

Inelastic scattering is the other type of scattering a neutron can undergo, but in this case kinetic energy is no longer conserved. Energy is transferred to or from an internal state of the target nucleus. This amount of energy, Q, is typically defined to be positive for reactions where energy is given to the neutron and target nucleus, i.e. Q is positive when the sum of the particles' kinetic energies is greater after the reaction than before. Therefore, Q values for inelastic scattering are negative, since energy is always lost to an internal state of the nucleus. In neutron-nucleus collisions, the target nucleus can be excited to a higher energy state than its ground state if the colliding neutron has a high enough energy to do so. If the colliding neutron has enough energy and the collision excites the nucleus, a discrete amount of energy is lost to the reaction. These excited states typically do not have long half lives, and a gamma ray is emitted when the nucleus relaxes to its ground state. This type of reaction is called inelastic *level* scattering because an excited energy level becomes occupied by the target nucleus.

Since this reaction is still a two-body interaction, the kinematics of the reaction are identical to elastic scattering except the Q value is nonzero and negative. These reactions have a threshold energy (below which their cross sections are zero) since the neutron needs to have enough energy to excite the target nucleus. Figure 2.4 shows the energy levels in ^{177}Hf, which is often used as a thermal neutron absorber due its large thermal capture cross

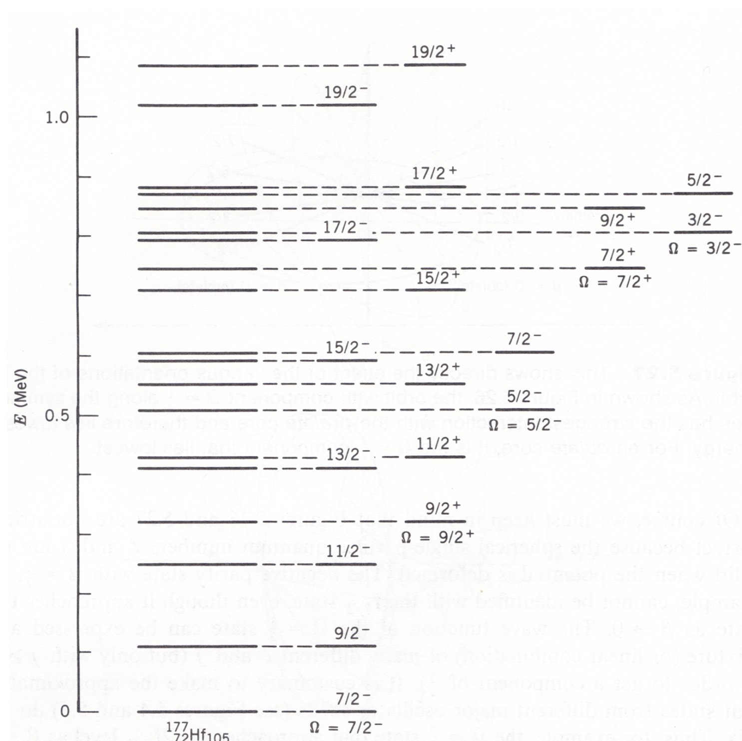


Figure 2.4: The energy levels of ^{177}Hf [2].

section, but it could also be used as a fast neutron moderator because it has many low-lying energy levels and large inelastic scattering cross sections.

Inelastic Continuum Scattering

At energies above the distinct levels there lies a continuum in the nuclear energy states. This isn't truly a "continuum" in a strict sense, but the energy levels become so close together they effectively form a domain where energy can take near-continuous values. Unlike the discrete Q values corresponding to a single excited state used in inelastic level scattering, the Q value of the reaction now follows a distribution [2].

Fission

Fission literally means "the splitting of something into two parts." This is exactly what nuclear fission is as well. Nuclear fission is when a nucleus splits into two smaller nuclei, called *fission fragments*. When heavy nuclei undergo fission, they release energy and typically emit a few other particles, including neutrons. That this reaction releases energy is the reason heavy fissile elements, like uranium, can be used as an energy source. Fission fragments have higher binding energy per nucleon compared to the parent nucleus, and as a result the total mass of the fission products plus initiating neutron is less mass than the parent, meaning

there is an energy release. Figure 2.5 shows the average binding energy per nucleon for a wide range of nuclides. Note that the peak of the curve is at ^{56}Fe, the most tightly bound nucleus, and that heavier nuclides are lower than it. Fission splits the parent into two lighter nuclides, and since the fragments are more tightly bound, the excess binding energy from the parent is released.

The released energy isn't deposited as heat directly, however. It is released in a range of forms, many of which are converted to heat in the immediate vicinity of the fission event. Table 2.1 shows the fraction of this total energy that is given to each entity [3]. Note that a 5% is given to neutrinos, which is essentially lost because materials have very small neutrino interaction cross sections. The kinetic energy of the fission fragments has the majority of the energy, and since they are heavy and charged, their energy is deposited as heat very near to the fission site. Other particles carry some of the fission energy further away from the fission site, but their energy is still almost completely converted to heat.

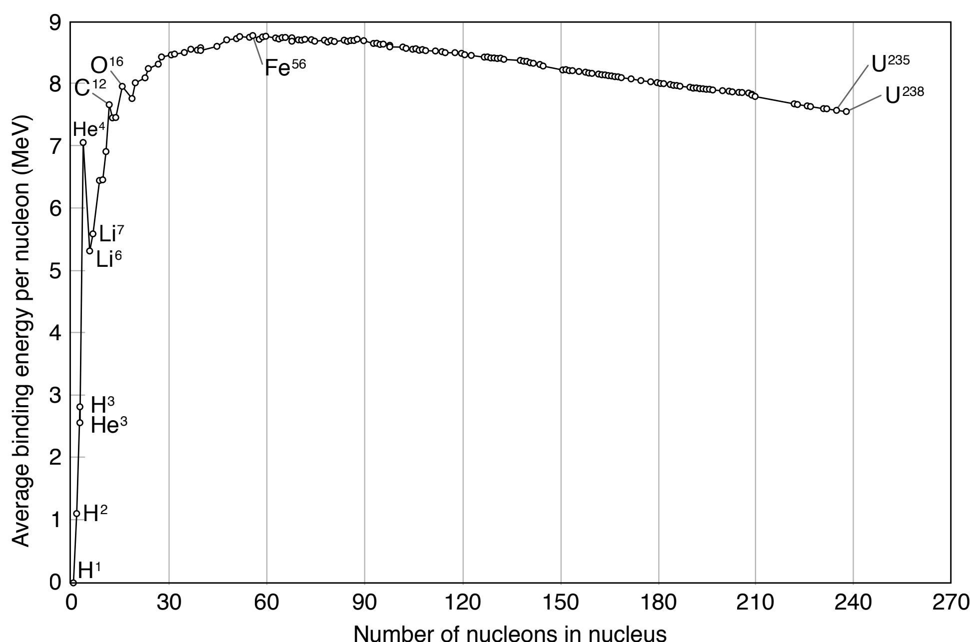


Figure 2.5: Average binding energy per nucleon [34].

Fission is technically considered an absorption reaction since the fission-inducing neutron enters the nucleus and creates a compound nucleus. Even though the neutron is absorbed, more than two new neutrons are released by each fission on average, enabling a fission chain reaction to occur. An important parameter of the fission chain reaction is ν_{T}, the average total number of neutrons emitted per fission. This number is called "total" since includes prompt neutrons, which appear immediately after fission, and delayed neutrons, which appear later. Delayed neutrons are mainly produced from fission product decay, but a small fraction also comes from photon-induced emission. These neutrons are not "prompt" in

Table 2.1: Average distribution of of ^{235}U fission energy [3].

Particle	Energy (% of 192.9±0.5 MeV)	Range	Time
Fission fragment kinetic energy	80	<0.1cm	prompt
Prompt neutrons	3	10-100 cm	prompt
Photons	4	100 cm	prompt
Fission product β decay	4	short	delayed
Neutrinos	5	extremely long	delayed
Nonfission reactions from n capture	4	100 cm	delayed

that they are not emitted immediately from the fission itself. The processes that create these "delayed" neutrons (decay and nuclear relaxation) take time to occur and these neutrons can therefore appear anywhere between 0.6 to 80 seconds after a fission event [3].

The kinetics of a nuclear chain reaction depends heavily on the mean neutron lifetime, which was touched upon in 2.1. If only prompt neutrons are considered, the mean neutron lifetime is approximately 10^{-4} seconds in light water (thermal spectrum) reactors fueled by ^{235}U [3]. Having a reactor that is critical solely on prompt neutrons means that it's power can change on the order of the prompt neutron lifetime. At this speed, it would be difficult for control systems to respond in time to suppress any power excursions before damage occurred. This is where delayed neutrons come into play. If a reactor is not critical with prompt neutrons, but is critical with the incorporation of the delayed neutrons, the long appearance time of delayed neutrons shifts the mean neutron lifetime to larger values; typically to around 0.1 seconds for light water reactors [3]. The reactor is much easier to control with this much slower mean neutron lifetime.

Figure 2.6 shows the energy dependence of the average total number of neutrons per fission, ν_{T}, for several fissionable isotopes. The average total number of neutrons per fission is shown at a single target material temperature since it has very weak temperature dependence for most incoming neutron energies. For all three isotopes shown, ν_{T} is nearly constant from low energies up to energies in the MeV range, where it increases sharply due to the energy the incident neutron provides being sufficient to eject additional bound neutrons.

Figure 2.7 shows the total fission cross sections for two fissile isotopes, ^{235}U and ^{239}Pu, and one fertile isotope, ^{238}U. The figure shows that ^{238}U has a fission cross section, but it isn't significant until above 1 MeV. For ^{238}U, fission is practically a threshold reaction. Simply absorbing a neutron does not provide enough energy to split the nucleus. Additional energy is required, which can be provided in the form of an incident neutron's kinetic energy. ^{238}U isn't fissile, but it can be a significant contributor of fission reactions in reactors where the neutron population is *fast*, i.e. mainly high-energy. As mentioned before, ^{238}U is considered fertile because it is converted to the fissile isotope ^{239}Pu after absorbing a neutron and radioactively decaying [3].

WARP uses ν_{T} and fission neutron energy spectra that incorporate delayed neutron

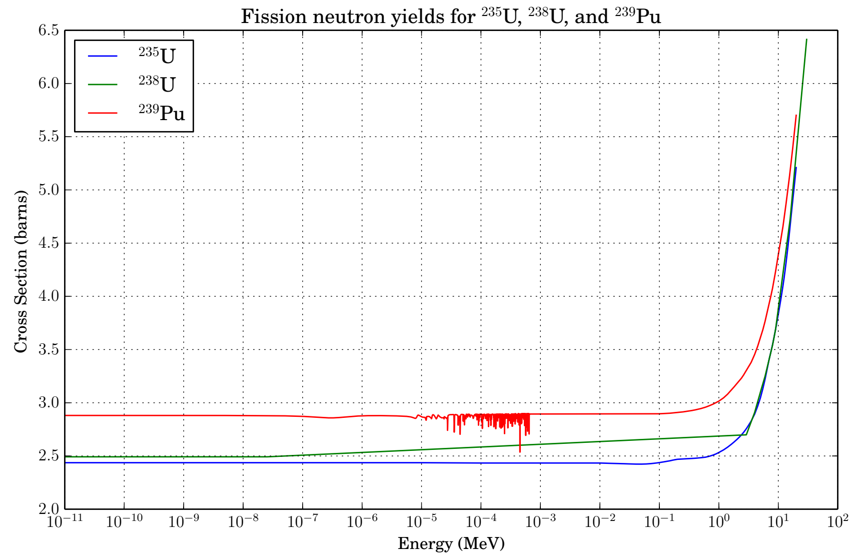


Figure 2.6: The average total number of neutrons per fission, ν_{T}, for ^{235}U, ^{238}U, and ^{239}Pu.

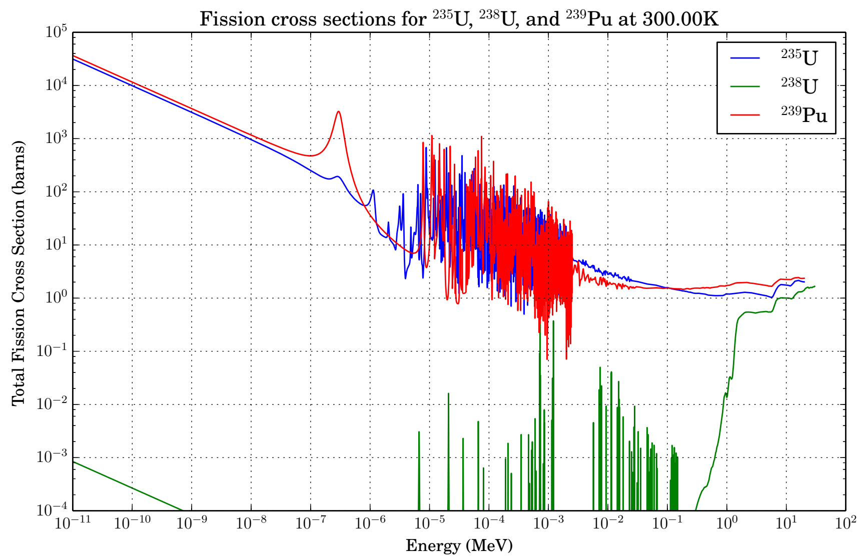


Figure 2.7: Fission cross sections for ^{235}U, ^{238}U, and ^{239}Pu.

energies since it is more physically accurate and produces the overall effective multiplication factor.

Capture Reactions

Unlike scattering reactions, where the energy and direction of the neutron is changed but continues to transport, *capture* reactions remove a free neutron. Typically this leaves the daughter nucleus in an excited state, which then relaxes to ground state by emitting a gamma ray, which is why captures are sometimes called (n,γ) reactions [2].

Other Secondary-Producing Absorption Reactions

This category encompasses all the other reactions neutrons undergo. There are two types: reactions that produce secondary neutrons in some amount and reactions that do not. Those that do not may still produce other particles, like alpha particles, tritons, protons, et cetera. Since these do not produce secondary neutrons, however, they are basically equivalent to capture reactions from a neutron transport standpoint. Even though they aren't strictly capture reactions, they can contribute significantly to an isotope's absorption of neutrons. Figure 2.2 shows that the (n,α) in ^{10}B is by far the main component of the total cross section, making ^{10}B a very strong absorber of low energy neutrons. ^{10}B is widely used in safety and control systems in thermal-spectrum reactors.

Of the reactions that produce secondary neutrons, the (n,2n) reaction is most significant because it has the lowest threshold energy. These reactions are called *multiplicity* reactions [11]. At higher incident neutron energies, (n,3n) and even (n,4n) can become possible. Other particle combinations are possible as well, such as (n,nα), and these reactions act like an inelastic scatter interaction where the relationship between scattering angle and energy no longer applies due to there being three bodies to distribute energy to instead of only two.

Bound Nuclei and Unresolved Resonances

Treating matter as a collection of free nuclei is a good approximation most of the time since the neutron energies are much larger than the thermal energies of the nuclei and the recoil energies imparted on the targets is typically much larger than any cohesive forces between them [12]. This assumption is not valid for many important moderator materials like water and graphite. In these materials, the atoms are bonded to each other and these bonds provide another degree of freedom for energy transfer in scattering. Since the bonds are of the same energy as a low energy neutron, a significant amount of energy can be lost to breaking them when the neutron scatters off bound nuclei. This changes the scattering kinematics significantly and the reactions rates can be affected from this change [12]. The kinematics of thermal scattering are handled via the S(α,β) coupled energy-angle representation, which replaces the free gas treatment discussed in the next section [11]. WARP does not included the S(α,β) treatment currently. Adding it will be a point of future development.

There is also a special treatment for sampling reaction types in the unresolved resonance region. The unresolved resonance region is the energy range above the resolved resonance region where the resonances are so closely spaced together that the cross section appears

to be smoothly-varying (e.g. 2.25 - 25 keV for ^{235}U) [11]. Treating the cross sections as smooth in this region does not account for resonance self-shielding effects, and can produce inaccurate results in systems where the flux is large in this region. Resonance self shielding is where the flux near a resonance becomes depressed due to strong absorption in the resonance, and this flux depression "shields" the nuclei from neutrons at this energy [3]. Probability tables are included in ENDF data to account for these resonances, but these tables are not loaded by WARP. Adding the unresolved resonance treatment will be another point of future development for WARP.

2.3 Temperature Effects

It is a good assumption that the thermal motion of the target nuclide is negligible when neutrons are at MeV-range energies. However, when neutrons scatter and lose energy they can come near the thermal energy of the material, which is on the order of .01 eV. When this happens, the target nuclide no longer appears stationary, and assuming that it has zero velocity in scattering calculations is inaccurate as discussed above.

MCNP sets the threshold above which the target nuclide can be assumed stationary at 400 kT, which corresponds to about 10 eV for materials at room temperature [11]. Below this threshold, it is important to model thermal effects. If this wasn't done, a neutron could keep scattering off of zero-energy targets and its energy could approach zero, which is not physical. Neutrons can only scatter down to a state where they are in thermal equilibrium with the material they are traveling through. This creates a "thermal peak" at low energies where neutrons collect, especially in materials where the absorption-to-scattering cross section ratio is small and neutrons scatter many times before they are absorbed.

Doppler Effect

The other effect that thermal motion has is the *Doppler effect*. The nuclei in a material are assumed to be in thermal equilibrium, and the velocities of the nuclei follow a Maxwell-Boltzmann distribution with a mean value corresponding to the material's temperature. Cross section data is adjusted to preserved true reaction rates even though target nucleus is assumed to be at rest. This adjustment basically involves convolving the cross sections with the thermal distribution of the target velocities. When temperature rises, the thermal velocity distribution becomes wider, and this manifests itself in the cross sections by broadening resonances. The effect is also known as *Doppler broadening* for this reason.

Figure 2.8 shows the Maxwell-Boltzmann distributions at various temperatures for a heavy nucleus and for a light nucleus. This is the distribution of speeds particles in a "gas" have if they only interact by scattering off of one another. Most solids can be modeled as a dense gas when there are no strong anisotropies in their structure, which is why modeling the target velocities in this way is called the "free gas model." Note that the broadening effect is much more pronounced for light nuclei [3].

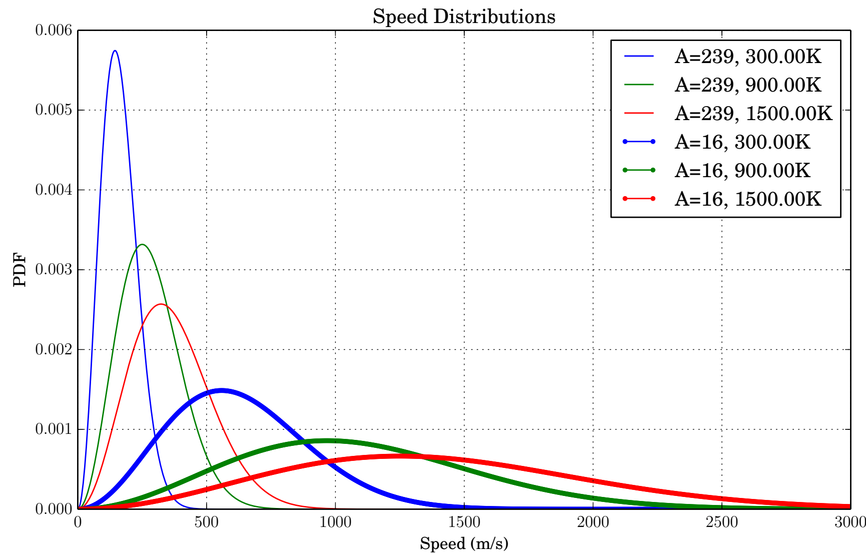


Figure 2.8: Maxwell-Boltzmann distribution for the speed of heavy and light nuclides at different temperatures.

The widening of resonances affects reactors in significant ways. The most notable is that absorption probability in resonances increases in the resonance region as neutrons scatter down to thermal energies. Since the number of neutrons lost to capture increases, Doppler broadening reduces the overall multiplication factor. This effect is important for reactor safety since it produces a negative reactivity feedback for fuel temperature increase and helps prevent power excursions and meltdowns. If the multiplication factor is above unity, the power starts to rise. In the short term, the fuel temperature will rise more rapidly than any coolant mechanism can respond, so a increase in power corresponds directly to an increase in fuel temperature. When the temperature goes up, the increased captures causes the fission rate and thus temperature to decrease, stopping the power from increasing further [3]. There are many different types of reactivity feedback phenomena, but the fuel temperature feedback is generally negative because of Doppler broadening.

Capture increases most in fission products since they often have strong absorption resonances and are lighter than fuel nuclides. Very light nuclides typically do not have absorption resonances, so Doppler broadening has littler effect on their absorption rates. This is why temperature feedback is least effective in fresh fuel where there are few fission products. Figure 2.9 shows the effect in ^{155}Eu, a fission product with a high capture cross section.

The way cross sections are processed assumes that the target nucleus will be treated as being at rest. Since the cross section actually depends on the relative velocity, the total reaction rate of a neutron at a specific energy is the sum of the reaction rates across all the relative velocities present due to thermal motion of the targets. The accurate total reaction rate can be preserved even if the target is assumed at rest if the thermal motion effect has

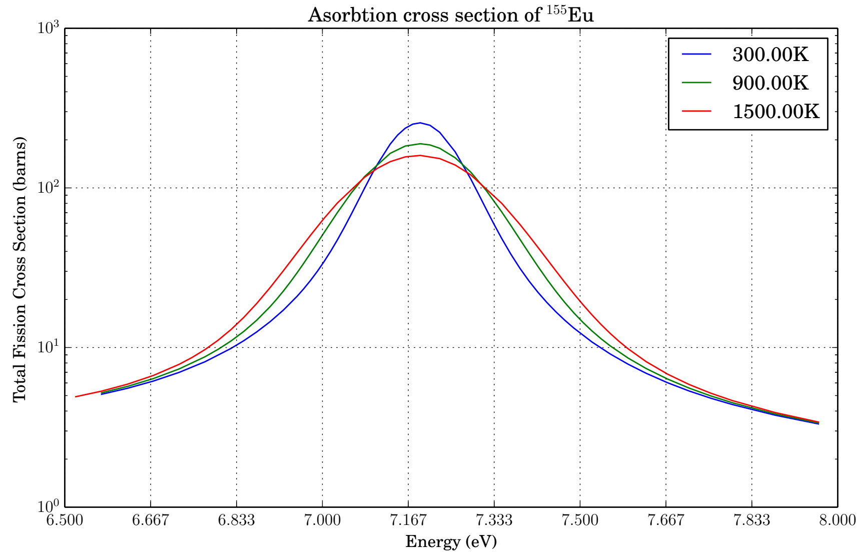


Figure 2.9: Doppler broadening of the 1 eV fission resonance in ^{155}Eu.

already been incorporated into the cross section. This temperature processed cross section is a called a *thermally averaged* cross section, $\bar{\sigma}$. It can be used in simulations assuming a target at rest, and the simulation will produce the same results as one that explicitly models the target thermal motion. The effects of thermal averaging are especially significant at resonances since slight movements in velocity can produce very large differences in cross section. The overall effect is that sharp resonances are effectively broadened since they start to contribute to reaction rates, and therefore thermally averaged cross sections, once the thermal distribution of relative velocities starts overlap them.

The expression for thermally-averaged cross sections is shown in Eq. (2.17) [13], where v_n and $\boldsymbol{v_n}$ are the speed and velocity of the neutron, respectively, $\boldsymbol{v_t}$ is the velocity of the target, $v_{\text{rel}} = ||\boldsymbol{v_n} - \boldsymbol{v_t}||$ is the velocity of the neutron relative to the target, and $M(v_t)$ is the thermal distribution of target speeds.

$$\bar{\sigma} = \frac{1}{v_n} \int v_{\text{rel}} \sigma(v_{\text{rel}}) M(v_t) dv_t \tag{2.17}$$

The fact that cross sections have been adjusted for a material temperature will be significant in Section 2.6 when the target-at-rest assumption breaks down and it becomes necessary to preserve the correct reaction rates while still using the temperature-processed cross sections.

2.4 Nuclear Data

Cross section data compiled by the United States is distributed by the Department of Energy in *ENDF* files. ENDF stands for "evaluated nuclear data file," and can contain data for nuclear decay, photons, atomic relaxation, fission yields, thermal neutron scattering, and charged particle reactions as well as neutron reactions. The data files are called "evaluated" because a group of experts decides, or evaluates, what data is included in them. The data includes theoretical calculations of cross sections based on well developed models as well as experimental data. They also decide how to represent regimes that haven't been measured yet by comparing simulation results to experiments. The first data released was ENDF/B-I in 1968 and the latest set is ENDF/B-VII, which was released in 2006.

The data is written in a standard format that dates back to when the data was stored on magnetic tapes, and data entries are sometimes referred to as "tapes" to this day. The format is rather archaic and contains a lot of redundant information about record locations, which was useful when the tape head had to physically move between points in the tape [35].

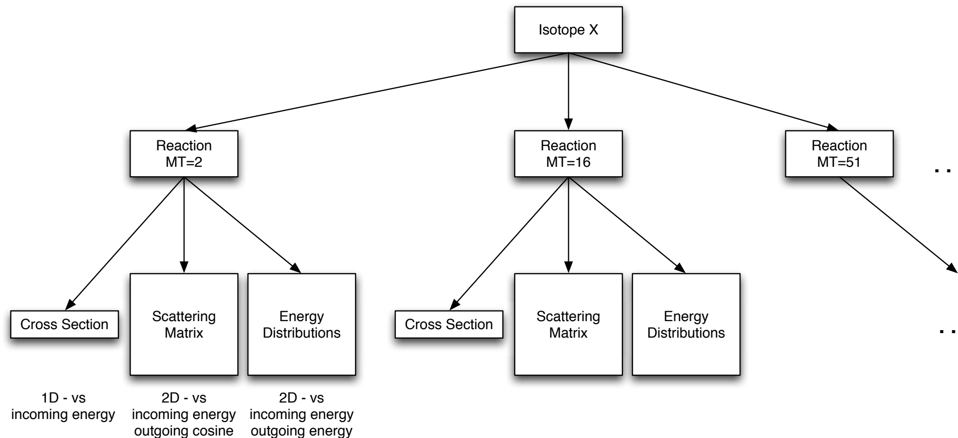


Figure 2.10: Hierarchy in ACE formatted neutron data libraries.

Many Monte Carlo codes read ACE-formatted data rather than the original ENDF file. ENDF files contain a lot of data so the dataset preserves the original evaluation, but most transport codes need the data in tabular format. This is why transport codes normally use ACE-formatted data files. [12] ACE stands for "a compact ENDF" and strips out a lot of the extra information unnecessary for neutron transport. ENDF tapes contain information that is valuable for charged particles and photons as well as neutrons, and this information is discarded for neutron transport. As Figure 2.10 shows, ACE files not only contain cross sections, but also angle and energy distributions used in scattering and fission. ENDF assigns a number to each type of reaction called the *MT* number. Table B.3 in Appendix B, taken from a LANL website, shows what these MT numbers mean [35].

It can clearly be seen that there are many reactions a neutron can undergo, most of which have very strong energy dependence. Most of the complexity in modeling nuclear reactors

comes from the fact that the data needed to model neutrons is very complicated. Data may be the most important part of the simulation; it is what ties the calculations to reality.

ACE data files typically come pre-processed at different temperatures. This processing can done by a code called NJOY [12], which Doppler broadens all the resonances in the cross sections and adjusts the unresolved resonance tables accordingly. It can also thin the energy grid if requested by the user, though this reduces the accuracy of the cross sections [12]. Thinning the energy grid is a computationally intensive task, and NJOY is not parallelized. Most production codes come with their own pre-processed datasets at several temperature intervals in order to save the user the time and effort needed to process data from ENDF files.

2.5 Neutron Transport

Now that the events that can happen to neutrons have been outlined, we will move to describing the neutron population itself. Since the neutron population in reactor cores is large, on the order of $10^8/\text{cm}^3$ [3], the neutrons themselves have very small radii, about 1.75×10^{-17} cm [2], and only the average behavior matters, their discrete distribution can be well-approximated by a continuous distribution function. We will eventually derive the *neutron transport equation*, which is a linearized version of the Boltzmann transport equation. It is linear since it is assumed that neutrons do not interact with each other. This is usually a good assumption in normal matter since the neutron density present in reactors is many orders of magnitude smaller than the material density and neutrons are much more likely to interact with the nuclei than each other [3].

Other than eliminating neutron self-interaction, there are several assumptions that go into the equation that will hold true for the rest of the derivations. The first is that neutrons are assumed to be points in space, so even at very high densities they still will not interact with each other. Treating neutrons as points also mean neutrons cannot be in more than one unique volume by overlapping boundaries. The next assumption is that any relativistic effects are negligible. The energies of importance in reactor physics are below 10 MeV, far below the neutron rest mass, and any changes in neutron mass will be below 1%. Since neutrons are neutral, they are assumed to move in straight lines between collisions. Materials are also assumed to be in thermal equilibrium and to have isotropic properties [3]. As mentioned previously, some common reactor materials, like graphite and water, do not have isotropic properties when it comes to scattering, however, but this can be corrected with $S(\alpha,\beta)$ tables.

Neutron Balance Equation

Before the transport equation is formulated, a neutron balance equation can be written for a differential volume. This equation describes the number of neutrons entering and exiting an infinitesimally-small volume with the difference being equal to the rate of change of the neutrons within the volume. The reactions that neutrons can undergo and how

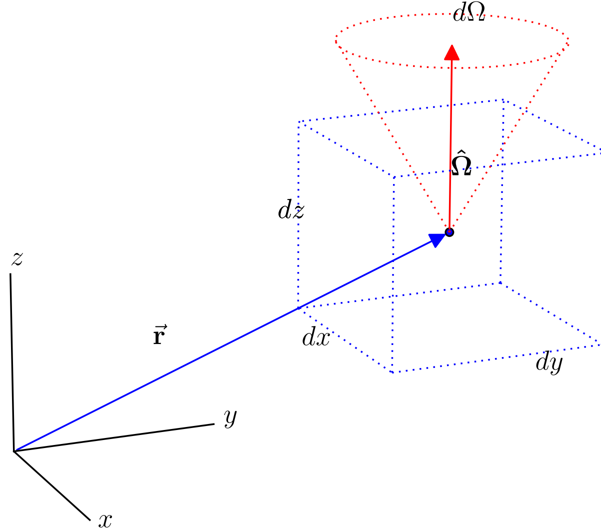


Figure 2.11: The differential volume of the neutron balance equation.

neutrons move within the material are known, so there is enough information to write such an equation. The balance equation for the neutron density, n, is shown in Eq. (2.18), and Figure 2.11 shows an illustration of the differential volume.

$$\begin{aligned} \text{rate of change} &= (\text{production and neutrons in}) - (\text{loss and neutrons out}) \\ \frac{\partial n}{\partial t} &= (\text{movement in} + \text{source} + \text{scatter in}) - (\text{movement out} + \text{absorbtion} + \text{scatter out}) \end{aligned} \tag{2.18}$$

Neutron Distribution Function

The *neutron distribution function*, $n(\vec{\boldsymbol{r}}, \hat{\mathbf{\Omega}}, E, t)$, is the number of neutrons in volume $d\vec{\boldsymbol{r}}$ around point $\vec{\boldsymbol{r}}$, in $d\hat{\mathbf{\Omega}}$ around angle $\hat{\mathbf{\Omega}}$, dE around energy E, and dt around time t.

$$n(\vec{\boldsymbol{r}}, \hat{\mathbf{\Omega}}, E, t)\ \ d\vec{\boldsymbol{r}}\, d\hat{\mathbf{\Omega}}\, dE\, dt \tag{2.19}$$

The angle vector, $\hat{\mathbf{\Omega}}$, is a unit vector that specifies direction only, not magnitude. It is easiest to express the directional vector in spherical coordinates (θ, ϕ), the polar angle and the azimuthal angle, respectively. Their Cartesian projections are given by Eq. (2.20) and shown in Figure 2.12.

$$\begin{aligned} \Omega_x &= \sin\theta \sin\phi \\ \Omega_y &= \sin\theta \cos\phi \\ \Omega_z &= \cos\theta = \mu \end{aligned} \tag{2.20}$$

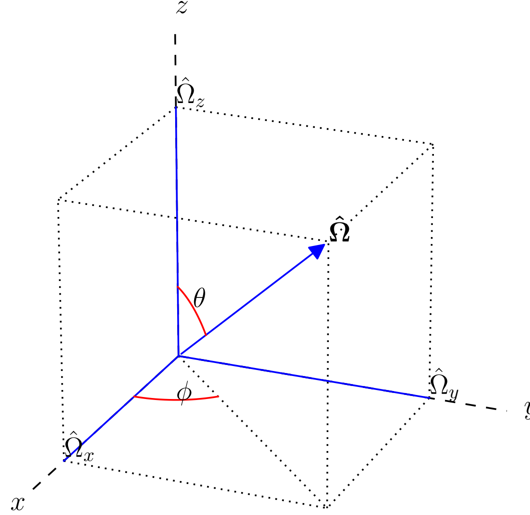


Figure 2.12: The Cartesian projections of the directional vector.

Like any continuous distribution, it must be integrated to calculate scalar or integral quantities. If we define the ith moment of the distribution according to Eq. (2.21), then the 0th moment would be the population and the 1st moment would be the mean if divided by the 0th moment.

$$M_i = \int_{-\infty}^{\infty} n(x)x^i dx \tag{2.21}$$

For example, if we are given a neutron distribution which has no time dependence, $n(\vec{\boldsymbol{r}}, \hat{\mathbf{\Omega}}, E)$, and wanted to calculate N_V, the total number of neutrons present in volume V, we would calculate this number by Eq. (2.22), whereas if we wanted to know the average energy, $\bar{E}$, we would do this by Eq. (2.23).

$$N_V = \int_0^{\infty} dE \int_{\hat{\mathbf{\Omega}}} d\hat{\mathbf{\Omega}} \int_V d\vec{\boldsymbol{r}}\, n(\vec{\boldsymbol{r}}, \hat{\mathbf{\Omega}}, E) \tag{2.22}$$

$$\bar{E} = \frac{1}{N_V} \int_0^{\infty} dE \int_{\hat{\mathbf{\Omega}}} d\hat{\mathbf{\Omega}} \int_V d\vec{\boldsymbol{r}}\, En(\vec{\boldsymbol{r}}, \hat{\mathbf{\Omega}}, E) \tag{2.23}$$

Reaction Rates

A *reaction rate* is the rate at which a certain reaction is happening in a volume [3]. Consider N particles that travel at speed v in one direction. If Σ is the interaction probability per unit length, multiplying it by the speed gives the probability of interaction per second, or the *collision rate*. Since there are N particles, multiplying the collision rate by N gives

the overall reaction rate, $Nv\Sigma$, of the particles in an infinite medium. If N is substituted for the neutron distribution function instead of a pulse, the expression becomes the reaction rate per distribution differential, or the *reaction rate density*, $R(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t)$. This expression is shown in Eq. (2.24) and is the first building block of the explicit neutron balance equation.

$$R(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = v(E)n(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t)\Sigma(\vec{\boldsymbol{r}}, E) \tag{2.24}$$

If we assume there is only one energy, E_0, and one direction that the neutron distribution varies along, $\hat{x}$, and integrate over the other dimensions of this equation, as shown in Eq. (2.25), we get an expression for the reaction rate in an infinitesimal slice, dx. A is the area perpendicular to the direction of motion where the neutron population is nonzero.

$$\int_0^\infty dE \int_{\hat{\boldsymbol{\Omega}}} d\hat{\boldsymbol{\Omega}} \int_V dxdydzv(E)n(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t)\Sigma(\vec{\boldsymbol{r}}, E)\delta(E-E_0)\delta(\hat{\boldsymbol{\Omega}}-\hat{x}) = v(E_0)\Sigma(E_0)An_{\hat{x}}(x)dx \tag{2.25}$$

This is effectively the loss term for a beam in the $\hat{x}$ direction, and if we set it as such, we recover the linear attenuation expression as shown in Eq. (2.26), which is equivalent to Eq. (2.1).

$$dn(x) = -v\Sigma An(x)dx \qquad \Rightarrow \qquad \frac{dN(x)}{dx} = -\Sigma N \tag{2.26}$$

Angular and Scalar Flux

Since the reactions rate density is dependent on the cross section and the velocity multiplied by the neutron distribution, the product of just the velocity and the neutron distribution is often defined as the *angular flux density*, $\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t)$, as shown in Eq. (2.27). It is called a "flux" because it represents a rate at which particles are passing through a surface and "angular" because it is angle-dependent. Since the reaction rates depend on this quantity, the neutron transport problem is usually written in terms of the angular flux density, which is then solved for instead of the neutron distribution.

$$\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = v(E)n(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.27}$$

Scattering cross sections have angular dependence, but absorption cross sections typically do not (their angular probability distribution function is a constant), so absorption reactions can be written in terms of the *scalar flux density* (or simply the *flux*), which is angular flux that has been integrated over all angles. The relation between the angular and scalar fluxes is show in Eq. (2.28). The scalar flux is usually the most interesting quantity in reactor physics since the reactor power profile is directly proportional to it (power comes from fission, which is an absorption reaction). The reaction rate for a reaction i that has no angular dependence is shown in Eq. (2.29).

$$\phi(\vec{\boldsymbol{r}}, E, t) = \int_{\hat{\boldsymbol{\Omega}}} d\hat{\boldsymbol{\Omega}} \psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.28}$$

$$\begin{aligned} R_i(\vec{\boldsymbol{r}}, E, t) &= \int_{\hat{\boldsymbol{\Omega}}} d\hat{\boldsymbol{\Omega}}\, \Sigma_i(\vec{\boldsymbol{r}}, E) \psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = \Sigma_i(\vec{\boldsymbol{r}}, E) \int_{\hat{\boldsymbol{\Omega}}} d\hat{\boldsymbol{\Omega}}\, \psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) \\ &= \Sigma_i(\vec{\boldsymbol{r}}, E) \phi(\vec{\boldsymbol{r}}, E, t) \end{aligned} \tag{2.29}$$

From Eq. (2.18) we can see that the time derivative is in terms of the neutron density, not the angular flux density. Thus, the time dependent term must be transformed to angular flux density by multiplying and dividing it by the velocity as shown in Eq. (2.30).

$$\frac{\partial}{\partial t} n(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = \frac{\partial}{\partial t} \frac{v(E)}{v(E)} n(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = \frac{\partial}{\partial t} \frac{\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t)}{v(E)} = \frac{1}{v(E)} \frac{\partial}{\partial t} \psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.30}$$

Scattering

Scattering is highly dependent on angle in addition to energy and requires a more detailed cross section expression than absorption reactions. Elastic scattering has a fixed relation between incoming and outgoing energy and angle, but inelastic scattering does not. To be completely general in order to describe either type of scattering, all scattering cross sections are assumed to depend not only on incoming energy (like all cross sections), but also on outgoing energy, incoming angle, and outgoing angle. Since two quantities are being related before and after the scattering event, the scattering cross section is considered *doubly differential* and is sometimes called the *scattering kernel*, whereas the integrated value, which only depends on incoming energy, is called the scattering cross section. An expression for the scattering kernel is shown in Eq. (2.31) with the outgoing values primed [12].

$$\Sigma_s(\vec{\boldsymbol{r}}, E) = \int_{\hat{\boldsymbol{\Omega}}} d\hat{\boldsymbol{\Omega}}' \int_0^\infty dE' \;\; \Sigma_s(\vec{\boldsymbol{r}}, E \to E', \hat{\boldsymbol{\Omega}} \to \hat{\boldsymbol{\Omega}}') \tag{2.31}$$

There is a practical reason for separating scattering into a cross section that describes the likelihood of a neutron entering the reaction, and a kernel that describes how it exits. In a Monte Carlo simulation, the cross section is used to determine *whether* scattering happens rather than *how* it happens. The scattering kernel is used to determine exit energy and angle only if a neutron has already been determined to scatter.

This separation is also useful in expressions for scattering in a differential volume. The cross section is used as the removal of a neutron from a particular energy and/or angle (it is the doubly differential scattering cross section integrated over all outgoing directions and energies), whereas the kernel is used to determine which other energies and angles contribute to neutrons scattering *into* a particular energy and/or angle. This can be seen in Eq. (2.31). If the outgoing energy and angle are held constant, the quantity is the probability for which other angles and energies scatter into it.

Using these two quantities, in-scattering and out-scattering terms can be written for the differential volume. The expression for loss uses the integrated cross section, shown in Eq. (2.32), and the kernel is used in what is often called the *scattering source* term, shown in Eq. (2.33). The source term needs to be integrated over all other energies, E', and angles, $\hat{\boldsymbol{\Omega}}'$, from which neutrons can scatter into energy E and angle $\hat{\boldsymbol{\Omega}}$.

$$R_{s,\mathrm{out}}(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = \Sigma_s(\vec{\boldsymbol{r}}, E)\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.32}$$

$$R_{s,\mathrm{in}}(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = \int_0^\infty dE' \int_{\hat{\boldsymbol{\Omega}}} d\hat{\boldsymbol{\Omega}}'\, \Sigma_s(E' \to E, \hat{\boldsymbol{\Omega}}' \to \hat{\boldsymbol{\Omega}})\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}', E', t) \tag{2.33}$$

Fission Source

The nuclear chain reaction is sustained by fission reactions producing neutrons, so it is essential to model it in any material that has fissionable isotopes. There are three determining factors for the fission source. The first is the fission reaction rate, which is represented by the flux multiplied by the fission cross section.

The second is the fission spectrum, which is represented by χ. Neutrons born from fission are not emitted at a single energy, and the fission spectrum describes the probability for a neutron to be emitted at a certain energy (it is a probability distribution function). The energy spectrum is weakly dependent of the incoming neutron energy, with higher incoming energies producing more higher energy neutrons. This dependence only starts making a significant difference in spectrum shape at energies above 10 MeV, higher than energies usually seen in a reactor. This is why the fission spectrum is usually treated as being independent of incoming neutron energy. The spectra of ^{235}U and ^{239}Pu for fission induced by a 0.5 MeV neutron are shown in Figure 2.13, and it can be seen that ^{239}Pu produces more high energy neutrons.

The last parameter is the average number of neutrons emitted in a fission event, represented by ν, and is relatively flat until about 1 MeV, as was shown in Figure 2.6. Since 1 MeV is within the typical energy range present in nuclear reactions, it is treated as a function of energy.

The fission source term describes the number of neutrons that are born in dE around energy E and angle $d\hat{\boldsymbol{\Omega}}$ around $\hat{\boldsymbol{\Omega}}$ from fissions occurring from any other energy, so like the in-scattering source, the fission reaction rate must be integrated over all other energies, E', and angles, $\hat{\boldsymbol{\Omega}}'$. Using the general differential cross section notation and integrating the fission reaction rate over all other energies and angles yields the fission source, shown in Eq. (2.34). Multiplying by the average number of neutrons emitted in fission, ν, scales the source to the proper strength.

$$R_f(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = \int_0^\infty dE' \int_{\hat{\boldsymbol{\Omega}}} d\hat{\boldsymbol{\Omega}}'\, \nu_T(E')\Sigma_f(\vec{\boldsymbol{r}}, E' \to E, \hat{\boldsymbol{\Omega}}' \to \hat{\boldsymbol{\Omega}})\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}', E', t) \tag{2.34}$$

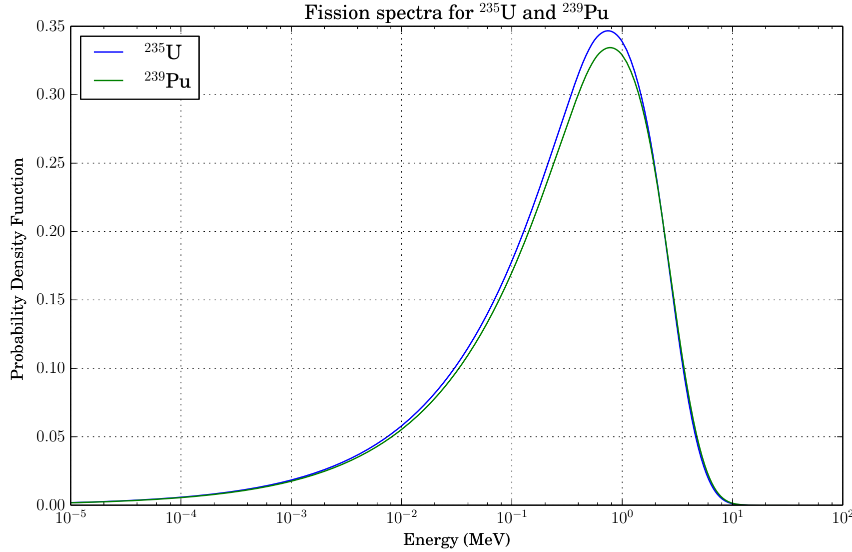


Figure 2.13: Prompt fission neutron spectra for ^{235}U and ^{239}Pu.

