GPU-Oriented Algorithms for Continuous Energy Monte Carlo Neutron Transport

by

Gavin Ridley

B.S. Nuclear Engineering, U.T. Knoxville (2018)

Submitted to the Department of Nuclear Science and Engineering
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTATIONAL NUCLEAR SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

Authored by: Gavin Ridley
Department of Nuclear Science and Engineering
August 8, 2024

Certified by: Benoit Forget
KEPCO Professor of Nuclear Engineering, Thesis Supervisor

Accepted by: Ju Li
Battelle Energy Alliance Professor of Nuclear Science and Engineering
Professor of Materials Science and Engineering

GPU-Oriented Algorithms for Continuous Energy Monte Carlo Neutron Transport

by

Gavin Ridley

Submitted to the Department of Nuclear Science and Engineering
on August 8, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTATIONAL NUCLEAR SCIENCE AND ENGINEERING

ABSTRACT

The advent of graphics processing units (GPUs) has brought computing to new heights with deep learning models, now deployed ubiquitously and touching the lives of many. While GPU hardware may be ideal for deep learning, its full potential has yet to be realized in a variety of scientific computing applications. Often, paradigm shifts in the data formalisms and algorithmic choices used to solve scientific computing problems must take place to fully leverage GPUs. A quintessential example of this shift has been the move towards matrix-free, high-order finite element formulations researched under the Exascale Computing Project. Similar groundbreaking shifts are only starting to take place in continuous energy Monte Carlo (MC) neutron transport simulations. These simulations play a crucial role in designing fission, fusion, and security systems that may play a pivotal role in the transition to a decarbonized world.

This work contributes to adapting continuous energy MC neutron transport simulations for the GPU computing era. We first summarize some changes made to other scientific computing applications that led to performance gains on GPUs, which informed our independent development of a CUDA-based version of OpenMC, an open-source continuous energy MC neutron and photon transport code. Fortunately, the historical event-based MC simulation modality developed extensively through the 1980s for vector computers provides an excellent basis for GPU computing. Adapting a full-physics, continuous energy MC neutron transport simulation for GPUs is a feat only completed by a few institutions across the world, so we share some software development tricks that facilitated this task. We then identify a variety of algorithmic optimizations that improved the performance of the baseline CUDA application, and identify areas for further development.

Based on experience adapting a full-physics continuous energy MC code for GPU, we identify two pieces of the simulation which can be improved for GPU computing: resonance upscatter handling and unresolved resonance modeling. Our new method for modeling resonance upscatter is based on a novel, fundamental observation regarding the resonance upscatter effect. The relative speed tabulation (RST) method developed by other GPU MC researchers can be underpinned by a universal special function we have named the incomplete Faddeeva function, which is closely related to the incomplete Goodwin-Staton integral. Our research develops numerical algorithms for efficient, accurate computation of the incomplete Faddeeva function and identifies some properties of the function. We then present a specialized root-finding algorithm that takes advantage of the structure of the problem to efficiently sample the resonance upscatter effect on GPUs. This obviates the need to rely on RST tables or a zero kelvin pointwise cross section, freeing precious GPU memory while using a GPU-friendly memory access pattern.

Continuing in the same direction, we focus on unresolved resonance region (URR) cross-section modeling, which was shown to induce a 20% computational efficiency degradation on GPUs. We review the requirements to model cross sections in the unresolved resonance regime, and provide what is to our knowledge the first rigorous demonstration that URR modeling can be reduced to a one-dimensional probabilistic model in addition to some expectation values of partial cross sections conditioned on the total. Through three asymptotic arguments covering different resonant behavior regimes, we show that the normal inverse Gaussian distribution is the natural choice for modeling the total neutron cross-section distribution. Rather than inducing a performance degradation, we show the new URR modeling technique in fact outperforms a pointwise infinite-dilute approach when it is used to model the URR region.

Thesis supervisor: Benoit Forget
Title: KEPCO Professor of Nuclear Engineering

Acknowledgments

I'd like to express deepest gratitude to me parents, Christine and Ted Ridley for inspiring me to learn things independently, instilling an appreciation for nature, and for supporting me through the toughest of times. My thanks also go to my sister for her enduring friendship, to my friends and mentors on the robotics team in high school who started my journey into engineering: Tracy, Gabe, Skylar, Reid, Joseph, Eric. It was when my dad encouraged me to stay in the robotics club in freshman year of high school that I was set on the path to engineering. A few years later, I decided that I couldn't get enough engineering, hence the next 200 pages of material.

My advisor Benoit Forget has provided support not just in figuring out tricky research problems, but also in supporting me when work became tough. Not every graduate student has the privilege of having a compassionate research advisor, and without his patience I could not have completed this thesis. I sought graduate school initially to learn about neutron transport theory and the related computational methods, and his excellently delivered reactor physics II class satisfied that wish.

Dr. Tim Burke provided valuable feedback and support throughout this thesis, particularly regarding resonance upscatter sampling using windowed multipole data. The opportunity he and Dr. Travis Trahan provided to work a summer internship at Los Alamos National Laboratory in the XCP-3 group was both pleasant and conducive to the success of my graduate research.

Our "family" dinners with Peter, Aaron, and Nick brought my nose out of the computer on Wednesday evenings. Taking time to decompress hiking with Lorenzo, Ralph, and Paul sustained me through this degree program as well. Reed, Amelia, and Isaac all helped put a smile on my face walking into the office every day. Discussions with Max fishing at the beach helped to polish off my knowledge on topological material science, which alas did not contribute to the completion of this thesis. Without Shay and Julian spotting me in the gym, I'd surely have been crushed under the bar and not been able to finish this thesis. Lastly, the MIT Brazilian Jiu Jitsu club under Alex, Remy, and Andrea's instruction gave a place for me to fight people in a socially acceptable setting—something every PhD student

must need at times.

This work was partially supported by the U.S. Department of Energy through the Los Alamos National Laboratory. Los Alamos National Laboratory is operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA000001). This material is also based upon work partially supported under an Integrated University Program Graduate Fellowship. This research was also partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Department of Energy Office of Nuclear Energy.

Biographical Sketch

Gavin Keith Ridley was raised in East Tennessee, where he earned his Bachelor of Science in Nuclear Engineering from the University of Tennessee, Knoxville in 2018. He was mentored by Dr. Ondrej Chvala, who played a significant role in shaping his academic journey. Currently, he is pursuing his PhD at MIT under the guidance of Professor Benoit Forget, concentrating on GPU-based Monte Carlo neutron transport.

In 2019, Gavin worked at the nuclear reactor startup Yellowstone Energy, where he conducted neutronics calculations. This experience laid a solid foundation for his future endeavors. Presently, he is involved with Aalo Atomics, contributing to the design of nuclear microreactors through a combination of innovative reactor design and advanced computational techniques.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reactor physics is an art of approximations, with one exception to this rule: the Monte Carlo (MC) method. Although some minor approximations exist therein, continuous energy MC simulations stand as the gold standard for reactor physics computations. These methods are, by construction, the closest a computer gets to mirroring the physical reality of neutron transport. In the most basic form of MC dating to the Manhattan project [1], the direct physical process of particle transport is followed from particle birth to death. Such a realistic method comes with a burden of computational expense compared to legacy deterministic methods for neutron transport, which are well-tuned approximate models used extensively within the nuclear power industry.

In an overarching way, the goal of deterministic neutronics models is to calculate power distributions in reactors in both steady state and perturbed conditions, along with the inventories of various nuclides which accumulate in nuclear fuel as it fissions. These codes are typically used in a cascading fashion, starting from simple or small geometric representations of a problem with fine energy discretization to capture neutron slowing down physics. The next step traditionally simulates a reactor fuel assembly to calculate parameters such that a diffusion equation can be used to approximate the gross movement of neutrons across an assembly. Full core programs then solve these diffusion equations in order to determine pin powers, assembly-averaged nuclide inventories, and behavior of the reactor in off-normal conditions. The books [2] and [3] provide further detailed information on the menagerie of deterministic neutron transport tools.

From the bottom of this toolchain to the top, various approximations are applied to the neutron transport problem. Historically, the applicability of the integrated stack of assumptions has been verified by experiment, particularly in the case of pressurized water reactors. Many conventional assumptions, however, remain untested for the myriad proposed advanced reactor concepts. As such, Monte Carlo models which are exact, in a sense, serve

as a reference point for proving veracity of a deterministic model of an untested reactor concept, reducing the need for neutronic experimentation involving critical experiments, hot cells, and so much more. In transients though, Monte Carlo calculations are not as frequently used.

With the publication of Sjenitzer's thesis in 2013 [4], the reactor physics community saw for the first time practical transient Monte Carlo on reactor-relevant timescales without the application of primitive approximations like the quasistatic method [5] which had been used for fifty years prior. Sjenitzer's methods have been implemented in a few Monte Carlo codes today, and have been tested against transient benchmarks and experiments such as C5G7-TD [6] and the SPERT tests [7].