The doubly-differential cross section can be split into a total cross section and a pair of probability distribution functions (PDFs) that describe the transfer probability, as shown in Eq. (2.35) [36]. Fission neutrons are born isotropically, so P_2 must be a constant. Since PDFs must integrate to 1, and angles are integrated over 4π seradians, P_2 must equal $1/4\pi$. The fission cross section has no incoming angular dependence and the fission emission energy is very weakly dependent on incoming energy, so P_1 is the total fission spectrum, $\chi_T(E)$, which includes both prompt and delayed neutrons. These PDFs, shown in Eq. (2.36), are not dependent on incoming angle or energy, and can be moved outside the integral. The other two fission quantities, $\nu_T(E')$ and $\Sigma_f(\vec{\boldsymbol{r}}, E')$, do not depend on angle, so the angular integral over angular flux density can be be replaced with the scalar flux. The final fission source expression is shown in Eq. (2.37).

$$\begin{aligned}\Sigma_f(\vec{\boldsymbol{r}}, E' \to E, \hat{\boldsymbol{\Omega}}' \to \hat{\boldsymbol{\Omega}}) &= \Sigma_f(\vec{\boldsymbol{r}}, E', \hat{\boldsymbol{\Omega}}')P_1(E \to E')P_2(\hat{\boldsymbol{\Omega}}' \to \hat{\boldsymbol{\Omega}}) \\ &= \Sigma_f(\vec{\boldsymbol{r}}, E')P_1(E \to E')P_2(\hat{\boldsymbol{\Omega}}' \to \hat{\boldsymbol{\Omega}})\end{aligned} \tag{2.35}$$

$$P_1(E \to E') = \chi_T(E) \qquad\qquad P_2(\hat{\boldsymbol{\Omega}}' \to \hat{\boldsymbol{\Omega}}) = \frac{1}{4\pi} \tag{2.36}$$

$$\begin{aligned}R_f(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) &= \frac{\chi_T(E)}{4\pi}\int_0^\infty \int_{\hat{\boldsymbol{\Omega}}} \nu_T(E')\Sigma_f(\vec{\boldsymbol{r}}, E')\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}', E', t)d\boldsymbol{\Omega}'dE' \\ &= \frac{\chi_T(E)}{4\pi}\int_0^\infty \nu_T(E')\Sigma_f(\vec{\boldsymbol{r}}, E')\phi(\vec{\boldsymbol{r}}, E', t)dE'\end{aligned} \tag{2.37}$$

Streaming

So far all that has been touched on is how neutrons interact within a volume and change in angle and energy. Now we will go over how they move in space, and to do this we consider a surface S around our differential volume. We want to find an expression for the net number of neutrons passing through this surface, and this will be the net leakage term (incoming minus outgoing). The angular neutron flux describes the number of neutrons crossing a differential surface at angle $\hat{\boldsymbol{\Omega}}$, but in order to perform any vector operations on it, we must multiply it by the unit vector. This new quantity is called the *current density*, $\vec{\boldsymbol{J}}$, and is shown in Eq. (2.38).

$$\vec{\boldsymbol{J}}(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = \hat{\boldsymbol{\Omega}}\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.38}$$

We can then write an expression for the leakage by performing a surface integral over the current and surface normal's dot product, shown in Eq. (2.39).

$$\text{Leakage} = \int_S \vec{\boldsymbol{ds}} \cdot \vec{\boldsymbol{J}}(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = \int_S \vec{\boldsymbol{ds}} \cdot \hat{\boldsymbol{\Omega}}\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.39}$$

We can turn this into a volume integral by applying the divergence theorem, shown in Eq. (2.40). If the volume of interest is shrunk to an infinitesimal volume and we switch the order of the dot product, we come to an expression that can be used in the balance equation. This expression, shown in Eq. (2.41), is often called the *streaming* term, since it describes the net movement of neutrons into the differential volume due to their physical movement.

$$\int_S \vec{\boldsymbol{ds}} \cdot \hat{\boldsymbol{\Omega}}\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = \int_V dV \nabla \cdot \hat{\boldsymbol{\Omega}}\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.40}$$

$$\lim_{V \to dV} \int_V dV \nabla \cdot \hat{\boldsymbol{\Omega}}\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) = \hat{\boldsymbol{\Omega}} \cdot \nabla\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E, t) \tag{2.41}$$

Neutron Transport Equation

Now all the terms of Eq. (2.18) have been explicitly defined. Substituting them all in yields Eq. (2.42), the *neutron transport equation*, written here with the neutron sinks on the left side and the neutron sources on the right side. An additional external source term, S_{external}, has been added to account for any neutron sources not induced by the neutron flux itself, i.e. external sources. This form of the equation also includes delayed neutrons. For delayed neutron precursor group j, C_j is the concentration, $\chi_{d,j}$ is the energy spectrum (d only signifies "delayed"), and λ_j is the decay constant [36, 3]. As was stated previously, delayed neutrons are important for reactor control. Since delayed neutrons are included explicitly in these equations, the fission spectrum, $\chi_p(E)$, and average fission neutron yield, $\nu_p(E)$, are for prompt neutrons only. The inclusion of delayed neutrons also means an additional six equations defining the precursor concentrations must be included (since the concentration depends on the flux), and are shown in Eq. (2.43) [36, 3]. WARP is concerned

with time-independent solutions, and the spectral and fission yield effects of delayed neutrons are incorporated into the data WARP uses.

$$
\begin{aligned}
&\frac{1}{v(E)}\frac{\partial}{\partial t}\psi(\vec{\boldsymbol{r}},\hat{\boldsymbol{\Omega}},E,t)+ \\
&\hat{\boldsymbol{\Omega}}\cdot\nabla\psi(\vec{\boldsymbol{r}},\hat{\boldsymbol{\Omega}},E,t)+ \\
&\Sigma_t(\vec{\boldsymbol{r}},E)\psi(\vec{\boldsymbol{r}},\hat{\boldsymbol{\Omega}},E,t) \\
&= \\
&\int_0^\infty dE' \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}'\ \Sigma_s(E'\to E,\hat{\boldsymbol{\Omega}}'\to\hat{\boldsymbol{\Omega}})\psi(\vec{\boldsymbol{r}},\hat{\boldsymbol{\Omega}}',E',t) \\
&+\frac{\chi_p(E)}{4\pi}\int_0^\infty dE' \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}'\ \nu_p(E')\Sigma_f(\vec{\boldsymbol{r}},E')\psi(\vec{\boldsymbol{r}},\hat{\boldsymbol{\Omega}}',E',t) \\
&+\sum_{j=1}^{6}\frac{\chi_{d,j}(E)}{4\pi}\lambda_j C_j(r,t) \\
&+S_{\text{external}}
\end{aligned}
\tag{2.42}
$$

$$
\frac{\partial}{\partial t}C_j(r,t) = -\lambda_j C_j(r,t) + \int_0^\infty dE' \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}'\ \nu_j(E')\Sigma_f(\vec{\boldsymbol{r}},E')\psi(\vec{\boldsymbol{r}},\hat{\boldsymbol{\Omega}}',E',t) \tag{2.43}
$$

The neutron transport equation in this form is an integro-differential equation since it has both derivates and integrals in it. Its spatial and temporal parts are differential, whereas its angular and energy parts are integral. It is linear and relatively easy to solve for simple geometries and reaction parameters, but in order to capture all the relevant physics for real-world problems, complex geometries and energy-dependent reaction parameters must be used. Despite it's linearity, the neutron transport equation can be difficult to solve because of the large, heterogeneous domains over which it must be solved and the complex energy dependence of the cross sections. The energy range of interest can span more then 12 orders of magnitude, from 1×10^{-11} to 1×10^{1} MeV and above, and the geometries involved can include millions of individual material regions containing many different mixtures of materials.

Time Independent Neutron Transport Equation

In most situations, the equilibrium state of the neutron population is of interest. By setting all time derivates and external sources to zero, Eq. (2.42) becomes the time-independent neutron transport equation shown in Eq. (2.44). The equation is no longer driven by an independent source term, and the transport equation becomes homogenous, turning it into an eigenvalue problem. There are infinitely many eigenfunction solutions to Eq. (2.44), and each has an associated eigenvalue. There is guaranteed to be a maximum spatial eigenvalue

that is real and positive which corresponds to a unique and non-negative eigenfunction [3]. This maximum eigenvalue has physical significance – it is equal to the multiplication factor, k_{eff}, which is why criticality calculations are often called "eigenvalue" calculations.

$$\begin{aligned} &\hat{\boldsymbol{\Omega}} \cdot \nabla\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E) + \\ &\Sigma_t(\vec{\boldsymbol{r}}, E)\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}, E) \\ &= \\ &\int_0^\infty dE' \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}' \ \Sigma_s(E' \rightarrow E, \hat{\boldsymbol{\Omega}}' \rightarrow \hat{\boldsymbol{\Omega}})\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}', E') \\ &+ \frac{1}{k_{\text{eff}}} \frac{\chi_T(E)}{4\pi} \int_0^\infty dE' \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}' \ \nu_T(E')\Sigma_f(\vec{\boldsymbol{r}}, E')\psi(\vec{\boldsymbol{r}}, \hat{\boldsymbol{\Omega}}', E') \end{aligned} \tag{2.44}$$

2.6 Monte Carlo

Deterministic methods solve the neutron transport equation directly. They treat the neutron population as a continuous distribution and discretize the spatial, angular, energy, and temporal dimensions that the distribution depends on and solve the transport equation at these discrete points. The Monte Carlo method takes a different approach in solving the transport problem on a computer. It attempts to directly simulate what happens in reality. Instead of the neutron population being continuous and the spatial, angular, and energy dimensions being discretized, it treats the problem dimensions as continuous and discretizes the neutron population. This is how the neutrons actually exist. They can be very well approximated by a continuous distribution, but it is still an approximation. Quantities of interest are then determined by integrating over the neutron population.

This way of integrating the neutron transport equation can be thought of as integrating "sideways." In the Monte Carlo method, the discretized neutrons "shoot through" the transport equation many times, sweeping out the entire phase space, effectively integrating it when a sum is done over the individual neutron realizations, or *histories*. It is important to note that the Monte Carlo Method does not actually solve the transport equation, per se, but rather is able to *estimate* the solution very well.

The Monte Carlo approach makes a physically analogous "experiment" on a computer. A computer thread "sits on top" of a neutron as it makes a *random walk* through the geometry. The computer thread uses pseudo-random numbers to sample probability distributions that describe the interactions the neutron makes as it travels. All the assumptions that went into deriving the neutron transport equation still hold, most importantly that neutrons travel in straight lines between interactions, that neutrons are points, and that they do not interact with each other. The other assumption is that interactions happen instantaneously and at a point, i.e. they do occur over a distance.

How surface crossing is detected in the simulation is very important since this is what changes the material data that specify the probability distributions. WARP, in its current

state, uses the traditional form of surface detection and material updates – ray tracing, the details of which will be discussed in Section 3.1. When a neutron is sampled to cross a surface, it is placed on the boundary, the material data is updated, and the interaction distance is sampled again with the neutron traveling in the same direction. Figure 2.14 shows a cartoon of a random walk of a neutron born in the center of a cube. The line colors represent a neutron sampling a specific material, the red "X" represents an absorption (walk termination), and the green "X" represents a surface crossing.

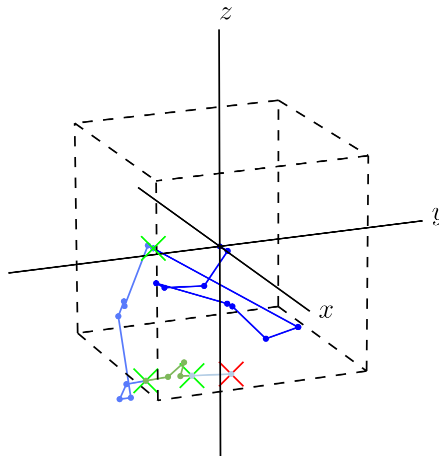


Figure 2.14: The Monte Carlo random walk process.

There are advantages in using the Monte Carlo method for neutron transport, some of which have been mentioned. The first and foremost is that very few assumptions must be made regarding the physics and the geometry of the problem. Paraphrasing Forrest Brown, the only reason people use Monte Carlo methods is that they are able to produce physically accurate results and the methods are trusted to do so. Anyone developing a Monte Carlo code should not abuse this trust and must put accuracy first and foremost in their implementations [37].

There are considerable drawbacks to using Monte Carlo as well; primarily convergence rates and silent inaccuracies. Since it is a statistical calculation, it is bound by statistical laws that dictate that the error in Monte Carlo calculations reduces as $1/\sqrt{N}$, discussed in more detail in the next section, where N is the number of histories performed. The slow convergence rate can be combated by using parallel computing, however. Requiring that neutrons do not interact with each other was an assumption that went into deriving the neutron transport equation, and can be exploited in Monte Carlo simulations. Since each neutron history is completely independent, histories can be run completely in parallel

without communication until the results are combined at the end of the simulation. This leads to very good scaling of parallel calculations, which can be used to reduce run times of large problems to acceptable values.

A hazard associated with the statistical requirements is that very small but very influential volumes within large geometries can be missed by the neutron random walk if the number of histories is not large enough. The estimated statistical error may be low at N histories, but there is a chance the entire phase space has not yet been sufficiently sampled to produce accurate results.

Statistics

To understand how the Monte Carlo Method estimates the solution to the neutron transport equation, the basics of distribution functions and statistics must be outlined. Given a continuous random variable, x, that follows the probability distribution function (PDF), $P(x)$, the mean value can be calculated by taking the first moment as was shown in Eq. (2.21) and is reproduced in Eq. (2.45). PDFs must always be positive, and their integral over all space must equal 1.

However, we are interested in determining the mean from a set samples rather than directly from the underlying distribution. If we have N independent measurements of quantity x, this is simply taking the arithmetic mean, as shown in Eq. (2.46). This value, computed from a finite set of measurements, X, is called the *sample mean*, $\bar{X}$, since it is based on *samples* from the underlying probability distribution $P(x)$. The *true mean* is μ, and is computed directly from $P(x)$.

$$\mu = \int_{-\infty}^{\infty} xP(x)dx \tag{2.45}$$

$$\bar{X} = \frac{1}{N}\sum_{i}^{N} x_i \tag{2.46}$$

The law of large number states that the sample mean converges to the true mean in the limit where $N \to \infty$.

$$Pr\left(\lim_{N\to\infty} \bar{X}_N = \mu\right) = 1 \tag{2.47}$$

The shape of the distribution of samples about the mean is important as well since it quantifies the uncertainty. The variance can be computed by performing the central second moment on P, shown in Eq. (2.48).

$$\sigma^2 = \int_{-\infty}^{\infty} x^2 P(x)dx - \mu^2 \tag{2.48}$$

$$\text{Var}(X) = \frac{1}{N-1}\sum_{i}^{N}(x_i - \bar{X})^2 = \frac{1}{N-1}\left(\sum_{i}^{N} x_i^2 - N\bar{X}^2\right) \tag{2.49}$$

Again, we are interested in determining the variance from a set samples rather than directly from the underlying distribution. The discrete equation for computing the variance of a set of independent measurements, X, is shown in Eq. (2.49). Dividing by $N-1$ instead of N is called Bessel's Correction [13]. It is useful that the sample mean can be removed from the sum of the sample squares since this means that only the sample sum and the sum of sample squares need be stored. The entire sample set does not need to be stored and used to compute the variance at the end when the sample mean is known.

The rate at which the sample mean converges to the true mean, i.e. how the uncertainty scales with the number of samples, comes from the central limit theorem. Shown in Eq. (2.50), the theorem states that the distribution of the means taken from a large set of independent random variables will be normally distributed [13].

$$\sqrt{N}\left(\left(\frac{1}{N}\sum_{i}^{N}x_i\right)-\mu\right)\xrightarrow{d}\mathcal{N}(0,\sigma^2) \tag{2.50}$$

We would like to know how close the mean of such an independent set of measurements will be to the true mean. In other words, the variance of the mean is desired. Luckily, this can be estimated with only one measurement for the mean instead of directly computing the variance of the mean with many measurements of the mean. The Bienaymé formula, shown in Eq. (2.51), states that the variance of a sum of uncorrelated random variables is equal to the sum of the variances of the variables.

$$Var\left(\sum_{i=0}^{N}x_i\right)=\sum_{i=0}^{N}Var(x_i) \tag{2.51}$$

Applying the Bienaymé formula and the variance relation shown in Eq. (2.52) to the sample mean results in an expression for the variance of the mean, $Var(\bar{X}_N)$, in terms of the variance of the sample, σ_N^2. This expression is shown in Eq. (2.54).

$$Var\left(aX\right)=a^2Var\left(X\right) \tag{2.52}$$

$$Var(\bar{X}_i)=Var\left(\frac{1}{N}\sum_{i=0}^{N}x_i\right)=\frac{1}{N^2}\sum_{i=0}^{N}Var(x_i)=\frac{N\sigma_N^2}{N^2}=\frac{\sigma_N^2}{N} \tag{2.53}$$

$$\frac{\sigma_N^2}{N}=\frac{1}{N(N-1)}\left(\sum_{i}^{N}x_i^2-N\bar{X}_N^2\right)=\frac{1}{(N-1)}\left(\frac{1}{N}\sum_{i}^{N}x_i^2-\left(\frac{1}{N}\sum_{i}^{N}x_i\right)\right) \tag{2.54}$$

It should be noted that Eq. (2.53) implies that the standard deviation of the mean always scales as $1/\sqrt{N}$. This is a blessing and a burden in that it means any calculation's variance of the mean will go to zero, i.e. the sample mean will converge to the true mean, as long

it is run long enough and enough samples are collected, but that it will converge as $1/\sqrt{N}$, which is slow [13].

Since the central limit theorem states that the sample mean is normally distributed, we can make use of the well-know properties of the normal distribution, namely the confidence interval. The normal distribution has well-defined confidence intervals: 68% of the population will lie within a single standard deviation, σ, and 95.5% will lie within 2σ [12]. From this knowledge, an expression for the relative error can be written. The expression for the relative error shown in Eq. (2.55) is for the 68% confidence level, and simply needs to be doubled for the 95% level [11].

$$\text{Rel.Err.} = \frac{\sigma}{\bar{X}_N} = \frac{1}{\bar{X}_N}\sqrt{\frac{1}{(N-1)}\left(\frac{1}{N}\sum_{i}^{N} x_i^2 - \bar{X}_N^2\right)} \tag{2.55}$$

Sampling Schemes

In order to actually transport a neutron across the geometry in a Monte Carlo simulation, the probability distributions of the reaction types must be sampled to reproduce accurate distributions when the histories are aggregated. Two methods that are commonly used are the direct inversion method and the rejection method.

The *direct inversion* method relies on being able to analytically integrate the probability distribution function to create a cumulative distribution function (CDF) and then being able to invert that CDF. The first step is to integrate the PDF to find the CDF, as shown in Eq. (2.56).

$$\begin{gathered} \{x \mid x_1 \leq x \leq x_2\} \\ CDF(x) = \int_{x_1}^{x} PDF(x')dx' \end{gathered} \tag{2.56}$$

The PDF is integrated from x_1, the beginning of the PDF domain, to x. x spans the entire domain of the PDF, i.e. that CDF(x_2)−CDF(x_1) = 1. Since a PDF must be positive and normalized by definition, the CDF must range from 0 to 1 and increase monotonically. Setting the CDF equal to a uniformly distributed random number, ξ, and solving for the value to be sampled yields the sampling scheme.

Figure 2.15 shows this graphically. The CDF describes the probability that a value of a random number, ξ, obeying the PDF will be less than or equal to the value x. Taking an integral gives the probability that x will be in the interval, as shown in Eq. (2.57). Since the width of the CDF is directly proportional to the PDF at the same value of x, as shown in Eq. (2.58), and if ξ is uniformly distributed on [0,1], the interval $\Delta\xi$ will contain a fraction of samples on [0,1] equal to $P(x_1 < x_i < x_2)$. As the interval width is reduced to 0, $P(x_i) = PDF(x)dx = d\xi$ (again, since ξ is uniformly distributed on [0,1]), leading to the sampling scheme shown in Eq. (2.59). This method is only useful if the CDF is simple enough to be inverted analytically.

$$CDF(x_2) - CDF(x_1) = P(x_1 < x_i < x_2) = \int_{x_1}^{x_2} PDF(x)dx \tag{2.57}$$

$$CDF(x_2) - CDF(x_1) = \Delta\xi = \int_{x_1}^{x_2} PDF(x)dx \tag{2.58}$$

$$x_i = CDF^{-1}(\xi_i) \tag{2.59}$$

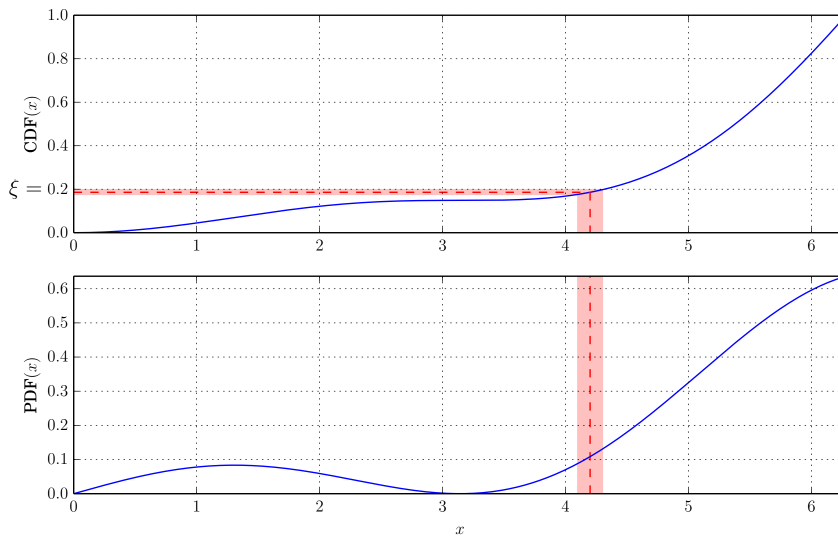


Figure 2.15: Sampling $PDF(x) = \frac{1}{2\pi^2}(x\cos x - x)$ by the direct inversion method on $[0, 2\pi]$.

When the CDF is not invertible, the *rejection sampling* method can be used. It uses a secondary PDF, $f(x)$, that is greater at all points in the primary PDF, $g(x)$ to be sampled from and whose CDF is easily inverted. Since the primary PDF is normalized, $f(x)$ now corresponds to probabilities greater than one. This is simply a scaling problem, and the random numbers used to directly sample from it must be uniform on $[0, \int_{\text{domain}} f(x))$ instead of $[0, 1]$. Then, two random numbers are generated. The first, ξ_1 is sampled from a $f(x)$ using a scaled random number as mentioned, and the second, ξ_2 is uniformly distributed on $[0, f(\xi_1)]$. If $\xi_2 < g(\xi_1)$, the sampled value (ξ_1), is added to the population, else it is *rejected*, or discarded from the population. The set of accepted values of ξ_2 will then follow $g(x)$. Figure 2.16 shows an illustration of rejection sampling. The red shaded area is the space between the primary and auxiliary PDFs where samples are rejected (red dots), and the blue shaded area is where they are accepted (blue dots). In this illustration, the main benefit of choosing a line as the auxiliary function instead of a constant is that a line produces less red shaded area and more samples will be accepted.

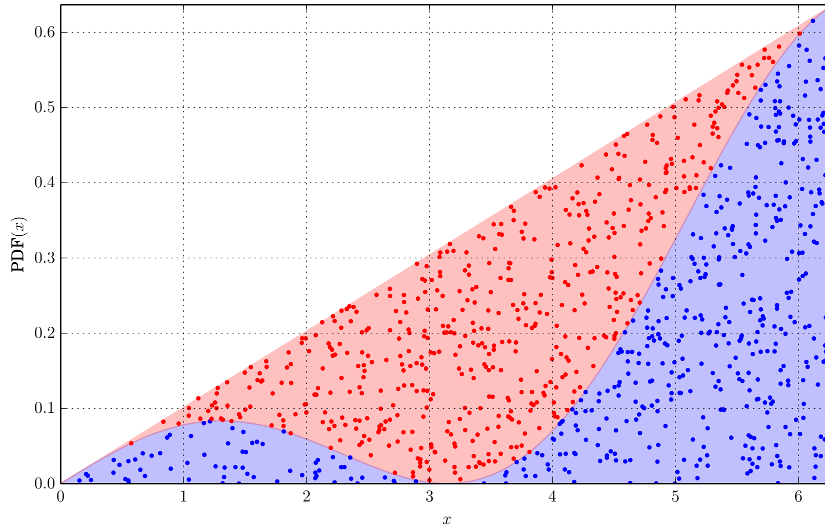


Figure 2.16: Sampling $PDF(x) = \frac{1}{2\pi^2}(x \cos x - x)$ by the rejection method on $[0, 2\pi]$. The auxiliary function is $f(x) = x/\pi$ and the random numbers used to sample from it are uniformly distributed on $[0, 2\pi]$ instead of $[0, 1]$.

Using these two methods of sampling from probability distributions, expressions can be found for every reaction type and transport phenomenon that a neutron undergoes in its random walk through matter. The specific schemes are described in the following subsections.

Cross Section Interpolation

When a Monte Carlo simulation is referred to as "Continuous Energy," it means that the cross sections are evaluated on-the-fly at the exact energy the neutron is currently at instead of using a fixed value for a range of energies. The cross section data is finite, however, and only includes data at discrete points in energy. Therefore, interpolation must be performed between the data points to get values for any specific energy. This is done by linear interpolation. This assumes that the cross section is a straight line between the surrounding energy points, E_i and E_{i+1}, and the interpolated value is calculated using the slope of the line via Eq. (2.60).

$$\begin{gathered} E_i < E < E_{i+1} \\ \sigma(E) = \frac{E - E_i}{E_{i+1} - E_i}(\sigma(E_{i+1}) - \sigma(E_i)) + \sigma(E_i) \end{gathered} \tag{2.60}$$

Distance To Interaction

The first step in stochastically transporting a neutron through matter is finding the distance to the next interaction. The expression for the probability of non-interaction was shown in Eq. (2.2). The probability of interaction in dx around x is $\Sigma_t dx$. Therefore the probability of not interacting from 0 to x and then interacting at dx around x is simply the product of these two probabilities. The resulting expression, shown in Eq. (2.61), is commonly referred to as the *first collision probability*.

$$P(x) = \Sigma_t e^{-\Sigma_t x} \tag{2.61}$$

This expression is simple and can be inverted for direct sampling as seen in Eq. (2.62). The sampling scheme is shown in Eq. (2.63). The expression $1 - \xi$ can be replaced with ξ since it is identically distributed.

$$CDF(x) = \int_0^x \Sigma_t e^{-\Sigma_t x'} dx' = 1 - e^{-\Sigma_t x} = \xi \tag{2.62}$$

$$x_i = CDF^{-1}(\xi) \qquad \Rightarrow \qquad x_i = \frac{-\ln(1-\xi)}{\Sigma_t} = \frac{-\ln(\xi)}{\Sigma_t} \tag{2.63}$$

Isotope and Reaction Selection

Once a neutron has been determined to interact, the isotope with which it interacts must be sampled. The discrete PDF of this process is shown in Eq. (2.64), where there are i isotopes in the material each having a total macroscopic cross section $\Sigma_{t,i}$. The distribution is sampled by generating a uniform random number, ξ_1, on $[0, 1]$ and performing a running sum over the individual isotope's total macroscopic cross sections. When $CDF_i > \xi_1$, the neutron collision is sampled to happen in isotope i.

$$PDF_i = \frac{\Sigma_{t,i}}{\Sigma_t} \qquad \Rightarrow \qquad CDF_i = \frac{1}{\Sigma_t} \sum_{n=1}^{i} \Sigma_{t,n} \tag{2.64}$$

Determining the reaction type is done in a similar fashion, except the number density of the material is no longer a concern since the isotope has already been selected. Therefore, the running sum of the CDF is done over isotope i's microscopic cross sections. Another random number is generated, and when $CDF_k > \xi_2$, the neutron is sampled to undergo reaction k in isotope i.

$$PDF_k = \frac{\sigma_k}{\sigma_t} \qquad \Rightarrow \qquad CDF_k = \frac{1}{\sigma_t} \sum_{n=1}^{k} \sigma_n \tag{2.65}$$

Tabular Distribution Interpolation

For reactions that have exiting neutrons, like scattering and fission, the nuclear data contains tables of CDFs that specify the probabilities for the energies and angles neutrons will be emitted. To sample from these tabular CDFs, a random number, ξ, is generated. Again, since the data is discrete, some type of interpolation must be performed to generate a value between the data points.

There are two interpolation methods specified in the ENDF format. The first is histogram interpolation, which is similar to the linear cross section interpolation shown in the previous subsection. Histogram interpolation assumes the CDF is a line between the data points, and the interpolated value is calculated to lie on this line. Since the CDF is the integral of the PDF, assuming the CDF is a line is the same as assuming the PDF is a constant, i.e. the PDF is a histogram. The data for the histogram could be obtained from using a tabulated PDF that is typically provided to compute the CDF. Using the PDF data is unnecessary, however, since these values can be computed from the CDF and this approach reduces the global memory access of a GPU kernel.

An example of using histogram interpolation to determine outgoing cosine, μ', is shown in Eq. (2.66), where C_i is the value of the CDF at point i and P_i is the value of the PDF at point i. μ'_i represents the corresponding value of μ' in the table at point i [13]. While this example is for μ', this method can be used to sampled a CDF for any quantity.

$$\begin{gathered} C_i < \xi < C_{i+1} \\ \mu'(\xi) = \frac{\xi - C_i}{C_{i+1} - C_i}(\mu'_{i+1} - \mu'_i) + \mu'_i \end{gathered} \tag{2.66}$$

The second method is called linear-linear interpolation. Instead of assuming the PDF is constant between the tabular points (and CDF is linear), it assumes the PDF is linear over the interval $C_i < \xi < C_{i+1}$. The linear expression for the PDF, shown in Eq. (2.67), is substituted into Eq. (2.57) and integrated to μ'. Again this derivation is shown for μ', but is valid for any CDF [13].

$$\begin{gathered} C_i < \xi < C_{i+1} \\ P(\mu') = \frac{P_{i+1} - P_i}{\mu'_{i+1} - \mu'_i}(\mu' - \mu'_i) + P_i = a\mu' + b \end{gathered} \tag{2.67}$$

Integration of Eq. (2.67) to μ' yields Eq. (2.68). This expression for $CDF(\mu')$ is set equal to the random number ξ and solved for μ' to give the interpolation function.

$$C(\mu') = C_i + \int_i^{\mu'} P(\mu'')d\mu'' = C_i + \int_i^{\mu'} (a\mu'' + b)\, d\mu'' = C_i + \frac{a}{2}(\mu'^2 - \mu_i'^2) + b(\mu' - \mu'_i) = \xi \tag{2.68}$$

Eq. (2.68) can be solved for μ', yielding Eq. (2.69), the final form of the linear-linear sampling scheme [13].

$$a = \frac{P_{i+1} - P_i}{\mu'_{i+1} - \mu'_i} \quad , \quad b = P_i - a\mu'_i$$
$$\mu' = \mu_i + \frac{1}{a}\left(\sqrt{P_i^2 + 2a(\xi - C_i)} - P_i\right) \tag{2.69}$$

Each tabular distribution in the ACE data file specifies if the histogram or linear-linear method should be used to perform interpolation on it. It is not up to the user to decide what scheme to use if results consistent with Serpent or MCNP are desired. The CDFs are tabulated with the specified interpolation method in mind, and using the incorrect scheme will produce incorrect results. The interpolation scheme must be checked for each distribution on a case-by-case basis.

Free Gas Treatment

As stated previously, the data in the cross section libraries are Doppler broadened for a certain material temperature. This is done to account for the effect the target nuclei's thermal motion has on the apparent width of resonances when the target nuclides are modeled as stationary. However, a stationary target model would incorrectly allow a neutron to scatter down to absolute zero. In reality, the targets follow a Maxwell-Boltzmann speed distribution, but this cannot be sampled directly for simulation purposes because of the Doppler broadening preprocessing done to the cross section data. Near resonances, portions of the thermal distribution within the resonance contribute much more to the overall reaction rate than the portions outside of it. Directly sampling a target velocity from the thermal distribution would produce incorrect exiting neutron energies. Rather, the distribution of target velocities that contributed to the reaction rate (and therefore the thermally averaged cross section) must be calculated and sampled from.

Elastic scattering is the only interaction that has a neutron at thermal energies in the exit channel, and therefore the only use for the target velocity is in its scattering kinematics. Temperature-preprocessed cross sections are used, however, so sampling the target velocities to reproduce an accurate thermal peak must be done in a way that preserves thermally-averaged reaction rates. For the sake of completeness, the following derivation is included and closely follows that in [13] and [38]. The reaction rate in terms of the target and neutron velocities is shown in Eq. (2.70) and is equivalent to Eq. (2.17) in Section 2.3.

$$R(\boldsymbol{v_t}) = ||\boldsymbol{v_n} - \boldsymbol{v_t}||\sigma(||\boldsymbol{v_n} - \boldsymbol{v_t}||)M(v_t) \tag{2.70}$$

Since elastic scattering is the only reaction of interest, $\sigma(v_{\text{rel}})$ can be assumed to be constant. This assumption is good for most nuclides since the elastic cross sections are in fact relatively constant at low energies. If there is a low-energy scattering resonance, this approximation may produce results that skew the distribution towards lower energies [13], but it is good for most nuclides.

Substituting the Maxwell-Boltzmann speed distribution for $M(v_t)$, the simplified PDF for target velocities can be written as Eq. (2.71) where C is the normalization constant calculated from Eq. (2.70) and m is the mass of the target nucleus.

$$\begin{aligned} C &= \int_0^\infty d\boldsymbol{v_t}\ R(\boldsymbol{v_t}) \\ \beta &= \sqrt{\frac{m}{2kT}} \\ PDF(\boldsymbol{v_t}) &= C v_{\mathrm{rel}} \sigma \left[\frac{4}{\sqrt{\pi}} \beta^3 v_T^2 \exp(-\beta^2 v_T^2) \right] \end{aligned} \tag{2.71}$$

Applying the law of cosines to $v_{\mathrm{rel}} = ||\boldsymbol{v_n} - \boldsymbol{v_t}||$ yields Eq. (2.72), which has changed from a vector equation to a scalar one. The new PDF has two unknowns, the target velocity v_T and the angle between it and the neutron velocity, μ.

$$\begin{aligned} \beta &= \sqrt{\frac{m}{2kT}} \\ PDF(v_t) &= C\sigma \sqrt{v_n^2 + v_t^2 - 2v_n v_t \mu} \left[\frac{4}{\sqrt{\pi}} \beta^3 v_T^2 \exp(-\beta^2 v_T^2) \right] \end{aligned} \tag{2.72}$$

Note that there is a restriction on μ. There cannot be cases where sampled values of μ and v_T produce a negative v_{rel}, which corresponds to the target neutron "running away" from the incident neutron. Such a case is non-physical since it would imply that the reaction never happened. This is the first reason why rejection sampling is needed to determine an appropriate target velocity. The PDF in Eq. (2.72) is sampled and the velocity is rejected if it violates the restriction on μ.

The second reason rejection sampling is used is that the PDF in Eq. (2.72) cannot be directly inverted. However, it can be split into two PDFs, $f_1(v_T)$ and $f_2(v_T)$ as seen in Eq. (2.73), which can be inverted. Sampling from the product of two invertible functions is possible by rejection. The sampling scheme for this is to normalize $f_2(v_T)$ (so its integral is one) and sample from it directly. The sample from $f_2(v_T)$ is accepted based on the probability in Eq. (2.74) [13]. The multiplication and division by $v_n^2 + v_t^2$ seen in Eq. (2.73) guarantees that f_1 is bounded between 0 and 1. This is necessary since v_{rel} is not bounded in any way. The normalization using f_2 to make "$CDF(v_T)$" is expanded and simplified in Eq. (2.76), where $y = \beta v_n$ and $x = \beta v_t$. The acceptance probability of the sample is shown in Eq. (2.75)

$$\begin{aligned} f_1(v_T, \mu) &= C\sigma \frac{\sqrt{v_n^2 + v_t^2 - 2v_n v_t \mu}}{v_n^2 + v_t^2} \\ f_2(v_T) &= \frac{4}{\sqrt{\pi}} (v_n^2 + v_t^2) \beta^3 v_T^2 \exp(-\beta^2 v_T^2) \end{aligned} \tag{2.73}$$

$$\text{“}CDF(v_T)\text{”} = \frac{f_2(v_T)}{\int f_2(v_T)} \qquad \Rightarrow \qquad v_T' \tag{2.74}$$

$$p_{\text{accept}} = \frac{f_1(v'_T, \mu)}{f_1(v'_T, \mu = 0)} \tag{2.75}$$

$$\begin{aligned} \text{“}CDF(v_T)\text{”} &= \frac{f_2(v_T)}{\int f_2(v_T)} = \frac{(v_n^2 + v_t^2)\beta^3 v_T^2 \exp(-\beta^2 v_T^2)}{\frac{1}{4\beta}\sqrt{\pi}\beta v_n + 2} \\ &= \left(\frac{4}{\sqrt{\pi}} \frac{\sqrt{\pi}y}{\sqrt{\pi}y + 2}\right) x^2 \exp(-x^2) + \left(2\frac{2}{\sqrt{\pi}y + 2}\right) x^3 \exp(-x^2) \end{aligned} \tag{2.76}$$

Since this CDF is the weighted sum of two independent distributions, the combined PDF can be sampled by sampling the second term with a probability of $2/(\sqrt{\pi}y + 2)$ and the first term otherwise. While both distributions appear complicated, each can be directly sampled via the schemes C49 and C61 provided in the Third Monte Carlo Sampler manual [39]. Sampling scheme C49, shown in Eq. (2.77), is for sampling distributions of the form $\nu^{2n-1}e^{-\nu^2}$ where n is an integer ≥ 1 and ν is a continuous random variable, i.e. for Gaussians multiplied by variables of odd powers. This is scheme is appropriate for sampling a value of x' from the second term in Eq. (2.76) when $n = 2$, and is shown in Eq. (2.78).

$$\nu = \sqrt{-\ln\left(\prod_{i=1}^{n} \xi_i\right)} \tag{2.77}$$

$$x' = \sqrt{-\ln(\xi_1 \xi_2)} \tag{2.78}$$

Sampling scheme C61, shown in Eq. (2.79), is for expressions of the for $\nu^{2n-1}e^{-\nu^2}$ where n is a half integer $\geq 1/2$, i.e. for Gaussians multiplied by variables of even powers. When $n = 3/2$, this is a sampling scheme for the first term in Eq. (2.76), and is shown in Eq. (2.80).

$$\begin{aligned} h &= n - 1/2 \\ \nu &= \sqrt{-\ln\left(\prod_{i=1}^{h} \xi_i\right) + \tau^2} \\ \tau^2 &= -\ln(\xi_1)\cos^2\left(\frac{\pi}{2}\xi_2\right) \end{aligned} \tag{2.79}$$

$$x' = \sqrt{-\ln(\xi_1) - \ln(\xi_2)\cos^2\left(\frac{\pi}{2}\xi_3\right)} \tag{2.80}$$

After the first or second term of Eq. (2.76) is sampled (RHS sampled with probability $2/(\sqrt{\pi}y + 2)$) to give x', it can be transformed to v'_T via $v'_T = x'/\beta$. Once a value for v'_T is determined, it must be accepted or rejected based on the angular constraint imposed by the rejection criterion Eq. (2.75). The rejection criterion requires a value for μ, however. Since this is for a thermal distribution, μ is sampled isotropically as shown in Eq. (2.81). Once

μ is sampled, $f_1(v_T, \mu)$ can be calculated, and the sampled values of v'_T and μ are accepted with the probability p_{accept} as shown in Eq. (2.82), i.e. the values are accepted if random number ξ satisfies $\xi < p_{\text{accept}}$. If the sample is rejected, the whole process is repeated until a pair of v'_T and μ are accepted.

$$\mu = 2\xi - 1 \tag{2.81}$$

$$p_{\text{accept}} = \frac{\sqrt{v_n^2 + v_t^2 - 2v_n v_t \mu}}{v_n^2 + v_t^2} \tag{2.82}$$

Even though this is a rejection method, it is fairly efficient, with samples being accepted 68% to 100% of the time as the neutron velocity goes from zero to values much greater than the target velocity [11].

Stochastic Mixing

The ENDF data tables have probability distributions for reactions at discrete energy points, but this is not physically accurate. The reactions do not abruptly transition from one PDF to another at a single energy, they smoothly transition. This is why *stochastic mixing* is prescribed in the ENDF tables. If the neutron energy, E, falls between the ENDF data energy grid values i and $i+1$, the neutron will use data from table $i+1$ with probability f, defined in Eq. (2.83) [13].

$$\begin{aligned} E_i < E < E_{i+1} \\ f = \frac{E - E_i}{E_{i+1} - E_i} \end{aligned} \tag{2.83}$$

As E approaches E_{i+1}, the probability it will sample from the $i+1$ distribution goes to 1 linearly. This ensures smooth transitions between reaction data tables.

Elastic Scattering

The kinematics of elastic scattering were outlined in Section 2.2, and even though it seems only the velocities and masses of the target and neutron are needed, data tables still need to be queried to carry out the interaction. At energies below the MeV range, elastic scattering is isotropic in the CM frame, but at these energies and above, it becomes significantly anisotropic. Figure 2.17 shows this in ^{16}O. At high energies, the scattering is both forward- and backward-peaked.

It is essential to model this phenomena accurately since the scattering angle factors heavily into the energy exchange between the particles. Figure 2.18 shows what can happen when elastic scattering is treated as isotropic for all energies. The figure shows the normalized flux per unit lethargy in a one meter cube of water with a 2 MeV point source at its center as calculated by WARP (red) and by Serpent (blue). The neutron flux spectra at scattering

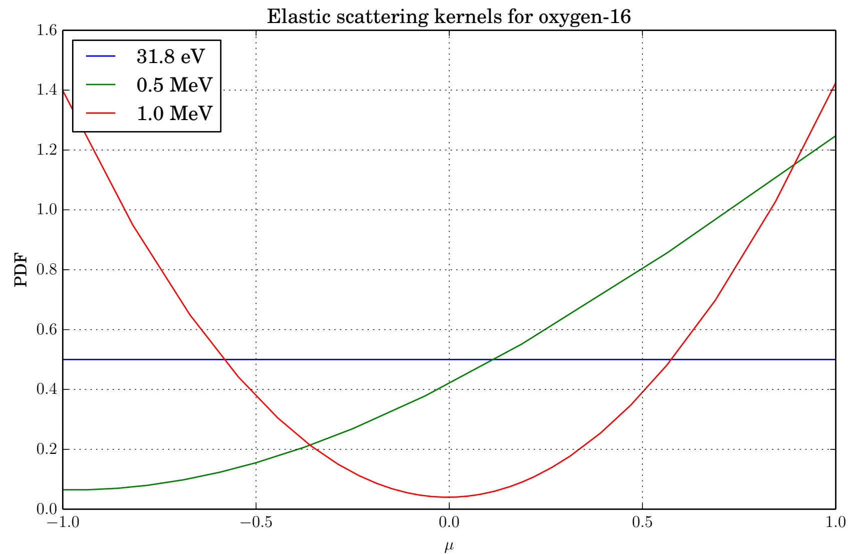


Figure 2.17: Elastic scattering anisotropy at high energies in ^{16}O.

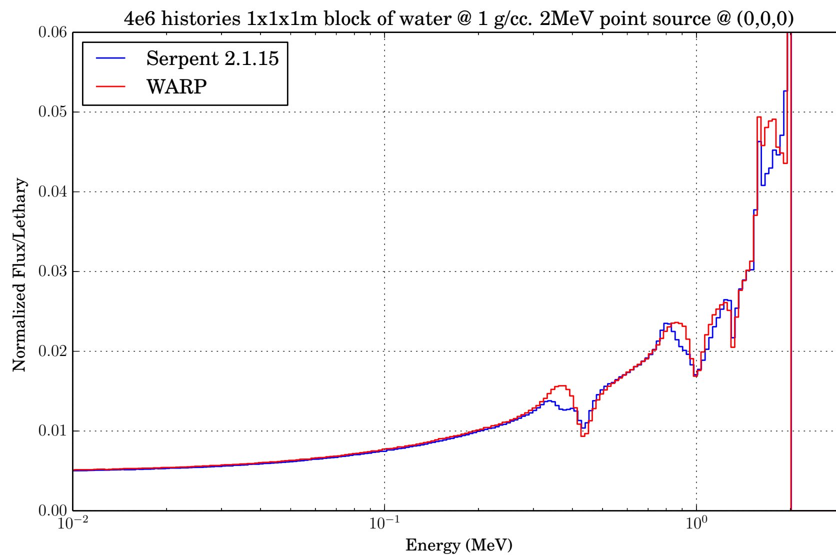


Figure 2.18: An inaccurate flux spectrum produced by WARP compared to a correct one from Serpent highlighting the errors caused by treating elastic scattering as isotropic for all energies.

resonances (which are apparent at 0.32, 1.0, and 1.2 MeV from the sharp dips in the flux) is too small, and the flux immediately below the resonances is too large.

The anisotropy shown in 2.17 at the 1 MeV resonance indicates that a neutron should

either lose a little (forward-peaked elastic scattering) or a large (backward-peaked elastic scattering) amount of energy in the elastic scattering event. It should not lose an intermediate amount. The minimum fraction of energy than can be lost to ^{16}O is zero and the maximum is $1-((16+1)/(16-1))^2 = 0.22$ [3]. The errors shown in Figure 2.18 typically span 2 octaves in the logarithmic flux plots, which is consistent with the maximum factional energy loss. In the completely isotropic flux, the higher flux at intermediate energies indicates that neutrons are being scattered to these energies at a higher rate than Serpent. It is low immediately on the resonance because neutrons are leaving the energy bin faster than they should be and are not scattering multiple times within the bin (as they would if they lost no energy in the elastic scattering event).

The point of highlighting this phenomenon is that sampling angular dependence of the scattering kernels is very important for obtaining correct results, especially for elastic scattering where the energy exchanges are completely defined by the scattering angle and the target momentum. The energy dependencies cannot be ignored and the distributions in the nuclear data must be sampled. Elastic scattering is very common (and often dominant) reaction and modeling it accurately is essential to obtaining correct results. WARP uses the tabular CDF format to sample the outgoing CM angle of the neutron in elastic scattering, which normally uses histogram interpolation. The process is as follows:

1. The neutron energy and direction are converted to velocity.
2. The target velocity is sampled as described in the previous subsection.
3. The center of mass velocity is calculated and the neutron and target velocities are transformed to the CM frame.
4. The polar scattering angle is sampled from the appropriate angular distribution for incoming neutron energy and the target isotope.
5. The azimuthal scattering angle is sampled isotropically in 2π.
6. The neutron velocity is rotated from its initial CM direction through the sampled polar and azimuthal angles according to Eq. (2.15).
7. The neutron velocity is transformed back to the Lab frame and its new energy and direction are calculated.

Tabular Energy Distributions

Continuous, independent energy distributions, like those specified for fission spectra, have different CDFs for various incoming neutron energies. Stochastic mixing probability, f, shown in Eq. (2.83), is used to select if the lower-bounding or upper-bounding distribution will be sampled from for $E_i < E < E_{i+1}$. After the distribution is selected, the CDF is sampled with histogram or linear-linear interpolation, whichever is specified in the library.

Once the sampled energy, E_{sampled}, is calculated, it must be scaled to the bounding incoming energy bins to preserve any thresholds. This is done via Eq. (2.84) where $E_{i,\text{first}}$ and $E_{i,\text{last}}$ are the first and last energy values of the CDF in E_i; and $E_{i+1,\text{first}}$ and $E_{i+1,\text{last}}$ are the first and last energy values of the CDF in E_{i+1} [11].

$$\begin{aligned}
& E_a = E_{i,\text{first}} + f(E_{i+1,\text{first}} - E_{i,\text{first}}) \\
& E_b = E_{i,\text{last}} + f(E_{i+1,\text{last}} - E_{i,\text{last}}) \\
\text{if i + 1 sampled}: \quad & \text{diff} = E_{i+1,\text{last}} - E_{i+1,\text{first}} \qquad E_{\text{start}} = E_{i+1,\text{first}} \\
\text{else}: \quad & \text{diff} = E_{i,\text{last}} - E_{i,\text{first}} \qquad E_{\text{start}} = E_{i,\text{first}} \\
& E' = E_a + (E_{\text{sampled}} - E_{\text{start}}) \frac{E_b - E_a}{\text{diff}}
\end{aligned} \tag{2.84}$$

Inelastic Reactions

Inelastic level scattering is treated identically to elastic scattering except that the Q value in Eq. (2.14) is now nonzero. This value is reaction-specific and must be obtained from the data tables.

Inelastic continuum scattering is treated differently, however. Instead of having a well-defined kinematic formula relating the scattering angle to the final neutron energy, continuum reactions are defined by correlated scattering and energy tables. They follow ENDF "law" number 44, the Kalbach-Mann correlated energy-angle scattering law [11] [13]. In this sampling scheme, a CDF is histogram or linear-linear interpolated and sampled from just like other reactions, except the sampled value is now a set of "precompound factors," R, and "angular slopes," A, instead of μ directly. These values correspond to a secondary PDF, shown in Eq. (2.85), which is again sampled from to find μ.

$$PDF = \frac{A}{2\sinh(A)}(\cosh(A\mu) + R\sinh(A\mu)) \tag{2.85}$$

The sampling scheme for this probability distribution shown in Eq. (2.86), where ξ_1 and ξ_2 are two random numbers. Since this is a correlated angle-energy distribution, there is an energy value along with the factors A and R that corresponds to a CDF bin. This sampled energy value is scaled via Eq. (2.84) just like those from a tabular energy distribution, which is the topic of the next subsection [39][13].

$$\begin{aligned}
\text{if } \xi_1 > R: \quad & T = (2\xi_2 - 1)\sinh(A) \\
& \mu = \frac{\ln(T + \sqrt{T^2+1})}{A} \\
\text{else}: \quad & \\
& \mu = \frac{\ln(\xi_2 \exp(A) + (1-\xi_2)\exp(-A))}{A}
\end{aligned} \tag{2.86}$$

Flux Estimation and Tallies

MCNP uses the word "tally" and Serpent uses the word "detector" for the same measurement [31, 11]. Evidently, "tally" does not translate into Finnish, which is why "detector" is used. "Tally" will be used here for terminological consistency with the majority of neutron transport codes.

Other than determining the multiplication factor, one of the main purposes of simulation reactors is determining the reactions rates in the core. The reaction rates and, therefore, power of the reactor are proportional to the flux, making the flux another very important quantity in determining how a rector will behave. Since reactions are explicitly simulated with the Monte Carlo method, a collision estimator can be used to estimate the neutron flux. A collision estimator does not directly calculate the flux, but relies on the fact that the total collision rate is the flux multiplied by the total macroscopic cross section, as was shown in Eq. (2.24). If every collision within a volume is counted in a Monte Carlo simulation, this corresponds directly to the total reaction rate density integrated over the volume and can be used to estimate the volume-averaged flux in that volume per Eq. (2.29). Once the reaction rate is known, the flux can be written in terms of it and the material's total macroscopic cross section as shown in Eq. (2.87).

$$\phi(\vec{\boldsymbol{r}}, E) = \frac{R_t(\vec{\boldsymbol{r}}, E)}{\Sigma_t(\vec{\boldsymbol{r}}, E)} \tag{2.87}$$

The reaction rate is simply the number of collisions at energy E, and Eq. (2.87) can be rewritten as a sum over all the collisions that happen at energy E, as shown in Eq. (2.88).

$$\phi(\vec{\boldsymbol{r}}, E) = \sum_{i}^{R_t(\vec{\boldsymbol{r}},E)} \frac{1}{\Sigma_t(\vec{\boldsymbol{r}}, E)} \tag{2.88}$$

To calculate the flux a computer, a discrete set of energy bins, $E_1 \rightarrow E_g$, must be specified. The average flux in group g, such that $E_g < E < E_{g+1}$, and in volume j is shown in Eq. (2.89).

$$\bar{\phi}_{g,j} = \frac{1}{V_j} \int_{V_j} dV \int_{E_g}^{E_{g+1}} dE \, \phi(\vec{\boldsymbol{r}}, E) = \frac{1}{V_j} \int_{V_j} dV \int_{E_g}^{E_{g+1}} dE \sum_{i}^{R_t(\vec{\boldsymbol{r}},E)} \frac{1}{\Sigma_t(\vec{\boldsymbol{r}}, E)} \tag{2.89}$$

The integrals in Eq. (2.89) are carried out over $R_t(\vec{\boldsymbol{r}}, E)$ as shown in Eq. (2.90), and the reaction rate is relabeled for clarity as $N_{g,j}$, the number of collisions in the energy group g and volume j.

$$\int_{V_j} dV \int_{E_g}^{E_{g+1}} dE \, R_t(\vec{\boldsymbol{r}}, E) = R_{g,j} = N_{g,j} \tag{2.90}$$

The average flux estimator can now be written as Eq. (2.91), where $\Sigma_{t,j}(E)$ is the total macroscopic cross section of the material in volume j at energy $E_g < E < E_{g+1}$.

$$
\begin{aligned}
E_g < E < E_{g+1} \\
\bar{\phi}_{g,j} = \frac{1}{V_j} \sum_{i=1}^{N_{g,j}} \frac{1}{\Sigma_{t,j}(E)}
\end{aligned}
\tag{2.91}
$$

The expression shown in Eq. (2.91) is a collision estimator, but it is also technically an *analog estimator* of the average flux. It is analog in the sense that the reactions are counted directly. Since every collision is proportional to the flux, counting every collision gives an estimate of the flux itself. Using an analog estimator for specific reactions, however, means they are only scored when the reaction is sampled to happen. If a reaction has a very small cross section, the reaction will not happen very often and its estimated reaction rate will have a high statistical uncertainly even though the flux may be well resolved. In an extreme case, the reaction may never happen and an analog estimate of its reaction rate would be impossible to make.

A collision estimator can be used for estimating specific reaction rates as well as the flux. For specific reaction rates, "collision" means they are scored for every collision, not just ones where the reaction is sampled to happen. This total score is simply weighted by the interaction probability for a particular reaction as shown in Eq. (2.92) [12], where $P_{k,m}(E_i)$ is the fractional interaction probability for reaction k in isotope m in the volume j and the energy group g.

$$
\begin{aligned}
E_g < E < E_{g+1} \\
P_{k,m}(E) = \frac{\Sigma_{k,m}(E)}{\Sigma_t(E)} \\
R_{g,j,k,m} = \frac{1}{V_j} \sum_{i=1}^{N_{g,j}} \frac{P_{k,m}(E)}{\Sigma_t(E)}
\end{aligned}
\tag{2.92}
$$

Neutron Sources

So far, all that has been mentioned is how to handle the different reactions in a Monte Carlo neutron transport simulation. This is important since reactions describe what happens to neutrons and where they go during their random walk, but how to handle where neutrons *come from* still needs to be defined. The source terms on the right hand side of the neutron transport equation, other than the scattering source, have not been defined yet. In this subsection, the other two source terms, the external source and the fission source, are discussed.

External Source

An external, or fixed, source is simply a source that does not depend on the neutron population itself. Physically, such a source could be from natural radioactive decay, an accelerator source, etc. This type of source is simpler than a fission source since the source is completely independent of the system response and the source distribution does not need to be converged before tallies are accumulated. Transporting neutrons in materials that produce no secondary neutrons is straight-forward – the source neutrons are initialized according to the specified source definition and are transported until they leak out of the system or are absorbed. Transport becomes more complicated when there are isotopes present than can produce secondary neutrons. If a single neutron is “shot” into a multiplying material, all the secondary particles the primary neutron induces also need to be simulated in order to simulate the response of the system.

Materials that produce secondary neutrons are called “multiplying” materials since the total number of neutrons needed to be transported is a multiple of the source number. The number of secondary neutrons produced is fully defined by the multiplication factor since it gives the ratio of subsequent neutron generations. An expression for the number of neutrons in an infinite series based on k is shown in Eq. (2.93), where N_0 is the number of primary neutrons [3][12]. WARP gives every neutron the same weight, so all secondary neutrons are transported. There is no differentiation between the importance of a primary neutron and one produced by five cycles of multiplication.

$$N_{\text{total}} = N_0 + k_{\text{eff}} N_0 + k_{\text{eff}}^2 N_0 + \cdots = N_0 \sum_{i=0}^{\infty} k_{\text{eff}}^i = \frac{N_0}{1-k} \tag{2.93}$$

This series only converges for $k_{\text{eff}} < 1$, which sets a restriction for fixed-source simulations. If the multiplication factor is greater than one it will require infinitely many secondary neutrons to be tracked, so this situation must be avoided. The algorithmic details of how secondary neutrons are transported will be discussed in Section 3.2.

Fission Source, k-Eigenvalue Method

When a fission source is specified, the simulation will run in a manner different from an external source problem. The neutron source now depends on the neutron population itself, and a method must be used that directly ties the source to the population. This is done by using any points where neutrons experience secondary-producing reactions as source points for future neutron histories.

When neutrons are born from these induced reactions, they must follow the emission laws specified by the data. For fission reactions, there is usually a tabulated energy spectrum, which is sampled and interpolated in the data-specified ways, and then scaled to the incoming energy bins via Eq. (2.84). The angle is isotropic in the Lab frame, and this is easily sampled via Eq. (2.94).

$$\begin{aligned} \phi &= 2\pi\xi_1 \\ \mu = \cos\theta &= 2\xi_2 - 1 \end{aligned} \tag{2.94}$$

An important result of criticality simulations is the effective multiplication factor, k_{eff}. It is defined by Eq. (2.95), where $N_{s,n}$ is the number of source neutrons in the current generation, and $N_{s,n+1}$ is the number of neutrons in the next generation [12]. WARP calculates $N_{s,n+1}$ as the sum of the yield of secondary particles from both fission and (n,2/3/4n) reactions. Calculating this quantity in a Monte Carlo simulation is straightforward, but neutron generation information must be preserved. Correct results cannot be obtained if yields from source particles in different generation are used to calculate the number of sources in the next generation. This is why criticality source simulations are usually run in a batched mode where a pre-set number of source particles from a single generation are all transported until termination. The secondary-producing reaction points are then used as the starting points for the next generation. The generations are not interleaved, and therefore correct values for the multiplication factor can be calculated. A GPU could be kept busier if generations could be interleaved, since processors would not have to wait for all neutrons in a current batch are complete before starting the next batch.

$$k_{\text{eff},n} = \frac{N_{s,n+1}}{N_{s,n}} = \frac{N_{f,n}}{N_{s,n}} \tag{2.95}$$

Modeling the fission chain reaction is simple in Monte Carlo simulations when a system is exactly critical. In batched criticality mode, the next generation of neutrons has starting points that are determined by the previous generation. When the system is exactly critical, there is a one-to-one mapping of induced secondary neutrons to a pre-set number of source particles that will be transported in the next batch. Each batch requires a fixed number of source neutrons to be transported, but when a batch yields a number of fission neutrons not equal to this fixed number, some method must be used to handle the difference and adjust the source to the prescribed number of neutrons. For example, if a simulation is run with cycles consisting of 1×10^4 source neutrons and the system has a multiplication factor of 0.73 (which is not known at this point in the simulation), the transport cycle will only yield $7,300$ fission neutrons. If these sites are to be used as the source points for the next cycle, a method is needed to initialize the remaining 2,700 neutrons required to start the cycle.

When the system is sub- or super-critical, there is no longer a one-to-one mapping and starting points either have to be reused or discarded, respectively. This is done by using the *k-eigenvalue method*, which is used by both Serpent and MCNP [12, 11].After k_{eff} is calculated via Eq. (2.95), it is used to renormalize the fission source of the next generation, i.e. the secondary yield values are divided by k_{eff}. This is an analog to dividing the fission source term in Eq. (2.44) by k_{eff} to enforce a time derivative of zero. By dividing the yield values by k_{eff}, the multiplication factor should be 1, and a one-to-one mapping is recovered. The spatial and energy distributions of the fission sites are unchanged, but the *number* of fission sites to start the next cycle is now appropriate.

It should be noted that WARP treats any secondary-producing reactions other than scattering, like (n,2n) reactions, as neutron sources. MCNP calculates k_{eff} by treating these kinds of reactions as negative absorptions, i.e. MCNP subtracts an absorption out of the global tally instead of adding it to a fission yield tally for an (n,2n) reaction [11]. The expression for k_{eff} that MCNP uses is shown in Eq. (2.96), and the expression that WARP uses is shown in Eq. (2.97). Normally, reactions other than fission are rare, but this scoring difference may be a source of disagreement when results are compared on a sub-10^{-2} level as they are with multiplication factors.

$$k_{\text{eff},n} = \frac{\int_V dV \int_0^\infty dE \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}\ \nu\Sigma_f\phi}{\int_V dV \int_0^\infty dE \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}\ \nabla\cdot\vec{\boldsymbol{J}} + \int_V dV \int_0^\infty dE \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}\ (\Sigma_c + \Sigma_f - \Sigma_{n,2n} - 2\Sigma_{n,3n})\phi} \tag{2.96}$$

$$k_{\text{eff},n} = \frac{\int_V dV \int_0^\infty dE \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}\ (\nu\Sigma_f + \Sigma_{n,2n} + 2\Sigma_{n,3n})\phi}{\int_V dV \int_0^\infty dE \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}\ \nabla\cdot\vec{\boldsymbol{J}} + \int_V dV \int_0^\infty dE \int_{\hat{\boldsymbol{\Omega}}} d\boldsymbol{\Omega}\ (\Sigma_c + \Sigma_f + \Sigma_{n,2n} + \Sigma_{n,3n})\phi} \tag{2.97}$$

Neutrons are discrete particles, and individual neutron yields of the fission events are integers. Using the k-eigenvalue method to adjust the neutron yields is done by dividing the integer yield values by k_{eff}, which is a continuous number. This division yields another continuous number, but yields must be integers. There is no way to initialize 2.76 neutrons in WARP since all neutrons have the same weight. Since k_{eff} is continuous, calculating an integer value of secondary particles is done stochastically to preserve the aggregate mean. The renormalized yield, y_r, is stochastically rounded based on a random number, ξ, as shown in Eq. (2.98). This ensures that, over a large batch size, the multiplication factor is renormalized to be as close to one as possible. In the case where y_r is slightly less than one, the last few source points from the previous generation are simply reused. If the number is slightly larger, the last few source points from the previous generation are simply discarded. This should not introduce much bias in the results if the source distribution is converged.

$$\begin{aligned} y_r &= \frac{\text{yield}}{k_{\text{eff}}} \\ \text{if} \quad & y_r - \text{floor}(y_r) < \xi \\ \text{then}: \quad & y_r = \text{ceil}(y_r) \\ \text{else}: \quad & y_r = \text{floor}(y_r) \end{aligned} \tag{2.98}$$

To start a criticality simulation, a guess for the initial neutron source points is required. As the simulation progresses, the source expands and converges to the true distribution. The initial batches, or cycles, in which the source distribution is likely far from the correct distribution are therefore discarded. No quantities are accumulated, and the multiplication

factors values are not used in calculating the final value. The discarded cycles only serve to converge the fission source so the accumulated quantities later in the simulation are not biased by an incorrect source distribution.

One way to specify the initial source distribution is by choosing a single initial point where all of the first generation neutrons are born; this is called the *point source* method. To accelerate the source convergence process, WARP uses a method similar to that used by Serpent to guess the initial source distribution. The materials in the geometry are all flagged as fissile or non-fissile. Then a uniform, random distribution of source neutrons are distributed across the geometry. If a particle lies in a fissile material, it is recorded in a buffer. This process is repeated until the required amount of source points are accumulated in the starting point buffer. This method is called the *flat source approximation*. Since the initial distribution has spatial extent, fewer discarded cycles will be needed to push neutrons into all geometrical regions than in the point source method [12].

When all the cycles are completed, the individual estimates of k_{eff} are averaged to make a final estimate for the system. This is typically called the *generation estimate* of k_{eff} [12]. As the simulation runs, a recursion relation can be made so the values for every cycle do not need to be stored individually. Expressions for the generation estimate are shown in Eq. (2.99). It may be of interest to keep values for every cycle in order to perform statistical checks to assure convergence and prevent roundoff error, however.

$$
\begin{aligned}
\bar{k}_{\text{eff},n} &= \frac{1}{n}\sum_{i=1}^{n} k_{\text{eff},i} \\
\bar{k}_{\text{eff},n} &= \frac{1}{n}\left[k_{\text{eff},n} + (n-1)\bar{k}_{\text{eff},n-1}\right]
\end{aligned}
\tag{2.99}
$$

2.7 GPUs

Now that the mathematical theory and simulation methods have been framed, the hardware will be discussed. As mentioned in the introduction, GPUs are an emerging technology in supercomputing. Their name, graphics processing units, tells their history. They started as specific-use coprocessor cards on computers in the early 1990s. These cards did one job: process graphics to be displayed on a monitor. This is a work-intensive job that could be offloaded from the main CPU to the graphics card, freeing up CPU resources and improving the overall performance of the computer. Most graphics computations require linear algebra operations on large datasets (projections, transforms, shading, etc.), so GPUs were tailored to do these jobs very well and were not able to do much else. GPUs were not programmable, APIs had to be used to send data to them, and the subtleties of GPU execution were abstracted away from the programmer. This was not a problem since the GPU/CPU system was balanced. The CPU did the complicated jobs and the GPU took care of the large but simple ones.

Since the turn of the century, CPU speeds have plateaued because of power density. Power dissipation in a processor goes as $P = fCV^2$, where f is frequency, C is capacitance, and V is voltage. There is a minimum voltage needed to avoid thermally-induced errors, and the capacitance is related to the process size of the chip. There is a maximum for power dissipation of a processor, or power density, when discussing a single core. At some point, it becomes impossible to remove enough heat from the chip without it becoming very unstable from thermal noise, or in the worst case, before it becomes impossible to prevent it from melting. This relation sets a maximum frequency of the processor and is the reason why overclockers must use exotic cooling methods to get CPU frequencies high. This ceiling was been reached, but CPU manufacturers continued to increase performance by including multiple processor cores on their chips. Moore's Law has resultantly stayed in effect, and the number of transistors on a chip is doubling every 18 or so months [40]. These transistors are in the form of additional cores or parallel resources, and chips are becoming wider, not faster. This spreads out the heat created in the chips and it is possible to dissipate enough heat while still adding more transistors. Thus, the overall power dissipation increases but the power density does not.

GPUs had been becoming wider, not faster, for many years prior to the introduction of a multi-core CPU. Once CPUs started widening, the similarities between CPU tasks and GPU tasks became blurred. Previously, the CPU handled complex tasks and the GPU handled simple but large tasks. With the introduction of multicore, CPU programs needed to implement some kind of parallelism in order to gain performance instead of relying on CPUs becoming faster. This trend led GPU manufacturers to ask the question, why not make GPUs programmable? Supercomputing had become massively parallel, and making supercomputers was a lucrative business. GPU manufacturers had been making high performance parallel processors for many years, and it was time to break into the supercomputing realm.

To break into the supercomputing market, manufacturers crated the first programmable, general purpose GPU, which focused on power efficiency and parallelism. As figure 2.19 shows, the maximum theoretical computational capacity of NVIDA GPUs has been increasing faster than that of CPUs [10]. The single-precision performance is much better than the double-precision, however. The traditional role of GPUs as graphics accelerators did not require double precision capabilities, which is why single precision performance was emphasized during their development. In recent years, double precision arithmetic is supported as well, but gaming cards in the GeForce series have much poorer performance than the high-end Tesla cards. The Tesla cards have substantially more double precision units than the GeForce cards, which is why GeForce double precision is not shown [41]. As Figure 2.19 also shows, CPUs have nearly equivalent single and double precision performance.

The GPU executes differently than a CPU and is a separate piece of hardware. To use GPUs for programmable computing, there needed to be an interface between the GPU and CPU and an easy way to program them, so CUDA was created. CUDA stands for "Compute Unified Device Architecture," and is NVIDIA's parallel computing platform [10]. The next subsections will talk about the details of GPU execution and how CUDA allows them to be programmed.

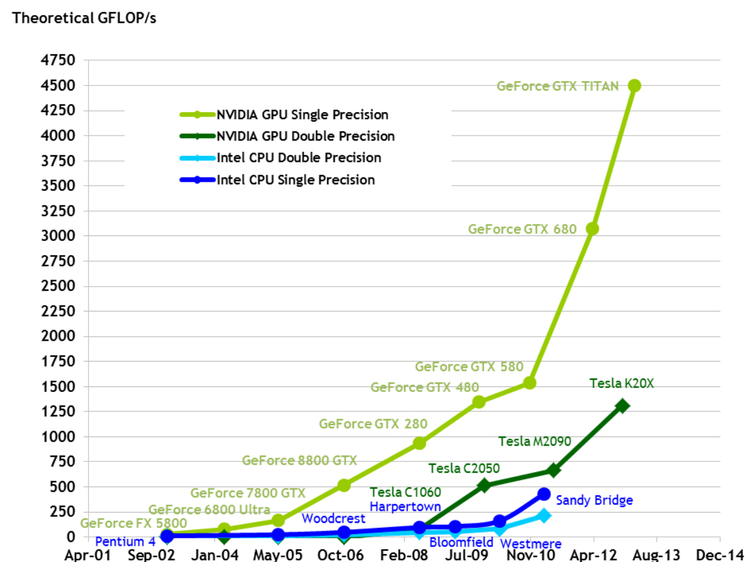


Figure 2.19: The maximum theoretical GigaFLOPs of various NVIDIA GPUs vs. flagship Intel processors [10].

Architecture

An NVIDIA GPU's basic processing unit is called a multiprocessor. These multiprocessors house many individual computational cores and can process jobs independently of each other. Figure 2.20 shows the architecture of a Fermi-family NVIDIA GPU. They are often called "streaming multiprocessors," or SMs, as well.

Part of the reason GPUs are able to perform efficiently is because they rely on single instruction, multiple data (SIMD) execution. This execution method uses the same instructions simultaneously carried out over multiple pieces of data. This reduces the amount of power used in control and therefore more math can be done per watt, which is the main reason why they are being used in supercomputers [14]. There are some tradeoffs for this power efficiency, however, such as requiring relatively simple tasks because of limited cache and control space and requiring data parallelism for full utilization.

The GPU programming model abstracts SIMD execution by using threads, which can be thought of in the traditional sense. Single-instruction multiple-thread (SIMT) and SIMD are similar in the sense that identical instructions are carried out across different pieces of data, but SIMT allows threads to act independently, albeit with a performance penalty. When a thread in a multiprocessor executes a different instruction than the other resident threads, the multiprocessor masks it from execution, executes the identically-executing threads, then masks the identically-executing threads and executes the *divergent* thread. This effectively serializes operations if they require different instructions. Masking and serializing makes many empty spaces in the SIMD lanes of the multiprocessor, and thus can cause severe performance penalties. The magnitude of the penalty depends on the rate at which data is

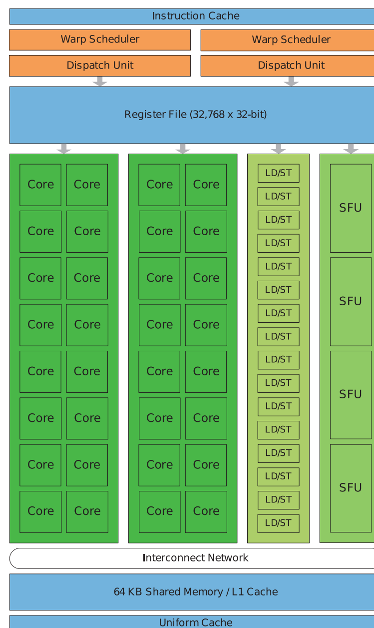


Figure 2.20: The architecture of a Fermi multiprocessor [10].

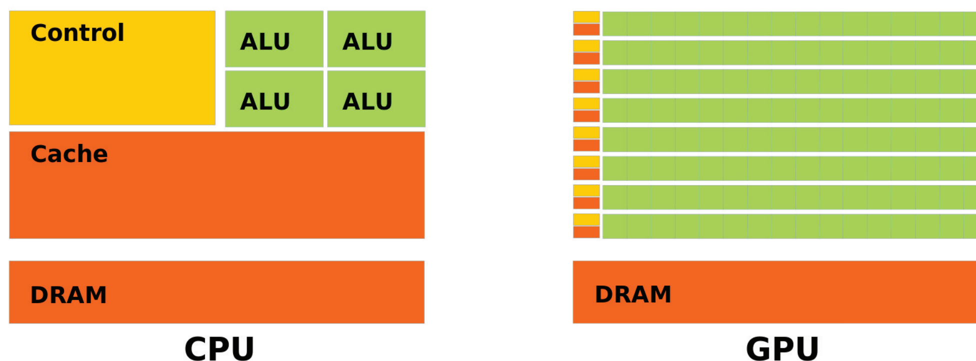


Figure 2.21: Relative transistor space use in CPUs and GPUs [10].

being delivered to the multiprocessor since work can only be done if data is present. If the multiprocessor can completes all jobs, even if it needs some serialization, before the next set of data arrives, then the penalty will be invisible [10].

Another benefit of using SIMD is that less transistor space is needed for control and storing instructions, and this frees up more transistors which can be used for arithmetic. This is evident in the amount of transistor space allocated to each kind functional unit in a CPU and a GPU. Figure 2.21 shows a cartoon of the space allocated to cache, control, and arithmetic units on typical CPUs and GPUs [10]. The CPU has more cache and control space since it needs to be able to execute complicated instructions quickly and to quickly switch between many thread contexts. The GPU has more arithmetic units since it is geared

for computational throughput on data-parallel tasks where less complicated instructions are needed [10].

CUDA

CUDA was first released in 2006 as NVIDIA's proprietary GPU programming platform. It makes minimal additions to C/C++, and any C programmer would be very comfortable programming in CUDA [10]. It was chosen over OpenCL, the open source GPU programming platform, in this work due to CUDA's greater feature support, stability, ease of programming, wider community usage, and ability to use new, cutting-edge features in NVIDIA GPUs that OpenCL is not.

The overarching theme for CUDA is that a host CPU thread directs the GPU's operation. Data must be transferred from the host to the GPU's global memory over the PCIe bus, *kernels* are launched to perform computations based on the transferred data, and then control returns to the host thread. "Kernels" are CUDA's parallel programs that each thread carries out over the data. Kernels must specify independent tasks for each GPU thread from the stnadpoint that thread execution order does not matter. There can be barriers where threads will wait for all other threads to arrive, but the order in which the threads execute to get to such a barrier is arbitrary and handled by the GPU hardware [10]. Blocks of threads have the same interleaving requirement. The order in which they execute is unspecified, and cannot be relied on for calculating values. Blocks of threads are executed simultaneously on a multiprocessor. The group of all blocks is called a "grid." The grid is analogous to the entire GPU device, blocks are analogous to the multiprocessors, and threads are analogous to the individual cores. Figure 2.22 shows the host-device execution and the organization of threads into blocks and blocks into the grid [10].

Grid and block dimensions can be 1D, 2D, or 3D arrays, which simply influences how the data is indexed. Every thread has unique variables, "threadIdx" and "blockIdx," that are automatically generated upon kernel launch. The grid dimensions (the number of blocks in each grid dimension) and the block dimensions (the number of threads in each block dimension) are also broadcast to every thread. Based on these quantities, a unique thread identification number can be computer and used to access the data. For example, if the grid is 2x2 and blocks are 4x4, a unique thread coordinate can be computed by doing id_y = blockIdx.y * blockDim.y + threadIdx.y and id_x = blockIdx.x * blockDim.x + threadIdx.x. If the grid and blocks are both 1D, indexing is much simpler, e.g. id = blockIdx.x*blockDim.x+threadIdx.x [10].

Even though blocks can be made of up to 1024 threads, when they are executed in a multiprocessor they are scheduled in smaller units that are 32 threads wide. These units are called *warps*. The multiprocessor executes the threads in a warp concurrently until all the threads in the thread block are complete. It then fetches another block form the queue and processes it. Multiple blocks can be resident in the SM if there are enough resources (registers, etc) to accommodate it [10].

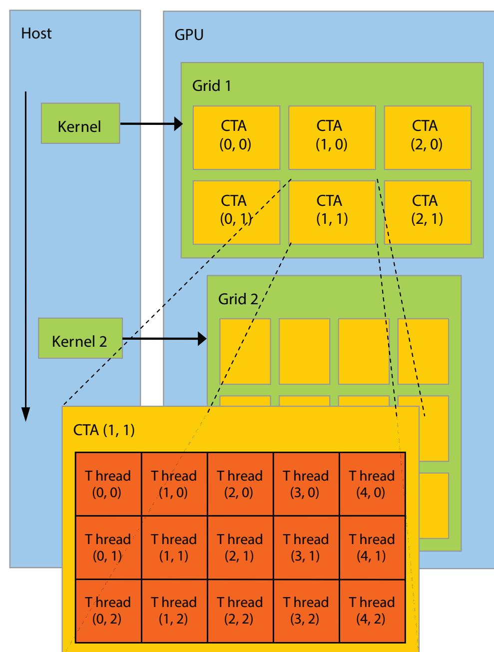


Figure 2.22: Host-device execution in CUDA and the organization of threads into blocks and blocks into the grid [42].

Warps are what abstract SIMD execution. Warps are like the SIMD vector that must be processed using the same interactions carried out over all the data elements of the vector. This is why every thread in a warp must execute the same instructions or they are split and serialized [10]. The low level hardware uses SIMD, but CUDA relaxes SIMD requirements since it allows threads to execute different instructions, albeit with a performance cost.

Memory

Bandwidth is the rate at which data can be read from or stored into memory by a processor. High bandwidth is needed to get data to the arithmetic units and to maintain high computational throughput. Most optimization work done of GPUs involves relieving memory bottlenecks. The bandwidths of recent Intel CPUs and NVIDIA GPUs are shown in Figure 2.23, and it can be seen that GPU's bandwidth far exceeds the CPU's. It is important to remember that this is maximum *aggregate* bandwidth, however, not the bandwidth available to each individual multiprocessor [10].

To maximize bandwidth and take advantage of spatial locality in memory, the memory subsystem on a GPU is also SIMD-like, as are most modern CPU subsystems. When a thread requests a piece of data, not just the piece that is requested is delivered by the subsystem, but rather a chunk of aligned data that contains the requested piece. If the other loaded values are not used, memory bandwidth is wasted. For best performance, CUDA

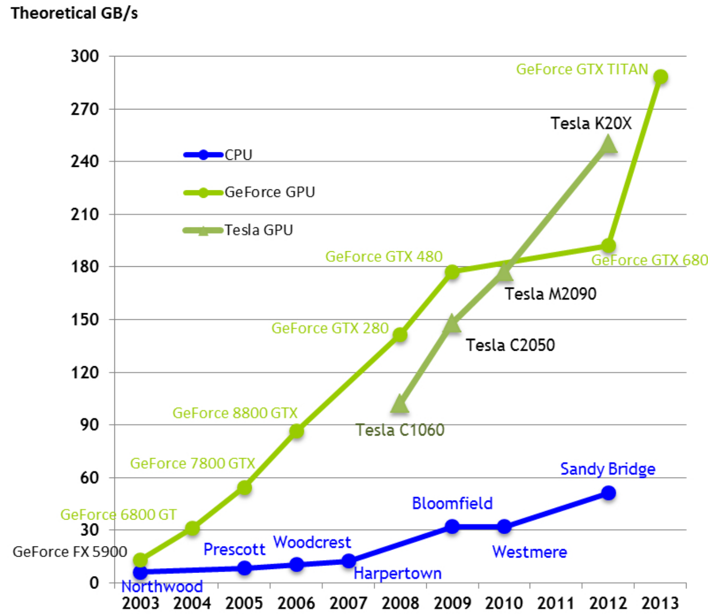


Figure 2.23: The total global memory bandwidth of NVIDIA GPUs vs. flagship Intel processors [10].

requires adjacent threads to access adjacent pieces of data. This way, memory transactions are "coalesced." In coalesced transactions, every piece of data in the memory payload is used and bandwidth is maximized [10]. Figure 2.24 shows this graphically [43]. If a thread needs to load an entire array, it is better to interleave values in memory so that adjacent memory is accessed at the *same time* by neighboring threads rather than loading adjacent data sequentially by a single thread [10].

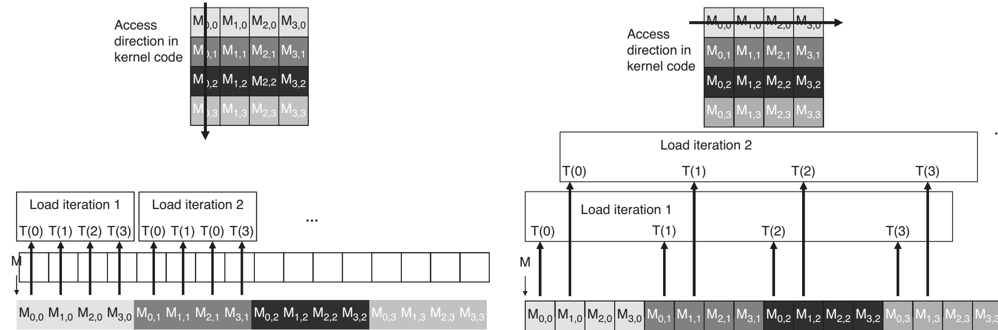


Figure 2.24: Memory transactions of CUDA threads. Coalesced access on the left and non-coalesced access on the right [43].

GPUs have even higher global memory latency than CPUs, from 200 clock cycles on newer cards up to 800 clock cycles on older cards, compared to about 50 clock cycles on a

typical CPU. CPUs, which are tailored for serial execution, want low latency memory access and do this through multilevel cache structures and large control spaces that allow them to do out-of-order and speculative execution. GPUs, which are geared for high throughput, want high bandwidth memory access and can tolerate high access latency by pipelining many threads. More transistor space is allocated for computation rather than for trying to minimize memory latency, and instead the memory latency is hidden though parallelism. Figure 2.25 shows how GPU threads are pipelined to hide latency whereas CPUs rely on low latency to quickly execute different threads in series [44]. Thus, GPUs perform best when as many threads as possible are launched [10].

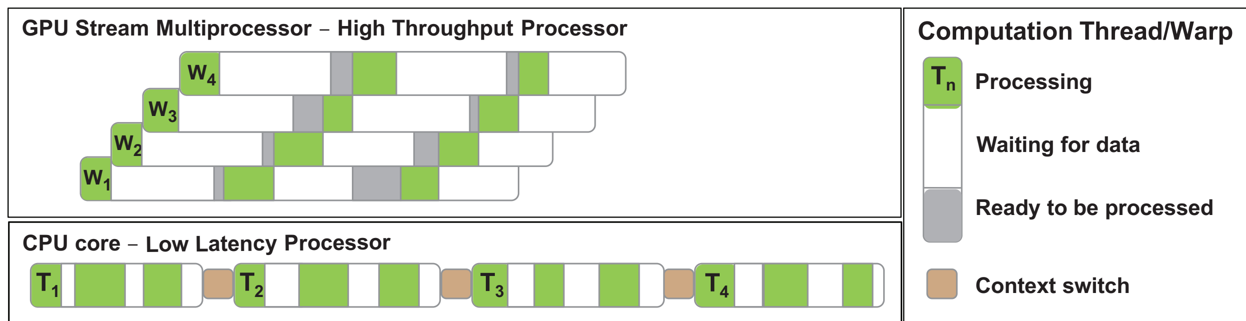


Figure 2.25: GPUs Pipeline many threads to hide high memory latency [44].

Even though there are fewer levels of cache in a GPU than a CPU, there are still many different memory spaces, each of which has different properties. Figure 2.26 shows the various memory spaces of an NVIDIA GPU. The global memory space is the largest memory space and has the greatest latency (∼200 memory clock cycles). This is the amount of total RAM specified for each card, which can be as small as 512 MB for very low-end cards and up to 12 GB for the largest high-end card. This memory space is accessible by all threads in all blocks.

The next level down is the shared memory space. This acts as a user-programmable cache and is not used unless explicitly coded. It resides on the multiprocessor and has very low latency (on the order of 1 processor clock cycle). The data can be seen by all threads within a block, hence the "shared" name. The shared memory space can allow intra-block communication without incurring the high penalty of global memory transactions and can therefore improve performance by acting as a cache, storing any data that is reused by threads [10].

The next level down is the local memory, which is thread-specific. The local memory serves to hold any data that cannot fit into the registers. It is not truly "local" since it is actually stored in global memory, but in Fermi and newer architectures the local memory is cached by an L1 cache on the multiprocessor. A cache is a small, but very fast, storage area for data between the processor registers and global memory. Caches hold data that has been loaded into the processor previously. If the data is required again, it can be loaded from the cache instead of the global memory. Loading from cache is much faster than loading form

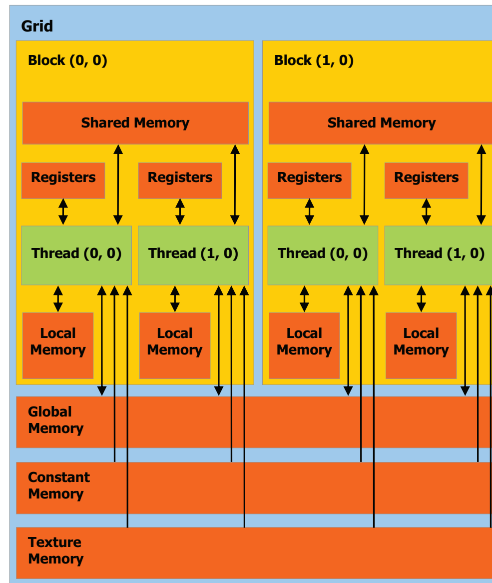


Figure 2.26: The memory spaces in CUDA and NVIDIA GPUs [10].

global memory, and caching increases performance by taking advantage of data reuse. There are multiple levels of cache as well, each of which are numbered. Lower numbers (e.g. L1) are faster and closer to the registers, and larger numbers (e.g. L2/L3) are typically slower and larger.

Registers, the lowest level of memory, are what the arithmetic units directly load their data from. They are fast, but have a limited size, which is why local memory is used to page them. Paging means that any data that does not fit into the registers is stored in the local memory. When the processor requests a value that has been paged, it is fetched from local memory and swapped for a register value. A Fermi GPU has a large register file (2 MB) compared to a typical CPU (∼6 kB), but each multiprocessor only has 128 kB of the registers, which is further subdivided by each thread (maximum register size per thread is 63 kB) [41].

The other memory spaces are the constant memory and the texture memory, which are both visible to all threads in all blocks. They aren't truly separate spaces, but rather global memory that is handled differently. Storing data in the constant space means that it cannot be written over (hence the "constant" name), but the data is cached so data reuse by threads can result in a cache hit and the global memory itself does not need to be queried [10]. Constant memory is limited to 64kB, however, and is cached by 8kB caches on each multiprocessor, so large datasets cannot be stored in constant memory. It also performs best when values are broadcast to all threads. Texture memory is also cached, but its cache is optimized for 2D spatial locality in the texture coordinate system instead of actual memory locality, and it does not have a maximum size. Another feature of texture memory is that is can perform low precision linear interpolation in the same transaction as a read [10].

2.8 OptiX

To save time and effort, NVIDIA's OptiX ray tracing framework is used by WARP for the geometry representation, surface detection, and material queries on the GPU. The programmer must write all geometric intersection programs for the geometry primitives as well as the geometry hit programs. OptiX provides the mathematical and computer science glue between the user-programmed ray tracing elements. Besides its flexibility, OptiX provides optimizations that would require many months for a single developer to replicate in hand-written code. It's main optimization is automatically producing high-quality acceleration structures for traversing the primitives in the geometry [45]. In this section, the general properties and workings of OptiX will be presented. Details about its implementation in WARP will be discussed in Chapter 3.

OptiX Programs

As stated before, the developer must provide all the programs for OptiX. These various programs are compiled to .ptx files, the paths to which are given to the OptiX API at compile time and are loaded by the executable at run time [45]. The programs are compiled to .ptx because it is a "virtual assembly language" that can be easily interpreted by GPU and requires no further compilation. Figure 2.27 shows the control flow in OptiX. The launch is just like a CUDA kernel launch. The API checks to make sure all the necessary data is present on the card, then launches kernels internally to carry out the trace on the GPU.

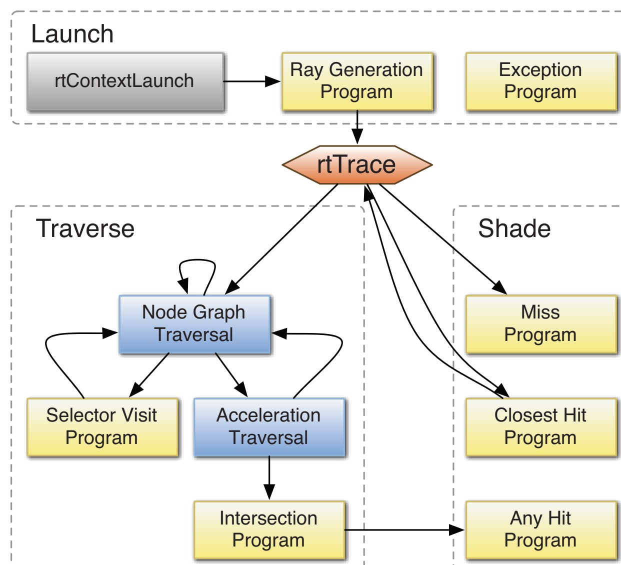


Figure 2.27: The program flow in a launched OptiX context [46].

The "ray generation" program is like a "main" function in the ray tracer. It initializes rays, their starting points and directions, and specifies the data that they report. Once

this is done, the trace is executed. The trace traverses the geometry and find intersection points. If *any* geometry primitive is intersected, the "any hit" program is executed. In graphics rendering problems this program is typically used for determining areas that are completely occluded from the light source, therefore terminating the trace. Shadow rays could potentially be used for an adjoint-type of calculation or variance reduction schemes to terminate neutrons according to some criteria, but are not used by WARP. As OptiX queries the primitives along a ray, it tightens an interval on t, the path length of the parameterized ray [45]. The equation for a parameterized ray is shown in Eq. (2.100), where $\vec{r}_P$ is the end point of the ray, $\vec{r}_O$ is the origin point of the ray, and $\vec{r}_D$ is the directional unit vector of the ray. Figure 2.28 shows an illustration of a ray-surface intersection.

$$\vec{r}_P = \vec{r}_O + t\vec{r}_D \tag{2.100}$$

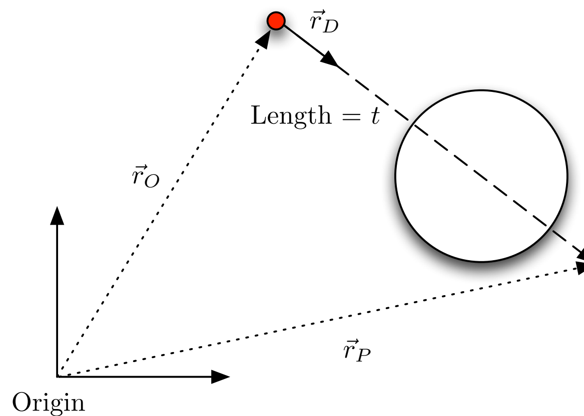


Figure 2.28: Ray-surface intersection: a sphere and a parameterized ray.