In 2018, Shaner's thesis [8] presented a few novel improvements for transient MC, but, most relevant to this discussion, parts with a few estimates regarding the future of transient MC. Shaner predicted, extrapolating from his results, that a 0.2 exaflop computer could perform a full core transient analysis of a PWR overnight, using around fifty million core hours. A 0.2 exaflop computer was built at the Oak Ridge National Laboratory that same year in 2018, Summit [9].

Yet, with near-exascale compute resources now available, a dearth of literature on execution of Monte Carlo calculations on pre-exascale machines has been published. Forrest Brown acutely predicted the burden of the exascale, post-Moore's-law world in his 2010 paper [10]. By observing an exascale computer would use nearly a half gigawatt of electricity using high performance computing technology of the day, Brown predicted the dominance of graphics processing units (GPUs) and other specialized architectures in the future. While Moore's law has not strictly ended yet in terms of available arithmetic performance on modern CPUs, it has ended with respect to the effective memory bandwidth, which is a potentially limiting factor for performance in Monte Carlo neutronics calculations. As we can see in Figure 1.1,the pace of CPU memory bandwidth growth rate has slowed, whereas server GPUs have continued to increase bandwidth at nearly the same exponential pace as CPUs in the 1980s and 1990s.

Harnessing the full memory bandwidth of GPUs for Monte Carlo particle transport requires carefully designed programming. As we will explore in the following sections, sometimes initially counter-intuitive programming changes that add more work to the calculation, such as sorting an array of particles, may greatly enhance code performance on GPUs. Tapping into that massive memory bandwidth requires a program designed with care. The overarching trend toward GPUs in scientific computing was predicted in [11] and [12], and today, of the five supercomputers from the United States making the top ten of the Top500 [13] list, all are based on GPUs. To keep pace with other high performance scientific soft-

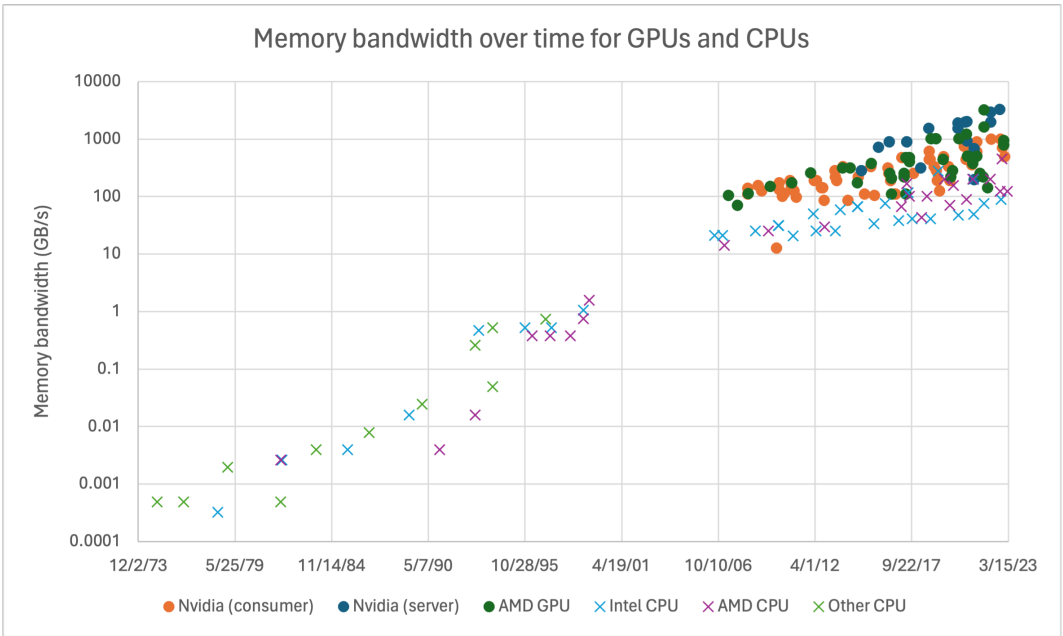


Figure 1.1: The memory bandwidth over time for CPUs (x marker) and GPUs (o marker) over time.

ware applications, the algorithms comprising the continuous energy MC neutron transport method must be redesigned.

1.0.1 Monte Carlo for neutral particle transport

How a sampling method can be used to solve partial differential equations such as the linear Boltzmann equation may not be immediately obvious. This approach relies on converting the Boltzmann equation's solution to an integral, rather than differential form. If the problem can be cast as a sampling process, we call it a Monte Carlo solution.

We now present a simple derivation of the Monte Carlo particle transport algorithm from the neutral particle Boltzmann equation, setting the stage for the later developments in this thesis. This derivation makes use of the fact that if an integrand is replaced by a random field with an expectation field equal to the original integrand, the expectation value of the random integral matches the integral of interest. A related but less intuitive derivation can be found in [14] as well.

We start with a simple case to illustrate, impressionistically, Monte Carlo. Suppose one wishes to evaluate the definite integral via Monte Carlo:

$$I = \int_0^1 x^2 \, \mathrm{d}x \tag{1.1}$$

A conventional Monte Carlo integration as presented in [14] would identify this integral as a product of a probability distribution $\pi(x)$ on $[0,1]$ with some arbitrary function of x, specifically:

$$I = \int_0^1 x^2 \frac{\pi(x)}{\pi(x)} = \mathbb{E}[\hat{x}^2/\pi(\hat{x})] \tag{1.2}$$

Where $\hat{x}$ is a random variable with distribution $\pi(x)$, and $\mathbb{E}$ is the expectation value operator. One then carries out this integration by sampling from $\pi(x)$ some number of times and approximating the expectation value above by a simple arithmetic mean.

To move towards how this method can be used to solve partial differential equations, an alternative but equivalent procedure instead replaces the above integrand by a random field consisting of a stochastically placed Dirac delta function:

$$x^2 = \mathbb{E}[\delta(x - \hat{x})\hat{x}^2/\pi(\hat{x})] \tag{1.3}$$

Where $\hat{x}$ is a scalar random variable distributed according to $\pi(\hat{x})$. That the equality holds can be easily shown by the linearity of integration.

So, we can approximate the integrand by an arbitrary random field with an expectation field matching the integrand:

$$\hat{I} = \int_0^1 \delta(x - \hat{x})x^2/\pi(\hat{x})\,\mathrm{d}x = \hat{x}^2/\pi(\hat{x}) \tag{1.4}$$

where it is again easily verified that $\mathbb{E}[\hat{I}] = I$, and we again approximate the expectation value by a finite arithmetic mean.

Now we turn to the stationary, fixed-source neutron transport equation, the keystone of reactor physics:

$$\hat{\Omega} \cdot \nabla\varphi + \Sigma\varphi = q + \int_0^\infty \mathrm{d}E' \int_{4\pi} \mathrm{d}\hat{\Omega}'\Sigma_s(\hat{\Omega}' \to \hat{\Omega}, E' \to E)\varphi|_{\hat{\Omega}',E'} \tag{1.5}$$

With $\hat{\Omega} \in \mathcal{S}_2$, the unit sphere $\mathcal{S}_2 \subset \mathbb{R}^3$, $E \in \mathbb{R}^+$, $r \in U \subset \mathbb{R}^3$, U being the spatial domain of interest. The function $\varphi : U \times \mathbb{R}^+ \times \mathcal{S}_2 \to \mathbb{R}^+$ is the angular flux. The function $\Sigma : U \times \mathbb{R}^+$ is called the total cross section. q is a function mapping between the same spaces as φ, and is called the angular source. We also impose vacuum boundary conditions to simplify coming arguments:

$$\varphi(r, \hat{\Omega}, E) = 0 \quad \forall r \in \partial U, \hat{\Omega} \cdot \hat{n} \leq 0 \tag{1.6}$$

with $\hat{n}$ being the unit normal at $r \in \partial U$. ∂U is the boundary of U as defined in [15]. For simplicity, we can simply lump the fission term, if present, as an isotropic contribution to

scattering above if that is present in a system. The results are identical. For more, the works [3], [16] cover the physical situations covered by this equation in greater detail. We also define the scattering operator $\mathbb{S} : U \times \mathbb{R}^+ \times \mathcal{S}_2 \to U \times \mathbb{R}^+ \times \mathcal{S}_2$

$$\mathbb{S}(\varphi) = \int_0^\infty \mathrm{d}E' \int_{4\pi} \mathrm{d}\hat{\Omega}' \Sigma_s(\hat{\Omega}' \to \hat{\Omega}, E' \to E)\varphi|_{\hat{\Omega}', E'} \tag{1.7}$$

Note that $\mathbb{S}$ is a linear operator as per the linearity of integration.