When the t interval becomes tight enough to determine the closest hit, the "closest hit" program is executed. This program typically returns the ray's closest intersection point to the ray generation program, which then uses this information to shade a pixel in an image, or in WARP's case, determines if a neutron travels past a surface. Miss programs tell a ray what to do if it does not intersect any primitives [45]. In rendering, this usually maps to a scene background image. Since a material must always be defined in WARP, a miss simply throws an error. This is analogous to a "lost particle" in traditional Monte Carlo codes.

Geometry and Intersection Programs

Geometrical primitives in OptiX are represented by their intersection programs. Intersection programs have two purposes – calculating the exact intersection points between a ray and a primitive and to providing axis-aligned bounding boxes around the primitives. These

bounding boxes must completely contain the underlying primitive, but should be as small as possible for best performance. Bounding boxes are queried when the acceleration structure is built over the geometry, which is discussed in the next subsection [45].

The main purpose of an intersection program is to calculate the intersection points, which must be reported back in terms of the t value. For example, the equation for a sphere is $r^2 = x^2 + y^2 + z^2$. Substituting Eq. (2.100) into this equation and solving for t yields Eq. (2.101). Since a sphere is a closed surface, there are two intersection points. The smallest nonnegative value is reported back by the intersection program.

$$\begin{aligned} a &= \vec{r}_D \cdot \vec{r}_D = 1 \\ b &= 2\vec{r}_O \cdot \vec{r}_D = 2b_2 \\ c &= \vec{r}_O \cdot \vec{r}_O - r^2 \\ t &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = -b_2 \pm \sqrt{b_2^2 - c} \end{aligned} \tag{2.101}$$

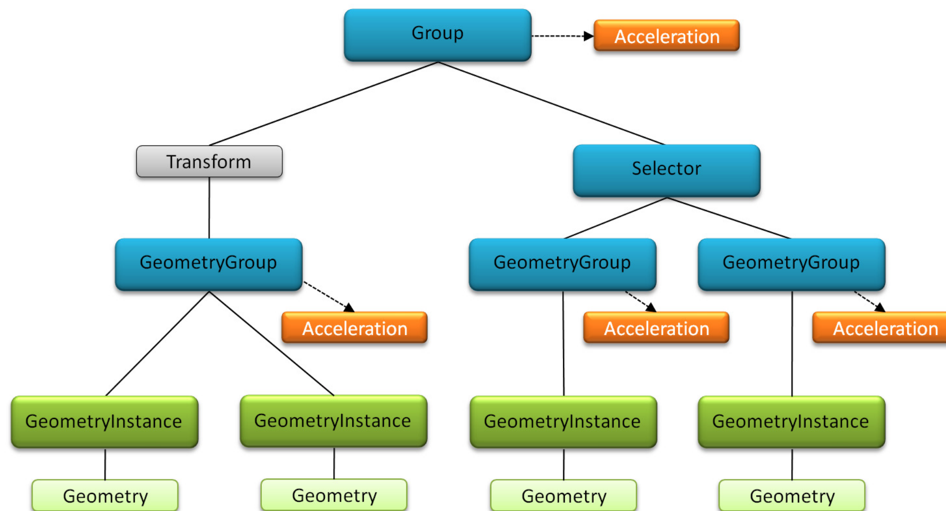


Figure 2.29: Node graph used for geometry representation in OptiX [45].

The framework by which geometric primitives are arranged into a scene is very flexible in OptiX. Figure 2.29 shows the node graph representation of the geometry that a ray traverses in OptiX [45]. The node types and connections are completely specified by the user. Rays enter the graph at the root node, or "entry point," and it must be specified before a trace can begin [45]. From the root node, rays start to query the geometry nodes in the rest of the graph. Intersection programs are run when a ray visits a geometry node to determine the t values of the geometry contained in the node. Some nodes, such as a "group" node, only serve to contain other nodes. A group can have children of any kind, whereas a "geometry group" can only have "geometry instance" children. A "geometry instance" is a object that

combines a geometry primitive, which describes the geometrical object, and a "material." A material is another name for the closest and any hit programs, and serves to package them into an object that they can be attached to multiple geometry primitives. OptiX can handle different ray types, typically one for radiance and one for shadow, and a geometry primitives can be attached to multiple materials to handle these ray types in different ways. WARP only uses radiance rays, and using shadow rays was not considered during its development. The way in which they traverse the graph depends on the acceleration structures attached to the nodes. These structures allow the rays to only query objects along their line instead of having to query all objects. How the geometry nodes are arranged in the graph greatly affects trace performance, as will be seen in Section 3.1.

There are two special node types that can have geometry group children - a selector node and a transform node. A selector node will decide which geometry group to send a ray down based on some type of criterion. This criterion can be based on a global variable that is assigned before a trace or a piece of data that is determined during the trace itself. It gives the framework more flexibility for the types of problems it can handle and the ease at which solutions can be implemented.

A transform node transforms the underlying geometry based on an affine transformation matrix. This node type can be useful for specifying one geometry instance node that is then instanced throughout the scene with different transform nodes. A transform node can also be used to move geometry based on time. For WARP's purposes this functionality could potentially be used for calculating the effects of geometric perturbations or even introducing time-dependent simulations. When transform nodes are specified, they must be attached to a geometry group and be given an affine transformation matrix. In three dimensions, this is a 4x4 matrix that can specify rotation, scale, translation, and shear. An example matrix for simultaneous translation and rotation is shown in Eq. (2.102) where dx, dy, and dz are translations in x, y, and z, respectively; and θ is a rotation around the z axis, i.e. a rotation in the x-y plane [47].

$$\vec{x} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad M = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & dx \\ \sin\theta & \cos\theta & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad M\vec{x} = \vec{x}' \tag{2.102}$$

The final, and arguably most important, part of the node graph is the acceleration object. This node specifies the acceleration data structure that is attached to each group or geometry group and *must* be present for rays to traverse these groups. Groups and geometry groups can share acceleration objects, but if any underlying geometry is changed, the structure must be rebuilt for both groups sharing the acceleration object. Simpler acceleration object layouts typically have better performance, however, since the acceleration objects higher up in the graph can only treat groups further down [45].

Acceleration Structures

The basic idea behind acceleration structures is that they provide a way to decompose the scene geometry into a hierarchical graph. Once this is done, parts of the scene that rays aren't close to intersecting can be pruned from the set of actual geometry primitives a ray must query to find intersection points [45]. OptiX provides different types of acceleration structures, namely, bounding volume hierarchy (BVH) trees and k-dimensional (k-d) trees.

BVH is object-centric in the sense that it places boundaries around groups of objects. Each leaf of the tree can contain one or more objects. K-d trees are space-centric in the sense that they partition the space objects lie in. K-d trees are binary trees that partition each volume into two smaller volumes until the lowest leaves only contain a portion of a single object. Figure 2.30 shows how BVH and k-d trees partition the objects and the space, respectively. BVH trees have the benefit of being shallower, and therefore fast to traverse, but some objects' higher bounding volumes can overlap, leading to situation where both subtrees must be queried.

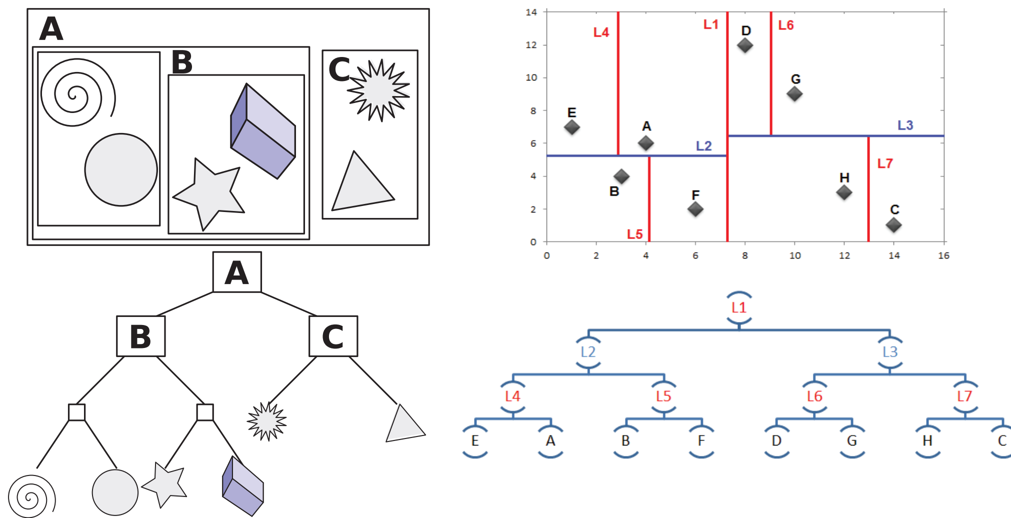


Figure 2.30: Spatial partitioning schemes (top) with the resulting trees (bottom). A BVH tree [48] is on the left and a k-d tree is on the right [49].

The benefit of these acceleration structures is that they can be queried in $\log(N)$ time rather than in linear time since major portions of the geometry do not need to be queried or intersected. OptiX automatically builds these trees over the geometry without any user input other than to specify the type. The two different types have tradeoffs in terms of size, build speed, and performance. Since WARP's geometry is static and relatively simple compared to photorealistic rendering jobs, the build speed and size have negligible impact. Determining which structure provides the best performance is the goal of one of the preliminary studies in Section 3.1.

The typical geometry representation in Monte Carlo neutron transport codes is combinatorial geometry, which uses union and intersection operations on the surface sense of a point in relation to second-order polynomials. This kind of algorithm is linear with the number of *surfaces* present in the geometry, so using OptiX in complicated geometries should provide provide a performance boost to geometry routines compared to MCNP or Serpent. The algorithms in both of these codes do use a form of acceleration, however. They use a "universe" representation to provide a level of acceleration in determining which material and/or cell a neutron is in. Universe "0" is typically the lowest level, i.e. what exists in the problem geometry. Each cell in universe 0 can either contain a defined material, or it can contain another universe. If it contains another universe, the neutron coordinates are transformed from the global values to the new universe's. Once the neutron is in this new universe, only surfaces contained in it need be considered instead of all the surfaces in the problem. Of course, cells in the new universe can contain other universes, and this cell query operation is done recursively until the actual material the neutron is traveling in is determined [12]. Using "universes" in this way is like traversing a k-d tree. It partitions the geometry based on their spatial relationships and allows a neutron to determine which cell it is in quicker by eliminating far-away cells from the set of cells that need to be queried. When the runtimes of Serpent, MCNP, and WARP are compared in Chapter 4, it will be seen how the various geometry representations scale with object number.

2.9 Previous Works

The impetus for developing WARP was the research done by Martin and Brown in 1984 [29] and by Vujić and Martin in 1991 [30]. In the 1984 paper, a method for mapping the Monte Carlo problem onto SIMD vector computers is described. The essential idea is to bank the neutrons into vectors based on their required operation. If a neutron is scattering, its data is placed in the scattering buffer. If a neutron needs to do a surface crossing calculation, it is put in the crossing buffer, and so on. Once a buffer becomes full, it is processed in a SIMD fashion by the vector computer. Processing the neutrons makes the buffers contain non-uniform reactions, however, and a "shuffle" operation is done that actually moves data back into contiguous blocks based on the reaction type.

This new approach was named "event-based" Monte Carlo, since the neutron events are tracked and processed as a group [29]. This was a very different way of performing a Monte Carlo simulation at the time. Almost all computers were strictly serial, and SIMD lanes were only available in supercomputers. Therefore, the pervasive method was the "task-based" method in which neutrons are tracked for their entire lifetime in series. Since GPUs are massively parallel and rely on SIMD, an event-based algorithm seems to be the appropriate approach to GPU-accelerated neutron transport.

In the 1991 paper, a vectorized approach is applied to the collision probability method (CPM), which is a deterministic way to solve the integral form of the neutron transport equation [30]. The paper does not directly deal with any type of Monte Carlo simulation, but

it addresses the topics of transferring a scalar, CPU-optimized program to a vector computer and discusses why global restructuring of CPU optimized codes is necessary to fully exploit highly vectorized architectures. This paper served as inspiration to start WARP from scratch instead of trying to completely restructure an establish CPU-based code.

Vectorizing the Monte Carlo algorithm should allow for efficient GPU execution, but a paper by Zhang [50] also shows that by remapping data references, the thread divergence in GPU warps can be minimized. This paper provides a simple way to "vectorize" the GPU data without actually moving data itself. This method may lead to sparse data access, but this will happen in a Monte Carlo algorithm anyway so it may be of use to reduce thread divergence and increase the number of active warps per active cycle on the GPU. Incorporating Zhang's idea means that the algorithm that WARP uses in its event-based transport algorithm differs from the methods in [29] and [30] in that WARP will use a reference remapping vector instead of a shuffle operation. Data will not actually be moved, rather only the references to the data.

Although there has been much research done on Monte Carlo neutron transport, there is relatively little research about performing Monte Carlo on GPUs since GPGPUs are a new technology. There have been two significant developments, however. One is by Adam Gregory Nelson, who published work on a GPU accelerated Monte Carlo neutron transport code he wrote for his Master's research at Pennsylvania State University [28]. In his thesis, he reports on developing a task-based Monte Carlo Neutron transport code, LADON, and an event-based Monte Carlo Neutron transport code, CERBERUS. Both codes have CPU versions that have the extension "c" and GPU versions that have the extension "g."

Comparing both codes, Nelson was able to attain a 24x speedup between LADONc and LADONg, and a 1.13x speedup between CERBERUSc and CERBERUSg. These results are very encouraging and show that significant speedups can be realized by performing Monte Carlo neutron transport on the GPU, but discouraging since the event-based speedup was so poor. Nelson also states that the poor performance of CERBERUS was due to the sorting of data was done on the CPU, and certain sections of his algorithm serialized execution completely. As a result, he abandoned CERBERUS and most of the thesis is about LADON [28].

There are also a handful of assumptions in his implementation that required his comparisons to be against a CPU code containing the same assumptions – which is why he wrote CPU versions of his codes. LANDON and CERBERUS use point-wise cross sections, but they are not read from ACE formatted data files, they do not perform any discrete inelastic scattering, and ν is fixed at 2.53. Comparing these codes against production codes that don't use these assumptions, like MCNP and Serpent could not be done, since the source of differences in solutions would be unclear. The codes seem to also be quite inflexible. They can only model spheres and cuboids, cannot transform these objects, cells cannot intersect, cell nesting must be explicitly defined and input in-order, and there is a total limit of 100 cells.

The GPU implementation may not be ideal either, since Nelson uses an array of structures (AOS) data layout instead of a structure of arrays (SOA) layout, which is not ideal

for coalesced loads as was shown in Figure 2.24 (though Nelson acknowledges this). CERBERUS does not seems to employ parallel algorithms other than for transport. None of the underlying operations necessary for implementing event-based Monte Carlo were parallelized effectively. The cross section data was also simply left as vectors in device memory and was not reformatted in any way to optimize the access pattern on the GPU [28].

The second major study done was that by Liu, et al. In the group's first publication, a GPU is used to perform a criticality calculation [26]. Very simple geometry is used, a bare slab and a bare sphere, and one group cross sections are used. The fission source operations and sorting are also done on the CPU rather than the GPU. Despite using the CPU to do these tasks, Liu et al. report a speedup over an identical CPU version of 7x for the bare sphere and 33x for the bare slab. They use a task-based algorithm for transport [26].

Further results were presented at the SNA+MC 2013 conference, where Archer, a general purpose Monte Carlo radiation transport code developed at Rensselaer Polytechnic Institute, was modified to run an event-based criticality simulation in a 1D slab. They found that control flow efficiency is increased, but global memory transactions are increased dramatically. Thus, the GPU is not kept busy when the number of particles is small, and performance is about 10x less than a task-based implementation [27]. No details about the implementation are given other than that data references are remapped similarly to the method that was presented at the ANS Winter Meeting in November of 2012 as a part of WARP's preliminary studies [51]. Single-energy cross sections were used again, which means that they can be entirely stored in the GPU's shared memory space for the task-based approach. Due to using small cross section data, any benefits from reduced divergence of the event-based algorithm is outweighed by the small amount of global memory transactions needed in the task-based approach. WARP uses real cross section data, and the memory traffic created by doing so should make the particle data traffic a small part of the problem.

Henderson implemented a GPU-accelerated algorithm for photon transport in Geant4 for the purpose of modeling CT scans [24]. Henderson uses interpolated cross sections, which are very smooth and small compared to those used in neutron transport. The geometry treatment is a voxelized approach, well-suited for reading in CT scan data. Henderson also only needed to consider water as a material since the human body is basically water. Criticality was also not considered, for obvious reasons. Secondary electrons were transported, however. Henderson uses a task-based approach, using the shared memory as a stack of to-be-transported photons and electrons, basically transforming the SM into its own small, independent processor (like a CPU). Henderson also emphasizes the positive effect of using an SOA access pattern, reporting 3-4x speedup over AOS. Henderson was able to achieve a total speedup of around 50-60x over the standard Geant4 routines [24].

A study lead by Qi Xu from Tsinghua University has investigated adding GPU acceleration to their in-house Monte Carlo code, RMC [52]. RMC is being developed as a drop-in replacement for MCNP since MCNP is export-controlled and very difficult for Chinese universities to obtain (source code might be impossible to obtain). They've published some of their results, including an eigenvalue test and geometry routine test. The eigenvalue test uses algorithms similar to those used by Liu et al. and Henderson. The Tsinghua team uses

static geometry, single group cross sections, and a pop-stack, task-based transport algorithm. They were able to obtain a 113x speedup over a CPU version without a flux tally and 36x speedup with a flux tally [53]. The study offloads the geometry processing routines of RMC to a GPU, where they were able to get 5-50x speedup over an all-CPU version of RMC when the number of fuel rods in their geometry went from 25 to 625. When more rods are used, geometrical processing becomes a large part of the computation time, and the GPU was able to process this well, leading to the very large speedups in the 625 rod case. Again, one group cross sections were used [52].

OpenMC is a "Monte Carlo particle transport simulation code focused on neutron criticality calculations ... [that] was originally developed by members of the Computational Reactor Physics Group at the Massachusetts Institute of Technology starting in 2011" [13]. It aims to be an open testing platform for novel Monte Carlo algorithms, has gained U.S. governmental approval to be released as open source software. It is cited often in this work due its freely available source code and thorough documentation, and the process of release WARP as open source software will be streamlined by the efforts done in releasing OpenMC. OpenMC is also being developed by NVIDIA to run on GPUs. Progress was presented at the 2014 GPU Technology Conference (GTC), but mainly consisted of accelerating xsbench, OpenMC's synthetic cross section processing benchmark tool, and did not mention any developments on OpenMC itself [54, 13].

WARP aims to be much more general and accurate than all previous codes while employing efficient parallel algorithms in order to have a completely GPU-accelerated code. WARP reads ACE-formatted data, performs all reaction types as prescribed by them, uses a Serpent-like unionized energy grid to regularize data access, uses an event-based transport algorithm with parallelized operations for sorts and sums, use OptiX for general 3D geometry representation (without explicit nesting), use a SOA for neutron history data, and perform *all* operations on the GPU unless strictly forbidden (the benefit of which will be discussed later). WARP takes these previous works into consideration and advances the state of the art by:

- Using an event-based transport algorithm to conduct continuous energy Monte Carlo neutron transport on a GPU
- Using a vector of neutron references to remap data access on-the-fly, therefore minimizing thread divergence and simultaneously eliminate completed histories
- Using nuclear data loaded from ACE formatted files
- Using a modified unionized energy grid data structure to minimize data traffic when performing cross section lookups
- Treating interactions exactly as specified by the nuclear data files
- Using the NVIDIA OptiX ray tracing framework for general 3D geometry representation

- Using the CUDPP libraries to perform parallel sorts, scans, and sums on the GPU
- Producing results (multiplication factors, fission distributions, flux spectra) comparable to production Monte Carlo neutron transport codes

Chapter 3

GPU Implementations

In this chapter, all GPU implementations will be explained in detail. First, the exploratory studies done in preparation for developing WARP will be discussed. These studies show the algorithmic benefits of a very important feature of WARP - remapping the data references. This chapter also covers how OptiX execution is optimized for best performance in reactor-like geometries. After the preliminary studies, the data layout for cross sections is explained and its similarities and differences from Serpent pointed out. The last topic discussed is the CUDA kernels written for WARP. These routines process the neutrons as they travel through the problem geometry and provide the "glue" to connect all the important tasks that WARP requires to process the neutron histories.

3.1 Preliminary Studies

Before any serious coding efforts were undertaken, a pair of smaller preliminary studies were done to determine the feasibility of doing Monte Carlo neutron transport on GPUs. The first study performs a simple, 2D, mono-energetic scattering game on a GPU. The second study tests the OptiX framework to determine if it can handle randomized ray tracing while maintaining acceptable performance. In both of the following studies and in WARP itself, CUDA 5.0 and OptiX 3.0.1 were used.

2D Scattering Game

The goal of the 2D, mono-energetic scattering game is to determine whether a sorted or an unsorted event-based algorithm was best for controlling thread divergence on a GPU, to the see the relative performance of the history- and event-based GPU implementations to an identical serial CPU version, and to determine if the performance of the CUDPP (CUDA Parallel Primitives) library is adequate for use in WARP. A set of three GPU implementations were written using CUDA C, and a single CPU implementation was written using C.

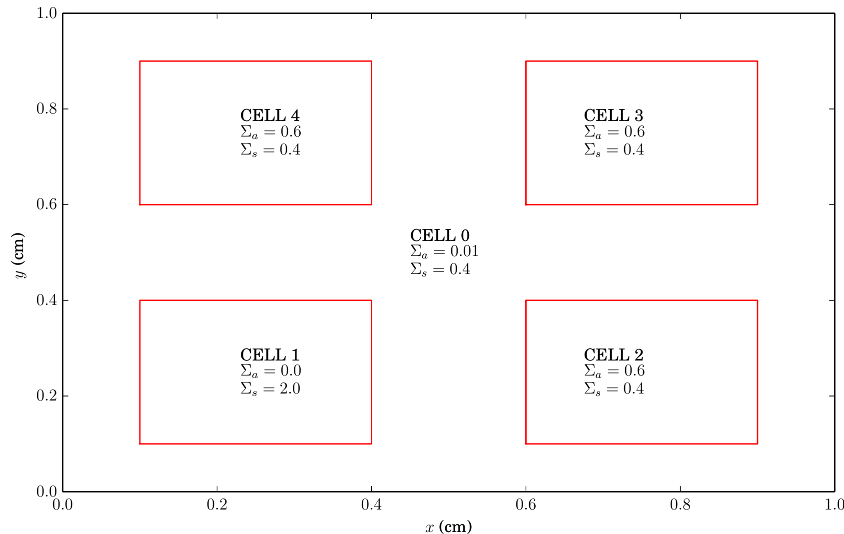


Figure 3.1: The 2D geometry of the scattering test.

In the simulation, only two reaction types are possible - scattering and absorption. Scattering is treated isotropically and, since it is mono-energetic, scattering only serves to change the direction of a particle. The geometry of the game is kept simple in order to highlight how the reaction divergence is handled rather than how geometry routines effect performance. The geometry used in the game is shown in Figure 3.1. There are five cells, all of which are square, and particles can only move in the x-y plane. Each cell has different reaction cross sections, and cell 0 extends to infinity but has a nonzero absorption cross section so particles cannot scatter forever. Particles are initialized with a uniform, random distribution in cell 1, which has no absorption cross section so that particles must cross a cell boundary at least once. Since the cells are few and square, the "where am I?" operation that determines the cell number each particle is in (and therefore the cross sections) is done with simple, hard-coded logic comparisons.

As particles scatter, they start to be absorbed and their histories are terminated. This means they are no longer transported, and their data should no longer be accessed by threads. Here is where the GPU implementations start to differ from one another. The task-based implementation performs the transport from a one-particle-per-thread standpoint. When the transport kernel is launched, each thread contains a while-loop that transports a particle until it terminates and the thread returns. If more histories are requested than the maximum thread number, transport is split into batches. The diagram of the simple task-based algorithm used is shown in Figure 3.2. The CPU version of the code also uses this algorithm on a single thread that processes the histories in series.

An event-based algorithm performs the same operations on the data as the task-based algorithm, but the operations are instead carried out over all the particle data simultaneously

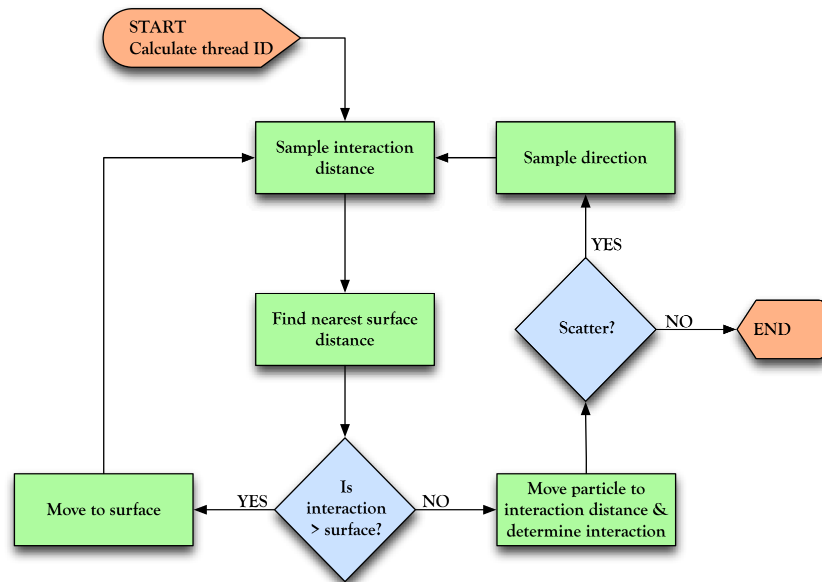


Figure 3.2: The task-based algorithm used in the 2D scattering study.

in parallel. Now, each kernel does *not* contain a while loop. The while loop is on the CPU host, the drawbacks of which are discussed later. Each kernel performs a single, simple task on the entire particle dataset in one step. Figure 3.3 shows an illustration of how the nearest surface distance and interaction distance comparison is done in a data-parallel way. The leftmost diagram in the figure shows a box containing blue dots, which represent the positions of a set of neutrons. These positions are all loaded from memory in a single step. The next diagram shows the directions are then loaded in a single step. The third diagram shows the sampled interaction distances for each of the neutrons, represented by green dots. After the interaction distance is know, the distance to the nearest surface, shown as red dots, is calculated for all the neutrons. Once the surface distance is known, the next step would be to update the neutron position to the smaller of interaction distance or the surface distance, at which point the process repeats. Each step of a data-parallel algorithm acts this way; it performs identical operations on the entire set of neutrons.

An analogy can be made with homework grading. A task-based algorithm is like grading a single student's homework in its entirety, then moving on to the next student's. An event-based approach is like grading a single *problem* for *every student*, then moving on to the next problem. This makes the transport loop *data parallel*, which GPUs need to keep warps coherent. Figure 3.4 shows the event-based transport loop used in the scattering game.

Two different implementations were made using an event-based algorithm, one that uses the CUDPP compaction to remap threads to active (non-terminated) data, and one that does not. In the non-remapping version, threads simply return and do no processing if they access data belonging to a terminated particle. In the remapping version, the number of

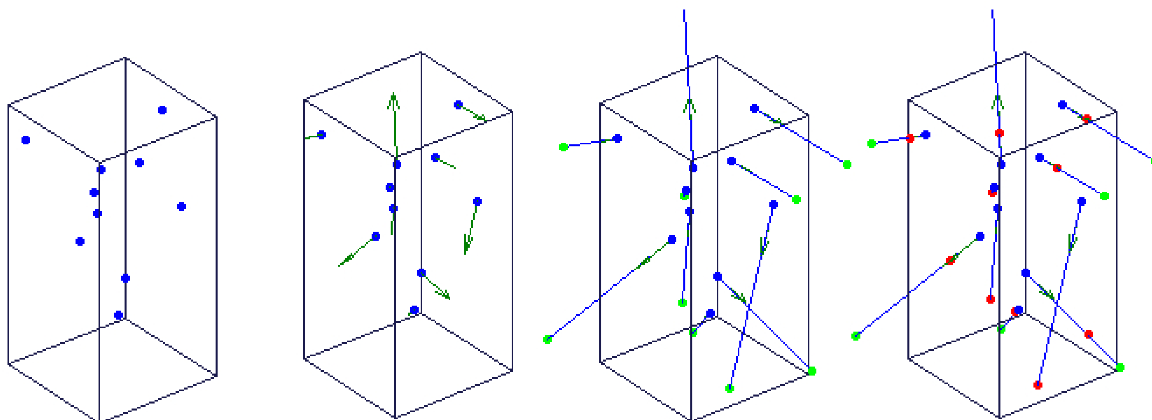


Figure 3.3: Ray tracing done in a data-parallel way.

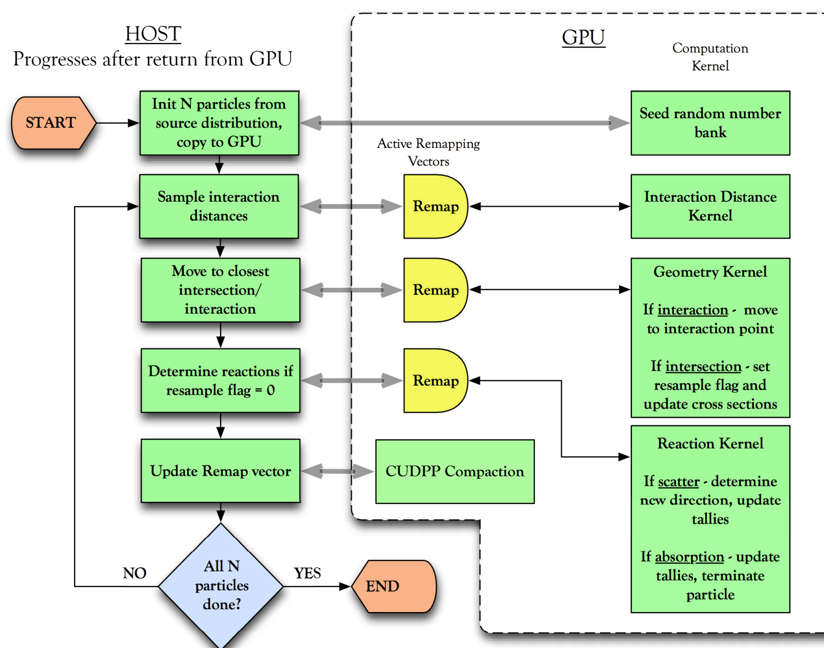


Figure 3.4: The event-based algorithm used in the 2D scattering study.

threads launched is equal to the number of active particles left in the dataset, not the size of the dataset itself. The threads access a remapping vector, which transforms their thread ID to a data index that still contains active data. Figure 3.5 shows how the remap vector transforms the initial thread ID to an active ID.

The CUDPP compaction function does just this - it takes an input vector and a valid

flag vector and returns a new vector containing only the valid elements, preserving the order they are in. For this test, the input vector is the thread ID vector (tid[i]=i), and the flag vector contains a 1 if a particle is unterminated and a 0 if it is not. The compact function then returns a remapping vector, which is as long as the number of unterminated particles and contains the indices of the unterminated data. Even though the neutron data access will not be coalesced, accessing the remap vector should be coalesced (adjacent threads access adjacent data) and should therefore be fast to load.

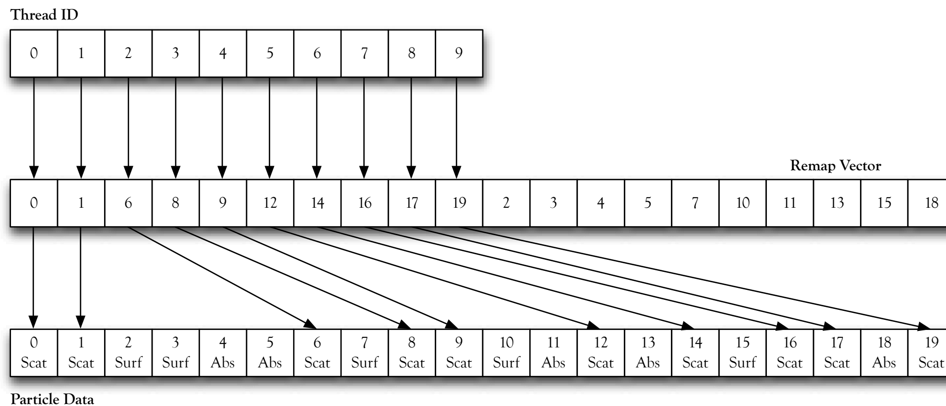


Figure 3.5: Mapping thread IDs to active data through a remapping vector

NVIDIA gives each GPU card a "compute capability" number which summarizes the features available on the card. For example, "dynamic parallelism" is a feature only supported on cards of compute capability 3.5 and above (dynamic parallelism allows kernels to launch additional kernels). When this preliminary test was first written, Fermi cards were the newest available, and the cards used were of compute capability 2.0. For these cards, there is a maximum grid dimension size of 65,536 blocks [10], and this limit was hardcoded into the applications. In other words, if the total number of particles was greater than what could be transported with 65,536 blocks, the particles were divided into separate batches. This limit was lifted in devices with compute capability of 3.0 and up (which includes the K20 used in this work), where the maximum 1D grid size was increased to $(10^{27} - 1)$ [10]. The hardcoded block limit was used to keep compatibility with Fermi cards of compute capability 2.0.

Libraries currently available for the GPU carry out a single task on a dataset; they are not coded for single-thread operation and must be launched as their own kernels. Having threads in different states wouldn't allow the library routines to be dropped in and used consistently across all the active particle data. Furthermore, much of the performance from optimized algorithms in the libraries comes from thread blocks cooperating, and using a task-based algorithm would be like treating each SM as a separate CPU (using a stack-popping method). The CUB library can perform sorts and scans in a per-block or per-warp fashion, but OptiX would be incompatible with a non-global transport scheme [45].

Figure 3.6 shows the speedup factors, $F_s = t_{\text{CPU}}/t_{\text{GPU}}$, of the GPU implementations over the CPU implementation versus the number of particles run. This benchmark was run on a server with an 8-core AMD Opteron 6128 CPU clocked at 2.0GHz and a Tesla K20 card. The task-parallel implementation always performs better than the non-remapping event-based approach. The remapping event-parallel implementation starts to overtake the non-remapping version at around 20,000 particles and overtakes the task version around 100,000 particles. No more that 10^8 histories could be run because of memory constraints of the event-based implementations. At this point, the speedups of the task and non-remapping implementations appear to be saturated around 5.9x and 4x, respectively. The remapping implementation does not seem to be saturated, but reaches a maximum speedup of 13x over the serial CPU implementation at the limit of this study. Why there is a noticeable jump in the remapping speedup at 10^7 particles instead of a smooth transition is as of yet unexplained.

The non-remapping approach sees a alight performance decrease at around 10^7 particles, the point where transport has to be broken into two batches ($128 \times 65,536 = 8,388,608$ threads in a single kernel launch possible). The finite address space affects the non-remapping implementation with 512 threads per block at larger dataset sizes, specifically at $512 \times 65,536 = 33,554,432$ particles. The task-based implementation also needs to break transport into two batches at the same boundaries as the non-remapping implementation, but is seemingly unaffected by it. The effect is likely unnoticeable since it only implies a second kernel launch in the task-based implementation instead of the hundreds more in the non-remapping. A similar test was done with a Tesla C2075 (Fermi) card where the performance hit at 65,536 blocks is more accentuated. The results of the test and speculation as to why it exhibits different behavior compared to the K20 is shown in Appendix A.

It should be noted that compiling the tests with compute capacity 2.0 (-arch sm_20 compiler flag) improved performance about 20% over compiling with compute capacity 3.0.

The most likely reasons that the task-based implementation outperforms the event-based for small particles datasets is because of the simplicity of the problem and the communication and setup overhead associated with many kernels being launched from the host. The task-based implementation only launches a single kernel that houses the transportation loop whereas the event-based implementations must launch kernels for every interaction *within* the transport loop. The event-based method therefore launches hundreds of kernels in every loop compared to the single kernel of the task-based implementation.

In addition, having only two reactions means threads only diverge when they terminate. WARP will use real data and have many reactions types to deal with, so divergence will be a greater problem. Only having two reaction channels also allows threads in a block to almost always be in the same step of the transport algorithm. The only real problem in this study is the idle threads left in the block after they terminate.

The non-remapping implementation is really just an intermediate case between the task-based and remapping implementations. It has all the drawbacks of the event-based algorithm, namely high kernel launch costs, but none of the benefits of breaking the transport up into coherent steps since any threads are left idle in blocks once their particle terminates.

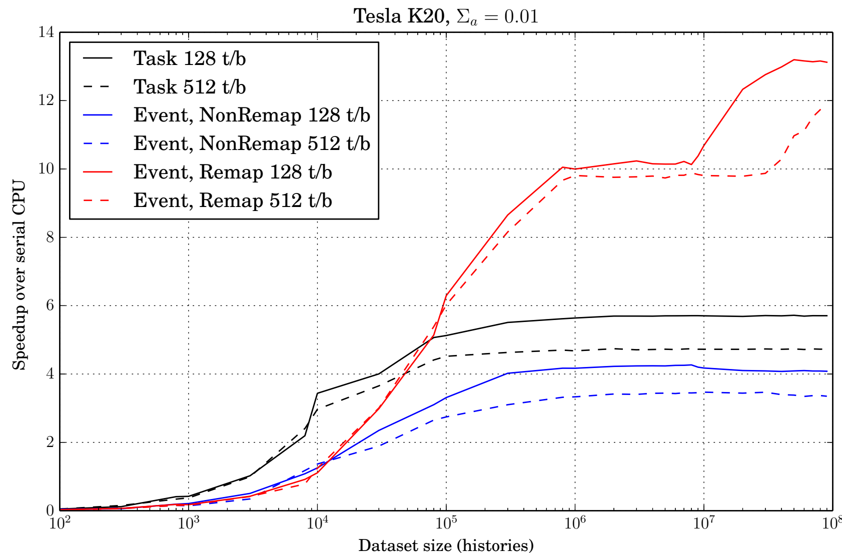


Figure 3.6: Speedup factors of the GPU implementations over the CPU implementation on a Tesla K20.

The remapping implementation shows the real benefits of an event-based algorithm in that a sort can be inserted into the transport algorithm to convert a “particle per thread” into “X particles shared by Y threads.” The remapping sweeps out references to terminated particles and allows the blocks to remain full of active data. This eliminates the cost of processing blocks with stale data, i.e. makes each block processed worth more on average.

The speedup curve highlights an important feature - that speedup plateaus at a large number of histories. This is because of the GPU’s ability to hide memory latencies through pipelining when there are many many active threads. It also may be related to the kernel launch overhead in that running large datasets spreads the overhead cost over many particles, reducing its cost per particle. Other than effecting the maximum number of threads resident on the card, the number of threads per block seems to make little difference between the implementations. Having fewer threads per block slightly outperforms having more for all cases. The reason for this could be the registers spilling to local memory (which is slower) since there are more threads resident in a multiprocessor and more variables need to be stored in the registers. This is speculation, profiler statistics about register spilling were not gathered.

Figure 3.7 shows the same test, but with the absorption cross section in cell 0 increased from 0.01 to 0.1. This means the potential difference in the number of scatters a particle undergoes before being absorbed is greatly reduced and, therefore, impact of thread divergence is also reduced. Again, no more that 10^8 histories could be run because of memory constraints of the event-based implementations. At saturation, the speedups of the task and non-remapping implementations appear to be around 6.2x and 4.5x, respectively. The

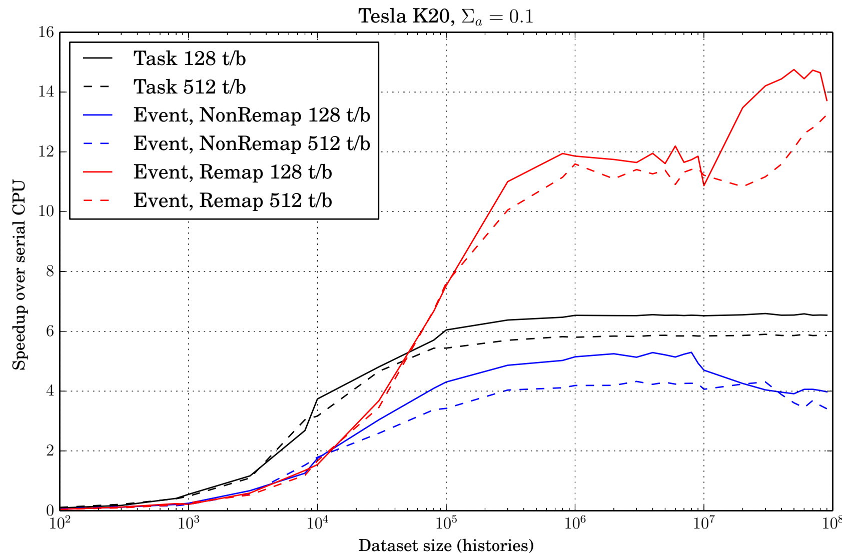


Figure 3.7: Speedup factors of the GPU implementations over the CPU implementation on a Tesla K20 with Σ_a increased to 0.1 in cell 0.

remapping implementation does not seem to be saturated, but reaches a maximum speedup of 14x over the serial CPU implementation. Again, there is a noticeable jump in the remapping speedup at 10^7 particles; the cause of which is not known. The curves have very similar trends as in the case with $\Sigma_a = 0.01$ in cell 0, but saturate at slightly higher values, most likely due to the fact that divergence is intrinsically decreased by the material parameters.

Figure 3.8 shows the number of active histories per transport iteration for the batches and remapping implementations. Since the remapping implementation eliminates all references to terminated particles, the data loaded into the threads blocks is all active and no threads return without doing work. The figure clearly shows that the blocks are kept full until the end of the simulation when the remaining number of particles becomes less than the maximum number of threads. It also shows how quickly particles are depleted from the blocks.

In this simple case, by far most interactions happen in cell 0, and particles are absorbed almost identically to Eq. (3.1), where i is the iteration number and Σ_a/Σ_t is the absorption probability in cell 0. This is expected behavior. Figure 3.8 does not reflect the time taken by each iteration, however. The iterations with more particles take longer than those with fewer for both implementations. More particles are transported per time in full iterations, however, as evidenced by the greater speedups of the remapping algorithm in Figures 3.7 and 3.6. The cycle time of a completely full iteration is approximately 0.2 seconds, whereas cycles times of iterations contain less than 5000 particles are around 0.01 seconds. The corresponding processing rates are 4.2×10^7 particles per second for full iterations and 5×10^6 particle per second for small iterations, however.

$$\frac{N}{N_0} = \exp\left(-\frac{\Sigma_a}{\Sigma_t} i\right) \tag{3.1}$$

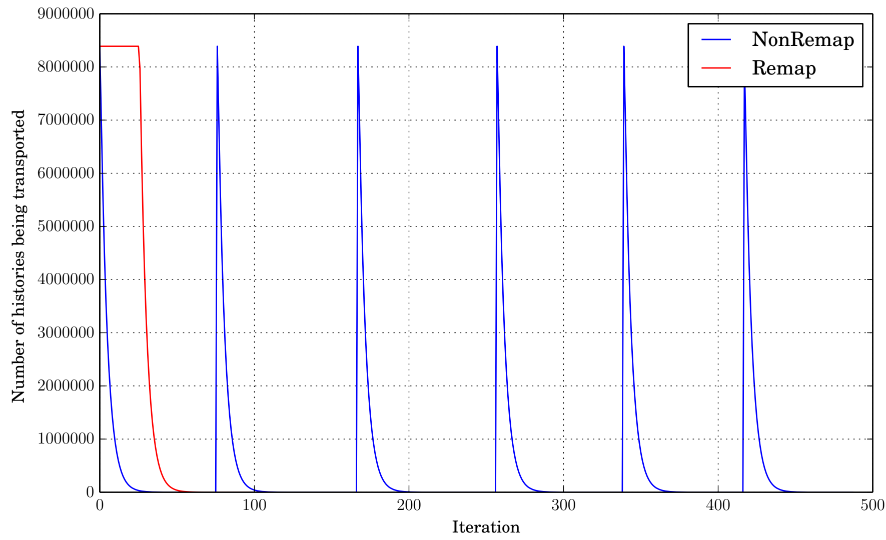


Figure 3.8: The number of active threads for the event-based GPU implementations.

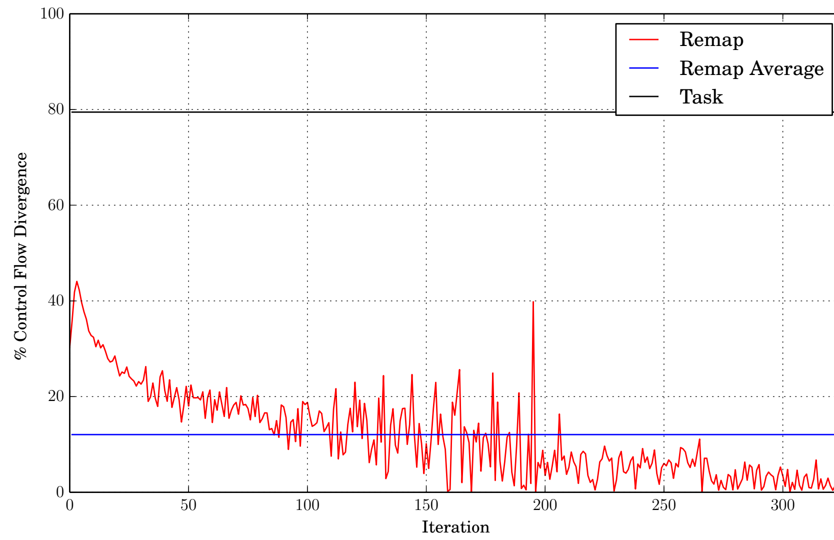


Figure 3.9: The percent of control flow divergence in blocks of the task- and event-based GPU implementations.