Next, defining:

$$\begin{aligned} \hat{\Omega} \cdot \nabla\varphi_0 + \Sigma\varphi_0 &= q \\ \hat{\Omega} \cdot \nabla\varphi_1 + \Sigma\varphi_1 &= \mathbb{S}(\varphi_0) \\ \hat{\Omega} \cdot \nabla\varphi_2 + \Sigma\varphi_2 &= \mathbb{S}(\varphi_1) \\ \hat{\Omega} \cdot \nabla\varphi_3 + \Sigma\varphi_3 &= \mathbb{S}(\varphi_2) \\ &\dots \end{aligned} \tag{1.8}$$

We can see that the function sum $\sum_{i=0}^\infty \varphi_i$, if it converges uniformly, solves the transport equation. This can be verified by summing all of the above equations, and using linearity of integration and differentiation to identify the summed quantity as solving the equation we wish to solve. Each term φ_0, φ_1, etc. has the physical interpretation of being uncollided neutrons, singly collided neutrons, doubly and so on. Of course some rigor could be included in this argument, but the conclusions remain unchanged. The goal here is to explain as intuitively as possible.

To obtain the Monte Carlo algorithm, we note that the above sequence of equations is simply a sequence of fixed-source, zero scattering transport problems. This problem with vacuum boundary conditions has a closed-form solution for each φ_0, φ_1, etc. The first is:

$$\varphi_0(r, \hat{\Omega}, E) = \int_0^s e^{-\tau(r, r-\hat{\Omega}s')} q(r - \hat{\Omega}s', \hat{\Omega}, E)\, \mathrm{d}s' \tag{1.9}$$

Where s is the smallest possible non-negative value such that:

$$r - \hat{\Omega}s \in \partial U \tag{1.10}$$

and τ is the *optical thickness*:

$$\tau(r, r - \hat{\Omega}s') = \int_0^{s'} \Sigma(r - \hat{\Omega}s'')\, \mathrm{d}s'' \tag{1.11}$$

In words, to find the angular flux moving in the direction $\hat{\Omega}$ at any point, we integrate all of the attenuated source behind that point moving along the line $r - \hat{\Omega}s'$ until the boundary is reached, which has zero incident flux as previously stipulated. This is a simple application of the method of characteristics as described in any textbook on partial differential equations [15], [17].

In order to approximate the problem via Monte Carlo, we first suppose that the source q satisfies the properties of a probability distribution, namely being non-negative and having an integral over all phase space $U \times \mathbb{R}^+ \times \mathcal{S}_2$ equal to unity. If the source is not normalized, one can simply divide the whole of the transport equation by the total source, and later multiply the solution by that "source strength". We can then replace q with this random field, using the aforementioned fact about random fields:

$$\tilde{q} = \delta(\tilde{\hat{\Omega}}, \hat{\Omega})\delta(\tilde{r} - r)\delta(\tilde{E} - E) \tag{1.12}$$

With the δ function on the unit sphere $\delta(\tilde{\hat{\Omega}}, \hat{\Omega})$ being defined as in [16], and the random variables $\tilde{\hat{\Omega}}, \tilde{r}, \tilde{E}$ being distributed according to the multivariate distribution $q(\tilde{\hat{\Omega}}, \tilde{r}, \tilde{E})$. We can then verify this stochastic approximation preserves the correct expectation field:

$$\mathbb{E}[\tilde{q}] = \int_0^\infty \mathrm{d}\tilde{E} \int_{4\pi} \mathrm{d}\tilde{\hat{\Omega}} \int_U \mathrm{d}\tilde{r}\delta(\tilde{\hat{\Omega}}, \hat{\Omega})\delta(\tilde{r} - r)\delta(\tilde{E} - E)q(\tilde{\hat{\Omega}}, \tilde{r}, \tilde{E}) = q(\hat{\Omega}, r, E) \tag{1.13}$$

So we then have the following stochastic estimate of φ_0:

$$\tilde{\varphi_0}(r, \hat{\Omega}, E) = \int_0^s e^{-\tau(r, r-\hat{\Omega}s')}\delta(\tilde{\hat{\Omega}}, \hat{\Omega})\delta(\tilde{r} - r)\delta(\tilde{E} - E)\,\mathrm{d}s' \tag{1.14}$$

This is simply an exponentially attenuating beam originating at the location $\tilde{r}$ moving in the direction $\tilde{\hat{\Omega}}$. This beam-like function similarly is now approximated by a stochastic estimate. It nominally appears that we can select a value $\tilde{\tau}$ from an exponential distribution, and simply approximate the above by:

$$\tilde{\tilde{\varphi}}_0(r, \hat{\Omega}, E) = \delta(\tilde{\hat{\Omega}}, \hat{\Omega})\delta(\tilde{r} + \tilde{\hat{\Omega}}x - r)\delta(\tilde{E} - E) \tag{1.15}$$

Where x is calculated by inverting the optical pathlength function:

$$\tilde{\tau} = \tau(\tilde{r}, \tilde{r} + x\tilde{\hat{\Omega}}) \tag{1.16}$$

However, we in fact may *not* naively sample from the exponential distribution above. Because the problem boundaries are finite, the exponential appearing in Eq. 1.14 is in fact not a

normalized probability distribution. As such, we should instead compute:

$$p_{\mathrm{NL}} = \int_0^s e^{-\tau(r, r-\hat{\Omega}s')}\,\mathrm{d}s' \tag{1.17}$$

which we can identify as the probability the source particle did not leak from the geometry. Thus, if we sample $\tilde{\tau}$ from the distribution:

$$\frac{1}{p_{\mathrm{NL}}} e^{-\tau} \tag{1.18}$$

with probability p_{NL}, and put (x calculated as before):

$$\tilde{\tilde{\varphi}}_0(r, \hat{\Omega}, E) = \delta(\tilde{\hat{\Omega}}, \hat{\Omega})\delta(\tilde{r} + \tilde{\hat{\Omega}}x - r)\delta(\tilde{E} - E) \tag{1.19}$$

else with probability $1 - p_{\mathrm{NL}}$ put:

$$\tilde{\tilde{\varphi}}_0(r, \hat{\Omega}, E) = 0 \quad . \tag{1.20}$$

It is at this point that we calculate the scattering source for the equation governing φ_1. We calculate this as:

$$\mathbb{S}(\tilde{\tilde{\varphi}}_0) = \int_0^\infty \mathrm{d}E' \int_{4\pi} \mathrm{d}\hat{\Omega}' \Sigma_s(\hat{\Omega}' \to \hat{\Omega}, E' \to E)\delta(\tilde{\hat{\Omega}}, \hat{\Omega}')\delta(\tilde{r} + \tilde{\hat{\Omega}}x - r)\delta(E' - \tilde{E}) \tag{1.21}$$

Which then simplifies to:

$$\mathbb{S}(\tilde{\tilde{\varphi}}_0) = \Sigma_s(\tilde{\hat{\Omega}} \to \hat{\Omega}, \tilde{E} \to E)\delta(\tilde{r} + \tilde{\hat{\Omega}}x - r) \tag{1.22}$$

This is a spatial dirac delta source with a scattering source distributed over the unit sphere, and is also distributed in energy. We sample from this a point estimate of the flux at a given energy and angle, and repeat the entirety of the previous process iteratively. Figure 1.2 shows how this procedure looks like with a shaded source region on a finite domain, making two point approximations of the flux, and culminating in a particle leak. The prior discussion does not allow particles to be absorbed, having their history terminated permanently. This is simply a selection of the zero function with a probability equal to the absorption probability.

In addition, the flux estimates are only collision estimates, although track-length flux estimates can be derived as a limit of a delta tracking process [14]. To derive track-length flux estimates Eq. 1.19 can be modified. Rather than sampling a Dirac delta with exponential distribution in the optical path length, the same can be one if $\tau < \hat{\tau}$ else zero. In other words,

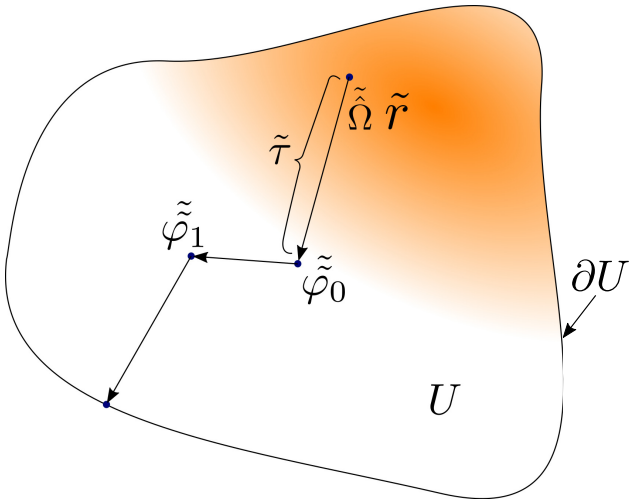


Figure 1.2: A neutron path through the domain U with shaded source q, using variables presented here. Collision flux estimates $\tilde{\tilde{\varphi}}_0$ and $\tilde{\tilde{\varphi}}_1$ are made for this track.

the exponential distribution can be approximated as a superposition of constant functions (assuming $x \geq 0$):

$$e^{-x} = \int_0^\infty H(\hat{x} - x) e^{-\hat{x}} \, \mathrm{d}\hat{x} \quad . \tag{1.23}$$

This "track length" estimator has the advantage that contributions to samples will be drawn in regions of low cross section more frequently than in the collision estimator. Figure 1.3 shows how these constant estimates average to match the exponential distribution required by the linear Boltzman distribution. For the collision flux estimator, the samples are instead Dirac deltas which are situated at the end of each of the line samples shown in Fig. 1.3.

Just as sampling from an arbitrary distribution could be carried out in Eq. 1.2, similar approaches can be applied to sampling the source in space, energy, or angle, so long as weight corrections are applied. Biased sampling can be used to reduce the variance of the samples of the simulation if the biasing distributions are chosen judiciously, and other works for example in [14] or by searching for the term "importance sampling" in any nuclear engineering journal. When biased sampling is not applied, the algorithm is called *analog*.

A convenient property of the exponential distribution is memorylessness, which states that for an exponentially distributed random variable X with rate parameter λ:

$$P[X > s + t \mid X > s] = P[X > t] \quad . \tag{1.24}$$

In other words, if a particle is stopped before reaching the end of a sampled track length as exhibited in the top of Fig. 1.3, the exponential distribution can simply be sampled again and the particle be allowed to follow that track length without regard to prior track

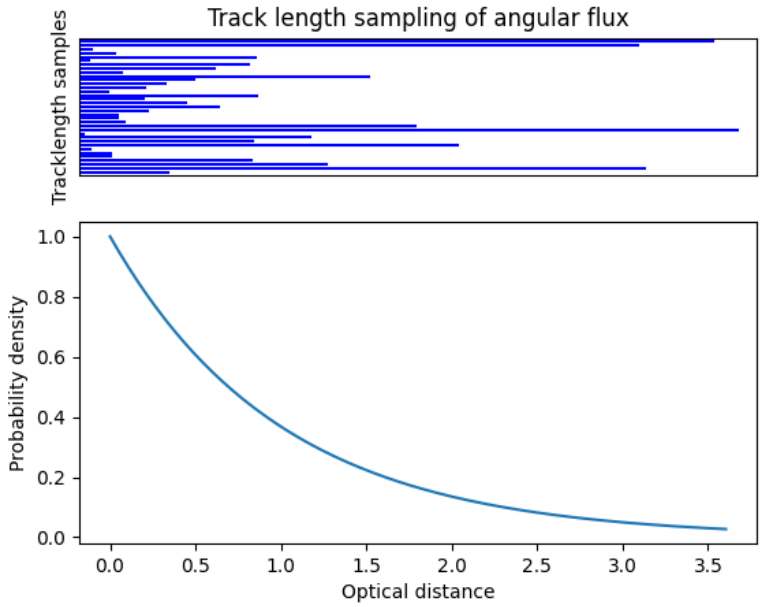


Figure 1.3: The tracklength approach to sampling flux averages piecewise constant samples (above) to produce exponentially distributed behavior on average.

length sample. Because in most practical nuclear engineering problems, the cross sections are spatially piecewise constant. Therefore, if a track crosses a material boundary, the total traversed track length need not be considered: a new track length may be sampled with the rate parameter Σ_t without explicitly integrating the track length using Eq. 1.11.

All of the aforementioned problems can be stitched together to define the *history-based* Monte Carlo algorithm for simulation of neutral particle transport, shown in Algorithm 1. As the track is followed through the problem, the aforementioned samples of the flux are accumulated on a mesh, dividing the phase space arbitrarily. The data structures used to accumulate the fluxes or functionals of the flux are called *tallies*, and the partitioning of phase space is typically done using a tensor product of a spatial mesh with an energy mesh. These individual meshes are typically called *filters*. Figure 1.4 shows how samples of the angular fluxes can be averaged together to estimate fluxes on a mesh on a nuclear fuel assembly problem.

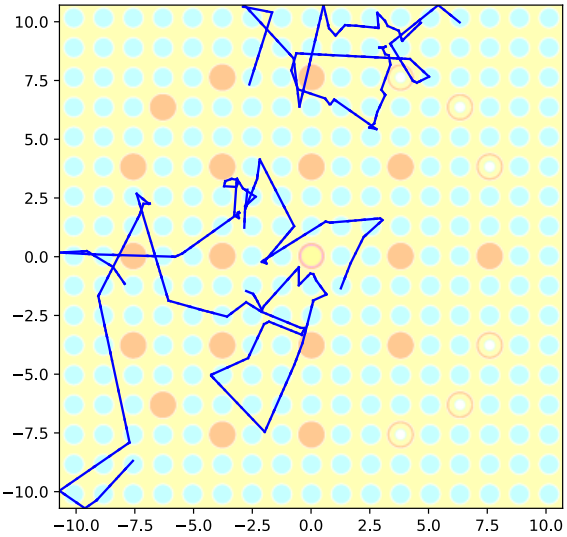
The fixed source, linear Boltzmann equation does not describe chain-reacting systems like a nuclear reactor. The problem is instead framed as a generalized eigenproblem:

$$\hat{\Omega}\cdot\nabla\varphi+\Sigma\varphi=\int_0^\infty \mathrm{d}E'\int_{4\pi}\mathrm{d}\hat{\Omega}'\Sigma_s(\hat{\Omega}'\to\hat{\Omega},E'\to E)\varphi|_{\hat{\Omega}',E'}+\frac{1}{4\pi k}\int_0^\infty \mathrm{d}E'\int_{4\pi}\mathrm{d}\hat{\Omega}'\nu(E')\chi(E'\to E)\Sigma_f(E')\varphi|_{\hat{\Omega}',E'} \quad (1.25)$$

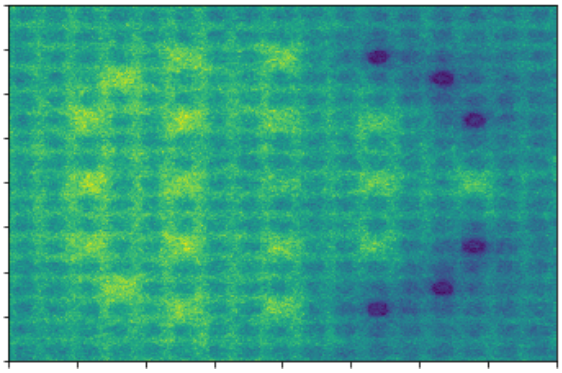
```
for each particle history do
    generate particle from source q;
    while particle not escaped or absorbed do
        sample distance to collision in material;
        sample distance to material interface;
        compute distance to cell boundary;
        advance minimum of collision and crossing distance;
        if Minimum distance was to collision then
            collide, sampling new energy E and angle Ω̂;
            compute new macroscopic cross section Σ_t at new energy;
        end
        if Minimum distance was to boundary crossing then
            compute new macroscopic cross section Σ_t in new material ;
        end
        if particle escaped spatial domain then
            end particle history;
        end
        if particle absorbed then
            end particle history;
        end
    end
end
```

Algorithm 1: Fixed source, history-based Monte Carlo algorithm

where ν is called the fission multiplicity, Σ_f is the fission cross section, and χ is the fission neutron spectrum. χ is often modeled as being independent of the incident energy, although continuous energy Monte Carlo codes typically model the dependence of this on incidence energy. The eigenvalue problem can be reduced to a series of fixed source problems in which each fixed source is defined as the set of fission sites created in the previous iteration.



(a) A pressurized water reactor fuel assembly with superimposed neutron tracks.



(b) Fluxes estimated by averaging many samples.

Figure 1.4: Flux samples generated with Monte Carlo sampling can be accumulated to approximate the Boltzmann equation's solution without bias.

Armed with an understanding of how Monte Carlo neutral particle simulation works, we can almost understand the shortcomings associated associated with running Algorithm 1 on GPUs. The peculiarities of GPU computing will provide the context for this.

1.0.2 GPUs and SIMD

GPUs bear a semblance to earlier vector computers, which were available in centralized setups up through the early 1990s. These were available in the early days of Monte Carlo neutron transport, and it has been well-known since the work of [18] in 1973 that the conventional Monte Carlo algorithm of following neutrons from birth to death sequentially performs dismally on vector computers. A spectrum of modified algorithms for *event-based* Monte Carlo (as opposed to history-based) have been developed since then, e.g. the naval lab code RACER3, which achieved a tenfold speedup using event-based mode versus scalar instructions on a vector machine, as documented in [19]. To understand the performance limitations of an event-based Monte Carlo algorithm, some further background on these

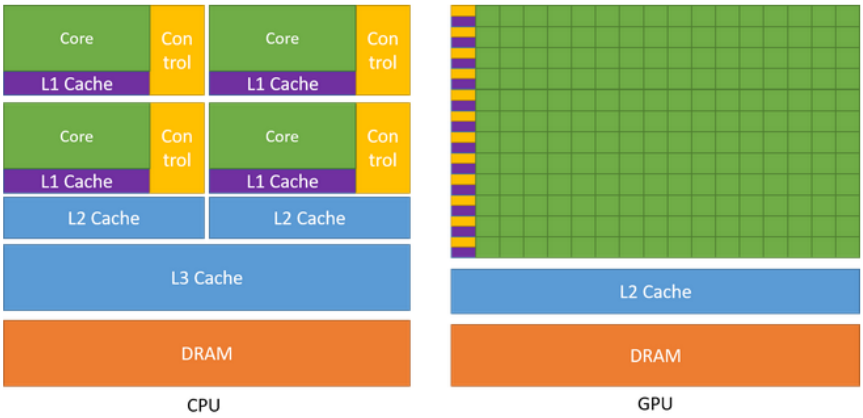


Figure 1.5: Rough sketch of typical CPU vs. GPU architecture, image from [23].

architectures is in order.