Actively remapping the data also reduces control flow divergence, as shown by Figure 3.9, the effect of which only manifests itself in this study as keeping threads full of active data. Despite this, remapping will allow the blocks in WARP to remain coherent (undergoing identical interactions) as well as full, which is another benefit of remapping. The figure shows the amount of divergence in a kernel launch, so the task-based algorithm only reports an average value since it only launches a single kernel.

From this preliminary, 2D, mono-energetic scattering study it can be concluded that using an event-based algorithm with a compaction/sort algorithm to eliminate terminated particles from being accessed by thread blocks drastically reduces control flow divergence and keeps warps coherent. There is a cost for adopting an event-based algorithm, however, namely the overhead of kernel launches. How these factors compete in WARP will be shown in the results in Chapter 4. WARP will adopt the event-based algorithm in hopes that thread coherency and its benefits (maximal load coalescing and little warp serialization) will outweigh launch overhead when real data is used, as well as to allow library usage to perform complicated parallel operations.

Ray Tracing with OptiX

Another preliminary study was conducted to investigate how OptiX performs when ray tracing is done from randomized points and to find the optimal OptiX configuration for WARP. In rendering, rays are initialized in a uniform array, but OptiX also allows for arbitrary starting points and directions to be used. This flexibility comes from being able to write custom ray-generation programs in OptiX. OptiX processes rays concurrently, so the ray tracing in WARP is done in a batched way, i.e. the traces are done for all neutrons in a single step (like that shown in Figure 3.3).

There are two types of scaling in OptiX that need to be characterized in order to ensure optimal performance – scaling with regard to the number of concurrent rays traced and scaling with regard to the number of geometrical objects in the scene. Determining how OptiX scales with the number of concurrent rays is important in knowing how large neutron batches need to be in WARP for the geometry routines to execute efficiently. Nuclear reactor simulations can contain thousands of material zones, and knowing how OptiX scales with the number of geometrical objects is important for choosing a configuration that will allow WARP to perform neutron transport well in complex geometries.

The first scaling study focuses on scaling with regard to the number of concurrent rays. Three different geometries were used in the test – an assembly-like hexagonal array of cylinders, an assembly of the same size but with two interleaved arrays, and a much larger version of the assembly-like geometry. The assembly-like configuration has 631 instances of a single primitive type, the interleaved array has 381 instances of three different primitives types, and the large array consists of 1801 instances of a single primitive type. These cases were chosen to determine if there are any large differences in the ray scaling when the number and types of objects are changed. Each array resides in a large hexagonal prism, which is in turn nested in a large cube that defines the outer limit of the scene.

The second scaling study focuses on how OptiX scales with regard to the number of geometrical objects in the scene using two different primitive instancing methods. Instancing refers to how individual geometric objects are created in the OptiX scene. The geometry used in this study is a hexagonal lattice of cylinders identical to the assembly-like geometry of the first scaling study. The volume, pitch to diameter ratio, and aspect ratios of the lattice is kept constant, but the cylinder radii are halved while doubling the number of elements on an edge. The smallest scene only has a single primitive in it, and the most complex has 42,843 primitives in it.

Point-in-Polygon / Material Query

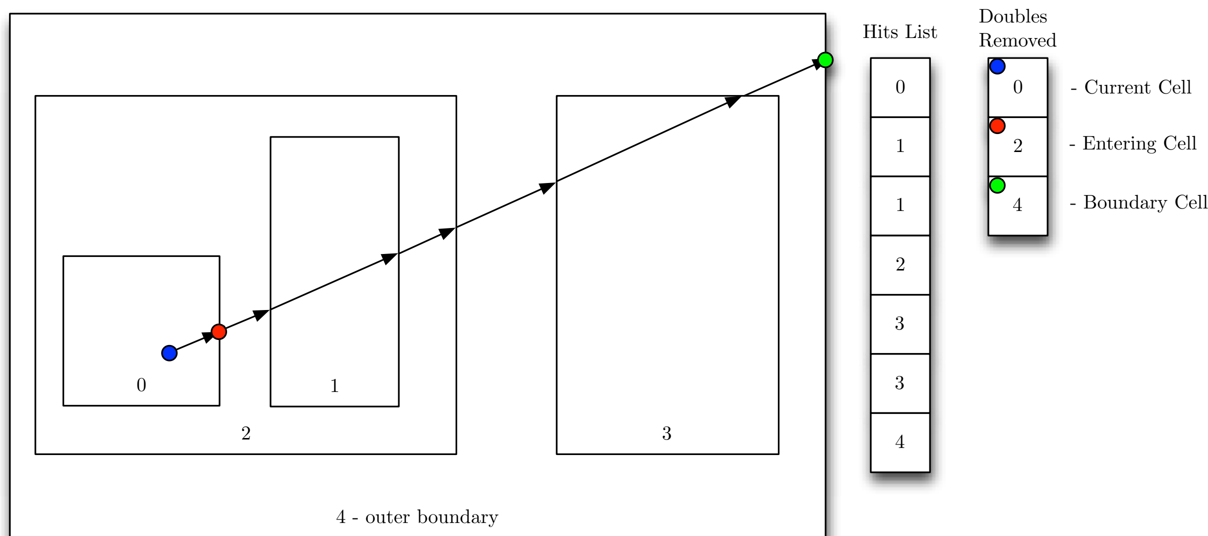


Figure 3.10: The point-in-polygon-like algorithm for determining the entering cell/material number by using ray tracing

An important job for the geometry routine in a Monte Carlo neutron transport code is to determine the cell and material IDs of a particle based only on its coordinates. In Woodcock tracking, surface intersections are not calculated and the material query is the only place where geometric information enters into the simulation. In WARP and other ray-tracing codes, material information is only updated when a sampled interaction distance is greater than the near surface distance. The neutron is then placed on the boundary, the material information is updated for the material the neutron is entering, and the interaction distance is sampled again using the same direction of flight as before. WARP will use an algorithm to determine the entering material number by using ray tracing, since all the geometric information is stored in the OptiX context. This also means the material query will be able to take advantage of the OptiX acceleration structures and should scale well (logarithmic).

The material query algorithm is shown in Figure 3.10. An ordered list of surface intersections is generated by iteratively ray tracing and adding the closest surface number to the hit list. Tracing is terminated when a predefined outer cell (that contains all other cells) is intersected. Since all surfaces are closed, the ray will intersect any cell surface twice that it isn't nested in. When the list is made, the double entries are removed, which yields a list of cells the neutron is nested in. The first entry will be its current cell and the second entry will be the cell it is entering into.

An issue with ray tracing is that mathematical cell descriptions are exact, but the numbers representing them are not. If these numbers are treated as exact, a situation can occur where a neutron is placed at a boundary but is actually slightly behind the boundary because of floating-point roundoff. When the next trace is started, the ray intersects the boundary it has already intersected instead of tracing into the next cell. This situation can be prevented by using a scene "epsilon," which determines the minimum intersection distance possible, i.e. the minimum distance away from the source point at which intersections are allowed to occur. Giving OptiX a scene epsilon helps guarantee that a trace starts after the boundary, and accurate results are calculated. It is important to make the scene epsilon an appropriate value for the geometry if this algorithm is to be used effectively.

Inaccurate material queries can also due to the scene epsilon. This can occur when intersecting the extreme corner of a box, for example. If the thickness of the object is less than the scene epsilon, the material query algorithm may only count a single intersection of the object, thus determining the box's material instead of skipping the box and determining the correct material. In many cases this can be avoided by performing the surface intersection in the neutron direction and then performing the material query in the z-direction only. Currently, the objects in WARP are all various kinds of z-aligned prisms (or spheres) and corner cases like this will not happen if the material query is done in the z-direction because the ray will only encounter planes perpendicular to it.

The last problem that can result from this algorithm is when cells have coincident surfaces. In this case, which cell is actually intersected is undefined and the next trace iteration will skip the intersection of the coincident cell (since it will be smaller than the scene epsilon value). A way to avoid this would be to ensure the desired coincident surfaces are more than a scene epsilon away from each other. In some cases, the neutron mean free path is much larger than the introduced gap, and this approximation will not change the results. Conversely, it could introduce errors in cases where the mean free path is smaller than the gap (e.g. in cells next to strong absorbers). Doing this by hand would be tedious, however, so an automatic way of doing this may be an area of future development in WARP.

This algorithm is very similar to the ray casting point-in-polygon (PIP) algorithm, which determines whether a point is inside or outside of an arbitrary polygon by counting the number of times it crosses a surface. An even number means it is outside and an odd number means it is inside. This algorithm is almost the same, but keeps track of cell numbers instead of binary logic for one surface. This way, the nesting of a neutron can be determined and the entering cell can be found. Both of these algorithms take advantage of the cell being closed, and in this sense are similar to Gauss's Law or the divergence theorem, which states

the flux integrated around a surface will be nonzero only if the surface contains a source. This material query algorithm is like a discrete, single field line version where the neutron's position is the source point and the cell boundaries are the integrating surfaces.

An important consideration in the type of geometrical representation used in WARP is that the volume of a cell is *always* the spatial intersection of the space inside of the cell with space outside any cells nested inside it. For example, if two cube cells were specified to be centered at the origin with cube 2 completely encompassed by cube 1, the space in-between cube 1 and cube 2 would belong to cell 1 while the space inside cube 2 would all belong to cube 2.

Instancing

Nuclear reactors can have very complicated geometries, but many rely on simple shapes that are repeated in arrays. There are a few ways in which identical cells can be instanced in OptiX. The first, and most convenient, is to define a single primitive in its own coordinate system, then use a transform node (shown in Figure 2.29) in the OptiX node graph to transform the primitive to its actual position via an affine transformation matrix. The resulting node graph is shown in Figure 3.11 for a scene that has three boxes in it. Note, a GeometryInstance object ties a hit program to the spatial geometry data in the geometry primitive object, acceleration objects are attached to all group objects, and transform objects can only have a *single child* that must be either a Group or GeometryGroup object. This is why each GeometryInstance must have its own GeometryGroup.

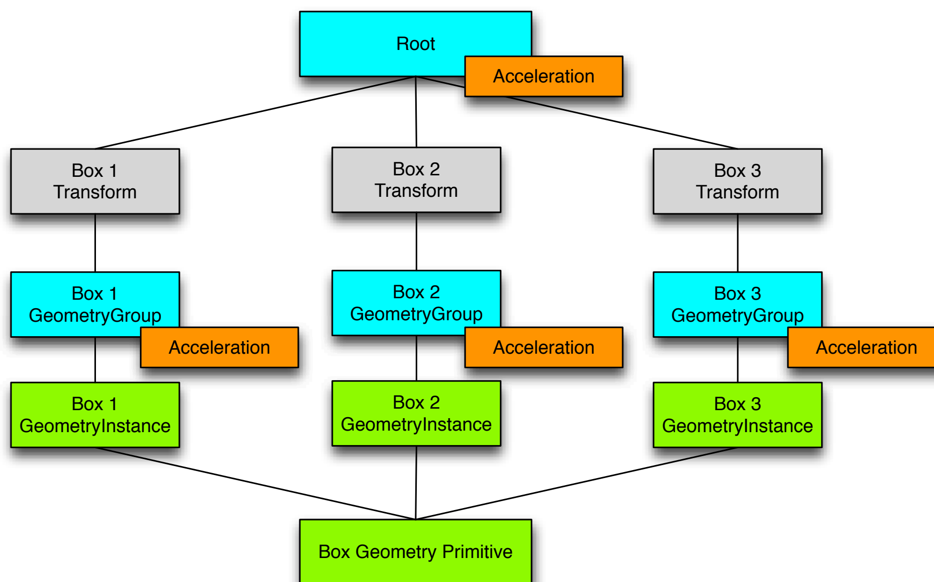


Figure 3.11: The OptiX node graph using transform instancing

Transform instancing is convenient since only a single primitive needs to be defined. If another instance is needed, one can simply apply a transform matrix to it and all the work is done by OptiX. This scheme has a lot of overhead, however, since each instance has its own group and its own acceleration object, producing a deeper node graph than necessary.

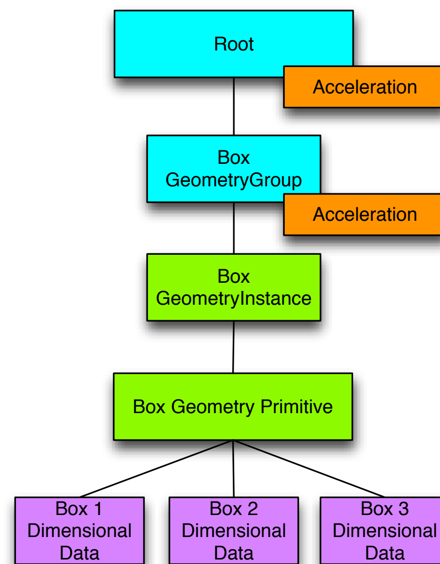


Figure 3.12: The OptiX node graph using mesh primitive instancing.

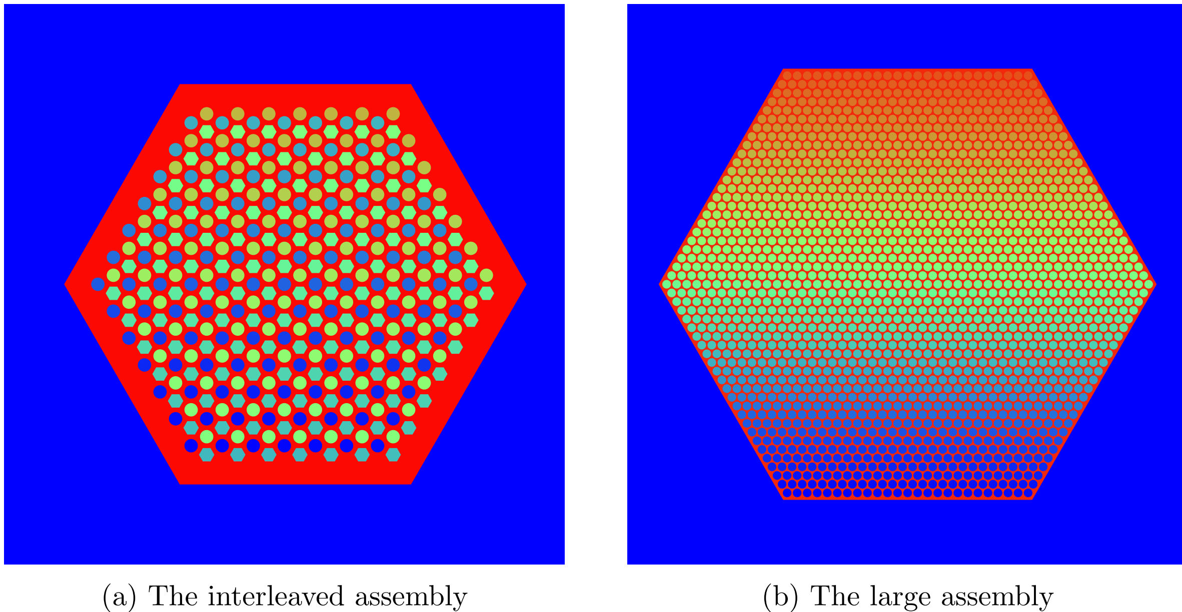
An alternative instancing method uses mesh-based primitives. In this scheme, there is still a single geometry primitive, but now the primitive is attached to an array of spatial data (and OptiX buffer) that contains the dimensions of each individual primitive. The transform node is no longer used since it would transform the whole group instead of a single primitive. The transforms must be applied to the data beforehand and the results are written into the OptiX buffer as separate elements.

This method produces a shallower node graph with only two acceleration objects and a single GeometryGroup for *all boxes*. These numbers would not change if there more boxes in the scene, only the number of data elements on the bottom of the graph would change. This is called mesh instancing since the data structure was envisioned for the many triangular surfaces in meshes of complex objects rather than for individual instancing of separate, simple objects.

Test Geometries

Figure 3.13 shows the geometry for the interleaved assembly and the large assembly cases used in the first scaling study. These figures were created by OptiX by performing a cell number query as prescribed in the previous subsection. The interleaved assembly consists

of three hexagonal arrays, two cylinders, and one hexagonal prism. The arrays are seven elements on a side, which corresponds to 127 elements each for a total of 383 elements (including the large hex cell around the arrays and the outer box cell). The large assembly has 25 cylinders on a side, for a total of 1803 elements. The smaller assembly is 15 cylinders on a side, for a total of 633 elements.



(a) The interleaved assembly (b) The large assembly

Figure 3.13: x-y cross sections of the geometry created by using the PIP cell/material query algorithm

The cell numbers are mapped to colors in Figure 3.13. The images appear correct upon visual inspection and no tracking errors were generated when creating these images, it therefore appears the routine is working and the algorithm can calculate the cell numbers. A helpful feature of OptiX is that variables can be attached to each geometry primitive. In addition to the cell ID number, the materiel ID is also attached, so the material query can be done directly by OptiX rather than by determining the cell number and then having to do an additional lookup or hash.

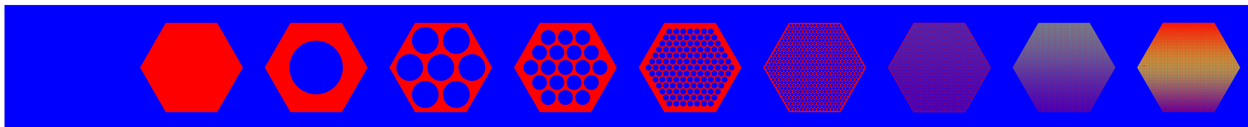


Figure 3.14: The geometries used in the scaling study from least number of objects on the left to most on the right.

Figure 3.14 shows the geometries used in the second scaling study. The hexagonal cell has the same dimensions as the assembly-like geometry case in the first scaling study, but

each successive case has smaller and more numerous cylinders in a hexagonal array inside the hexagonal cell. The most sparse scene only has a single primitive in it, shown on the left side of Figure 3.14, and the most complex scene, shown on the right side of the figure, has 42,843 primitives in it.

Results

Both of the scaling studies were run on an NVIDIA Tesla K20 card. Cell queries were done with the PIP algorithm from a uniformly random and isotropic distribution of source points in addition to finding the closest intersection point. Figure 3.15 shows the ray trace rates versus number particle starting points for the three test geometries. There are cases for each geometry (assembly, large, and interleaved), whether the trace uses primitive or transform instancing (prim or xfrm), and whether it uses a split bounding volume hierarchy or regular bounding volume hierarchy (SBVH or BVH). K-d trees are not used since they require a mesh vertex buffer be provided by the user, which implies that they only work for triangularly-meshed objects, not ones instanced by simple geometric primitives like spheres and cubes. Handling geometries that are meshed in such a way may be a future area of development in WARP.

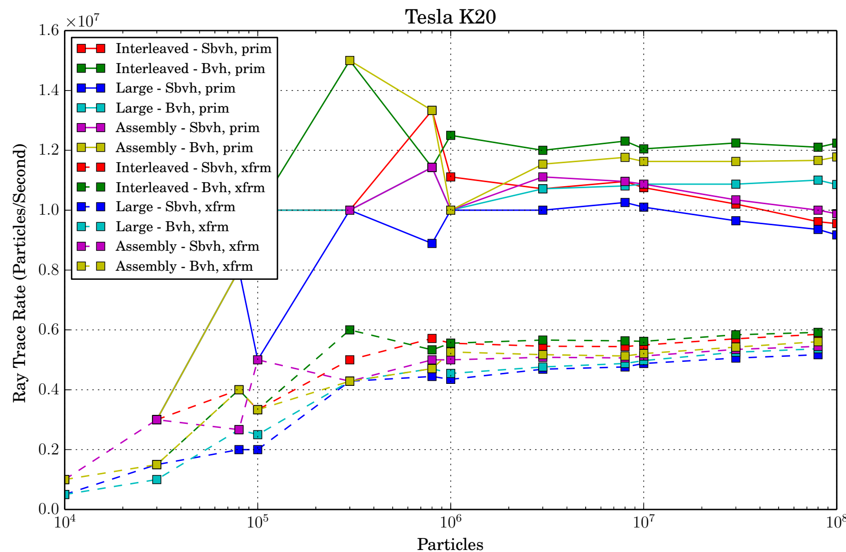


Figure 3.15: Trace rates of an NVIDIA Tesla K20 performing cell queries with the PIP algorithm.

The trace rates are fairly constant after 10^6 particles, but primitive instancing is always faster than transform instancing, and using BVH acceleration is always faster than SBVH. The final data point at 10^8 starting points is missing for the transform instancing because the card ran out of memory and these cases would not run.

The number of source points used in the second scaling study was set to 10^6 and 10^8 to see if the trace rate scales with object number differently if done on the extreme edges of the trace rate plateau shown in Figure 3.15. A BVH acceleration structure was used since it showed the best performance in all cases. Figure 3.16 shows the results of the scaling test on an NVIDIA K20 card. It can be seen that the ray trace rates for both primitive and transform instancing plateau after about 20 objects in the scene. Primitive instancing also always outperforms transform instancing. It should be noted that the trace rates for the two dataset sizes start quite far apart for one object then converge to equal performance qucikly as more objects are included in the scene. This may be because the trace becomes more dependent on acceleration traversal rather than output buffer access (where smaller, slower sizes makes a difference when there are few objects).

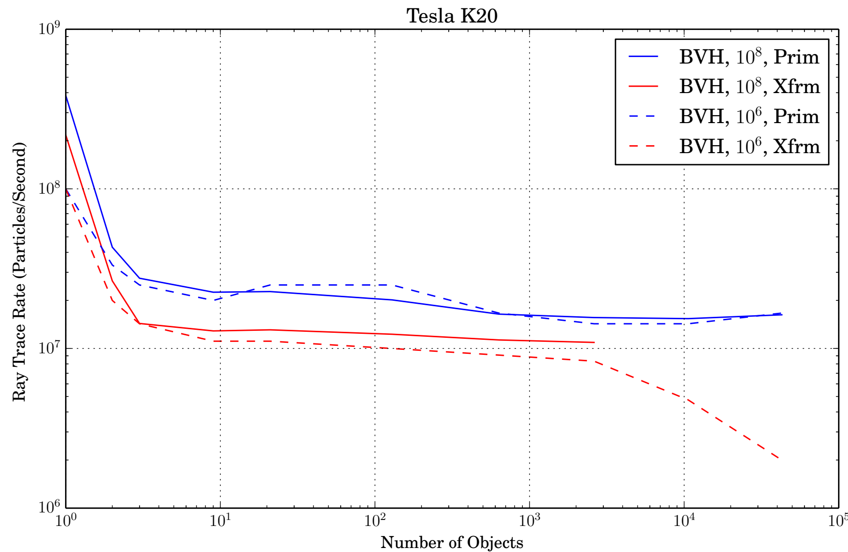


Figure 3.16: Trace rate scaling on an NVIDIA Tesla K20 performing cell queries with the PIP algorithm.

Using transform instancing fails for the 10^8 dataset size when somewhere between 2613 and 10,623 objects were included in the scene. This is why the transform instancing trace stops at 2613 cells in Figure 3.16. The most likely reason is because transform instancing requires a large amount of memory, and the card runs out of memory when a large particle dataset is also present on the card. This is also most likely why performance of the transform instancing method drops at the same point for the 10^6 dataset size. At this point, OptiX started paging GPU memory to the host memory, and this degraded performance significantly. Structure construction for transform instancing prior to the trace also became very slow compared to primitive instancing, presumably due to the independent acceleration objects attached to every instance and the overhead of computing all the affine transformations at acceleration structure build time.

Figure 3.17 shows the trace rate versus dataset size test, but run on an NVIDIA GeForce GT 650M (the discrete graphics card in a MacBook Pro, Mid-2012 Retina model) instead of a Tesla K20. The same trends can be observed except that overall trace rate is much slower, which is not surprising, considering the 650M has 1/5 of the memory and 1/7 the number of CUDA cores as the K20. The impact of transform instancing is very pronounced in runs with fewer particles. This study was done to compare the performance of a smaller non-compute card compared to a larger compute card.

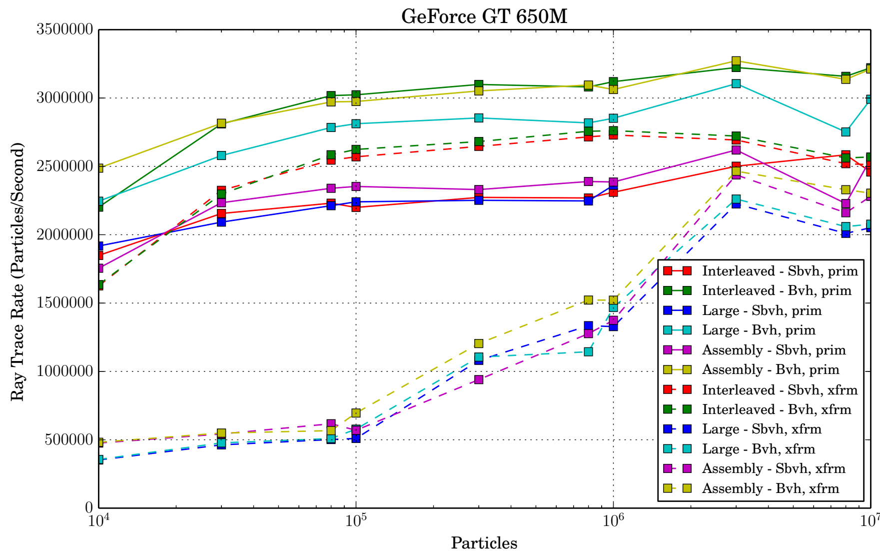


Figure 3.17: Trace rates of an NVIDIA GeForce GT 650M performing cell queries with the PIP algorithm.

From the scaling studies it can be concluded that OptiX can be used to handle the geometry representation in a Monte Carlo neutron transport code. Using a primitive-based geometry instancing method, a BVH acceleration structure, and running as many parallel rays as possible provides the best performance. The OptiX ray trace rate also seems to be insensitive to the number of objects in a scene after about ten primitives.

3.2 WARP in Detail

Thus far, no details about WARP's transport routines have been discussed. The preliminary studies showed that thread divergence can be effectively reduced with CUDPP without incurring prohibitive costs, that large datasets need to be run to hide memory latency, and that a BVH acceleration structure and primitive instancing in OptiX perform best for randomly-oriented distributions of rays. The mathematical and physical background information relevant to the Monte Carlo method was covered in Chapter 2. This section builds

on the previously presented information to explain the implementation details of WARP's neutron transport subroutines.

As stated before, WARP is designed to read ACE-formatted data, perform all reaction types as prescribed by the data, use a Serpent-like unionized energy grid to regularize data access, use an event-based transport algorithm with parallelized operations for sorts and sums, use OptiX for general 3D geometry representation (without explicit nesting), use an SOA for neutron history data, and perform all operations on the GPU unless strictly forbidden. In terms of implementation, OptiX is used for 3D geometry representation; physics and tally routines are written in CUDA; the CUDPP radix sort is sued to create the remapping vector; and the CUDPP prefix sum is used to augment CUDA routines in running criticality and fixed source simulations.

The host-side code in WARP is written in C/C++ with some Python (which will be explained later in this section). Single precision floating point numbers are used throughout in order to realize the full computational capacity of the GPU and to allow simulations to be carried out on more affordable and higher clocked GeForce cards. Using single precision numbers may be dangerous when there are very dilute isotopes or very rare reactions reactions present, as roundoff error may make their contributions zero. Buffer overflow and roundoff error in the tallies may also be a problem with single precision, but this can be mitigated by accumulating the tallies frequently in a double precision vector. Roundoff error may also be problematic in calculating the multiplication factor, as progressively smaller and smaller numbers are accumulated into the multiplication factor value. If double precision data is found to be needed, WARP can easily be changed, and doing so may be an interesting experiment in the future.

Data Layout

As described in Chapter 2, the nuclear data required in Monte Carlo simulations are very heterogeneous and the way the data is accessed in a Monte Carlo simulation is random. This section describes the methods used to deal with the data heterogeneity and how data access in WARP has been optimized for the GPU.

Unionized Cross Sections

A material's macroscopic cross sections dictate the reaction probabilities as a neutron travels through it. In order for the reaction probabilities to be computed from the macroscopic cross sections, the material's total macroscopic cross section (the sum of all the isotopic macroscopic cross sections of a material) must be known. This value is used to normalize the isotopic macroscopic cross sections to one, thus making them probabilities. The formula for doing this was shown in Eq. (2.64).

The atomic densities change according to the material the neutron is traveling through, and WARP recomputes a material's total macroscopic cross section at every interaction. This is done in order to not store preprocessed macroscopic cross sections for each material. If

macroscopic cross sections are computed on-the-fly, a single set of microscopic cross sections and a small vector of material number densities can be stored instead. Of course, recomputing the macroscopic cross sections at every inner transport iteration adds additional work, but if the material densities are stored in the GPU's shared memory, they can be accessed quickly and an equivalent number of global loads are needed for loading the microscopic cross sections as would be need for loading the preprocessed macroscopic cross sections. If the material total cross section is precomputed and stored, this may save global loads at the cost of increased memory usage.

Since cross sections need to be interpolated between energy points, It is particularly troublesome that each cross section has its own independent energy grid. Using point-wise data for continuous energy simulations requires an interpolation be performed between points, and to do this, the code must somehow scan the energy grid array to find the points between which the neutron's energy falls. If every isotope has its own grid, this search must be done for every isotope, and can become very expensive. This is why a unionized energy grid structure, like that implemented in Serpent, is used in WARP [55].

Unionizing the cross sections means that the energy grids of the cross sections are all unionized into a single, larger structure that contains the energy points of every isotope. Including every energy point ensures that no information is lost in the unionizing process. Since the unionized grid is then used to index every isotope's cross section vectors, there are energy points which do not correspond to values in the original nuclear data. These holes in the data are filled by linear interpolating between the closest two grid points that have values. The unionization process transforms many 1D reaction cross section vectors into a single 2D matrix indexed by incoming energy and MT reaction number. Figure 3.18 shows the unionization process with two small, arbitrary energy grids and their corresponding cross section vectors. The colors in the unionized energy grid represent which isotope the grid value came from, the green cross section data blocks are interpolated values, and the red blocks are placeholder zeros to preserve thresholds. It is clear that the resulting dataset is larger than the sum of the individual cross sections since it contains redundant data, but it will be much easier to search.

The interpolated data is redundant in the sense that it contains no new information. If a cross section was required between two data points of the original grid, linear interpolation would be performed to calculate it. The filled-in holes of the unionized dataset contains values on a straight line which could be calculated using the original data. The tradeoff between regularizing the data in this way, as opposed to accessing the energy grids separately, is worthwhile because only a single grid search is needed to find the vector indices a neutron's energy falls in between.

Because the cross sections unionized in this way the cross section values for every isotope can be read along a single row once the bounding energy indices, $E_i < E < E_{i+1}$, are found for a given interaction. This format reduces the number of energy searches needed and promotes data locality by keeping all the data points for all isotopes in a given energy range adjacent to one another in memory (if row-major matrix format is used). On CPUs, the data for an energy value can be read as a cache line and accessed quickly by the cores. On

Figure 3.18: Unionizing two cross section vectors.

a GPU, since a thread needs to read the entire energy line (one energy over all isotopes) in sequence in order to compute the macroscopic cross section on-the-fly, the memory loads are not coalesced and memory bandwidth is wasted (adjacent threads do not access adjacent memory at the same time).

To mitigate this inefficiency, the data can be recast to use the float4 datatype, which is a vector type that is 16 bytes long. When requested, the entire 16 bytes will be loaded in one transaction rather than 4 separate 4-byte transactions, maximizing bandwidth and minimizing the number of global requests (and reducing the impact of latency). Also, since all of the total cross sections for a material are all needed at the same time, they are stored together in the first columns of the unionized data array. After this block of total cross section columns, the individual reaction cross section are stored as blocks in the same material order as the total cross section block (all the cross sections for an isotope are in a single block). This is done so the macroscopic or microscopic kernel needs only scan a single, contiguous

block of data in the array. Figure 3.19 shows an illustration of the unionized cross section dataset structure used in WARP.

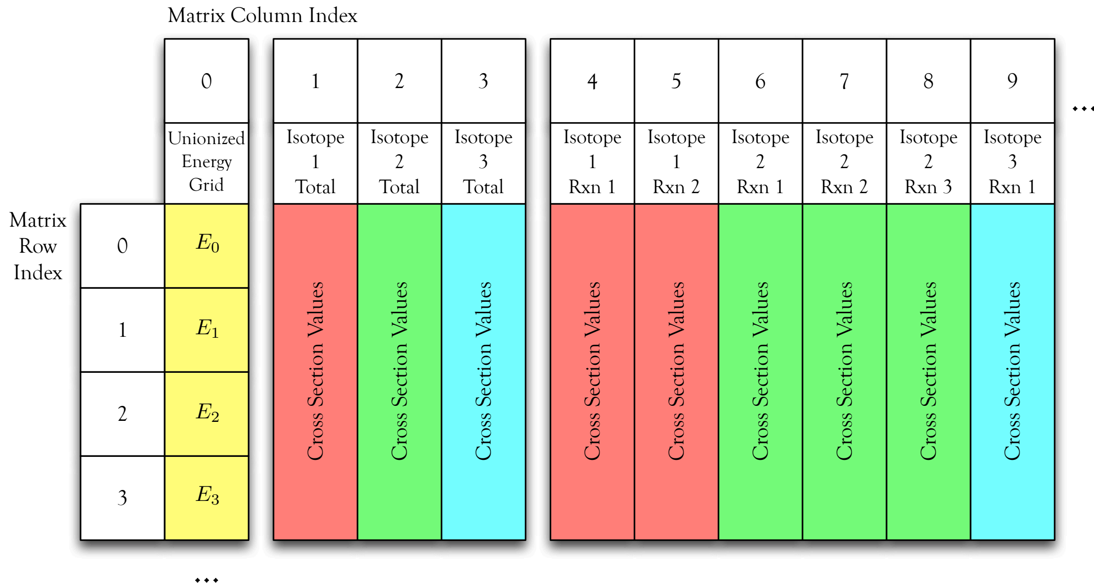


Figure 3.19: The unionized cross section layout used in WARP.

Distribution Data as a Linked List

The cross section data can be regularized with the unionized grid method, but there are other data distributions that WARP needs to conduct accurate simulations. The most prevalent of which are scattering matrices that give tabulated probability distributions for outgoing CM angle μ and incident neutron energy. Since these matrices have their own incoming energy grid, these values can be unionized into the cross section dataset as well. This transforms many 2D scattering matrices into a 3D matrix indexed by incoming energy, outgoing μ, and reaction MT number.

Since the energy grid of the scattering matrices is often much, much coarser than the main cross section energy grid, this operation produces a *very large* amount of redundant data. It was attempted with ^{235}U, and resulted in a 3D matrix that was 6.4 GB (30,991 energy grid points, 598 μ CDF points, 51 reactions). This is just for a single isotope's angular distributions of all reactions, and the dataset already cannot fit on the on-card DRAM on most GPU cards.

After determining that unionizing with the main energy grid was an unacceptable method, another approach was tested where the energy grids of the scattering matrices were unionized between themselves, but not with the main energy grid. This would reduce the amount of unnecessary data, but would require an additional search be made on the unionized scattering

matrix energy grid. The memory usage of this method with ^{235}U was reduced to 12 MB (103 energy grid points, 598 μ CDF points, 51 reactions). When tested with ^{235}U and ^{238}U, usage for the scattering data increased to 68 MB (171 energy grid points, 1072 μ CDF points, 97 reactions). When tested with ^{235}U, ^{238}U, ^{16}O, and ^{1}H, usage became 1.1 GB (1325 energy grid points, 1237 μ CDF points, 170 reactions). Therefore, this method was also deemed too costly for systems with a practical number of materials. The distributions are therefore left in their original formats for use on GPUs, where storage space is much more limited compared to CPUs.

Figure 3.20: Making a link to distribution data in the unionized dataset.

After these failed attempts for resolving the scattering data heterogeneity problem, a method was formulated to eliminate the secondary energy search without having massive data replication. This method introduces two new matrices identical in size to the unionized cross section matrix, but instead of containing data values, they contain pointers to the location of the appropriate distribution data for the reaction at the energy index determined by the grid search. Two matrices are needed - one for the scattering distributions and another for the energy distributions. This way, the unionized dataset is only replicated 3 times, the distribution data can be copied to the card in its original format, and only a single gird

search needs to be performed. This is similar to a linked list, where a there is a pointer at the end of an object pointing to the next object in a list.

Structuring the arrays like this means once a reaction is sampled to occur, there is a pointer readily available at the same index in a second matrix to the appropriate distributions, and no searching needs to be done. Since the distribution PDFs need to be read serially by each thread, the float4 format can also be of use here. If there are no distributions for a particular reaction or energy, a null pointer is inserted into the matrix. For fission reactions that do not have scatting distributions (neutron emission can be assumed to be isotropic [13]), the ν value for the grid energy is stored in the scattering pointer matrix instead of a pointer. This way a search for the appropriate ν value does not have to be performed either. Figure 3.20 shows a pointer matrix for the unionized cross section matrix in Figure 3.18.

Serpent stores pointers in its unionized cross section layout [12]. The pointers in Serpent's layout point to arrays that contain contain both cross section and distribution data. This way, only a single unionized array needs to be stored as opposed to the three needed by WARP. Since Serpent's layout only contains pointers, however, the pointer needs to be loaded before any cross section values can be loaded. This means that for every cross section loaded, a pointer needed to be loaded as well. WARP's layout stored the cross section values directly in the layout, so only cross sections are loaded during the isotope and reaction sampling routine instead of pointers and cross sections. A pair of pointers is only loaded when when the reaction has been selected and the angular or energy distribution data is needed. Such a layout may use more space, but reduces the amount of memory traffic needed to sample reactions.

Embedded Python

Now that the data layout has been discussed, how the data is loaded or reformatted from ACE-formatted data files will be explained. An initial effort was made to write an ACE-parsing script in C from scratch, but this was abandoned in favor of using the existing ACE module of the PyNE (Python for Nuclear Engineering) package [56] (why reinvent the wheel?). The PyNE package "is a suite of tools to aid in computational nuclear science & engineering. PyNE seeks to provide native implementations of common nuclear algorithms, as well as Python bindings and I/O support for other industry standard nuclear codes." [56]. PyNE contains a Python module that can parse ACE data files into Python objects that can be more easily handled than flat data arrays. This module was written by Paul Romano as a preliminary project for OpenMC, whose ACE library was later written in Fortran based on the methods developed in this Python module [56, 13].

Some parts of PyNE have a C++ application programming interfaces (APIs) as well as a Python API, but this is not the case for the ACE module. It was originally written in Python, not C++, so it has no C++ API. This created a problem for WARP since WARP is written in C/C++ and could not directly make use of the ACE module. Fortunately, there is a very effective C API for Python! This API allows one to initialize and run a Python instance from a C program. In WARP, this API is used to start a Python instance where

PyNE can be used to load cross sections from ACE data files. NumPy [57] is then used to unionize the energy grids of the requested isotopes and perform the linear interpolation to fill in the gaps. Once this is done, the Python instance returns the NumPy array to WARP as a C data structure. The data is copied to an internal array and the Python object is cleared.

WARP then loops through the unionized array, requesting distribution data from the Python instance. The scattering and energy distribution data are also copied for the requested energy range in a similar fashion, and a device pointer for the distribution is written into the scattering and energy pointer matrices. When all the needed data has been copied over to WARP, the Python instance is terminated and its memory freed. Since the unionized energy grid is an index of the matrix that needs to be searched before a row of the unionized cross section matrix can be accessed, the energy grid is stored as its own contiguous array rather than a column of the matrix. Figures 3.18, **??**, and 3.20 show the energy grid as a column simply for illustrative purposes.

The unionized cross section dataset in WARP is resident in the global memory of the GPU. Storing it in constant memory would seem to make sense, but since it is limited to 64kB, the entire dataset simply cannot be stored here. Texture memory was also considered, as this could possibly contain all the data, but it is optimized for 2D data locality, whereas the dataset format is mostly accessed in a linear fashion across rows and randomly across columns (since energy-changing reactions are sampled randomly).

Both the constant and texture memory spaces are cached, so the random access inherent in Monte Carlo simulations may make cached access worse than non-cached because of cache miss penalties. This effect was not studied in the initial development of WARP, however. Nelson reports using the constant memory space in his simulations, with little-to-no performance improvement [28]. Using the texture memory to store the cross section matrix might benefit from the free linear interpolation that can be performed with the texture element load, however. With all of these points considered, it was decided to store all data, whether it be for history data or cross sections, in global memory in WARP.

In its current state, WARP does not use the thermal scattering ($S(\alpha,\beta)$) tables or unresolved resonance parameters. These tables improve the physical fidelity of the simulation, but these features can be turned off in production codes and direct comparisons can be made without them. Their incorporation may lead to more divergent program flow and is left as an area of future work.

Python Wrapper

Work is also underway for wrapping the WARP shared library with Python. This would be done via SWIG [58], a piece of software that automatically wraps compiled languages like C/C++ in high-level scripting languages. This is being done for convenience and usability reasons. With the C++ classes exposed in Python, the main() function can be replaced with a Python script, eliminating the need to recompile WARP applications when different geometries or different run parameters are desired. In its current state, WARP is compiled to

a shared library with an API. This requires a small main function to be written to make an executable that calls the WARP library routines. The library does not need to be recompiled, but the main function and executable does for the simulation parameters to be changed.

The Python wrapping approach deviates from the standard flat text input file structure that most Monte Carlo codes use. Flat text input relies on keywords and adds a layer where input files need to be parsed and data structures are then built in the application based on the information parsed from the input. Using Python to directly access the classes and their data removes this layer, and allows a user to build complex applications. Since the results would also be resident in a Python session and would therefore be easily available to the user for potting scripts or analysis tools. To process data in the same way from text-file-based output, the output needs to be parsed with a user written function or processed by hand, which is time consuming and can lead to human error.

CUDA Kernels and Data-Parallel Tasks

Now that it has been discussed how the necessary data is loaded and reformatted in a GPU-friendly way, the details of the actual process of transporting neutrons will be described in this section. Neutron transport consists of an inner and outer transport loop. The inner loop consists of routines that are needed to process a batch of neutrons and the outer loop consists of routines that are needed to connect neutron batches to each other. Both loops are discussed in shallow detail here enumerate the routines that comprise them and to establish this section's structure. In-depth descriptions are given on each of the individual routines later in this section.

The inner loop, shown in Figure 3.21, actually transports the neutrons through the problem geometry and samples the reaction CDFs. The blocks in the figure represent independent kernel launches, and are executed left-to-right. The quantities listed under the routine names are what the routine calculates. The block color corresponds to the library used to perform the task. The first step of the inner loop is using OptiX to perform the material query, since material information is needed to query the proper cross sections and to determine the distance to the nearest surface along the neutron's path of travel. Once this is known, a kernel is launched to do a search on the unionized energy grid to determine the grid index i that satisfies $E_i < E < E_{i+1}$. Next, the macroscopic kernel is launched. This kernel computes the total macroscopic cross section for the material, samples the interaction distance, samples which isotope the neutron interacts with, moves the neutron to the nearer of the interaction/intersection distance, and sets the neutron's reaction number to the resample flag if the intersection is closer. Since the macroscopic cross section has been computed at this point, the tally kernel is launched and scores a specified flux tally. After the flux is scored, the microscopic kernel is launched to determine which reaction type in the determined isotope occurs. The radix sort is done next since the reaction has been determined, and this operation sorts the neutrons by reaction type and updates the values in the remapping vector. The next four kernels are launched concurrently, as indicated by being in the same horizontal position in Figure 3.21. Each of these kernels performs the

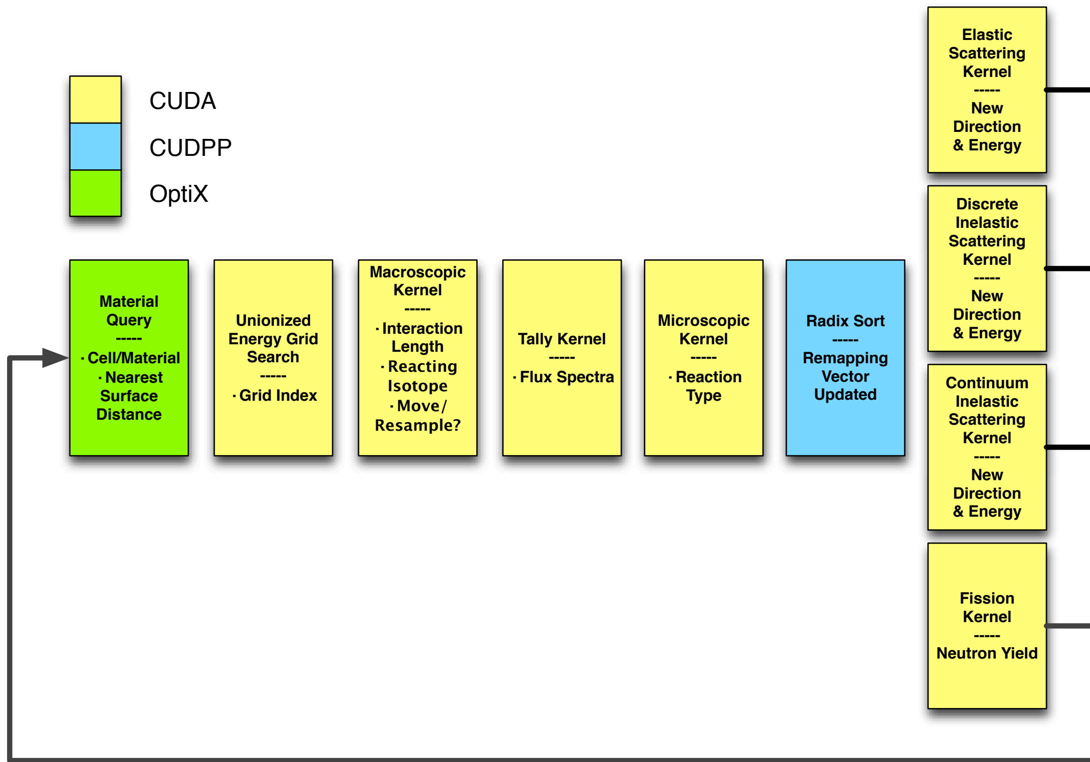


Figure 3.21: WARP inner transport loop that is executed until all neutrons in a batch are completed.

specific functions necessary to model their different reactions, and since a neutron cannot undergo two reactions simultaneously, the data they access does not overlap and these kernels can be launched concurrently. If a neutron's reaction number is the resample flag, it skips the microscopic and reaction kernels and isn't processed again until the material query, which resets the reaction number to zero. This loop cycles until all neutrons in a batch are terminated through absorption or leakage.

The outer loop, shown in Figure 3.22, uses the result from the inner loop to set up the next batch of neutrons. The inner loop serves to determine the yields of the batch's neutrons. The yield array is then reduced to determine the multiplication factor, k_{eff}, of the batch. The yields are then divided by k_{eff} to make the yield as close to one as possible without biasing the fission source distribution. Since dividing by k_{eff} is unlikely to yield an integer, the rebased yield is sampled between the bounding integers to preserve the mean across many neutrons.

After the yields are rebased, a prefix sum (also called a scan) is performed on the yield array. The value for each particle's element of the prefix sum is the total number of fission neutrons before it. Since this is known, the "pop" routine can insert the sampled fission neutrons into the next batch's history vector with no risk of writing into the same element.

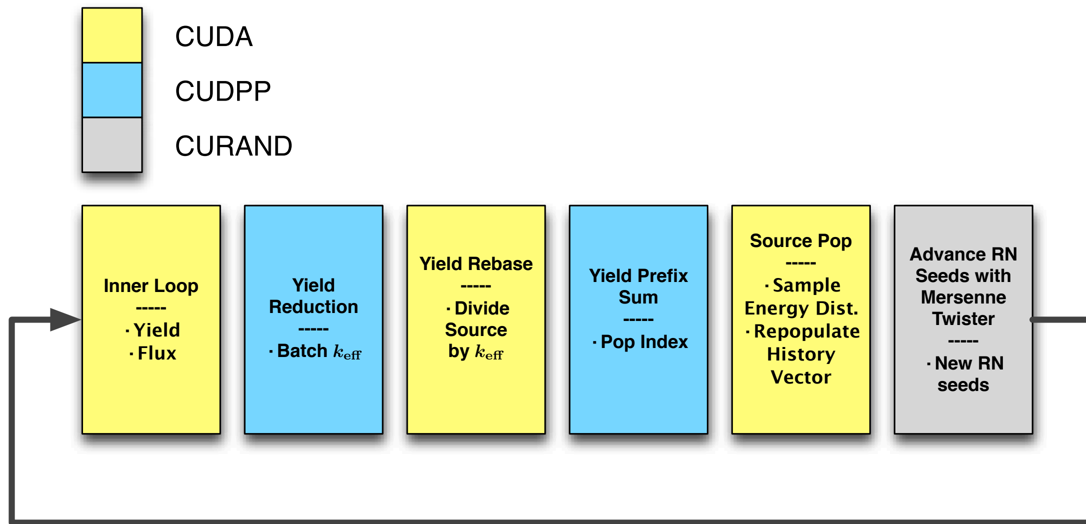


Figure 3.22: WARP outer transport loop that is executed in between neutron batches for criticality source runs.

Since the yields were rebased to make $k_{\text{eff}} \approx 1$, the yield vector will contain enough secondary neutrons to fill the next generation's entire history vector. The pop kernel inserts neutrons into the history vector at points where fissions occurred, but with individually sampled energies and (isotropic) directions. After the pop completes, the inner loop transports this newly initialized batch of neutrons.

Fixed source runs do not have an outer loop since the source is not dependent on the flux. If there is enough memory available, the entire number of requested neutrons are initialized based on the source definition. The neutrons are transported simultaneously, and any secondary neutrons produced from fission or (n,2/3/4n) reactions are popped into the history vector at the end of the inner loop to be transported with the original source particles in the next pass of the loop.

Grid Search Kernel

Much effort has gone into ensuring that only one search on the main unionized energy grid is required to find the index i that satisfies $E_i < E < E_{i+1}$, but an algorithm to perform the search itself has not yet been established. It is convenient that the values in the energy grid are monotonically increasing, but since the array is composed of floating point values with arbitrary spacing, an inverse function cannot made to calculate an index in $O(1)$ time. Conversely, a naïve approach would be to scan the array from beginning to end, performing comparisons along the way, and therefore calculating the cross section array index in $O(n)$ time, where n is the number of elements in the array. This method does not scale well and will not be considered.

An extreme method taking advantage of the space-time tradeoff would be to use a simple lookup vector where the points are linearly spaced. In order to include all original data, the spacing of the lookup vector, dx, would have to be smaller than the smallest found in the data. The lookup vector index could simply be calculated by $i = E/dx$, and cross sections could be searched in constant time. The spacing around resonances in the nuclear data is very small, however, and makes this method's memory usage unacceptably large. For uranium-235, the smallest difference is 10^{-12} MeV over a range of 10^{-11} to 20 MeV, indicating that the vector would need to be around $20/10^{-12} \approx 10^{14}$ elements long. Logarithmic spacing could also be used, but again assuring that all original data is included may make the regular grid too fine, resulting in unacceptable memory use.

Another approach could be to use a high order polynomial approximation of the unionized grid for the initial guess of an iterative method. This would be similar to the Serpent pointer search method, but would eliminate the pointer vector and use the analytical computation to find an initial index in the main grid instead of computing the index of the pointer vector.

The binary search is the classic search algorithm for searching irregular data. Instead of progressing through the array from beginning to end, it is continually bisected until the correct interval is found. In other words, the search begins at the center of the vector. If the searched-for value is less than the middle value, the next loaded value is the middle value of the first bisection. If the searched-for value to greater than the middle value, the next loaded value is the middle value of the second bisection (of the first bisection). This process repeats iteratively until the value is found. The binary search algorithm is attractive because it runs in $O(\log_2(n))$ loads/comparisons in the worst case, and uses no storage except for the loaded and comparing values [59]. WARP currently uses a simple binary search algorithm because of this algorithm's logarithmic scaling and ease of implementation. It will be determined if the grid search is a significant portion of the simulation time, and if it is, WARP could benefit by implementing more advanced searching algorithms in the future.

Pseudo-Random Numbers

Random numbers are the basis of the Monte Carlo method, and generating them must be done efficiently. Since these numbers must be computed, they are not strictly random, but rather pseudo-random since they are randomly distributed but can be deterministically computed. NVIDIA provides a random number library called CURAND that can generate large arrays of randomly-distributed numbers [60]. Initially, WARP precomputed all the random numbers it would need for a single transport loop iteration. This could be done since direct sampling methods were used everywhere (which was incorrect for the target velocity in scattering) and there was a known maximum number of numbers required. The threads would simply access random number values stored in this giant array (about 20 numbers were needed per thread). This is a very bandwidth-intensive way to access random numbers since they are always written to and read from global memory. In these early implementations, the random number generation and access was taking about a quarter of the transport loop time. In addition, the introduction of the correct rejection sampling method for the target

velocity disqualifies precomputation since there was no longer a predetermined number of numbers needed.

Taking these points into consideration, it was decided to use a linear congruential random number generator (LCRNG) for computing random numbers in the transport kernels. It allows the kernels to generate random numbers with the recursion relations shown in Eq. (3.2), where x_{n-1} is the previous random number. The values a, c, and m are taken from Pierre L'Ecuyer's manual for configurations with a high figure of merit [61].

$$\begin{aligned} x_n &= a x_{n-1} + c \mod m \\ a &= 116646453 \\ m &= 2^{30} \\ c &= 7 \end{aligned} \tag{3.2}$$

OpenMC uses a similar LCRNG implementation but with different values for a, m, and c since it uses double rather than single precision. The values were chosen close to the full 32-bit range for maximum floating point resolution. Having a modulus as a power of two is very convenient as well since the modulus can be performed by bit truncation rather than a full modulus operation [13]. In other words, the operation can be done by a bitwise AND operation between $ax_{n-1} + c$ and $2^{30} - 1$, which in binary is all ones below the 30th bit.

Using a LCRNG keeps global access down and improves performance greatly, since a single seed value is loaded at the start of the transport loop, stored in fast registers, then written back at the end of the loop. CURAND is used outside of the LCRNG cycle to advance the seed bank with the Mersenne Twister algorithm (which has a period of $2^{19937} - 1$) to make sure no correlated numbers are used as intermediate seeds.

Macroscopic Cross Section Kernel

The macroscopic kernel in WARP computes all values related to macroscopic cross sections. These include total macroscopic cross section for the material, interaction distance, which isotope the neutron interacts with, moving the neutron to the nearer of the interaction/intersection distance, and setting the neutron's reaction number to the resample flag if the intersection is closer.

Since the isotopes' total cross sections are stored together in the unionized cross section matrix, the total macroscopic cross section can scan the contiguous row of this part of the matrix, interpolating the value between i and $i + 1$ (which has been determined by the grid search kernel). The interpolated values are multiplied by the number density vector (in atoms/barn-cm) and the sum is accumulated as shown in Eq. (3.3), where M_k is the number density and $\Sigma_{t,k}(E)$ is the macroscopic cross section of isotope k. When the material vector has been completely scanned and total cross section is computed, the array is scanned again, but this time a random number is used to determined which isotope the neutron interacts with via Eq. (2.64).

$$\Sigma_t(E) = \sum_{k=0}^{N_{\text{isotopes}}} M_k \, \Sigma_{t,k}(E) \tag{3.3}$$

The material's total macroscopic cross section is written to an array so the flux tally routine does not have to recompute it, which would take many more global memory transactions than loading it from the array. Next, the interaction distance is sampled via Eq. (2.63) and is compared against the nearest surface distance, which was computed with OptiX in the first step of the transport loop. If the interaction distance is nearer, the neutron's coordinates are changed to this location. If the surface is closer, the neutron is moved to the surface and its reaction number is changed to the resample flag value, which will cause it to skip all other kernels until OptiX determines the new material and resets its reaction flag. As stated in the beginning of this section, preprocessed values are not stored in memory and WARP recomputes macroscopic cross section at every iteration of the inner transport loop.

Flux Tally Kernel

The flux tally kernel scores a collision in a predefined cell into a bin via Eq. (2.91). Currently, WARP only allows evenly spaced logarithmic bins since their indices can be calculated analytically (as opposed to an arbitrary input grid). The tally index, j, out of N_{tally} bins for a neutron with energy E is calculated via Eq. (3.4). Since multiple threads could be adding to the element located at j, "atomic" operations are used to perform the sum, specifically the atomicAdd() function. Atomic functions perform the load-compute-store operations in a single, uninterruptible transaction. This eliminates the risk of data races occurring.

$$j = \text{floor}\left(\frac{\log\left(\frac{E}{E_{\text{min}}}\right)}{\log\left(\frac{E_{\text{max}}}{E_{\text{min}}}\right)} (N_{\text{tally}} - 1) \right) \tag{3.4}$$

Since WARP uses single precision floats and integers, buffer overflow can be a problem when a tally is scored frequently (or at least more of one than if double precision were used). Because of this, the tallies are copied from the device during each outer loop iteration. The tally values are divided by the total number of source neutrons and are accumulated into double or long integer arrays, as appropriate, on the host. After copying, the device arrays are zeroed. Dividing and accumulating also serves to increase the accuracy of the arithmetic since the tally values are kept within a smaller range of values. Roundoff error is exacerbated by adding small floating point numbers to large ones, which is more likely when simply accumulating all of the tally data without periodically normalizing by particle count. The required intermittent copying during transport increases the device-host communication, but this impact is minimal and can be overlapped with other routines since it is only done once in the outer loop.

Microscopic Cross Section Kernel

Unlike the macroscopic kernel, the microscopic kernel only has one job - to determine which reaction the neutron undergoes in the isotope it has been sampled to interact with. This is done by scanning the isotope's subrow in the unionized cross section matrix as show in Eq. (3.5). $N_{\text{start},k}$ is the starting index of the row for isotope k, $N_{\text{end},k}$ is the index of the isotope's last reaction cross section, and $\sigma_{t,k}(E)$ is isotope k's total microscopic cross section. The starting and ending indices for an isotope are precomputed and stored in a separate array. For most runs, this array and the material isotope density array can be stored in fast shared memory.

$$m < N_{\text{end},k}$$
$$PDF_m = \frac{1}{\sigma_{t,k}(E)} \sum_{z=N_{\text{start},k}}^{z=m} \sigma_{i,z}(E) \tag{3.5}$$

The reaction is sampled by generating a new random number and using Eq. (2.65). Once the reaction m is sampled, the sorting routine is launched to remap data references and sweep completed histories out, then reaction kernels can be launched to carry out the individual reactions.

Interaction Kernels

There are only four reaction kernels since all capture reactions are taken care of in the radix sort, leaving the three different kinds of scattering and fission ((n,2/3/4n) reactions follow laws of continuum scattering but have yields that are taken care of in the pop routine). The scattering reactions change the direction and energy of the neutrons, and their distributions are sampled as outlined in Section 2.6.

The implementation of these methods is where the pointer array becomes useful. Once the row, i, and column, m, of the unionized cross section matrix have been determined, the scattering kernels simply need to access these coordinates in the scattering and energy pointer matrices to load the appropriate distribution data. No additional searching needs to be done.

Concurrent Kernels

As mentioned previously, the reaction kernels should never access overlapping data so that they can all be launched concurrently. NVIDIA cards with a compute capacity of 2.0 and higher support up to 16 concurrent kernel launches. Kernel launches are submitted to nonzero execution streams by declaring additional streams to be present and passing the stream objects to the kernel launches as an extra launch parameter. Kernels launched on different streams can be executed concurrently and their blocks can be interleaved, which is why it is very important for them to act on independent data. Launching the reaction kernels concurrently means they are launched in parallel and they do not need to wait for

each other to complete. If not launched concurrently, kernels must be launched sequentially, and the inelastic scattering kernel would have to wait for the elastic scattering kernel to complete before it could be launched, for instance.

Kernels launched without an explicit stream are launched on the default stream 0, which are synchronous (program flow does not continue on the host until the launch is complete). If kernel synchronization is important after execution completes (as it is here, reactions must be carried out before the OptiX trace), the streams must be synchronized with an explicit stream synchronization command or with a synchronous operation like a cudaMemcpy(). If the kernels are not synchronized, control can continue to the OptiX trace, and launching OptiX before the reaction kernels are done processing could lead to errors, incorrect results, or program crashes.

Parallel Operations

The array-wide parallel operations in WARP are done with the CUDPP library. The functions used are reductions for calculating k_{eff} via Eq. (2.95), prefix sums for calculating the indices for the source pop kernels, and sorts for keeping warps coherent and full after the microscopic kernel.

Remapping with Radix Sort

Since the reaction numbers are integers they can be sorted by very efficient algorithms like a radix sort, which performs in $O(kn)$, where n is the number of values to be sorted and k is the number of significant digits of the values. The reaction numbers in WARP are four digits or less, so $k = 4$. The reaction number encodings in WARP are shown in Appendix B in Table B.1. Since reactions greater than 900 represent a terminated particle, the sort will push all the completed history references to the end of the remapping vector.

Another benefit pushing all of the completed references to the end of the vector is that, as neutrons start to complete, the length of the sort can be reduced and it will be computed faster. Figure 3.23 shows how the radix sort will arrange the remap vector and how threads will be mapped to the data. CUDPP's radix sort conveniently sorts for key-value pairs, and WARP uses the reaction numbers as keys and the data index as values, effectively creating a remap vector.

There is one more piece of information that is needed to launch the reaction kernels concurrently – the number of particles undergoing each type of reaction. This is determined by launching a comparison kernel on the sorted array, which detects the places where the reaction numbers change. The algorithm to do this is a simple adjacent comparison. If there are N elements in the array, $N - 1$ threads are launched that load values tid and tid+1. If the values are different, tid is the last index of a reaction type. If a transition is determined to be valid, its thread can write to a small array and no data races should occur because there are coarse boundaries for the reaction kernels and each boundary is unique. There are only two boundaries for each reaction group, and writing the indices of the reaction block

After Microscopic Kernel, Before Radix Sort

Data Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Reaction Number	2	800	2	2	51	61	1102	81	67	91	2	2	816	51	91	810	91	818	54	800

After Radix Sort

Data Index	0	2	3	10	11	4	13	18	5	8	7	9	14	16	1	19	15	12	17	6
Reaction Number	2	2	2	2	2	51	51	54	61	67	81	91	91	91	800	800	810	816	818	1102
	Elastic Scatter Kernel					Discrete Inelastic Kernel						Continuum Inelastic Kernel			Resample - Rxn kernels skipped		Fission Kernel - Terminated after processing yields			Complete - No longer referenced

Figure 3.23: A radix key-value sort creating a remapping vector.

edges can be done without atomic operations since only one thread should ever write to an element.

The array of reaction block boundaries needs to be read by the host in order to determine the number of blocks that need to be launched for each reaction kernel and where the kernels start accessing the remapping vector. Therefore, it makes sense to have this edge array as mapped memory, where values are implicitly copied from the device. Declaring the array as mapped allows CUDA to handle when it makes sense for the copy to happen and can potentially hide the communication overhead. It also makes the code simpler, since explicit copy commands are not needed.

If a launched thread lies on a boundary, it writes its index into a vector containing 11 elements, the start and end indices for the four reaction kernels (three types of scattering kernels and the fission kernel), and the resample block. Terminated neutron data are pushed to the end of the remapping vector and will not be accessed by active blocks, eliminating the need for an explicit absorption kernel (which *is* needed in the non-remapped implementation of WARP).

The standard MT numbers are not well suited for producing contiguous blocks of reaction types with a single sort. Fission and other neutron-producing reactions that terminate a primary history lie between scatter reactions, and the resample flag (MT=800) is larger than every reaction type other than leakage (MT=999) and geometry miss (MT=997). All disappearance reactions, like (n,γ), are greater than 102. To resolve this problem and make clear, contiguous reaction blocks, the MT reaction numbers are slightly modified in WARP.

Any reaction over 900 signifies a terminated history, so when the cross section data is loaded, any reaction larger than 102 has 1000 added to it to ensure it will be pushed out of the remapping vector upon sorting. Since fission also terminates particles but needs a yield value sampled after the microscopic kernel samples it to occur, any reactions in the MT=11-45 block have 800 added to their MT numbers. Therefore, after the sort is performed, the fission-like reactions are already near then end of the remapping vector. After the fission kernel processes these histories, their reaction number have 100 added to them, making them part

of the completed history block. The reaction numbers for scattering (MT=2, 51-91) are not changed. MT=11-45 not including the fission numbers can usually be treated as continuum scattering (ENDF law 44), but may not be true for all isotopes. With the reactions laid out as such, a contiguous remapping vector can be formed and the total number of active neutrons can be calculated from a single, efficient radix sort operation.

Criticality Source

As was mentioned previously, when the simulation is run in criticality mode, the source points for neutron batches depend on the fission points in the previous batch. But if k_{eff} is not equal to one, each source neutron no longer corresponds to a secondary neutron. This is the same problem in the time-independent neutron transport equation, where getting rid of the time-dependent term causes the equation to be inconsistent if the multiplication factor is not one. To remedy this, the fission source term is divided by the multiplication factor to force the equations to be consistent (assuming the multiplication factor is known). Similarly, the fission yield vector of a batch of neutrons can be renormalized to artificially make the multiplication factor equal to unity by rearranging Eq. (2.95) as shown in Eq. (3.6), where n is the batch number, N_d is the length of the dataset, y_j is the secondary neutron yield of particle j, and N_f is the total number of secondary neutrons.

$$\sum_{j=0}^{N_d} \frac{y_j}{k_{\text{eff},n}} = \frac{N_{f,n}}{k_{\text{eff},n}} \Rightarrow N_{s,n+1} = N_{s,n} \tag{3.6}$$

Since WARP handless all neutrons with equal weight, yields must be integers, but y_j/k_{eff} rarely will be. Stochastic rounding is used to round y_j/k_{eff} to the lower integer with probability $y_j/k_{\text{eff}} - \text{floor}(y_j/k_{\text{eff}})$. Since yields range from 2-5 neutrons, there will be many individual instances of each yield, and yields will approach the mean y_j/k_{eff}.

Once the yield vector is rebased by dividing by k_{eff} and stochastically rounding to a nearest integer, a prefix sum is done on the yield vector. The jth value of an exclusive prefix sum, or exclusive scan as CUDPP calls it, is sum of the array values before index j as shown in Eq. (3.7). Such a vector can be used in the pop kernel, which writes the secondary particle data into the next batch's initial data based on the yield vector.

$$p_j = \sum_{0}^{j-1} x_i \tag{3.7}$$

After the prefix sum is done, the source pop kernel is launched. A thread is launched for every element of the history dataset, and if a thread encounters a yield value of zero, it returns. If a nonzero value is encountered, the thread samples y_j secondary particles and writes them into the history data from p_j to $p_j + y_j$. Since the yields have been rebased, this should either just fall short of or slightly over the total dataset size, N_d. The yield vectors are immediately set to zero for all particles in the next generation.

Needing to store generational information to determine k_{eff} and the next generation's source distribution forces criticality calculations to use a batch-like transport approach. Transport on the GPU becomes inefficient when few active neutrons remain in the dataset. Formulating a way where generations did not need to be run in series could benefit the GPU greatly. A potential way would be to associate generational information with neutrons, transporting them as in a fixed-source problem, then post-processing the results to determine k_{eff}, but a clever way to assure fission source distribution convergence would have to be developed.

Fixed Source & Subcritical Multiplication

In fixed source mode, the secondary particles can be popped back into the active particle dataset at the end of the inner loop since generational information is not needed and only the total system response to the primary neutrons is desired. In this mode, the number of source neutrons is given as an input and, memory allowing, a dataset of source neutrons this size is made. These neutrons and transported, and any secondary-producing reactions are popped back in to the actively transporting dataset. This is done by performing a compaction operation on the *completed* dataset indices and a prefix sum is done on the yield vector.

A pop routine is used in a manner similar to a criticality source run, but the threads with a nonzero yield write into the indices where a neutron has been terminated. This is specified by writing into indices referenced from p_j to $p_j + y_j$ of the compaction vector instead of p_j to $p_j + y_j$ of the dataset itself. This reactivates terminated particle data for transporting secondary neutrons. The terminated neutron data is all replaced by data appropriately sampled for the secondary neutron reaction. As mentioned previously, it is necessary to be subcritical in fixed source mode, or subsequent generations will grow instead of shrink and the total number of neutrons needing to be transported to calculate the response will diverge.

All the necessary pieces have been outlined to perform Monte Carlo neutron transport in general 3D geometries using continuous energy cross sections. Algorithms and libraries have been chosen so each step of neutron transport can be preformed efficiently on a GPU. The next chapter will discuss the consequences of these choices and compare WARP's results and runtimes to those of Serpent and MCNP.

Chapter 4

Results

The results of six simple tests of WARP compared to Serpent 2.1.18 and MCNP 6.1 are presented in this section. WARP uses the same nuclear data ACE libraries that Serpent uses. MCNP 6.1 uses the nuclear data distributed with it. Both data sets are derived from ENDF/B-VII data. Since the library files used in Serpent and WARP are identical but the one used in MCNP is different, the error comparisons in this section are made against Serpent, not MCNP. WARP only uses collision estimators for flux tallies and k_{eff} estimations, so the collision estimates from Serpent and MCNP are also used. Serpent and MCNP are run serially, which gives them the best theoretical performance since no time is lost to parallelization inefficiencies; both codes are then assumed to scale linearly. The GPU card that WARP runs on can be approximated as a certain number of CPU cores, and a conservative performance per cost comparison can be made.

All of the test cases were run on a server containing two AMD Opteron 6128 Magny-Cours processors. These processors each contain eight cores which are clocked at 2.0 GHz and have a 512 kB L2 cache. The server has 32 GB of DDR3 clocked at 1.333 GHz between the two processors. The GPU used in the tests is a NVIDIA Tesla k20. It has 2496 "CUDA cores," a multiprocessor clock of 706 MHz, and 5 GB of 2.6 GHz GDDR5 global memory. The Opteron 6128 was released in Q2 of 2010, whereas the k20 was released in Q4 2012 [23, 62]. This is a comparison of one of the newest Tesla cards with a older CPU, but it is a standard CPU for many supercomputer systems currently running and is sufficient if given every advantage in the comparison (i.e. perfect scaling linearity). WARP was built using CUDA 5.0, OptiX 3.0.1, CUDPP 2.1, and PyNE 0.4-dev.

The multiplication factor differences are reported in "per cent mille," which is a thousandth of a percent, or 10^{-5}. This is a standard way of reporting differences in the multiplication factor, as any small deviation from unity can cause a reactor to change its power level. The flux spectra are normalized per source neutron and per unit lethargy. Normalizing per source neutron serves to reproduce the same results for different numbers of histories run. The uncertainty will be lower in results with more histories, of course, but the magnitudes should have the same mean values. "Lethargy" means the logarithm of the neutron energy, $\ln(E_0/E)$ [3]. Normalizing the flux per unit lethargy accentuates the high energy region,

but used in conjunction with plotting on a logarithmic scale, it also yields a plot where the area under the line gives fraction of neutrons (flux) in a specific energy bin. Plotting per unit lethargy is a change of base where plotting the flux $\phi(E)$ to transformed to $\phi(u)$, where $u = \ln(E_0/E)$ [63]. Changing the base is shown in Eq. (4.1). In the discrete energy group case, normalizing the flux to per unit lethargy means that the bin value is multiplied by the average (mid-point) energy of the bin.

$$\begin{gathered} \phi(E)dE = \phi(u)du = \phi(u)\frac{dE}{E} \\ \phi(u) = E\phi(E) \qquad \Rightarrow \qquad |\bar{\phi}_{g,j}| = \bar{E}\bar{\phi}_{g,j} \end{gathered} \tag{4.1}$$

To normalize the raw tally values (Eq. (2.91)) to per unit lethargy, the values must be divided by the total number of source neutrons run, divided by the energy bin width, and multiplied by the average bin energy [63]. The expression for normalizing the raw tally scores is shown in Eq. (4.2), where $\bar{\phi}_{g,j}$ is the raw tally value in cell volume j and energy group g. Since WARP does not include S(α,β) or unresolved resonance tables (yet), these features were not activated in MCNP or Serpent in the test runs. WARP is currently only able to handle black, or vacuum, boundary conditions, so all the test runs use black boundary conditions as well.

$$\begin{gathered} E_g < E_i < E_{g+1} \\ |\bar{\phi}_{g,j}| = \left(\frac{1}{N_{\text{total}}}\right)\left(\frac{1}{E_{g+1}-E_g}\right)\left(\frac{E_g+E_{g+1}}{2}\right)\bar{\phi}_{g,j} \end{gathered} \tag{4.2}$$

Six test cases were considered, four that are criticality type, and two that are fixed-source type. WARP can represent spheres, cuboids, cylinders, and hexagonal prisms, and all but the hexagonal prism are represented in the tests. Most of the cases have very simple geometries with few cells, but the assembly case has 632 total cells to accentuate the geometry processing routines and the impact of the material resampling as the neutrons move into different regions without interacting. Every material in every test case uses some combination of the reactor-pertinent isotopes ^{239}Pu, ^{238}U, ^{235}U, ^{16}O, ^{10}B, and ^{1}H. This is a relatively small set of isotopes to have in a reactor simulation. In the debugging process of WARP's development, fixed-source simulations were done using ^{27}Al, ^{208}Pb, ^{12}C, ^{6}Li in addition to the previous isotopes, but the results of these short debugging simulations are not reported here. These isotopes are only mentioned in order to fully express the amount of testing done on the data loading routines. Currently a temperature of 300K is hardcoded into WARP, so cross sections used in these tests are all processed at this temperature. The summary of the test geometries and materials is shown in Table 4.1.

The "Jezebel" test is a bare plutonium sphere, and is a standard criticality test. The fission neutron rate from ^{239}Pu is balanced by the leakage rate from the 5.1 cm radius to give a k_{eff} of approximately 1. Since this system is so leaky, producing results consistent with MCNP and Serpent ensures that the boundary conditions are correctly being enforced. The Jezebel test is a standard test used to validate neutron transport codes. It is described

in the International Handbook of Evaluated Criticality Safety Test Experiments under the name "Pu-MET-FAST-001" [64]. The test here uses a slightly higher density, and therefore a slightly smaller critical radius than the handbook's version.

The "Homogenized Block" test consists of a single cell and a single material as well, but that material has multiple isotopes in it. This particular material is a mixture of 1% ^{235}U enriched UO_2 and water at a 1:1 ratio. Since the single cell is entirely made up of material containing fissile isotopes, fissions can happen anywhere and the fission source is spread over a large volume compared to the other tests. The cell dimensions are small so the amount of time requires to converge the fission source is not extremely long.

The "Pin Cell" test consists of a bare UO_2 cylinder surrounded by a block of water. This test now has two materials, each with multiple isotopes, and two cells. The water block dimensions are not very large, so leakage should play a part. This test serves to highlight that all the processing routines work simultaneously, and the effect of introducing more than one cell, which should have a significant effect on the ray tracing rate as pointed out in the preliminary OptiX study.

The "Hex assembly" test consists of 631 bare UO_2 cylinders laid out in a hexagonal lattice surrounded by water. The material compositions, densities, and the cylinder dimensions are identical to the pin cell test case, but since this test has two orders of magnitude more objects, it serves to highlight the effect of introducing many geometric objects into the problem and will further validate that the geometry processing routines work correctly if consistent results are obtained.

The "Fixed-source Block" refers to two separate tests where WARP is run in fixed-source mode instead of criticality mode. The first test illustrates WARP's capability to perform subcritical multiplication from secondary neutrons in fixed-source mode. The geometry is a cube that is 2 meters on a side so leakage is minimized and a thermal peak should be produced. The materials are identical to the homogenized block test, except for the addition of a small amount ^{10}B to make the system more subcritical. A 1 eV isotropic point source at the origin was chosen as to highlight the effectiveness of popping any secondary neutrons back into the active transport cycle. Since the source is at 1 eV, it will induce fissions in ^{235}U which will produce neutrons in a fission spectrum that does not overlap with the source. The second fixed-source test is a 2 meter cube of water that contains a 2 MeV point source at the center. This test does not contain any fissile materials, so there are no secondary particles transported. There are also no highly absorbing materials, so neutrons will scatter many times before they are absorbed or leak out of the cube. This test serves as a counterpoint to the first since it will be seen what effect eliminating the secondary pop routine from the inner transport loop will have on performance.

Table 4.1 summarizes the geometry and materials used in the test cases. The numbers in parentheses preceding the isotopes their ratios to each other (i.e. they are unnormalized atomic fractions). The pin cell and assembly cases contain water at 3 g/cm^3, but water cannot ever have a density this high. This was an oversight when deciding on the material parameters. Changing it to 1.0 or lower will not affect the conclusions of the simulation, however, since the Serpent and MCNP simulations were run with these unrealistic densities

as well. The comparison should be consistent, even if the results are unphysical.

Table 4.1: Geometry and materials used in the six test cases.

Test	Number & Type of Cells	Materials	Isotopes	Densities
Jezebel	1 sphere, r=5.1cm	Fuel	(1.00) ^{239}Pu	19.816 g/cm^3
Homogenized Block	1 cube, r=10cm	Hom. Fuel.	(0.90) ^{238}U (0.10) ^{235}U (3.00) ^{16}O (2.00) ^{1}H	10 g/cm^3
Pin Cell	1 cuboid, 10x10x50cm 1 cylinder, r=1cm z=40cm	Fuel	(0.90) ^{238}U (0.10) ^{235}U (2.00) ^{16}O	15 g/cm^3
		Water	(1.00) ^{16}O (2.00) ^{1}H	3 g/cm^3
Hex Assembly	1 cube, 84cm^3 631 cylinders, r=1cm z=40cm	Fuel	(0.90) ^{238}U (0.10) ^{235}U (2.00) ^{16}O	15 g/cm^3
		Water	(1.00) ^{16}O (2.00) ^{1}H	3 g/cm^3
Fixed-source Block	1 cube, 2x2x2m	Water	(1.00) ^{16}O (2.00) ^{1}H	1 g/cm^3
	1 cube, 2x2x2m	Hom. Fuel w/ ^{10}B	(0.90) ^{238}U (0.10) ^{235}U (0.10) ^{10}B (3.00) ^{16}O (2.00) ^{1}H	10 g/cm^3

4.1 Criticality Tests - Multiplication Factors and Runtimes

The goal of WARP is to be the first step in creating a full-featured continuous energy Monte Carlo neutron transport code that is *accelerated* by running on GPUs. The crux of the effort is to make Monte Carlo calculations faster but still produce accurate results. Table 4.2 shows the multiplication factor deviations and the speedup factors for the four criticality tests compared to MCNP 6.1 and Serpent 2.1.18 when 10^5 neutrons per batch are used. Table 4.3 shows the same information, but for cases where 10^6 neutrons per batch are used. 10^6 was the maximum number of neutrons that could be run before the GPU ran out of memory, so larger batches could not be considered. The ΔM column is for WARP's difference from MCNP, and the ΔS column is for WARP's difference from Serpent. The

differences are reported in pcm for multiplication factors, and speedup factors (t/t_{WARP}) for runtimes. The (y) in the Δ columns signify if the WARP value is inside (y) or outside (n) two standard deviations of the production code's value. Error estimators have not yet been implemented in WARP, so a confidence interval is not reported for WARP's values.

Table 4.2: Summary of k_{eff} single-run results of the WARP criticality tests with 20/40 discarded/active criticality cycles and 10^5 histories per cycle.

Test	MCNP 6.1	Serpent 2.1.18	WARP	Δ M	Δ S
Jezebel					
k_{eff}	1.027509±0.0005	1.02748±0.00052	1.02789	-38.1 pcm (y)	-41 pcm (y)
Runtime	2.32 m	9.50868 m	0.2752 m	8.4x	34.6x
Homogenized Block					
k_{eff}	1.216842±0.0005	1.21494±0.00047	1.21463	221 pcm (n)	-31 pcm (y)
Runtime	17.28 m	13.6 m	0.48 m	36.0x	28.3x
Pin Cell					
k_{eff}	0.381435±0.0008	0.380511±0.00128	0.380586	84.9 pcm (n)	-7.5 pcm (y)
Runtime	55.85 m	40.0035 m	2.81583 m	19.8x	14.2x
Hex Assembly					
k_{eff}	1.437465±0.0004	1.44704±0.00046	1.4442	-673 pcm (n)	284 pcm (n)
Runtime	25.34 m	26.3349 m	3.2395 m	7.8x	8.1x

Table 4.3: Summary of k_{eff} single-run results of the WARP criticality tests with 20/40 discarded/active criticality cycles and 10^6 histories per cycle.

Test	MCNP 6.1	Serpent 2.1.18	WARP	Δ M	Δ S
Jezebel					
k_{eff}	1.027942±0.0002	1.02837±0.00017	1.0279	4.2 pcm (y)	47 pcm (n)
Runtime	22.75 m	95.8 m	2.0 m	11.3x	47.6x
Homogenized Block					
k_{eff}	1.215533±0.0002	1.21414±0.0002	1.21346	207 pcm (n)	68 pcm (n)
Runtime	150.7 m	137.0 m	2.8 m	53.8x	48.6x
Pin Cell					
k_{eff}	0.381564±0.0003	0.380624±0.00037	0.38043	113.4 pcm (n)	19.4 pcm (y)
Runtime	578.62 m	404.2 m	7.1 m	81.9x	57.2x
Hex Assembly					
k_{eff}	1.437326±0.0002	1.44722±0.00014	1.4454	-807.4 pcm (n)	182 pcm (n)
Runtime	252.77 m	267.7 m	8.0 m	31.7x	33.6x

It can be seen that in the 10^5 neutrons per batch case, WARP performs 8 to 36 times faster than the production codes, and the multiplication factor differences are slightly different.

Multiplication factors are most notably different in the assembly test, where WARP's answer falls between MCNP and Serpent, but is still very far from either. Similar trends are shown in the multiplication factor for the 10^6 runs in Table 4.3, but the speedup factors are much higher, with WARP being 11 to 82 times faster than the production codes. The greatest speedup is seen in the pin cell test, followed by the homogenized block, assembly, then finally the bare sphere. The speedup factors compared to Serpent in the 10^6 neutrons per batch runs are relatively constant, ranging from 33 to 56, whereas the speedup factors compared to MCNP range from 11 to 82.

In the Jezebel test, all codes agree, but as things get more complicated, deviations start to occur. WARP is often outside of a single standard deviation for either code. It is important to point out that Serpent and MCNP are also often more than a standard deviation away from each other, with WARP falling somewhere in between, or if not in between, near one of the two codes.

It is also interesting that in the Jezebel test, MCNP is actually much faster than Serpent. Presumably, this is due to the different methods they use for neutron tracking. Serpent uses Woodcock tracking, and moves the neutron in small steps across the geometry, but does not have to calculate any surface intersection points. MCNP uses ray tracing and does calculate intersection points (as does WARP). Since the Jezebel sphere is a very leaky system, many neutrons stream out of the sphere without interacting many times (or at all). Since MCNP and WARP both use ray tracing, they can terminate these particles in a small number of geometry queries. Serpent, however, needs to query the geometry many times in order to propagate the neutrons to the edge of the sphere where they are leaked. This difference is not seen in the homogenized block test, where leakage plays very little role in neutron loss.

Tables 4.2 and 4.3 show that the multiplication factors calculated by WARP sometimes fall outside the uncertainty of the production codes. The exact reason for this has not been determined, but can be speculated. The deviations are the worst in the assembly test, implying that there may be a bug error in the geometry processing routines. The fact that WARP treats (n,2/3/4n) multiplicity reactions as neutron sources instead of negative absorptions could also be effecting the results, as per Eq. (2.96) and Eq. (2.97). WARP also produces spectra that have slight differences from Serpent and MCNP (shown in the next section), indicating that the reaction rates are different at certain energies. And even though they are small, spectral differences may point to a bug in the reaction sampling routines. WARP also uses single precision float point numbers, and the way that k_{eff} is accumulated between batches may also be leading to roundoff errors in the final result. Another reason could be that the fission source distributions are not completely converged. Adding a estimator for the fission source convergence, such as Shannon entropy [11], could be incorporated into WARP in the future to ensure full convergence. The reproducibility of results has also not been tested in WARP. CURAND can be seeded with the same number across runs, which should lead to the same sequence of random numbers in the kernel LCRNGs, but exact reproducibility has not been attempted. Reproducibility is an important feature in validating results, and will be an area of future development for WARP.

4.2 Criticality Tests - Flux Spectra and Fission Distributions

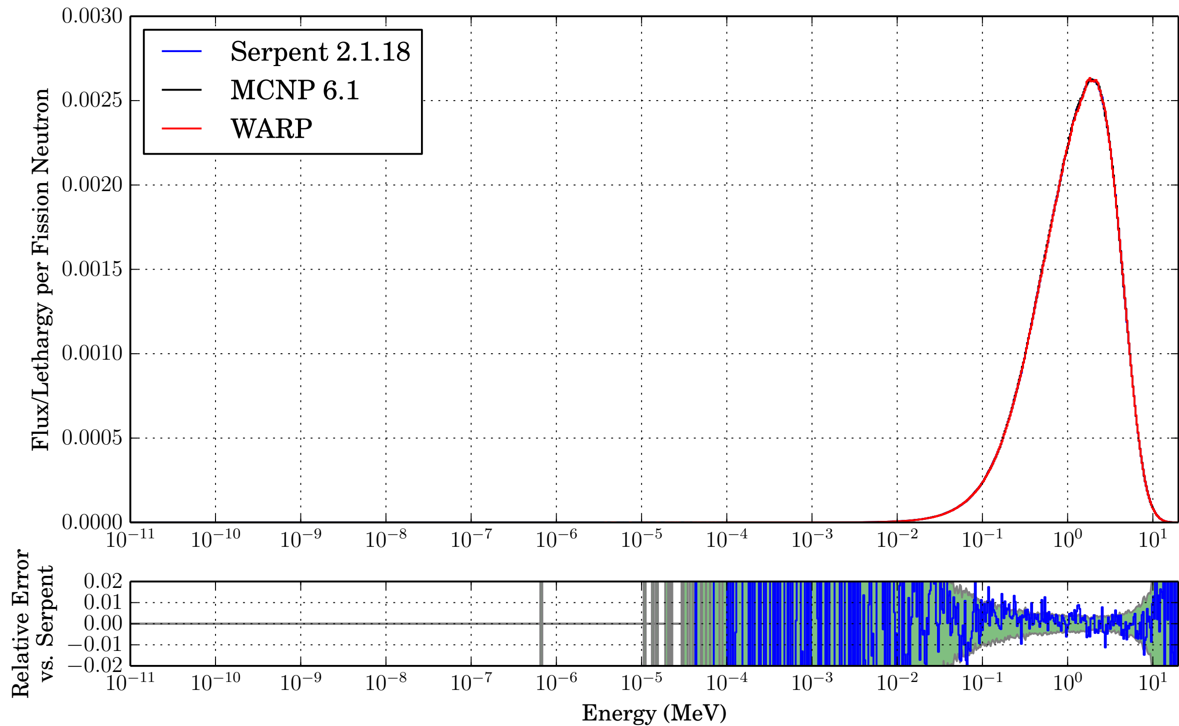


Figure 4.1: Spectrum comparison in a "Jezebel" bare ^{239}Pu sphere.

The flux and fission source distributions for the criticality tests are shown in the following figures. The relative difference subplots for the spectra also have the 2σ error from Serpent shown in transparent green. The error levels shown are from Serpent since the relative difference is calculated from the Serpent results. The green area has a black border line to highlight where it terminates. The colors in the fission source distribution plots represent the relative probability of a fission neutron being born at that point. The distributions are normalized so the maximum value is one. Plots showing the relative difference compared to the fission source distributions produced by Serpent are also shown to attest to the accuracy of the fission source. These plots were made by dumping mesh plots produced by Serpent and re-normalizing them to a maximum of one. The mesh plots have a resolution of 250x250 pixels for every plot. The fission points produced by WARP were binned at an identical resolution, and the relative difference distributions calculated by simply subtracting bin-by-bin (there is no smoothing done). The average relative difference of each distributions is shown in the plots as $\bar{\Delta}_{xy}$ and $\bar{\Delta}_{xz}$. The Serpent fission source mesh plots are included in

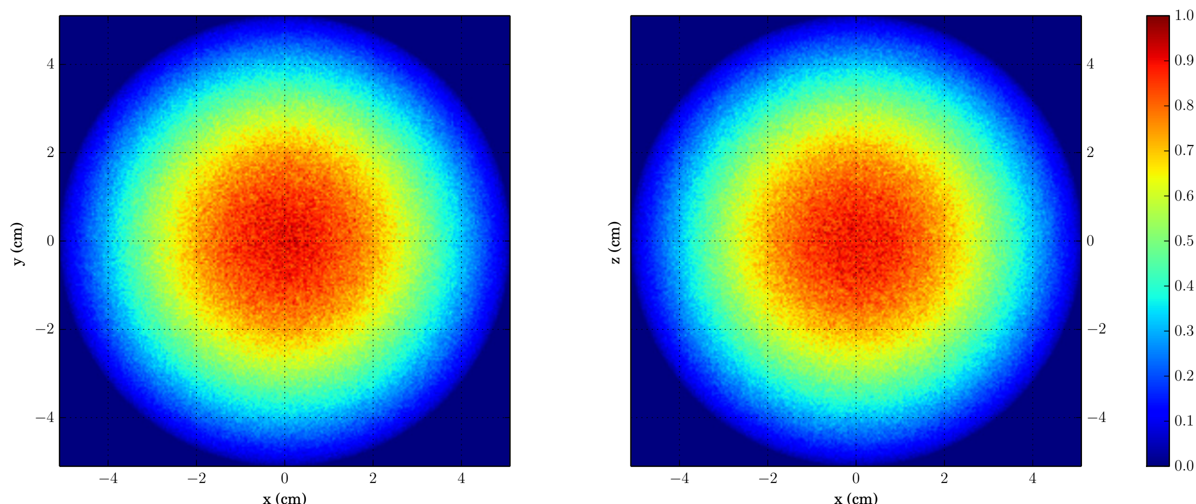


Figure 4.2: Fission source distribution of a "Jezebel" bare ^{239}Pu sphere calculated by WARP.