Flynn's taxonomy of computer architectures [20] broadly classifies the commonly used shared memory parallel computer architectures as either SIMD or multiple instruction, multiple data (MIMD). Both of these parallelism modes are encountered on the modern desktop. Multicore computer processors realize the MIMD approach in that each core can execute different instructions on multiple data simultaneously. Typically, each of these cores can realize parallelism individually by the use of special SIMD instructions which carry out the same operation on several adjacent pieces of data. These operations are typically restricted to addition, multiplication, or rudimentary logic operations. By using these special instructions on modern processors, throughput of numerical programs can be increased by a factor roughly approaching the SIMD data length. For more information on parallel computing in the context of nuclear engineering, the chapter [21] expounds the aforementioned concepts further.

GPUs are often considered as operating using a SIMT, or single instruction multiple thread, model [22] which means that many threads can carry out a single instruction at a time on arbitrary data. This is distinctly more flexible than SIMD. In SIMT, a full-featured instruction set may be executed on data in parallel; this contrasts from CPUs where specific instructions must be used to leverage SIMD. Because of this, branching statements can be used in parallel with ease, and although each branch must execute in lockstep, thus degrading performance, some degree of parallelism is achieved. Fig. 1.5 contrasts the general layout of a typical CPU and a typical GPU; the biggest takeaway being that many GPU compute cores (green) share a single instruction unit (yellow).

The group of threads each executing an instruction is called a warp, and there are thirty-two threads per warp on an Nvidia GPU [23]. As previously mentioned, it is commonplace for a warp of threads to encounter a point in the program where some threads enter an "if" block,

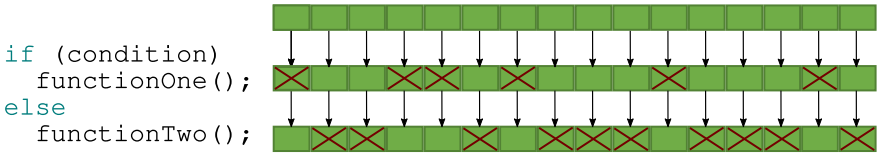


Figure 1.6: In this example, some threads have `condition=true`, and others not. The branching statement must be executed in lockstep, with x's indicating idle threads.

and others do not. This situation is called *warp divergence*, and should be avoided if possible since it degrades the computational throughput of the GPU. Figure 1.6 illustrates warp divergence on some branching code. This thesis work identifies Monte Carlo algorithms for neutron transport that can be tuned to reduce warp divergence, thus enabling more efficient utilization of the hardware of modern exascale computers, GPU-equipped machines, and SIMD units on modern CPUs.

The previous paragraph is an oversimplification but serves as a reasonable model of how GPU programs should be written. In reality, modern GPUs operate in a MIMD manner, heavily flavored by SIMT. Since the Volta architecture GPUs released in 2017, Nvidia GPUs are capable of independent thread execution [24]. As a result, Fig. 1.6 does not adequately represent the situation. The threads may in fact execute the different code paths simultaneously as long as all the code fits into the thread block's instruction cache. Therefore, without considering the complicating factor of memory access, sufficiently "shallow" conditional branches incur negligible performance penalty. A `syncwarp` function is provided in GPU architectures with independent thread execution that behaves similarly to a threading barrier used in CPU code.

Despite the capability of independent thread execution, diverged thread behavior can still seriously impact GPU program performance. This could come as a result of instruction cache misses, which are hard to quantify the behavior of due to the proprietary nature of GPU hardware implementation. Similar to execution of code, memory accesses can also be *diverged*. Fig. 1.7 shows the desirable pattern to access memory in on GPUs, and Fig. 1.8 shows a memory access pattern which may yield poor performance. The randomized nature of Monte Carlo simulations can make it very difficult to avoid uncoalesced memory accesses. Threads usually operate in groups of 32 called *warps*, and in the case of coalesced memory accesses, memory accesses happen at the maximum bandwidth. For a purely uncoalesced memory access, the memory accesses can become completely serialized, completing in an order of magnitude longer time. Intermediate situations provide a blend between the uncoalesced and coalesced speed.

Again, the previous paragraph is an oversimplification of reality. Modern GPUs have less trouble with the order that threads access memory in, and more trouble with memory

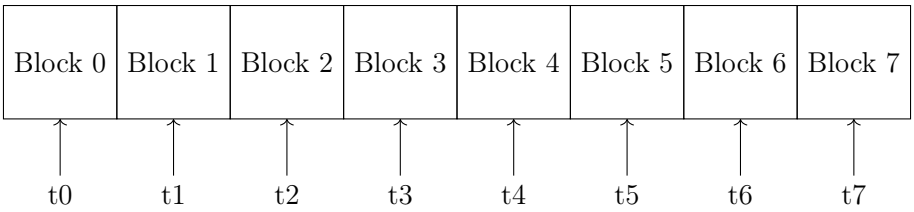


Figure 1.7: A quintessential coalesced memory access. t0, t1, etc. indicate threads 0, 1, ...

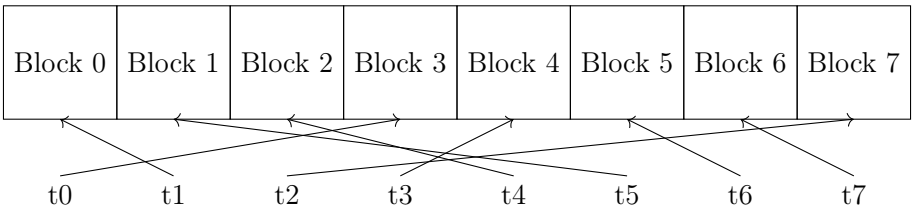


Figure 1.8: A quintessential uncoalesced memory access.

accesses from threads to locations in memory scattered far apart. This explains why sorting the particle cross-section look-up queue can greatly enhance continuous energy MC code performance, and also explains why caching microscopic cross sections counter-intuitively hurts performance. Both of these situations will be explained in greater detail in later text.

A more nuanced understanding of memory accesses on GPUs, which are currently the limiting aspects of continuous energy MC codes, requires understanding the types of memory on a GPU. *Global* memory is the number reported that represents GPU RAM, and can be accessed from any thread albeit with a considerable amount of latency. *Shared* memory can be accessed an order of magnitude faster, but may only be shared amongst *blocks* of threads and is of more limited size, tens of kilobytes. Thread blocks are groups of thread warps that range between the warp size and ten or more warps, around 320 threads. *Constant* memory is optimized for broadcasting to many different threads across many blocks; this is ideal for storing run-time settings such as whether to toggle different sampling approaches like weighting in lieu of absorption in an MC code. *Texture* memory is optimized for 2D memory accesses like used in sampling textures on 3D models. Lastly, *registers* are the storage immediately local to each thread, and cannot be shared across threads with the exception of *warp primitives*, but explaining warp primitives is beyond the scope of this introduction.

Thread block size was introduced in the context of shared memory. The optimal block size for fastest execution of an application depends on the application in question. While GPU memory access latency (a different concept from bandwidth) may be relatively slow, GPUs have the ability to over-saturate the system with more thread blocks than can simultaneously execute. *Latency hiding* is the strategy GPUs automatically employ of switching

to other thread block contexts when waiting through the latency period of memory access. By switching to other thread blocks while waiting on latency, the delays induced by memory latency can be effectively hidden. The amount of blocks which can simultaneously run are governed by limitations on the total number of registers and the total amount of shared memory. Choosing how these resources are divided up to thread blocks can imply tremendous performance differences.

Occupancy is related to the number of threads able to run on a GPU for a given piece of code relative to the theoretical maximum number of threads that could be executing on a GPU. For continuous energy MC codes, our experience has suggested occupancy fractions are typically around 40%, far from ideal. This originates as a result of the complexity of the code, which requires a large number of registers. An important technique in programs bound by memory latency is intentionally using more registers and decreasing occupancy in an effort to reduce the number of global memory transactions [25].

The requirement for threads to execute similar code paths and access nearby memory nearby implies that the conventional history-based algorithm tends to underperform. As particles randomly cross surfaces, enter materials with varying numbers of nuclides, and scatter to different energies, the data and code needs of each particle becomes more and more different. Eventually, all the particles executing in parallel on each GPU thread would be executing entirely different instructions, leading to serialized memory accesses and constant instruction cache misses. A new paradigm outside following each particle from birth to death on each thread is required.