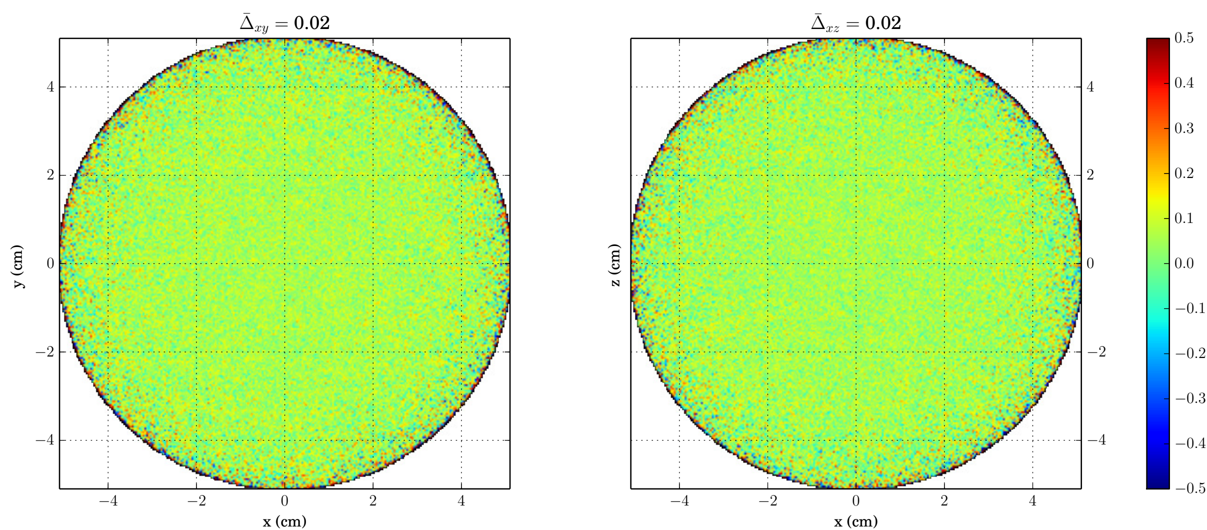


Figure 4.3: Relative difference of the WARP fission source distribution compared to Serpent's for the Jezebel bare ^{239}Pu sphere.

Figures A.2-A.5 in Appendix A as they are somewhat redundant to the information shown by the WARP fission distributions and the relative difference distributions. To show the most accurate and highest resolution results, the data in all these figures are from runs using 10^6 neutrons per batch.

Figure 4.1 shows the volume-averaged flux spectrum in the Jezebel sphere. The relative

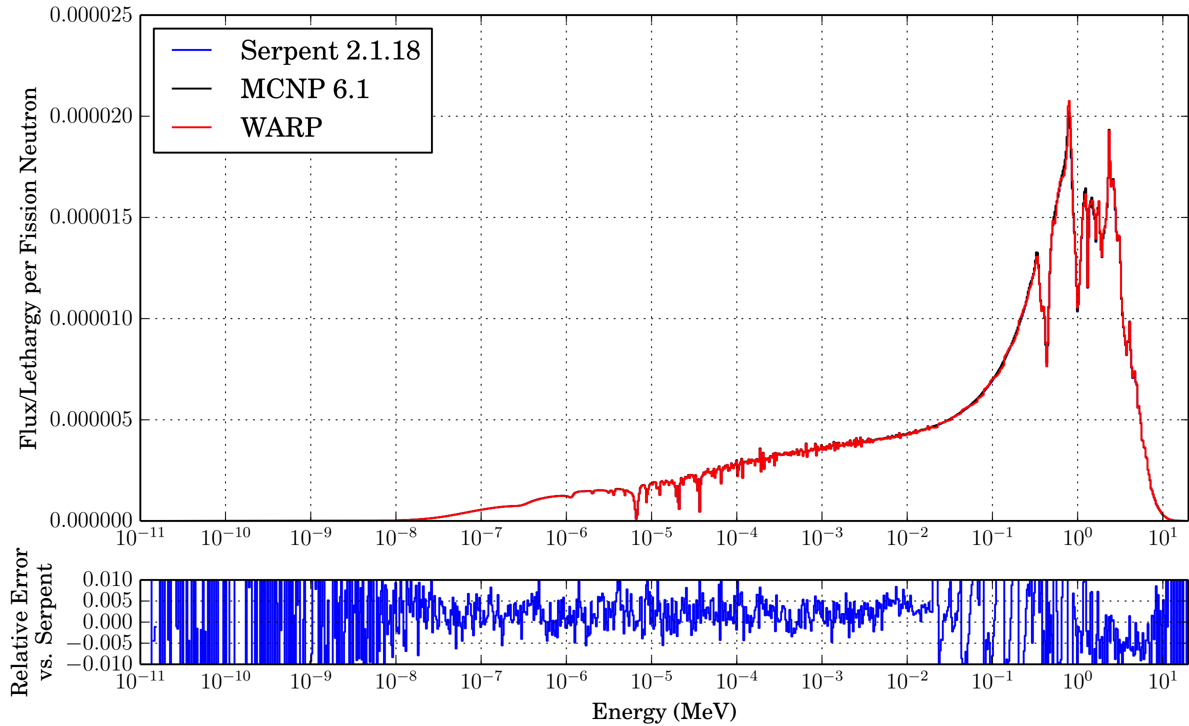


Figure 4.4: Spectrum comparison in a homogenized block of UO_2 and water.

difference compared to the Serpent spectrum is shown in the subplot below the main spectrum plot. The relative difference is very low compared to Serpent, with the normalized tally bins being less than 1% from each other in regions where the flux is large. Of course, when the flux is small, the statistical uncertainty becomes much higher, and the relative difference becomes noisy. For the most part, the relative difference of the WARP spectrum is within 2σ of Serpent's, and appears to have an average of zero. The spectra produced by the three codes lie very close together and are almost indistinguishable from each other in the plot.

The fission distribution, shown in Figure 4.2, appears very uniform with neutrons being preferentially born near the center. The distribution is visually identical in an z-integrated slice compared to the y-integrated slice, as it should be. The distribution is not as smooth as that produced by Serpent, however, and the noisiness of the WARP distribution can be seen in the relative difference plot in Figure 4.3. On a whole, the relative difference is near zero and appears to be mostly green. The white areas are where a NaN value is produced by dividing zero by zero, indicating that WARP and Serpent are in agreement that no fissions occur there.

Figures 4.4 shows the volume-averaged flux spectrum in the block of homogenized fuel. Since this material is slightly leaky and contains a strong thermal neutron absorber (^{235}U),

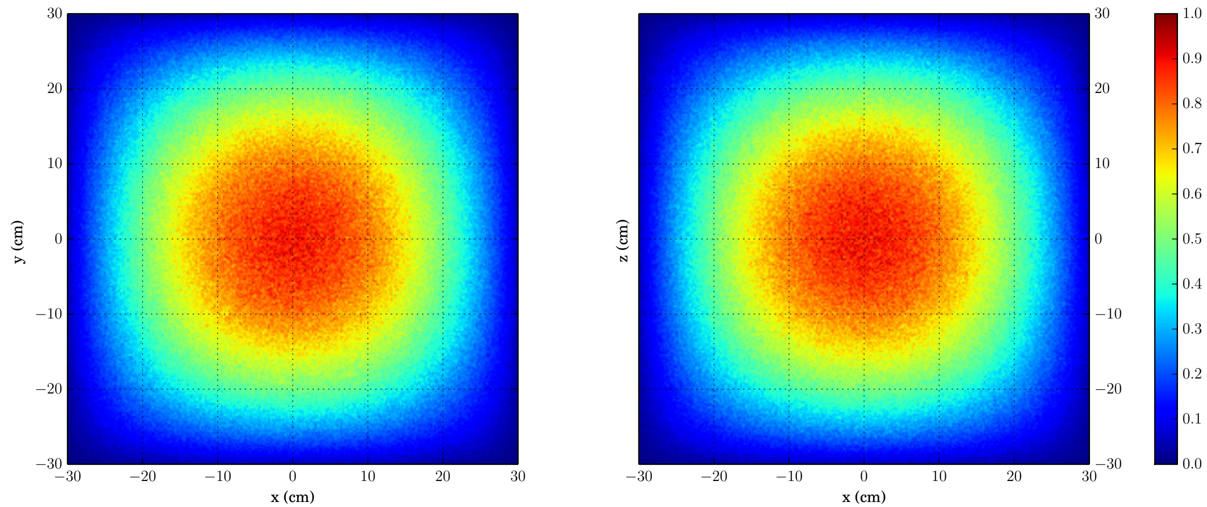


Figure 4.5: Fission source distribution of a homogenized block of UO_2 and water calculated by WARP.

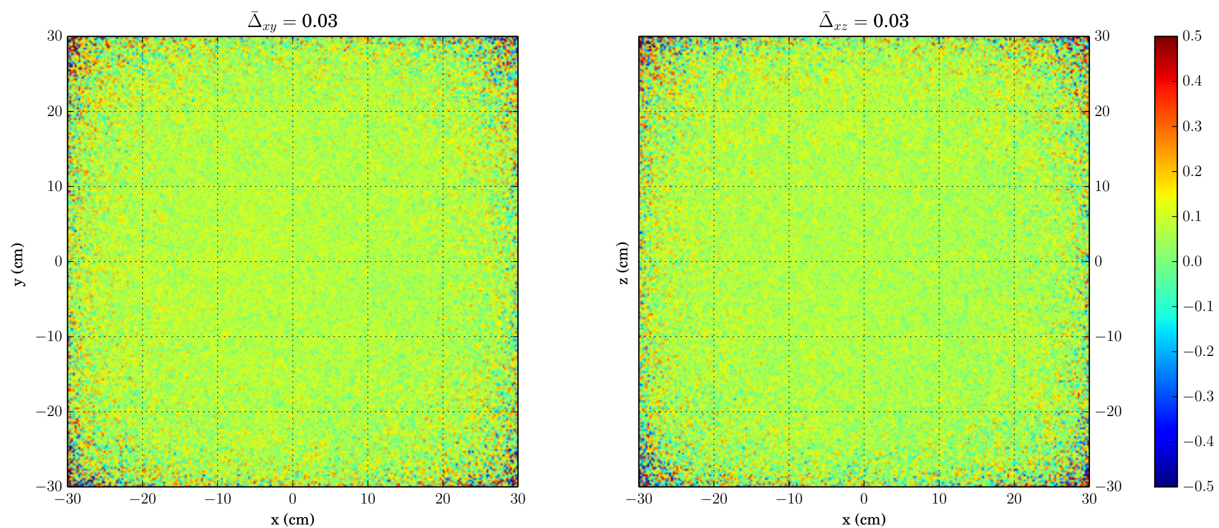


Figure 4.6: Relative difference of the WARP fission source distribution compared to Serpent's for the homogenized block of UO_2 and water.

no substantial thermal peak is seen since neutrons do not have time to accumulate around 0.026 eV before they are absorbed or leak out of the system. Again, the relative difference compared the Serpent spectrum is very low, at 0.5% or lower for energies 10^{-9}-10^{-2} MeV where the flux has good statistics, but has much larger variations at energies above 10^{-2}. In this region, the errors become much larger, which could indicate that a minor sampling

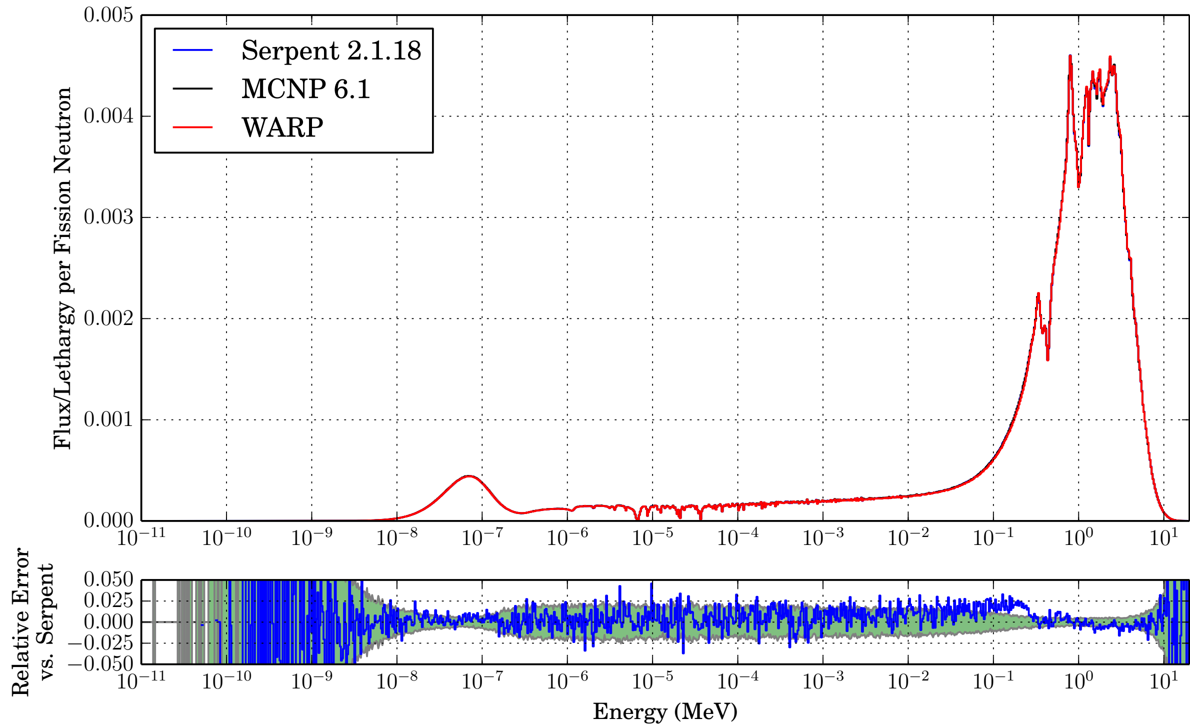


Figure 4.7: Spectrum comparison in a single UO_2 pin surrounded by a block of water.

problem is occurring or physics isn't being handled exactly right. It could also indicate a flux accumulation problem or roundoff error, since this is a high count rate region. The cause will be investigated as WARP continues to develop.

The fission source distribution of the homogenized block, shown in Figure 4.5, has a structure similar to the Jezebel sphere, with an area of high probability in the center that drops to zero at the edges. The noise seems to be more pronounced in this test, since the fissile volume is much larger than in the Jezebel sphere. The relative difference of the distribution, shown in Figure 4.6, shows that edges are also more noisy, especially in the corners, since the flux is low there. Despite the noise, the fission distribution appears to have a fairly uniform error distribution with an approximate mean of zero as indicated by the predominately green color.

Figures 4.7 and 4.8, show the volume-averaged flux spectrum and fission source distribution, respectively, in the UO_2 pin of the pin cell test. This system is slightly leaky, and again contains light nuclides (water), but now they are spatially separated from the fissile material. The relative difference compared to Serpent is again low, around 3% or lower for energies 5×10^{-9}-10 MeV where the flux has good statistics. The relative difference is also within 2σ of Serpent's values on the entire energy range, except for the slight deviation around 0.2

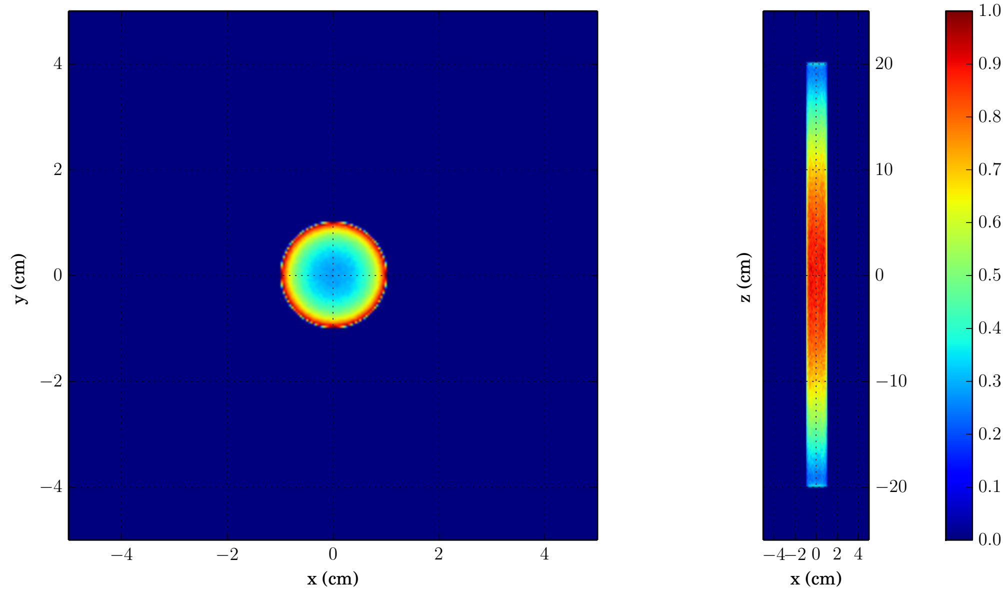


Figure 4.8: Fission source distribution of a single UO_2 pin surrounded by a block of water calculated by WARP.

MeV where the error exceeds this bound. This slightly increased error might be related to the resonance at 0.6 MeV. There is a small thermal peak in the spectrum around 0.026 eV, indicating that the target velocity sampling scheme is working correctly, as well as the vector transformation and rotation schemes. This is expected behavior since in this geometry the fissile material is separated from the moderator, and neutrons have a chance to accumulate at thermal energies before they are absorbed or leak.

The fission source distribution of the pin cell, shown in Figure 4.8, appears to be very smooth and well-converged radially and only slightly noisy axially. The effect of the fuel having a high fission cross section at low energies can also be seen in the fission source distribution. The high cross section manifests itself as a ring of high fission probability around the edge of the pin. This is cause by thermalized neutrons re-entering the fuel pin and immediately fissioning a ^{235}U nucleus. This phenomenon is also called “spatial self-shielding” since the uranium at the surface shields the interior uranium from resonance neutrons and has implications in depletion and power distributions. The axial fission source distribution follows an expected behavior of a finite cylinder as well, roughly following a cosine with the peak at $z = 0$ [3]. Figure 4.9 shows the relative difference distribution of the pin cell compared to Serpent, and is almost entirely green in both the radial and axial directions, indicating very close agreement with Serpent.

Figures 4.10 shows the volume-averaged flux spectrum in the center pin of the hexagonal assembly test. This problem is much more heterogeneous than any of the previous, with

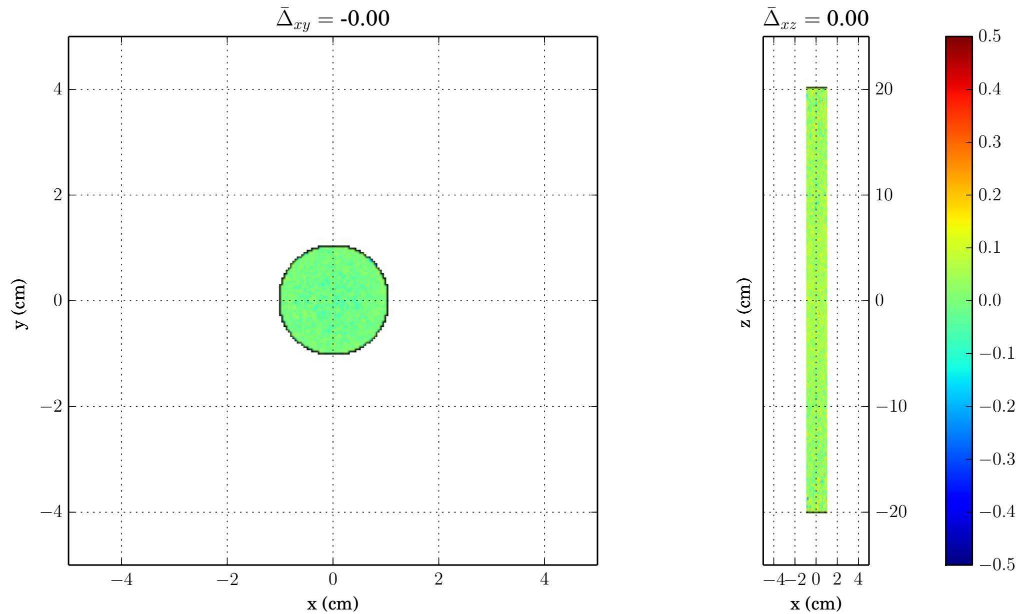


Figure 4.9: Relative difference of the WARP fission source distribution compared to Serpent's for the single UO_2 pin surrounded by water.

300 times more objects in the domain compared to the pin cell test. The WARP spectrum matches very well with Serpent's, and the relative difference is with 15% of Serpent for energies 10^{-8}-9 MeV where the flux has good statistics. The spectrum is much noisier than any of the other tests since the volume of the center pin is relatively small compared to the volume of the fission source, even though this pin has the highest flux of any of the pins in the assembly. The mean of the error appears to be close to zero, however, and the high error is most likely due to statistics. Despite the large relative difference, the WARP spectrum is within 2σ of Serpent's results for the entire energy range. The MCNP spectrum is slightly higher than both the WARP and Serpent spectra, which might be a small normalization error or a real difference.

The fission source distribution, shown in Figure 4.11, appears as expected, with the highest average probability in the center of the array both axially and radially. Self-shielding is shown again, as every pin has a ring of higher probability on near its surface. The assembly also shows some reflection effects from the thick layers of water surrounding it. The pins at the edges have a higher fission source probability at the surfaces facing the water rather than the interior of the assembly. This is due to neutrons that slowed-down in the water moderator surrounding the fuel assembly eventually scattering back into an edge pin where they are absorbed. This phenomenon is apparent in both the axial and radial directions. The relative difference of the assembly's fission source distribution, shown in Figure 4.12, shows good agreement with Serpent. The radial agreement is very good, but the axial has

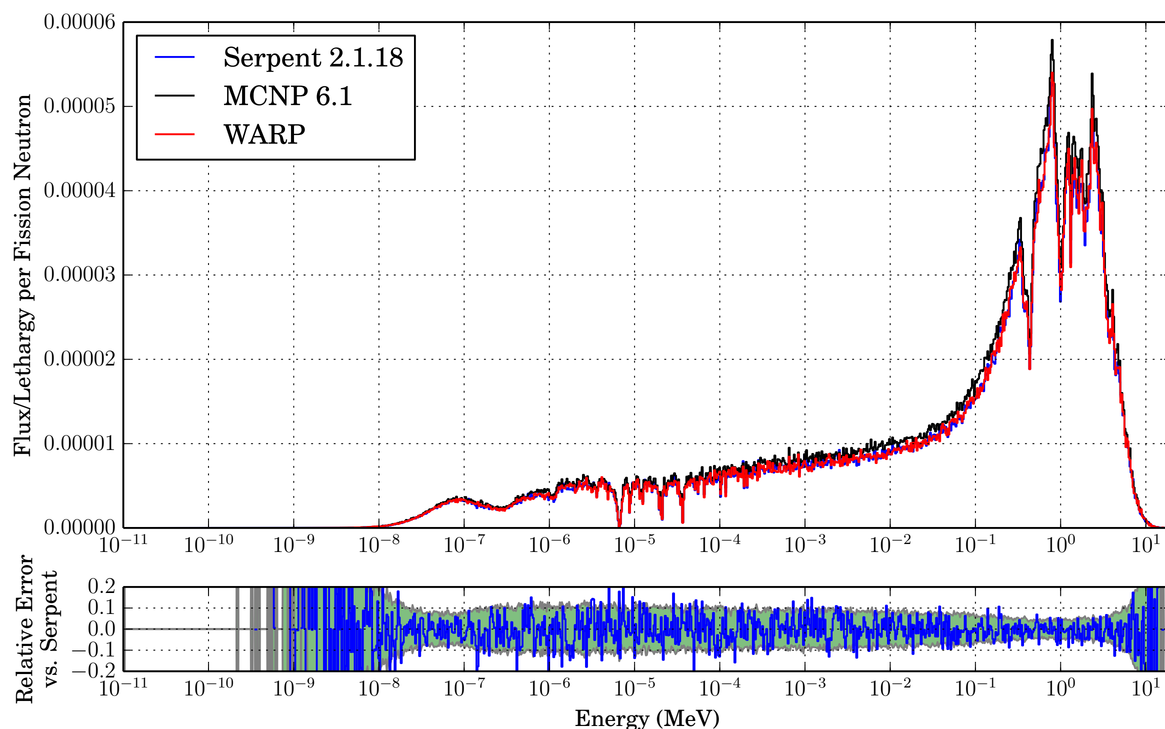


Figure 4.10: Spectrum comparison in the center UO_2 pin of a hexagonal pin array in water.

some noise in it due to the flux being small and the fissile volume being thin at the vertical edges of the assembly when the source points are summed in the y-direction.

4.3 Fixed-Source Tests

WARP is also able to run simulations in fixed-source mode, where the initial neutron energies, directions, and locations are predefined and do not incorporate any feedback from the flux like the criticality source. This mode adds any induced secondary neutrons back into the active particle dataset to be transported in conjunction with the source neutrons, which is also why these problems must be subcritical. The first test is a 2x2x2 m cube of homogenized fuel material with a 1 eV point source at its center. In this case, fissions occur in uranium and secondary neutrons are produced and transported. A small amount of ^{10}B was included to ensure the system is subcritical so multiplication, and therefore runtimes, are modest. The second test is a 2x2x2 m cube of light water with a 2 MeV point source at its center. No secondary neutrons are produced in this case.

Table 4.4 shows the runtimes of WARP, Serpent, and MCNP for the two fixed-source

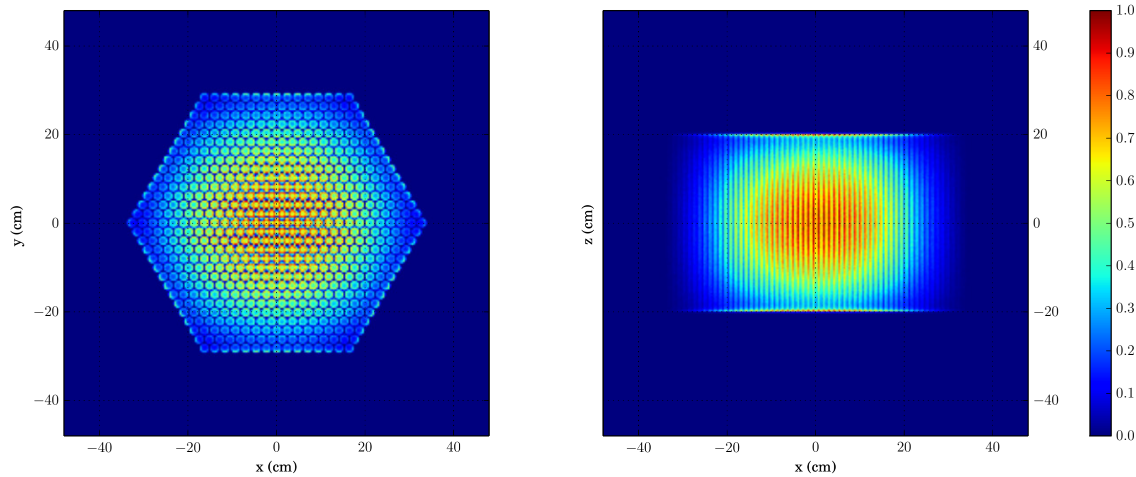


Figure 4.11: Fission source distribution of a hexagonal array of UO_2 pins in water calculated by WARP.

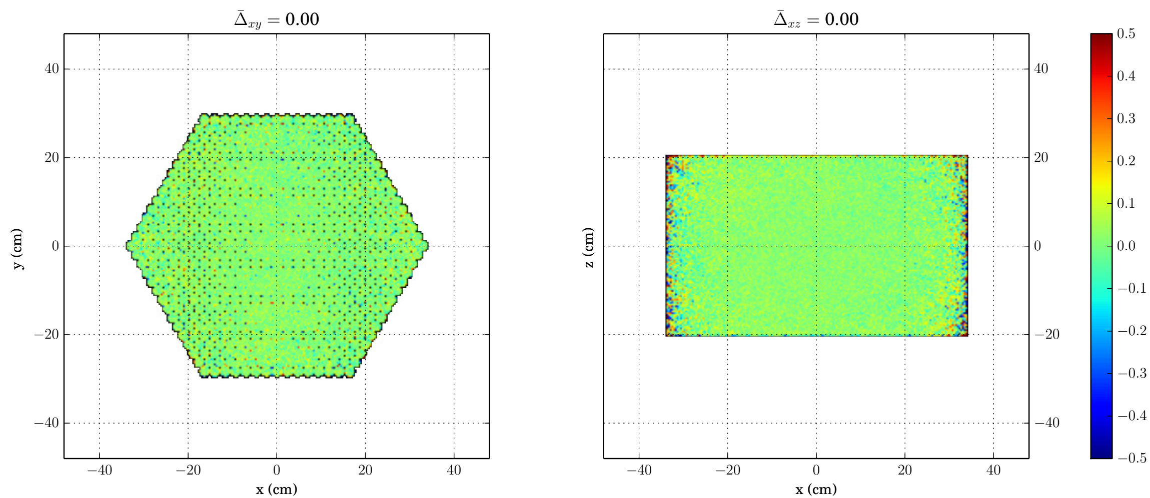


Figure 4.12: Relative difference of WARP fission source distribution compared to Serpent's for the hexagonal array of UO_2 pins in water.

tests. WARP is able to perform the transport faster than either code on a single CPU, but the water test performs about twice as fast as the homogenized fuel test. The runtimes of the water test is about five times longer than the homogenized fuel test, indicating that neutrons survive much longer and undergo many more scattering reactions in water than the homogenized fuel before they are absorbed or leak out. The homogenized block contains two very strong thermal neutron absorbers, and low energy neutrons are quickly absorbed.

Table 4.4: Summary of runtimes of the fixed-source tests with 4×10^7 total histories.

Test	MCNP 6.1	Serpent 2.1.18	WARP	Δ M	Δ S
Water with 2 MeV point source					
Runtime	748.9 m	519.7 m	15.8 m	47.4x	32.9x
Homogenized fuel with 1 eV point source					
Runtime	82.3 m	71.1 m	3.4 m	24.2x	20.9x

Even though there are more neutrons transported in total in the homogenized block due to the secondary neutrons produced, the number of scatters in water still makes the runtime much longer.

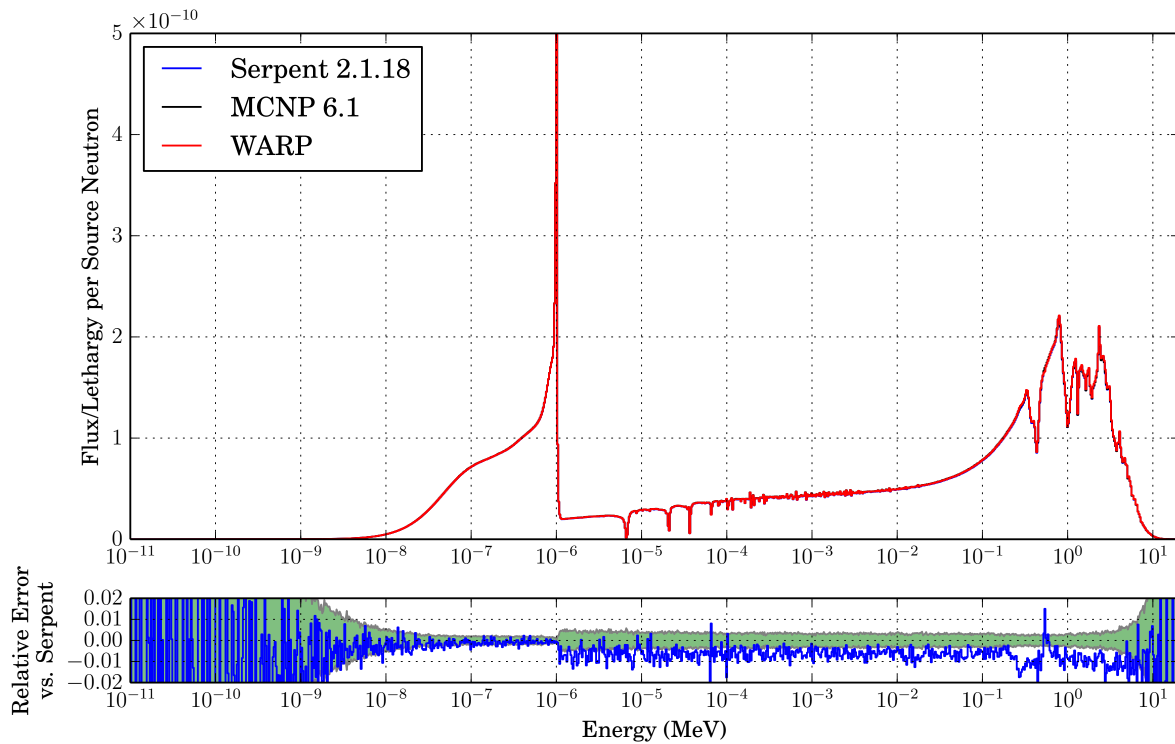


Figure 4.13: Volume-average flux spectra of a fixed-source simulation with a 1 eV point source in a homogenous block of fuel, water, and boron.

Figure 4.13 shows the volume-averaged flux spectra produced by WARP, Serpent, and MCNP for the homogenized fuel fixed-source test. The source spike extends beyond the plot scale, but the plot is on this scale to show more structure at other energies. Even though the height of the source flux is not shown, the error at source is still calculated with the actual value and can still be seen in the error plots. The source spike at 1 eV is clearly visible, and

despite the presence of very strong thermal absorbers, there is a slight population below 1 eV due to the strength of the source. Even though the absorbers are strong, some neutrons will interact with the ^{1}H, ^{16}O, or ^{238}U and will not immediately be absorbed. The difference in magnitude of the source spike and the knee immediately below it is about a factor of 30, attesting the strength of the absorbers since leakage is low. Subcritical multiplication is also evident, since there is significant flux at energies higher than the source. The structure looks very similar to that of the criticality test in Figure 4.4, as it should. The relative difference is below 1% for most energies and within the statistical error of the Serpent results below the source spike, but has a constant negative offset above the source. If this offset were not there, it appears that the error would also be within the Serpent bounds. The error also exhibits a deviation at 0.6 MeV, similar to that in the pin cell spectrum.

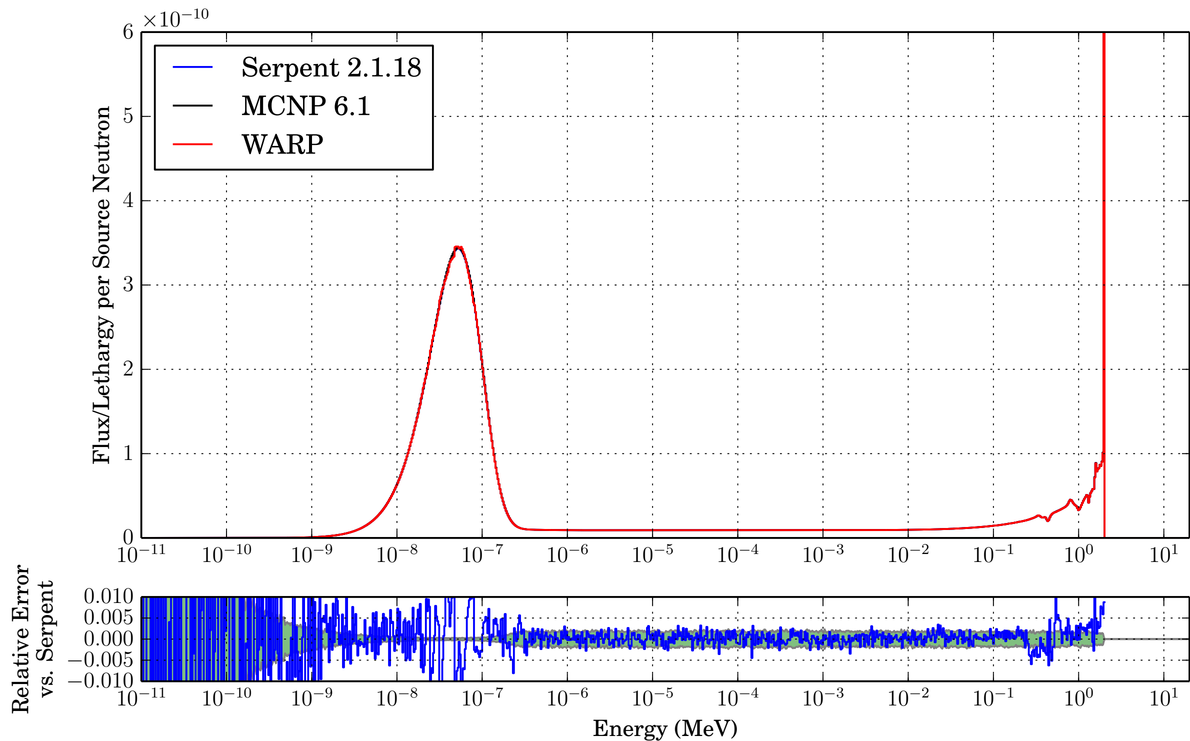


Figure 4.14: Volume-average flux spectra of a fixed-source simulation with a 2 MeV point source in a block of water.

Figure 4.14 shows the volume-averaged flux spectra produced by WARP, Serpent, and MCNP for the water fixed-source test. Again, the source spike extends beyond the plot scale to show more details at other energies. This spectrum is much simpler than those with heavy nuclides. There are no resonances below about 0.6 MeV, and the reactions are almost entirely elastic scattering. There is very little structure until the very large thermal peak

where neutrons accumulate until they are most likely absorbed in ^{1}H. The relative difference is below 0.5% and within the uncertainty of Serpent's results for most of the energy domain except in the thermal peak and again at 0.6 MeV. The thermal peak exhibits the same large swings as the higher energy region of the homogenized fuel criticality test. The large swings may indicate that there is a small error in how the tallies are accumulated and there is some roundoff error in regions where they tallies are scored frequently. Since the 0.6 MeV deviation is present in almost all spectra and so is ^{16}O, this may indicate that there is a small error in handling the 0.6 MeV scattering resonance in ^{16}O.

4.4 Comparison to Non-Remapping

To measure the benefit of sorting and remapping references in WARP, a comparison to a non-remapping version of WARP must be made. Table 4.5 shows the runtimes and multiplications factors of the criticality benchmarks for remapping and non-remapping versions of WARP. The remapping version performs much better than the non-remapping version when more complexity is added to the problem. For problems where neutrons die out very quickly, such as the Jezebel test, remapping does not benefit the simulation much because there are very few inner loop iterations done compared to a more complex problem, such as the assembly test.

Table 4.5: Comparison of the non-remapping and remapping versions of WARP for the four criticality test cases. 20/40 discarded/active criticality cycles and 10^6 histories per cycle.

test	Remapping	Non-Remapping	Δk or Ratio
Jezebel			
k_{eff}	1.0279	1.0279	0 pcm
Runtime	2.0147 m	1.84 m	0.91x
Homogenized Block			
k_{eff}	0.9426	0.941857	74.3 pcm
Runtime	2.8 m	2.9 m	1.04x
Pin Cell			
k_{eff}	0.3804	0.38063	-23 pcm
Runtime	7.0682 m	10.52 m	1.49x
Hex assembly			
k_{eff}	1.4454	1.4457	-30 pcm
Runtime	7.9703 m	95.85 m	12.03x

One of main benefits of remapping is that it eliminates completed neutrons from the GPU's address space. If remapping is not done, the GPU does not know where the active neutrons are and must check if each neutron is done at every kernel launch. This leads to spending a lot of time checking already completed data. In problems where neutrons

die out quickly, a neutron that is terminated on the first iteration may only be checked a small number of times compared to a problem where neutrons do not die out quickly. For example, if a problem only takes five inner loop iterations to process an entire neutron batch, a neutron that completes in the first iteration is only checked 4 additional times and 80% of its data loads result in no work being done. In a problem that takes one hundred iterations, a neutron that completes in the first iteration is checked 99 additional times and 99% of its data loads results in no work being done. Remapping results in 100% of a neutrons loads being used for useful work, since no already completed data is ever loaded. This is a major benefit in slowly-attenuating batches where stale data could be loaded repeatedly, resulting in wasted time.

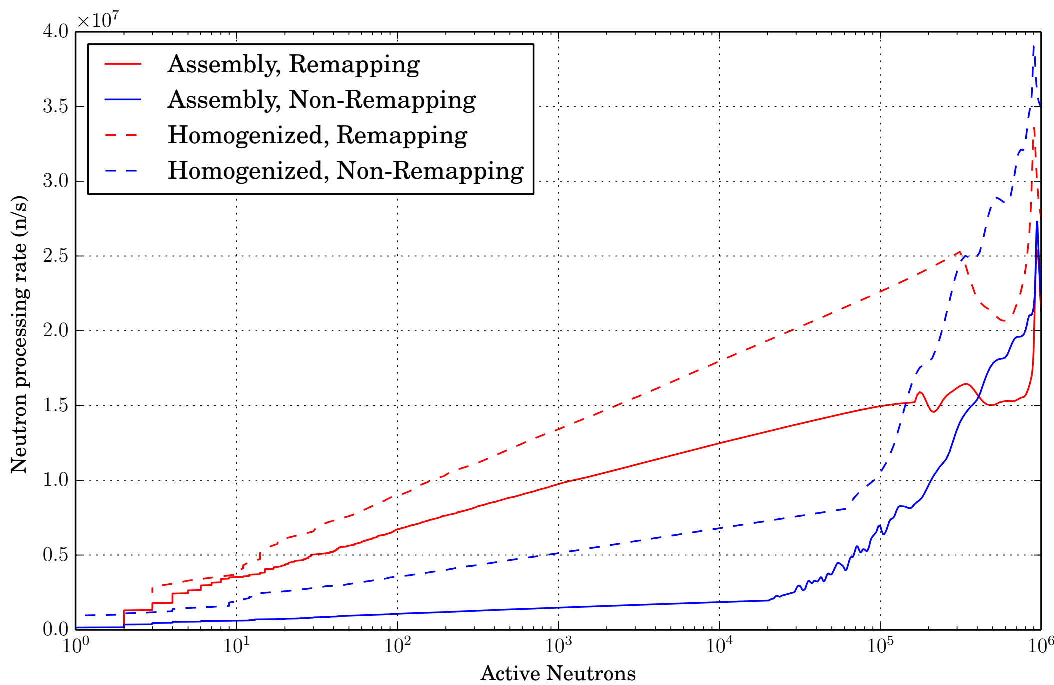


Figure 4.15: Neutron processing rate of WARP in the homogenized block and assembly test cases.

These effects of remapping are more clearly seen in the trends of Figure 4.15. This figure shows the rate at which each version is able to process neutrons vs. the number of active neutrons left in the batch for the homogenized block and the hex assembly. The data in the figure was quite noisy when there are few neutrons left to process since the iteration time is very small (∼1 ms) then. The timer used had trouble resolving times at this rate, and the data shown in the figure has been smoothed with a window 11 data points wide,and linearly interpolated where the timings were too close together and gave Inf or NaN results for processing rates.

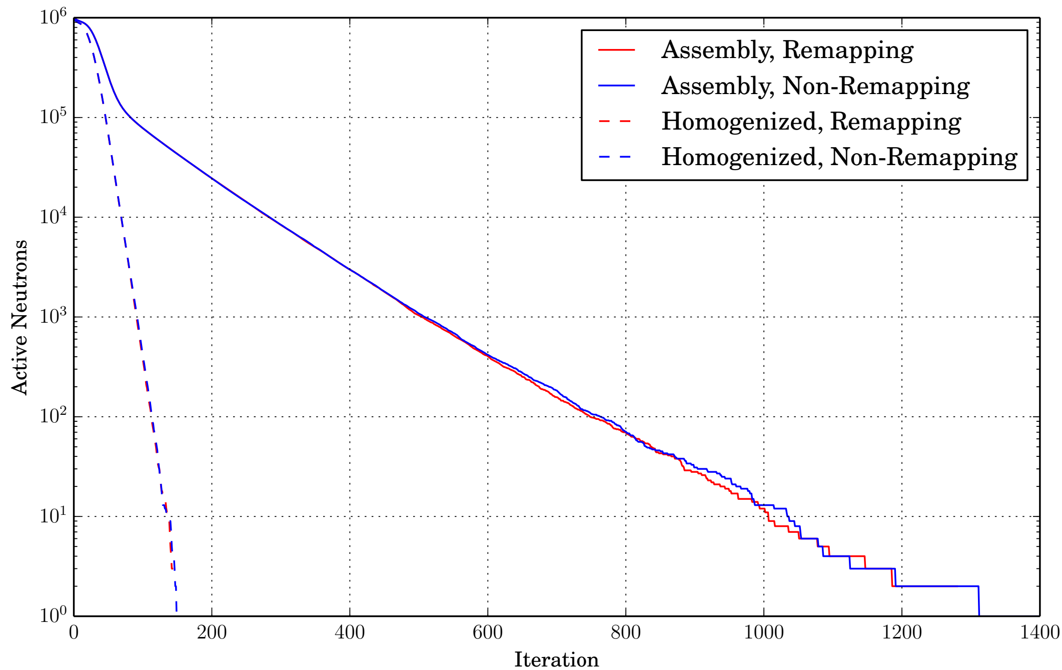


Figure 4.16: Active neutrons per iteration in WARP in the homogenized block and assembly test cases.

The non-remapping version is faster at large numbers of active neutrons (above $\sim 5 \times 10^5$) since accessing the entire dataset generally yields active data anyway. Conversely, the remapping version is faster when there few active neutrons left in the batch and accessing the entire grid generally yields terminated data. The remapping version always has an associated overhead. When there are few active neutrons left eliminating completed data and reducing the grid size is beneficial enough to overcome the overhead. When there are many active particles, the overhead dominates. Since the completed histories are pushed to the end of the remapping vector, the sort is only done the active particles, meaning its overhead scales with the active particle number. Since the radix sort is a very efficient operation, the addition cost of remapping is small (about 30% slower) and the benefit at small active neutron numbers pays for this (about 3x faster). In the non-remapping version, the additional cost at low numbers of active neutrons is due to launching a large grid where most of the blocks access terminated data and simply return without doing work. Compute cycles and, more importantly, loads from global memory are wasted simply to check if a neutron is terminated already or not. The dip in the processing curves around 8×10^5 is most likely an artifact of the data smoothing.