1.0.3 Event Monte Carlo

In event Monte Carlo, a few approaches have been proposed, the most promising being stack-based event Monte Carlo as proposed by Brown [19]. Generally, rather than dealing with a single particle at a time, event Monte Carlo deals with an array of particles. First, source locations and directions of many particles in a particle array are determined. After sampling the distance each neutron should travel, which has a known distribution, particles in the array advance until they collide or cross a boundary. The computer then treats boundary crossings of each neutron for those entering new regions, and similarly processes collision physics of colliding neutrons. After that, the algorithm repeats, and particles are advanced again, until all neutrons either leak from the geometry or are absorbed. Over the course of the algorithm, the various events such as collisions, boundary crossing, and passage through various parts of the geometry may be tallied according to the neutron properties, e.g. energies, directions, or positions, for output to the analyst. Algorithm 2 presents the

basic functioning of the event-based tracking algorithm.

In practice, a subtlety arises: how should one mark which particles in the particle array are yet to collide, cross a surface, or do something else? Early work shuffled particle arrays such that all colliding particles are contiguous, all crossing surfaces are contiguous in the array, etc. This shuffle requires a burdensome amount of computation, however, which motivated the stack-based approach, where one simply records indices of particles about to undergo a given event.

The stack-based event algorithm can take on a whole spectrum of possible implementations. [26] describes a few of the possibilities. For instance, one code maintained a separate event queue for each of the geometric cells in a problem. Another code kept a separate event queue for each of the possible reactions a given neutron could undergo. Considering this, the vast range of possible stack-based event algorithms can be clearly seen. One could, in principle, construct a stack-based event code that handles each surface type crossing, each reaction, perhaps even each range of energies of various particles separately. But the management of all these queues comes with its own computational cost, so a balance must be struck. The optimal balance will depend on the hardware characteristics of the computer. Writing each permutation of event queue layouts, however, is inefficient.

The creation of a production level event-based Monte Carlo codes can require thousands of person-hours, and even more for subsequent verification and validation [26]. All loops in the program, and similarly data structures, must be adjusted to match the new layout of the event-based code [27]. Clearly then, if the optimal granularity of a given event depends on hardware characteristics of a system and an immense effort is required to create a realization of a given event-based algorithm, a given code will perform optimally on the platform it was designed for, but underperform on many others.

The work [27] sought to estimate the efficiency of an event-based MC code relative to history-based tracking. This study used empirical estimates of the time to complete various particle events from OpenMC, and predicted the performance of an event-based MC code based on that. The conclusion was that event MC codes can only degrade performance, under the assumptions in the study. However, the study did not account for the intricate sensitivity of GPU memory accesses to the access pattern. In fact, if the event timings were adjusted to represent the upper bound memory bandwidth and a model of cache hits implemented, a model such as this should predict event mode to outperform history mode as has been observed empirically in codes like [28], [29].

Lastly, every time a code adds a new feature, this potentially increases the divergence of each event. For instance, in a code using a monolithic collision processing kernel, adding new collision physics types adds more thread divergence to that kernel, like in Figure 1.6.

```
while particles remain do
    generate all particles in batch from source;
    put particle indices to cross section lookup queue;
    while particles remaining in batch do
        for Choose event E in (collision, flight, material interface crossing, cross
          section lookup) from longest queue do
            if E == collision then
                collide;
                handle collision tallies;
                add particle to cross-section lookup queue;
                if particle absorbed then
                    kill particle;
                end
            end
            if E == flight then
                advance particle, accumulate track length tallies;
                if particle collided then
                    add particle to cross-section lookup queue;
                end
                else if particle leaked then
                    kill particle;
                end
                else
                    add particle to material interface crossing queue;
                end
            end
            if E == material interface crossing then
                determine cell crossing to from neighbor list;
                lookup new material;
                add particle to cross-section lookup queue;
            end
            if E == cross section lookup then
                Loop over nuclides, get microscopic cross sections;
                Save particle macroscopic cross section;
            end
        end
    end
end
```

Algorithm 2: Event-based Monte Carlo algorithm

For production Monte Carlo codes like MCNP6 [30] seeking to support numerous particle types and a plethora of physics, the book-keeping associated with many event queues could become burdensome. The former would become more divergent with the addition of each new collision type, and the latter would require large programming efforts to update the queue processing code whenever new physics is added. As a result, we foresee that most practically used GPU MC codes will be custom tailored for specific applications in the foreseeable future, for example the PRAGMA [31] code which will be reviewed in the next section.

1.1 Review of Previous GPU Monte Carlo Projects

1.1.1 Generic GPU Background Material

In addition to ubiquitous application in deep learning, GPUs have been employed in successfully accelerating a variety of other scientific computing applications. A common theme in distilling the literature on scientific computing by way of GPU is that algorithms, which may work well, fail to provide optimal or at times remotely acceptable performance on GPUs. This happens as a result of the fundamentally different threading model on GPUs, fundamentally different memory access mechanisms, and the corresponding optimal floating point to memory access ratios to obtain peak performance to solve a given problem. We'll first explore how other applications have been modified to obtain excellent GPU performance, then transition to how Monte Carlo neutron transport algorithms can be tailored for GPU.

Goddeke's 2011 dissertation [32] is one of the early works detailing the conceptual shift required to achieve optimal GPU performance in finite element PDE solvers. Goddeke suggests the use of finite elements of high order that admit a tensor product basis. This allows a sum factorization trick in evaluating the action of the system matrix on the unknowns vector. In general, matrix-free techniques using higher order elements prove to be the best choice on GPUs, whereas first or second order elements using a sparse matrix representation of an FEM problem are most commonly used in CPU-based FEM codes such as MOOSE [33].

[34] discusses the acceleration of Runge-Kutta methods on GPUs. Again, in contrast to converting scientific computing applications from for example single to for multithreaded CPUs, fundamental programmatic changes were required. In this case, discrepancies in the amount of work distributed to each thread have to be mitigated, as on the GPU, significant discrepancies in the amount of work assigned to each thread cause all threads to wait on the longest executing task. This work explores some task distribution strategies in the context of Runge-Kutta methods that mitigate this effect. In particular, by assigning each thread

multiple tasks, the distribution of task length can be made more narrow. This was shown to improve performance by an order of magnitude.

[35] explores ray tracing to simulate photon passage through vegetation canopies. Again, in this application, the best parallelization technique avoided a naive mapping of one sampled photon to each thread. Instead, it was shown that using one photon per thread block, and using threads within the block to compute ray-surface intersections over a large number of distance-to-boundary checks, provided the optimal task layout.

[36] details experience in optimizing Monte Carlo computations of atmospheric balloon landing locations. A key takeaway from this work is that "a good understanding of the physical problem permits one to optimize the data locality and hence to improve the performance of the parallel application." Indeed, this same theme echoes throughout the latter chapters of this thesis in which we design modifications to typical continuous energy Monte Carlo neutron transport operations that aim to enhance data locality.

In the world of nuclear engineering, plenty of work has been done on GPU acceleration of characteristics-based deterministic codes, starting from the work on OpenMOC [37]. The later work on nTRACER [38], [39] highlights the importance of reorganizing nested data structures to fit contiguously in memory, this case pertaining to resonance self-shielding models and the hierarchical ray data. In addition, a specialized linear solver was developed again to optimize the memory access patterns for GPU-friendly operation.

Similarly, advancements have been made in making pin-wise full-core nodal calculations more practical via GPU acceleration [40]. In this case, the code was designed from the bottom up with the niceties of GPU programming in mind. A specialized cross section compression algorithm was devised to handle the memory limitations. The code serves as an advancement towards practical reactor design with pin-by-pin diffusion calculations because hundreds of statepoint evaluations are required for core design, so accelerating the calculation to make this feasible on a single workstation serves as a valuable contribution.

More work has been carried out building an MOC code designed specifically for GPU execution [41]. From our read of this work, no unique algorithms designed for GPU have been developed as part of this project so far. However, if the focus on unstructured mesh MOC continues, we expect some unique developments will be discovered in the coming years.

Very recently, [42] drives home again the importance of GPU-specific modifications to obtain optimal performance in a finite element framework, MFEM [43]. Again, this work introduces novel algorithmic changes that enhance performance on GPUs.

Related to the developments in the MOC field, the "immortal ray" technique was recently proposed for improved utilization of GPU resources [44]. The new method was shown to perform equivalently to the conventional random ray algorithm on CPU architecture, but

demonstrated a clear advantage on GPU. Again, we see the recurring theme that new ways of looking at a problem are required to obtain optimal or even acceptable performance on GPUs.

Another work presented a unique means of solving the transport equation that works well with the transport equation. In this case, the particle scattering source was discretized on a mesh, the GPU was used to source individual straight lines of flight from the scattering source in a randomized fashion [45]. GPUs were also used to apply nonlinear diffusion acceleration in this work. The method in this case was applied to a simplified one-dimensional problem, and we see no barriers preventing generalization of this method to more complex multidimensional problems. While the method is indeed GPU-friendly, it presents no clear advantage over random ray or the immortal ray method.

Digesting all of the above together, we can observe that bringing scientific computing to GPU architecture is a heavier task than a simple "port". Fundamental algorithmic changes are required in many cases to achieve optimal performance.

1.1.2 Performance portability frameworks

It has been said that performance is not portable [46]. In parallel with the rise of GPU computing, a variety of performance portability frameworks have arisen. Applications developed for platform-specific GPU acceleration often cannot be converted to use other hardware without some potentially high effort. Some tools have been developed to automatically convert CUDA code for other platforms [47]. Other approaches to performance portability abstract away parallelizable operations, and toggle between various methods of execution in the backend. Kokkos [48], RAJA [49], and OCCA [50], [51] are just three examples among many.

With tools like this existing, why write platform-specific code like CUDA in the first place? In this project, we opted to use CUDA for a few reasons. All of these reasons, however, pertain to the technological maturity of the tool-chain. When this research project started, no full-featured, continuous energy Monte Carlo neutron transport code had yet been run on GPUs in a fashion that yields impressive speedups on par with the advancements in other fields. WARP [52] had achieved modest performance, but not a speedup substantial enough to justify the performance per dollar or watt that motivates GPU use in other fields like machine learning.

In the author's opinion, attempts at bringing new scientific computing applications to GPU should use the most mature tool-chain possible initially with a focus on performance portability coming as a step beyond performance alone. Using the most mature possi-

ble framework is in line with the guidance presented in [53]. Of course, the right tool for this job is CUDA as a result of its deeply detailed documentation, mature debugging and profiling utilities, and dedicated compiler. To our knowledge, no profiling tool tapping into GPU performance counters is available for GPU performance portability frameworks. Even using CUDA, our early attempts at GPU ports of OpenMC ran into errors fundamentally related to the compiler, for example the workaround introduced by the author in `https://github.com/mpark/variant/pull/73`, which had knock-on impacts to PyTORCH `https://github.com/pytorch/pytorch/pull/33230`. If even the CUDA compiler still failed to parse complex C++ in 2020, using less mature tool-chains seemed like a recipe for disaster.

Eventually though, remarkable performance-portable results were presented in an OpenMP-based version of OpenMC [54]. These results, though, were only made possible by first converting the OpenMC codebase to remove many modern programming conveniences and flattening the nuclear data structures entirely. Even after this work, specific compiler optimizations in Clang for OpenMP device offloading were implemented to improve the performance of OpenMC on GPUs.

1.1.3 GPU Monte Carlo Neutron and Photon Transport

Monte Carlo particle transport simulations on GPUs was attempted only four years after the release of CUDA in a piece of research introducing the "particle-per-block" approach [55]. Specifically this work focused on x-ray photon transport physics. Tickner's approach observed that because Monte Carlo particle transport codes tend to introduce significant branching, the only feasible means towards running particles on GPUs would be to assign particles to entire blocks of threads, and attempt instead to parallelize operations carried out on particles. For example, the intersection of the ray with triangles geometry could test multiple potential intersections simultaneously, or the sum across various elements' contributions to the total cross section could be distributed over threads.

Tickner's approach, however, never caught on. While it likely could provide a performance benefit in some applications, particularly for tallying like in Sweezy's volumetric ray casting method [56] or potentially variants of it. In the context of neutron transport, however, the particle-per-block approach misses the opportunity for massive coalesced memory accesses in nuclide cross section lookups that drives performance upward as more particles run in flight, as demonstrated in works like [28], [29], [54]. The differentiating factor between the photon problems Tickner studied and neutron transport problems lies in the cross section grid; neutron transport are dominated by larger grids such as what XSBench models [57].

While the computing of GPUs may be a paradigm shift over pure single-threaded CPUs, it's fortunate that vector-based computers were considered for simulating neutron transport from the earliest days of computing [18]. The demands of vector-based computers are somewhat similar to that of GPUs: in this case, one instruction acts on many pieces of data. This is known as single instruction multiple data (SIMD) programming, and the event-based Monte Carlo method was developed to more completely utilize the resources of a SIMD computer.

This work continued through the 1980s in the works from Brown and Martin [19], [26], [58], culminating in an event-based Monte Carlo neutron transport code called RACER that offered excellent performance on vector computers. A tangible benefit over the history-based algorithm was proven, and the concepts developed therein laid the foundation for GPU computing when it blossomed three decades later. It was in this period that the event-based Monte Carlo technique was polished off; rather than reorganizing whole arrays of particles, it was realized here that arrays of pointers corresponding to each event queue offered a benefit by reducing particle shuffling. This technique was later used in the work on WARP [52] and named "reference remapping". Towards the end of the vector computing era, a code called VMONT was developed that yielded 10x speedups over non-vector MC code [59]. Despite the impressive speedups, this computing paradigm fell into disfavor as SIMD arose, which operated on smaller vectors. To our knowledge, no MC codes were developed with the goal of leveraging SIMD instructions specifically.

As best we know, the first attempts to GPU-based Monte Carlo neutron transport were presented in the master's thesis [60]. In that work, Nelson developed a mini-app that represented the core computational difficulties associated with this problem: extensive branching and randomized memory access patterns. The work found speedups of 15-20x compared to CPU-based versions of the same code, and recognized the applicability of event-based algorithms to GPU Monte Carlo. This work found that history-based tracking outperformed event-based, though.

Following Nelson's work, some insightful research on GPU-based Monte Carlo neutron transport was produced from within Nvidia [61]. While lacking a level of depth of Nelson's work, this research better underpinned some of the programming techniques that can enhance performance of this type of application on GPUs. For example, while the conventional wisdom goes that maximizing occupancy can be the key to unlocking performant GPU code, using more registers and lowering occupancy can reduce the number of required global memory transactions, enhancing performance [25].

The WARP [52], [62], [63] code demonstrated speedups in more detailed continuous energy Monte Carlo transport problems. The developers of WARP used proprietary geometry

libraries optimized for GPU to achieve fast particle flight kernels.

Some implementations of multigroup Monte Carlo appeared not long after, demonstrating excellent performance [64]–[66]. Unfortunately, the relative simplicity of multigroup-based Monte Carlo code makes it a poor model of continunous energy applications. Specifically, studies which compare history and event based tracking tend to conclude that history-based tracking is superior for multigroup Monte Carlo, but event-based tracking is superior for continuous energy [28]. It appears that a threshold in the complexity of the code, including a sufficient representation of all types of physics and geometry required for practical nuclear engineering, tends to cause event-based tracking to eventually outperform history-based tracking. This complexity threshold is crossed when including effects like thermal scattering, unresolved resonance handling, inelastic scattering, etc. With the addition of routines to handle these effects, the code becomes more divergent both in the instruction and memory access sense, favoring the use of event mode.

A glimpse at the performance difference between a performance portability framework and CUDA was given in [67]. In this work, the NVIDIA Thrust library was considered to be representative of a performance portability framework, and factors of 2-5 slower performance were observed compared to CUDA. Notably, this work focused on mono-energetic problems in one dimension only; the code complexity came as a result of modeling binary stochastic medium properties. The code complexity therefore likely had substantially fewer branches and data types than a continuous energy, constructive solid geometry Monte Carlo code: a factor of 100x less in our estimation. Nonetheless, this work found that event-based algorithms outperformed history-based ones. This contrasts the findings in [66] where history mode was found to outperform event-based tracking for multigroup, simplified geometry problems.

Work on the PATMOS code [68] explored the viability of performance portability approaches to continuous energy GPU Monte Carlo. As part of that, that work explored a few variations on the cross section lookup problem within performance portability frameworks and CUDA. The variations on the cross section lookup algorithm included the binary, n-ary, double-indexing [69], and hash map [70] approaches. The concluding sentence of this thesis is: "Programming models like StarPU, Kokkos and OpenACC should be evaluated instead of the low-level CUDA library." and we wholeheartedly disagree with this statement unless the developer has access to the compiler team. Because this work only explored simplified models of neutron transport, the full complexity of the application changes the optimal solution. Excluding the OpenMP-based OpenMC work in which the developers collaborated with the OpenMP Clang development team, only codes which have been implemented in pure CUDA have attained performance on GPUs worthy of use outside a research setting:

PRAGMA and Shift.

Around 2019, the first two continuous energy MC codes with notable performance gains compared to CPU execution were presented: PRAGMA and Shift [31]. The developers of SHIFT brought one of the most full-featured physical simulations done on GPU yet, including effects such as thermal scattering and unresolved resonance treatment. PRAGMA has achieved the highest performance yet of any GPU MC code. Interestingly, the initial presentation of PRAGMA used windowed multipole [71].

Later, both PRAGMA and Shift's practical applicability due to their novel performance on GPUs led to implementation of domain decomposition handling [72], [73]. Because memory is far more constrained on GPU architectures than CPU, this stood as a necessary step towards practical full-core reactor analysis using GPU Monte Carlo.

PRAGMA eventually developed a few novel optimizations for continuous energy Monte Carlo neutron transport [29]. It was shown that a hybrid history-based method can outperform pure event-based Monte Carlo. This method requires particles to undergo a small number of events and then be saved back to a queue for sorting with respect to the particle energy. Their work on optimization of the tracking rate also included a novel "data-based hash" which is an advancement over the logarithmic hashing approach [70] because it provides an equal amount of search work to be done to each GPU thread.