It therefore makes sense that the remapping version performs much better when the simulation spends significant time in the sub-300k active neutron region on the processing

Table 4.6: Summary table of the fraction of total kernel time spent in each WARP subroutine in criticality mode for each test case.

Remapping

Subroutine	Jezebel	Homogenized Block	Pincell	Hex assembly
OptiX Trace	3.8	18.5	26.5	41.1
Grid Search	0.5	6.5	5.1	4.7
Tally	0.2	1.3	0.5	0.4
Macroscopic	1.7	22.3	19.2	19.0
Microscopic	1.3	11.3	8.6	8.4
Radix Sort	1.7	5.3	8.5	10.3
Elastic Scatter	1.0	15.1	14.6	11.1
Inelastic Scatter	0.4	0.4	0.0	0.1
Continuum Scatter	0.3	0.2	0.0	0.1
Fission	0.2	0.2	0.2	0.2
Yield Reduction	0.1	0.0	0.0	0.0
Yield Rebase	0.0	0.0	0.0	0.0
Yield Prefix Sum	0.0	0.0	0.0	0.0
Pop Source	87.3	18.3	14.5	5.2

Non-Remapping

Subroutine	Jezebel	Homogenized Block	Pincell	Hex assembly
OptiX Trace	14.2	38.0	46.7	45.2
Grid Search	1.3	6.3	4.7	2.1
Tally	0.2	0.7	0.4	1.2
Macroscopic	1.3	8.8	6.6	2.8
Microscopic	1.5	5.6	2.6	1.9
Elastic Scatter	2.5	10.5	8.3	3.6
Inelastic Scatter	1.2	1.6	1.5	2.4
Continuum Scatter	1.1	1.5	1.6	2.4
Fission	0.4	0.7	0.9	1.3
Absorption	0.3	0.6	0.8	1.3
Active Reduction	3.8	7.3	11.2	20.1
Yield Reduction	3.4	6.2	9.6	17.3
Yield Rebase	0.0	0.0	0.0	0.0
Yield Prefix Sum	3.3	6.1	9.6	17.2
Pop Source	67.9	12.8	5.7	0.5

rate curve (about where the lines cross). In the Jezebel, homogenized block, pin cell, and assembly tests, 90%, 77%, 91%, and 96% of the transport iterations occur in sub-300k region,

respectively. It also makes sense that the relative number of *iterations* spent in the decades of Figure 4.15 is vaguely constant. The particles die out exponentially, and it takes roughly the same number of transport *iterations* to go from 10^7 to 10^6 active neutrons as it does to go from 10^2 to 10^1 active neutrons. This phenomenon manifests itself as the number of active neutrons per iteration being a straight line in Figure 4.16. The neutrons dying out exponentially is also why Figure 4.15 is plotted logarithmically with respect to active neutron count – to emphasize that equal spaces under the curves typically take the same number of iterations to complete. This shouldn't be confused for the amount of time the iterations take to complete, however. Iterations with more active neutrons typically take more time to complete, but have overall higher processing rates since the large neutron payload pays for overheads and latencies.

In a high leakage problem, like Jezebel, or a highly absorbing problem, like the homogenized block, the neutrons die off quickly. This means that the non-remapping simulation has a generally higher processing rate than problems where neutrons do not die off quickly. It has been mentioned that this is due to there being fewer stale loads, but another way to think about why this happens is that the active neutron population "leaps" down the curve faster and can skip very slow processing regions. This effect is especially pronounced at the end of the simulation where 10 particles can suddenly become 0, effectively skipping a large portion of slow processing. Adding many surfaces also keeps neutrons alive longer, since they often have to resample the materials as they cross boundaries.

Table 4.6 shows a breakdown of the amount of time spent in each subroutine in the remapping and non-remapping versions of WARP. The values shown are average values for an entire simulation. The values will also will not add to 100% since there are many other kernels launched and memory copies made, but these are typically only done at problem startup and are negligible compared to those listed. The kernels included in the table are the compute kernels of both the inner and outer transport loops.

In the non-remapping version, WARP spends most of its time in global routines like OptiX, macroscopic, scan, and source pop kernels, whereas the remapping version spends most of its time in OptiX and the reaction kernels (except in the Jezebel test, where almost all time is spent in the source pop). This is due to the fact that the non-remapping version never knows where the active data is and must launch kernels over the entire grid at every iteration. This makes the global operations expensive since they are always going through 10^6 elements. In the remapping version, the global functions access the remapping vector and only sort or scan the active data, making them much cheaper as neutrons are terminated. Remapping is almost always worth the effort, especially since radix is done in-place rather than having to be done and copied, but its main benefit is eliminating stale access, and therefore grid size, not reducing divergence in the reaction kernels.

In conclusion, WARP can produce results that are near those of Serpent and MCNP, but there are slight deviations in both the multiplication factor and the flux spectra that are outside of statistical error. Remapping data references is an effective way to keep GPU execution efficient when using an event-based transport algorithm, and WARP can produce results an order of magnitude faster than Serpent and MCNP.

Chapter 5

Conclusions and Future Work

WARP has been tested in criticality and fixed-source modes performing continuous energy Monte Carlo neutron transport in geometries containing 1 to 632 individual material regions. The materials used in the tests were combinations of ^{239}Pu, ^{238}U, ^{235}U, ^{16}O, ^{10}B, and ^{1}H, which were loaded from ENDF/B-VII ACE-formatted data libraries using the PyNE package.

Compared to Serpent 2.1.18 and MCNP 6.1, WARP produced the most accurate results in a UO_2 pin cell surrounded by water. In this case, 10^6 neutron histories were run and WARP calculated a multiplication factor 113 pcm and 19 pcm away from ones calculated by MCNP and Serpent, respectively, produced a flux spectrum within 2σ of Serpent's, and computed a fission source distribution almost identical to Serpent's. WARP is able to compute results in the pin cell case 82 and 57 times faster than MCNP and Serpent, respectively. Bare sphere, homogenized cube, and hexagonal lattice geometries were also tested. Considering all these cases, multiplication factors calculated by WARP vary between 4 and 807 pcm away from either Serpent or MCNP, and speedup factors over these codes vary between 11 and 82. Flux spectra compare well for most energies, as do fission source distributions.

5.1 General Conclusions

The WARP code developed in this work is currently the most detailed and feature-rich program in existence for performing continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs. It implements a novel adaptation of an event-based transport algorithm; loads standard data files; unionizes the nuclear data in a new, high performance way; accurately simulates each reaction type specified in the data; and uses a flexible, scalable, and optimized geometry representation.

The ultimate purpose for developing WARP was to accelerate accurate, continuous-energy neutron transport simulations in general, 3D geometries by using GPUs. By that metric it has been successful in its mission. It can produce results within fractions of a percent of qualified and benchmarked production codes like MCNP and Serpent. It can produce these results 11-80 times faster, depending on problem parameters and hardware. Along the

way, useful information regarding the importance of thread divergence, neutron termination, geometric acceleration structures, and dataset size has been collected and analyzed.

WARP's secondary goal of using standard nuclear data files, running in both fixed-source and criticality-source modes, calculating multiplication factors, and producing neutron spectra has also been achieved. Despite these milestones, WARP can only handle a single tally volume, and loading data from many isotopes has not been tested. It also does not have routines to handle thermal scattering data or unresolved resonance tables. There is still much work to be done in testing, stabilizing WARP's execution, adding physics and features, as well as continuing to optimize algorithms for efficient GPU execution.

This initial development phase of WARP is the first step in creating a full-featured reactor simulation program that runs on GPUs. Such an effort was needed to ensure the nuclear engineering community's codebase keeps abreast of modern programming techniques and hardware. WARP is by no means a mature code ready to be used in everyday nuclear reactor calculations, but rather it is a good starting point for more advanced development.

In its current state, WARP may be a good tool for multiplication factor searches, like determining reactivity coefficients by perturbing material densities and temperatures, since these types of calculations typically do not require many reaction rate tallies. WARP is also useful in a workstation environment since it currently can only be run on a single GPU card and can significantly accelerate calculations where only a few CPU cores would be available otherwise. Conversely, it is currently not good for depletion calculations since they can require many, many reaction rate tallies. This is not to say that GPUs couldn't perform depletion well, but rather that WARP is currently not suited to the task.

5.2 Specific Conclusions

The development of WARP has led to important conclusions regarding how to conduct neutron transport on GPUs. The first is that running with large datasets, and therefore a large number of threads, is important for good performance. It was found that in some cases, moving from 10^5 to 10^6 neutrons per batch in criticality mode increase performance by a factor of four or more. It is important to keep the GPU saturated with threads so it can effectively pipeline data loads.

The second conclusion is that remapping threads to active data is an effective way of raising the processing rate when the number of active neutrons becomes small; this also reduces thread divergence in reaction kernels. Using a radix sort to do the remapping is effective since it segregates reactions into contiguous blocks, efficient since it can be done in place and in $O(kN)$ time, and can eliminate completed data from being accessed if slight modifications to the standard reaction number encodings are made.

Most of the performance gain in remapping data references comes from being able to launch grids that are sized for only the active data rather than the entire dataset for both global and reaction kernels. A non-remapping algorithm does not keep track of where active data is, and therefore must launch a grid that covers the entire dataset. When the number

of active neutrons drops below about 30% of the initial number, the overhead and memory bandwidth cost of launching these extra threads, which only load a "done" bit and return, is more than the cost of performing the radix sort and edge detection. The majority of the transport iterations occur while there are less than 30% of the initial neutrons left, and remapping references is usually worthwhile.

Using the NVIDIA OptiX ray tracing framework was also shown to be an effective way to handle the geometry representation in WARP. OptiX is flexible, allows attachment of material and cell number to individual geometric primitives, can perform surface detection with a randomly-distributed and directed dataset, can incorporate the remapping vector created by a radix sort, and can do so fast enough to be used in WARP. The acceleration structures that OptiX can automatically build over the scene geometry was the initial reason for using it, and it was determined that the BVH builder and traverser provide the best performance as does using mesh primitive instancing rather than a transform node approach. The number of objects present in the scenes in reactors is small compared to many rendering scenes, and the SBVH acceleration structure does not perform as well. This is presumably due to some additional overhead related to traversing the objects that is not offset when few objects (less than a few hundred thousand) are present in the scene. Primitive instance provides better performance since using transform nodes requires traversing a deeper geometry tree, which also has more (redundant) data associated to it.

Further, assuming the capital prices for the GPU and CPU servers outlined in Table 1.2 and that CPU code scales linearly, the capital price per Monte Carlo "history power," or histories run per second, of a GPU is 2.8 times lower than that of a CPU on average for the tests done in this work, indicating that GPUs are a sound hardware investment for running Monte Carlo neutron transport.This conclusion only takes the results of WARP's initial development in account, i.e. simple materials and a single tally. Determining how to maintain high performance when both the number of materials and tallies are increased will be part of the future development of WARP.

5.3 Future Work

The initial goals of WARP have been completed, but there is much work still to be done if it is going to be of real use to the nuclear engineering community. Basic functionality is currently good enough to assure the GPUs can accelerate high-fidelity Monte Carlo neutron transport calculations, which was the point of this work, but many capabilities need to be expanded and ensured to scale well to large numbers of neutrons, isotopes, and geometrical zones.

The geometry is currently handled by OptiX since it provides a convenient way to obtain high-performance results, but its execution had to be coerced into providing WARP with the information it needed, namely the material number via the point-in-polygon algorithm. The way OptiX executes this algorithm is not efficient since it must be done iteratively using OptiX's native functions instead of calculating the ordered list of intersections in a

single trace. NVIDIA is releasing "OptiX Prime" with OptiX 3.5 [65], which promises to provide a more "to the metal" ray tracing experience, and might be leveraged to provide more efficient single-traverse functionality. OptiX could also be replaced by Rayforce [66], a high-performance GPU ray tracing library developed by VSL that has this functionality built-in and is currently available free of charge for noncommercial use.

The geometry routines could also be replaced by handwritten routines that use combinatorial solid geometry like Serpent and MCNP. This would make writing input for WARP more like what most nuclear engineers are already used to and, more importantly, could provide a potential performance increase. A universe-based CSG representation may map very well to the GPU and may even be able to fit inside of shared memory for small numbers of surfaces. The surfaces may also be able to be bound to texture memory, which could provide a performance boost since it automatically caches for spatial locality.

Using an efficient CSG method would further lend itself to using Woodcock delta-tracking for the neutrons and thus getting rid of the tracing algorithms and libraries altogether. These algorithms account for about 50% of execution time, and WARP's performance could be doubled by making the geometry routines more efficient. If OptiX is determined to be the best option out of these others, however, a routine would need to be developed to automatically space coincident surfaces appropriately without specific user input.

WARP's execution can also be improved. The amount of memory required per neutron was not tracked in this initial development, much less optimized, since developing a functioning code was the main priority. Reducing the memory needed per neutron would be highly beneficial in the sense that more concurrent neutrons could be launched using the reclaimed memory space. Dynamic parallelism can be implemented to minimize kernel launch overhead and host-device communication in the inner transport loop. Dynamic parallelism is a feature introduced into the NVIDA Kepler GPUs that allows kernels to be launched from kernels, and could eliminate the host needing to contain the main transport loop. Neither CUDPP or OptiX support its use, however. CUDPP could be replaced with newer, higher-performance libraries (e.g. CUB) that do support dynamic parallelism, but OptiX would have to be replaced by a handwritten kernel to perform the necessary geometric tasks. Textures could be thoroughly investigated as they might provide better performance in tasks where their free linear interpolation and spatial caching could provide a performance boost, like in an energy grid search on a tree structure. Alternatively, using optimized graph search libraries, such as "gunrock," could be used to perform the energy grid search.

WARP also currently only supports fixed-source mode in the non-remapping version as it requires popping secondary neutrons back into the active neutron pool after every iteration of the inner transport loop. This operation is expensive since it is a global operation that must be done often. This algorithm could be changed to be more like a criticality run, where the primary neutrons are all transported together, then the next (smaller) generation of secondary particles are transported, then the next, and so on. This way, the pop routine is only executed in the outer loop, and could produce faster results. Multi-GPU support should also be added so that WARP can be used effectively on computers with more than a one GPU.

If an entire overhaul of the WARP transport algorithm is feasible, using a SM-based algorithm might be investigated instead of current global one. This type of algorithm would treat each SM as an independent processor, and would provide each a bank of neutrons to transport, as is done by Liu and Henderson [26, 24]. This way, neutron data could be stored in very fast shared memory, but using this memory space would compete with storing geometric information there. Also, since a smaller set of neutrons could be stored, the SMs would need to communicate to determine which of the next neutrons they would take out of the global bank, or they would need to periodically rendezvous to shared source information and ensure that the distributions they use are each converged. This type of transport algorithm would also preclude using OptiX, since it does not have SM-level functionality [45].

Data access patterns are very important on the GPU, and there are a few straight forward modifications that could be made to WARP in the future. The first is using Legendre expansion data for the angular dependencies of anisotropic scattering instead of using tabular data. This method uses more computation and less data than tabular distributions, and would probably perform well on the GPU. Since global memory comes at a premium on GPUs, an on-the-fly temperature treatment for nuclides would likely be required if more than a handful of isotopes are desired at more than one temperature. Methods like those used in Serpent could be adapted for use on the GPU [31]. On-the-fly methods reduce the amount of storage needed, but they require more computation per data element loaded since the loaded value is adjusted according to the temperature of the material. This kind of method may work well on the GPU since GPUs have a larger FLOP/byte ratio than CPUs and the additional work may cost little. Other than the how to represent and adjust the data, an efficient way to handle situations where there are many different material and isotopes present needs to be explored. The work done by Scudiero on porting OpenMC's macroscopic cross section processing benchmarking tool, "xsbench," may elucidate this endeavor [13, 54].

WARP would gain usability if more features were incorporated as well. Currently, WARP treats all neutrons equally, but adding neutron weight would allow many variance reduction techniques to be implemented. Importance cutoffs could be used to terminate secondary neutrons in fixed-source runs, leading to shorter runtimes; cell importances and implicit absorption could help improve tally statistics. Developing an efficient way to include many reaction rate tallies would also make WARP useful for performing depletion analysis. It would also be helpful for WARP to have statistical tests like Shannon entropy to ensure the fission source is fully converged before tallies and multiplication factors are accumulated.

WARP still has bugs, and a large part of future development will be tracking them down to ensure that accurate results are produced. Reproducibility using the same random number seeds will also be investigated to ensure consistent results can be produced and that there are no systematic errors present in WARP. Ensuring reproducibility will also be necessary in making a test suite for WARP, so future users can have confidence in their results and future developers can know their modifications do not introduce new errors into WARP.

Releasing WARP as open source software is in progress and is pending University approval. OpenMC has opened the door for open source neutron transport codes, and WARP will hopefully follow in its footsteps. Releasing the source openly has obvious benefits like

providing potential users with a convenient and transparent way of obtaining the software, as well as allowing for valuable contributions from the community.

Appendix A

Supplementary Results

This appendix includes results that were redundant or not directly relevant to the discussions in the chapters. They are provided here as supplementary and complimentary information.

A.1 2D Scattering on an NVIDIA Tesla C2075

As was mentioned in Section 3.1, different results were obtained in the 2D scattering study when the GPU implementations were run on a NVIDIA Tesla C2075 card instead of a Tesla K20. Figure A.1 shows the speedup factors, $F_s = t_{\text{CPU}}/t_{\text{GPU}}$, of the GPU implementations discussed in Section 3.1. This benchmark was run on the same server with a 8-core AMD Opteron 6128 CPU clocked at 2.0GHz, but on a Tesla C2075 card. The task-parallel implementation performs best, with a maximum speedup of around 29x. The remapping implementation has the next best performance, with about a 20x speedup over the CPU. The batched implementation's performance departs from the remapping implementation at 10^5 particles and even starts to deteriorate between 10^6 and 10^7 particles. This is due to the transport having to be done in batches at this point due to the maximum block number of 65,536.

These results are quite different than those shown in Section 3.1 where the remapping version was decidedly faster than the others at large particle numbers. The C2075 card is a Fermi architecture card, and has slightly different memory characteristics that the newer K20, which is a Kepler architecture card. The likely reason that the older architecture is faster for these applications is that an important change was made in Kepler regarding the L1 cache. The L1 cache was introduced in Fermi, and sits closest to the registers and processing units on the SMs. "Devices of compute capability 2.x come with an L1/L2 cache hierarchy that is used to cache local and global memory accesses [41]." In Fermi, the L1 cache acts like a traditional CPU L1 and buffers access to the L2, which in turn buffers access to global memory. In these simple applications, all particle data is access in global memory, but checks the L1 first. A cache hit would increase access rate considerably, and since the history-based

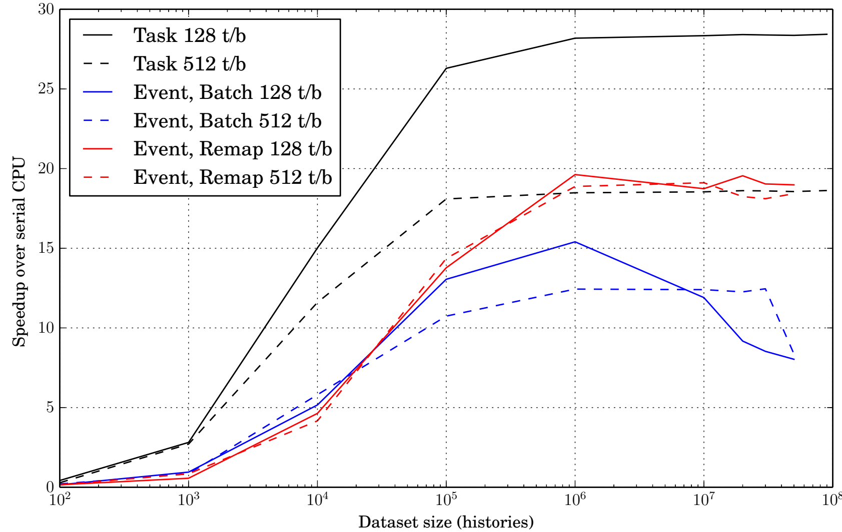


Figure A.1: Speedup factors of the GPU implementations over the CPU implementation on a Tesla C2075.

version keeps the same particle data in it (or registers) for its entire lifetime, and the particle does not carry much data, it most likely lives in L1 for the entire life of the particle. In the event-based versions, multiple kernels are launched for every operation, and the caches are cleared in between launches. This means that the particle data is fetched from global memory every time instead of being loaded from much faster L1. This may also explain why the task-based performance drops to that of the remapping version when the threads per block is increased from 128 to 512. The additional pressure on the registers causes the data to spill into L2 and the benefits of L1 caching (or always keeping the data in registers) are lost.

In Kepler, the L1 is reserved for register spills and local data access. "L1 caching in Kepler GPUs is reserved only for local memory accesses, such as register spills and stack data. Global loads are cached in L2 only (or in the Read-Only Data Cache) [67]." This means that on Kepler, the task-based version does not benefit as much from from L1 cache hits as it did on Fermi, and must go to L2 or global memory every time particles data is loaded or spilled from registers.

The remapping version performs better than the non-remapping version for the very same reasons the remming version of WARP performs better than the non-remapping version. Remapping creates more efficient, if non-coalesced, data access. Since the amount of data required per neutron is greater in WARP than that of the particles in this preliminary study, the L1 cache would most likely be overflown anyway, and the remapping algorithm would be the best performing on Fermi as well as Kepler architectures.

A.2 Serpent Fission Source Distributions

Included here are the fission source distributions of the criticality tests produced by Serpent.

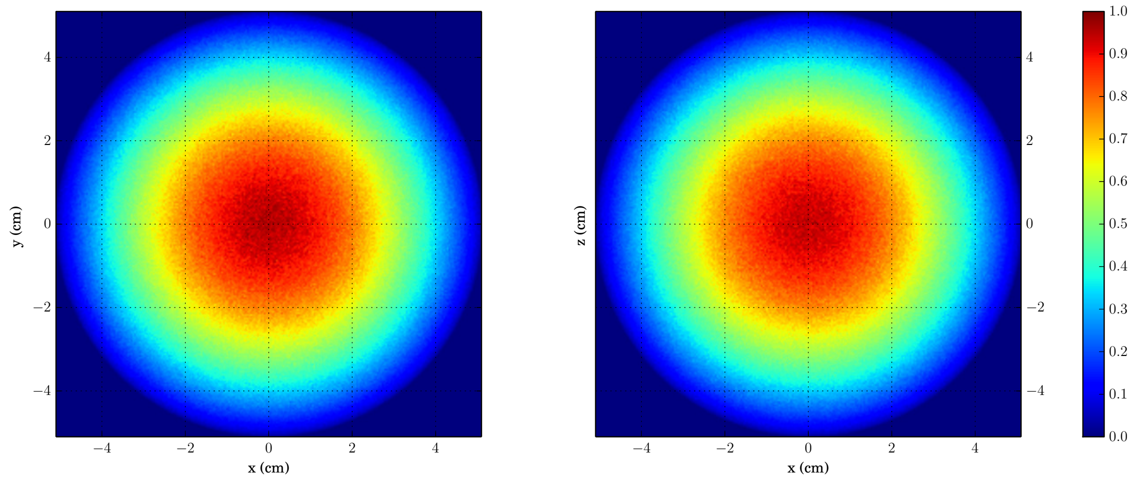


Figure A.2: Fission source distribution from Serpent 2.1.18 of a “Jezebel” bare Pu-239 sphere.

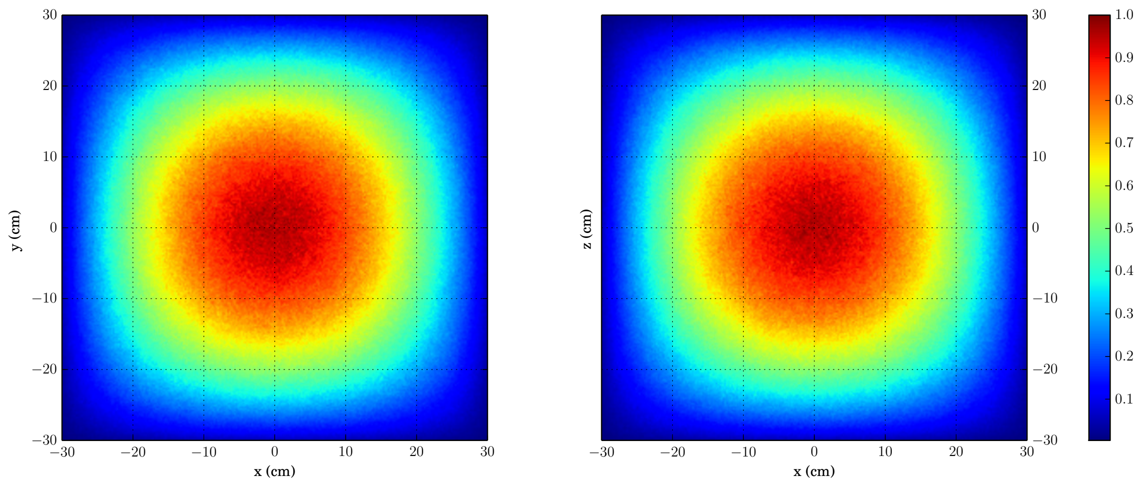


Figure A.3: Fission source distribution from Serpent 2.1.18 of a homogenized block of UO_2 and water.

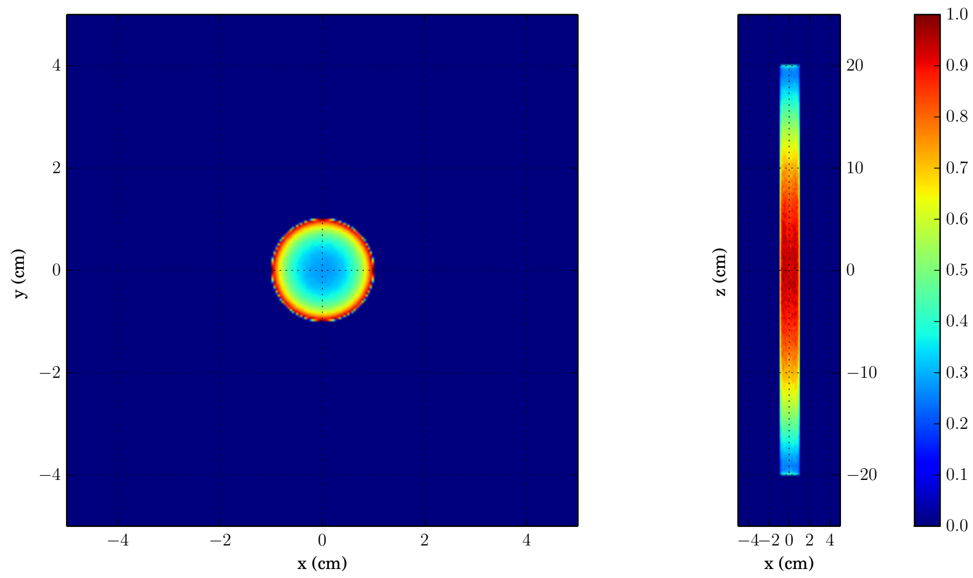


Figure A.4: Fission source distribution from Serpent 2.1.18 of a single UO_2 pin surrounded by a block of water

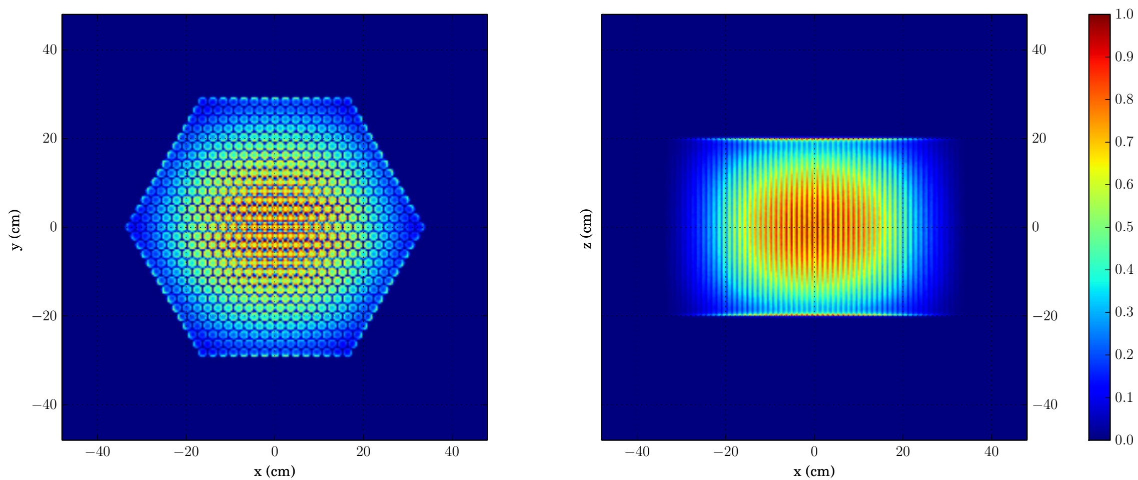


Figure A.5: Fission source distribution from Serpent 2.1.18 of a hexagonal array of UO_2 pins in water.

Appendix B

Additional Tables

Table B.1: Reaction number encodings in WARP.

Number	Description
2, 51-91	Scatter reaction unmodified from standard MT number
800	Resample flag History is skipped until OptiX determines the material
811-845	Secondary-producing absorption reactions that have NOT yet been processed by the fission kernel
911-945	Secondary-producing absorption reactions that have been processed, history terminated
997	OptiX miss (error)
999	Leakage
>1102	Capture reactions 1000 added to standard MT number

Table B.2: Geometric primitive number encodings in WARP.

Number	Description
1	Axis-aligned Cuboid
2	Z-parallel finite cylinder
3	Z-parallel finite hexagonal prism
4	Sphere

Table B.3: ENDF MT numbers and the reactions they stand for. [35]

MT	Reaction	Description
Sum and Elastic Cross Sections		
1	(n,total)	Neutron total cross section.
2	(z,z0)	Elastic scattering cross section.
3	(z,nonelastic)	Nonelastic cross section.
4	(z,n)	Production of one neutron in the exit channel. Sum of MT=50-91.
5	(z,anything)	Sum of all reactions not given explicitly in another MT number.
10	(z,continuum)	Total continuum reaction; excludes all discrete reactions.
27	(z,abs)	Absorption. Sum of MT=18 and MT=102-117.
101	(z,disap)	Disappearance. Sum of MT=102-117.
Neutron-Producing & Continuum Reactions		
11	(z,2nd)	Production of two neutrons and a deuteron, plus a residual.
16	(z,2n)	Production of two neutrons, plus a residual.
17	(z,2n)	Production of three neutrons, plus a residual.
18	(z,fission)	Total fission.
19	(z,f)	First-chance fission.
20	(z,nf)	Second-chance fission.
21	(z,2nf)	Third-chance fission.
22	(z,na)	Production of a neutron and alpha particle, plus a residual.
23	(z,n3a)	Production of a neutron and three alpha particles, plus a residual.
24	(z,2na)	Production of two neutrons and an alpha particle, plus a residual.
25	(z,3na)	Production of three neutrons and an alpha particle, plus a residual.
28	(z,np)	Production of a neutron and a proton, plus a residual.
29	(z,n2a)	Production of a neutron and two alpha particles, plus a residual.
30	(z,2n2a)	Production of two neutrons and two alpha particles, plus a residual.
32	(z,nd)	Production of a neutron and a deuteron, plus a residual.
33	(z,nt)	Production of a neutron and a triton, plus a residual.
34	(z,n3He)	Production of a neutron and a 3He particle, plus a residual.
35	(z,nd2a)	Production of a neutron, a deuteron, and two alpha particles, plus a residual.
36	(z,nt2a)	Production of a neutron, a triton, and two alpha particles, plus a residual.
37	(z,4n)	Production of four neutrons, plus a residual.
38	(z,3nf)	Fourth-change fission.
41	(z,2np)	Production of two neutrons and a proton, plus a residual.
42	(z,3np)	Production of three neutrons and a proton, plus a residual.
44	(z,n2p)	Production of a neutron and two protons, plus a residual.
45	(z,npa)	Production of a neutron, a proton, and an alpha particle, plus a residual.
Neutron-Producing Discrete Reactions		
51	(z,n1)	Production of a neutron, nucleus in the first excited state.
52	(z,n2)	Production of a neutron, nucleus in the second excited state.
...		
90	(z,n40)	Production of a neutron, nucleus in the 40th excited state.
91	(z,nc)	Production of a neutron in the continuum.
Reactions That Do Not Produce Neutrons		
102	(z,gamma)	Radiative capture.
103	(z,p)	Production of a proton, plus a residual.
104	(z,d)	Production of a deuteron, plus a residual.
105	(z,t)	Production of a triton, plus a residual.
106	(z,3He)	Production of a He particles, plus a residual.
107	(z,a)	Production of an alpha particle, plus a residual.
108	(z,2a)	Production of two alphas, plus a residual.
109	(z,3a)	Production of three alphas, plus a residual.
111	(z,2p)	Production of two protons, plus a residual.
112	(z,pa)	Production of a proton and an alpha particle, plus a residual.
113	(z,t2a)	Production of a triton and two alphas, plus a residual.
114	(z,d2a)	Production of a deuteron and two alphas, plus a residual.
115	(z,pd)	Production of a proton and a deuteron, plus a residual.
116	(z,pt)	Production of a proton and a triton, plus a residual.
117	(z,da)	Production of a deuteron and an alpha particle, plus a residual.

Bibliography

[1] World Nuclear Association. *Energy for the World - Why Uranium?* Dec. 2012. URL: http://www.world-nuclear.org/info/Nuclear-Fuel-Cycle/Introduction/Energy-for-the-World---Why-Uranium-/.

[2] K.S. Krane. *Introductory Nuclear Physics*. Wiley, 1987. ISBN: 9780471805533. URL: http://books.google.com/books?id=ConwAAAAMAAJ.

[3] James J. Duderstadt and Louis J. Hamilton. *Nuclear Reactor Analysis*. New York, NY: John Wiley & Sons, Inc., 1976.

[4] Richard S. Treptow. "E = mc2 for the Chemist: When Is Mass Conserved?" In: *Journal of Chemical Education* 82.11 (2005), p. 1636. DOI: 10.1021/ed082p1636. eprint: http://pubs.acs.org/doi/pdf/10.1021/ed082p1636. URL: http://pubs.acs.org/doi/abs/10.1021/ed082p1636.

[5] Gene Rowe and Mark Abkowitz. "NUWASTE Results, Scenario 2.1, Characteristics of U.S. Spent Fuel Inventory as of December 2009". In: U.S. Nuclear Waste Technical Review Board Workshop on Evaluation of Waste Streams Associated with LWR Fuel Cycle Options. June 2011.

[6] World Nuclear Association. *The Economics of Nuclear Power*. Feb. 2014. URL: http://www.world-nuclear.org/info/Economic-Aspects/Economics-of-Nuclear-Power/.

[7] *Popular GPU-Accelerated Applications*. Pamphlet. NVIDIA Co. 2012.

[8] Victor W. Lee et al. "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU". In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 451–460. ISSN: 0163-5964. DOI: 10.1145/1816038.1816021. URL: http://doi.acm.org/10.1145/1816038.1816021.

[9] Mike James. *Why We Don't Need Even More Programming Languages*. Dec. 2011. URL: http://www.i-programmer.info/professional-programmer/i-programmer/3466-why-we-dont-need-even-more-programming-languages.html.

[10] *CUDA C Programming Guide*. PG-02829-001_v5.5. NVIDIA Co. Santa Clara, CA, 2013.

[11] X-5 Monte Carlo Team. *MCNP - A General Monte Carlo N-Particle Transport Code, Version 5*. Volume I: Overview and Theory. (Revised 2/1/2008). Los Alamos National Laboratory, Los Alamos, NM, Apr. 2003.

[12] Jaakko Leppänen. "Development of a New Monte Carlo Reactor Physics Code". D.Sc. Dissertation. Helsinki, Finland: Helsinki Institute of Technology, 2007.

[13] *OpenMC Documentation*. 0.5.3. http://mit-crpg.github.io/openmc/index.html. Cambridge, MA, 2013.

[14] P. Kogge et al. *Exascale computing study: Technology challenges in achieving exascale systems*. Tech. rep. University of Notre Dame, CSE Dept., 2008.

[15] *Top500 Supercomputer Rankings*. `http://www.top500.org/lists/2013/11/`. November 2013.

[16] Brian Catanzaro. *An Introduction to CUDA/OpenCL and Manycore Graphics Processors*. Lecture slides. University of California, Berkeley CS267 - Applications of Parallel Computers. Feb. 2011.

[17] Joshua Ruggiero. *Measuring Cache and Memory Latency and CPU to Memory Bandwidth*. Tech. rep. Intel Co., 2008.

[18] Yunsup Lee et al. "Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators". In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), pp. 129–140. ISSN: 0163-5964. DOI: `10.1145/2024723.2000080`. URL: `http://doi.acm.org/10.1145/2024723.2000080`.

[19] *The AMD Opteron processor is the ideal solution for High Performance Computing*. Tech. rep. Advanced Micro Devices, Inc., 2011. URL: `http://sites.amd.com/us/Documents/AMD_Opteron_ideal_for_HPC.pdf`.

[20] *PSSC Labs PowerWulf Cluster Upgrade*. Quotation #33-111610-06. PSSC Labs. Lake Forest, CA, 2011.

[21] *PSSC Labs CMAS Contract #3-07-70-2493A*. Quotation #33-011912-02. PSSC Labs. Lake Forest, CA, 2012.

[22] *Tesla C2075 Computing Processor Board*. BD-05880-001_v02. NVIDIA Co. Santa Clara, CA, Sept. 2011.

[23] *List of AMD Opteron microprocessors*. `http://en.wikipedia.org/wiki/List_of_AMD_Opteron_microprocessors`. Accessed April 2014.

[24] Nicholas Henderson et al. "A CUDA Monte Carlo simulator for radiation therapy dosimetry based on Geant4". In: Supercomputing in Nuclear Applications and Monte Carlo (SNA+MC). Paris, France, Oct. 2013.

[25] *Archer Website*. `http://www.rpi.edu/dept/radsafe/public_html/GPU_project/ARCHER.html`. Apr. 2014.

[26] Tianyu Liu et al. “A Monte Carlo Neutron Transport Code For Eigenvalue Calculations on a Dual-GPU System and CUDA Environment”. In: PHYSOR. Knoxville, Tennessee, Apr. 2012.

[27] Tianyu Liu et al. “A comparative study of history-based versus vectorized Monte Carlo methods in the GPU/CUDA environment for a simple neutron eigenvalue problem”. In: Supercomputing in Nuclear Applications and Monte Carlo (SNA+MC). Paris, France, Oct. 2013.

[28] Adam Gregory Nelson. “Monte Carlo Methods for Neutron Transport on Graphics Processing Units Using CUDA”. M.S. Thesis. University Park, Pennsylvania: Pennsylvania State University, 2009.

[29] 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: `http://dx.doi.org/10.1016/0149-1970(84)90024-6`. URL: `http://www.sciencedirect.com/science/article/pii/0149197084900246`.

[30] 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: `http://dx.doi.org/10.1016/0149-1970(91)90033-L`. URL: `http://www.sciencedirect.com/science/article/pii/014919709190033L`.

[31] *Serpent Website*. `http://montecarlo.vtt.fi/`. Apr. 2014.

[32] *U.S. Nuclear Regulatory Commission (NRC) 2013-2014 Information Digest*. Tech. rep. U.S. Nuclear Regulatory Commission, August 2013.

[33] M. Holloway and R. Baker. *Note on the Origin of the Term “Barn”*. Tech. rep. LANL No. LAMS-523, 13 March 1944.

[34] Wikimedia Commons user Fastfission. `http://commons.wikimedia.org/wiki/File:Binding_energy_curve_-_common_isotopes.svg`. Public domain image. Sept. 2012.

[35] Los Alamos National Laboratory. `http://t2.lanl.gov/nis/endf/`. An Introduction to the ENDF Formats. Jan. 1998.

[36] Jasmina Vujić. *NE-150/250 Class Notes*. 2014.

[37] Forrest Brown’s general assembly speech. Supercomputing in Nuclear Applications and Monte Carlo (SNA+MC) Conference, Oct. 30, 2013.

[38] E. M. Gelbard. *Epithermal Scattering in VIM*. Tech. rep. Argonne National Laboratory, 1979.

[39] C. J. Everett and E. D. Cashwell. *A Third Monte Carlo Sampler*. Tech. rep. LANL No. LA-9721-MS, March 1983.

[40] C.A. Mack. “Fifty Years of Moore’s Law”. In: *Semiconductor Manufacturing, IEEE Transactions on* 24.2 (May 2011), pp. 202–207. ISSN: 0894-6507. DOI: `10.1109/TSM.2010.2096437`.

[41] *Tuning CUDA Applications for Fermi*. DA-05612-001_v1.5. NVIDIA Co. Santa Clara, CA, May 2011.

[42] *Parallel Thread Execution ISA*. v3.2. NVIDIA Co. Santa Clara, CA, July 2013.

[43] David B. Kirk and Wen-mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington, MA: Morgan Kaufmann Publishers, 2010.

[44] Peter Messmer. *GPU Computing Overview*. HRSK-II GPU Computing Tutorial, Technische Universität Dresden. May 2013.

[45] *OptiX Programming Guide*. v3.0. NVIDIA Co. Santa Clara, CA, Nov. 2012.

[46] Steven G. Parker et al. "OptiX: A General Purpose Ray Tracing Engine". In: *ACM Transactions on Graphics* (Aug. 2010).

[47] Donald H. House. *Matrix Algebra and Affine Transformations*. Lecture Notes. Clemson University, Computer Science 401, Technical Foundations of Digital Production II. Spring 2013. URL: `http://people.cs.clemson.edu/~dhouse/courses/401/notes/affines-matrices.pdf`.

[48] Wikimedia Commons user Schreiberx. `http://commons.wikimedia.org/wiki/File:Example_of_bounding_volume_hierarchy.svg`. Creative Commons BY-SA 3.0 License. Dec. 2011.

[49] Wikimedia Commons user Prométhée33. `http://commons.wikimedia.org/wiki/File:KD-Tree_part2.png`. Creative Commons BY-SA 3.0 License. Feb. 2013.

[50] Eddy Z. Zhang et al. "On-the-fly elimination of dynamic irregularities for GPU computing". In: *SIGPLAN Not.* 46.3 (2011), pp. 369–38.

[51] Ryan M. Bergmann, Jasmina L. Vujić, and Noah A. Fischer. "2D Mono-Energetic Monte Carlo Particle Transport on a GPU". In: ANS Winter Meeting. San Diego, California, Nov. 2012.

[52] Qi Xu, Ganglin Yu, and Kan Wang. "Research on GPU Acceleration for Monte Carlo Criticality Calculation". In: Supercomputing in Nuclear Applications and Monte Carlo (SNA+MC). Paris, France, Oct. 2013.

[53] Qi Xu et al. "A GPU-Based Local Acceleration Strategy for Monte Carlo Neutron Transport". In: ANS Winter Meeting. San Diego, California, Nov. 2012.

[54] Tony Scudiero. "Monte Carlo Neutron Transport - Simulating Nuclear Reactions One Neutron at a Time". In: GPU Technology Conference. San Jose, California, Mar. 2014.

[55] Jaakko Leppänen. "Two practical methods for unionized energy grid construction in continuous-energy Monte Carlo neutron transport calculation". In: *Annals of Nuclear Energy* 36.7 (2009), pp. 878–885. ISSN: 0306-4549. DOI: `http://dx.doi.org/10.1016/j.anucene.2009.03.019`. URL: `http://www.sciencedirect.com/science/article/pii/S0306454909001108`.

[56] Anthony Scopatz et al. *PyNE: The Nuclear Engineering Toolkit*. 2014. URL: `pyne.io`.

[57] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001–. URL: http://www.scipy.org/.

[58] D. M. Beazley. “Automated Scientific Software Scripting with SWIG”. In: *Future Gener. Comput. Syst.* 19.5 (July 2003), pp. 599–609. ISSN: 0167-739X. DOI: 10.1016/S0167-739X(02)00171-1. URL: http://dx.doi.org/10.1016/S0167-739X(02)00171-1.

[59] D.E. Knuth. “Optimum binary search trees”. English. In: *Acta Informatica* 1.1 (1971), pp. 14–25. ISSN: 0001-5903. DOI: 10.1007/BF00264289. URL: http://dx.doi.org/10.1007/BF00264289.

[60] *CUDA Toolkit 5.0, CURAND Guide*. PG-05328-050_v02. NVIDIA Co. Santa Clara, CA, Sept. 2012.

[61] Pierre L’Ecuyer. “Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure”. In: *Mathematics of Computation* 68.225 (2009), pp. 249–260.

[62] *NVIDIA Tesla K20 Specifications*. http://videocardz.com/nvidia/tesla-nvidia/tesla-k20. Accessed April 2014.

[63] Massimiliano Fratoni. *Spectrum Plot from MCNP Data*. Internal department guide. Created July 26, 2007.

[64] OECD Nuclear Energy Agency. *International Handbook of Evaluated Criticality Safety Benchmark Experiments*. Nuclear Energy Agency, OECD, 1995.

[65] *OptiX Programming Guide*. v3.5. NVIDIA Co. Santa Clara, CA, Nov. 2012.

[66] Christiaan Gribble and Lee A. Butler. “Advances in High-Performance GPU Ray Tracing for Physics-Based Simulation”. In: GPU Technology Conference. San Jose, California, Mar. 2013.

[67] *Tuning CUDA Applications for Kepler*. DA-06288-001_v6.0. NVIDIA Co. Santa Clara, CA, Feb. 2014.