Deviating slightly from GPU programming, other work explored the ability to speed up OpenMC using many integrated core (MIC) architectures which differ from GPUs in that many cores able to carry out independent instructions are employed [74]. This work presented the highest single node particle processing rate of any MC code, but the MIC architecture fell into disfavor over time as GPUs rose to dominance in other applications.

The PRAGMA code has been applied to solving the BEAVRS benchmark with cycle depletion including multiphysics feedback and critical boron search [75]. It has also been adopted to use hardware-accelerated ray tracing on unstructured tetrahedral meshes for analysis of advanced microreactors [76].

The first time-dependent GPU Monte Carlo neutronics was developed in [77]. This work implemented a two group MC method implementing Sjenitzer's method [4] for a real-life reactor and gave reasonable comparison to experiment. However, as has been previously mentioned, the multigroup paradigm fails to present the pressing challenges of GPU Monte Carlo both from a programming and performance point of view.

The applicability of performance portability frameworks to Monte Carlo and method of characteristics model problems was explored in [78]. The most relevant discussion there pertained to acceleration of the XSBench [57] model problem, where it was shown that OCCA [50], [51] outperformed CUDA. Considering that OCCA uses CUDA as a backend,

we find this result surprising. The compiler optimization flags were perhaps different in this case.

Disappointed by the modest performance improvements demonstrated by efforts to improve the speed of GPU Monte Carlo applications, Sweezy [56] proposed offloading volumetric ray casting (VRC) tallies to GPU. These tallies generate straight-line rays that propagate through an entire problem at each collision site, with exponential attenuation. They can be interpreted as the expected contribution of many track length estimators of potential particles originating from a collision site. By tracing these rays through the problem, it was shown that the Monte Carlo figure of merit, or inverse product of variance and run time, could be improved on mesh tallies. VRCs have not yet been attempted in large-scale reactor computations, and may offer some performance benefits. Due to the rapid attenuation in optically thick reactor environments and the correlation of the samples, our experience with particle splitting methods suggests that this research direction is not particularly promising for GPU-based continuous energy reactor simulations.

Bossler honed in on the performance impacts of tallying [79] on GPUs. One critique of the study is that the problem is likely a bit too simplified to capture the idiosyncracies of full-physics Monte Carlo particle simulations; the work focused on tallying exponential attenuation of mono-energetic photons along a line. The study demonstrated that replicating tally data on a GPU can offer a 10x performance benefit compared to shared data, which requires atomic memory transactions. However, in practical reactor calculations, tally memory already forms one of the main constraints, driving developers to implement domain decomposition methods [72], [73] to distribute massive tally data across many GPUs. Bossler's work also highlights the benefit of collective thread communications before carrying out atomic transactions, something that our work later confirms particularly regarding writing particle IDs into event queues. This will be discussed in detail in the next chapter.

In contrast, Shriver's work [80] sought to develop a computational mini-app for assessing cross-section lookup operations on GPUs that models the complexities of event-based MC methods more accurately. It sought to use more realistic energy grids which do not balance the work perfectly between threads, reflective of production GPU MC code environments. It was shown in that work that their code, VEXS, more accurately mirrored the behavior of a production GPU MC code, Shift [28]. Therefore, we suggest that researchers seeking to develop new cross-section lookup algorithms for GPUs test their methods with VEXS in addition to XSBench [57] to examine the sensitivity of the method on realistic, irregular energy grids.

Our effort is not the first to demonstrate a speedup of the OpenMC code using GPUs. The first author to achieve this tested small, leakage-heavy neutron transport problems

with a single nuclide [81]. That work used history-based algorithms. Because the problems tested in that work were so leaky, particles were sourced, had their cross sections looked up, and leaked. Collisions rarely happened, compared to reactor simulations. Moreover, only one nuclide was included in each problem. The problem geometries were meshes, and the authors adapted their code to use proprietary NVIDIA libraries for surface mesh ray tracing. Because the problems considered were heavy on ray-tracing operations, fantastic speedups were reported. Further work never explored the applicability of this code to reactor simulations with hundreds of nuclides and far more events per particle. We suspect that the approach would perform poorly for reactor simulations.

The development of a robust GPU-based MC capability with Lawrence Livermore National Laboratories has been documented well over the past few years in the works [82]–[84]. While some initial results accelerating LLNL MC applications with GPUs were not promising [84], showing slowdowns on a per-compute-node basis, continual development has brought GPU modes of computation to the forefront. The work at Livermore has focused on the Imp and Mercury codes, which respectively model thermal photon transport and neutral particles. The authors report the use of a few conveniences known to be sub-optimal for GPU computing: double precision floating point numbers, recursion, and virtual functions. In addition, similarly to our CUDA-based OpenMC code discussed later, the authors use managed memory. Converting the numbers from the report [84], we see that Mercury on one NVIDIA V100 GPU performed equivalently to about seventy CPU cores.

Work at Argonne National Laboratory has delivered some of the most performant continuous energy Monte Carlo code with a modified version of OpenMC [54]. Their approach refactored OpenMC to use entirely flat data structures. Their approach also refactored the whole OpenMC codebase to not use polymorphism, pivoting to tagged union data structures instead. The approach to GPU programming used OpenMP [85], which supports offloading of computations to accelerators in general, a superset of GPUs. The developers collaborated with the Clang OpenMP 5 development team as part of this project, and were able to integrate optimizations into the compiler itself (rather than modifying the simulation code, as all other works cited in this section have) to produce better results. In fact, this compiler optimization delivered a 10x performance increase. The code produced as a part of this thesis pales in comparison, attaining roughly half the performance on the Hoogenboom-Martin benchmark [86].

A recent conference was held in which developers from across the United States shared their successes and failures in adapting their Monte Carlo particle transport applications for GPUs [87]. A newcomer code to the GPU MC world was introduced there, MC/DC which seeks to employ just-in-time compilation methods to Python code to generate GPU

code via Numba [88]. Results from the aforementioned Livermore codes were presented, as were results on the aforementioned Shift code. For the first time, to our knowledge, public statements were made about performance improvements of the Jayenne and MCATK codes from Los Alamos National Laboratory which are respectively thermal photon and neutron transport codes. Due to the differences in the types of problems solved by each code, a single benchmark problem could not be identified to measure the performance of GPU particle transport simulations. The needs of thermal photon simulation codes, neutron transport, and photon transport all differ tremendously.

With a variety of GPU-based applications laid out, and a comprehensive overview of the attempts at GPU-based continuous energy particle transport in mind, we can quantify some of the performance bottlenecks of bringing continuous energy MC codes to GPU. One ubiquitously used algorithm employed in Monte Carlo codes is rejection sampling. In the next section, we present our novel contribution in quantifying the performance impact of rejection sampling on SIMT architectures.

We have introduced how Monte Carlo algorithms can be used to solve neutral particle transport problems, proceeding from an assumption that the Boltzmann equation adequately models the situation. The history-based particle tracking algorithm serves as the baseline implementation of the Monte Carlo method for continuous energy neutron transport simulations. We then introduced the intricacies of GPU hardware and its implications on GPU programming, which motivates the use of the event-based tracking method. Before proceeding, we present some novel work that enables the analysis of algorithms which use rejection sampling on GPUs.

Chapter 2

Impact of Rejection Sampling in SIMT Programming

This chapter is based on the following paper:

> Ridley, Gavin, and Benoit Forget. "A Simple Method for Rejection Sampling Efficiency Improvement on SIMT Architectures." Statistics and Computing 31, no. 3 (March 30, 2021): 30. https://doi.org/10.1007/s11222-021-10003-z.

As previously mentioned, it is commonplace for a warp of threads to encounter a point in the program where some threads enter an if-block, and others do not. This situation is called *warp divergence*, and should be avoided if possible since it degrades the computational throughput of the GPU. Warp divergence may be impossible to avoid in rejection sampling algorithms where some of the threads in a warp may have to sample again in order to obtain an accepted sample. This section finds a closed form result for how the number of iterations required to complete rejection sampling is distributed on a warp as a function of the number of threads per warp and the rejection probability. This distribution is found to exactly match the exponentiated geometric distribution proposed by [89], which to the authors' knowledge has not been previously found to exactly govern any other processes.

Although rejection sampling typically takes more time to draw a sample on a computer than other methods, the method is robust and can sample any distribution for which the maximum probability is known. To restate the definition of rejection sampling given by [90], rejection sampling from a distribution X with support in $\mathbb{R}^n$ with density f. Then, if g is another distribution in the same space we can draw samples from f by the following procedure if

$$\exists c \geq 1, c \in \mathbb{R} \quad \text{s.t.} \quad f(x) \leq cg(x) \forall x \in \mathbb{R}^n. \tag{2.1}$$

With a value of c satisfying the above, a sample X may be drawn by Algorithm 3. Algorithm 4 shows how this can be modified to execute in parallel.

repeat
  sample U uniformly from $[0, 1]$;
  sample X with density g;
until $Ucg(X) \leq f(X)$;
return X;

Algorithm 3: Rejection sampling requires a loop which terminates stochastically.

repeat
  if $U_i cg(X_i) \leq f(X_i)$ **then**
    continue;
  end
  else
    sample U_i uniformly from $[0, 1]$;
    sample X_i with density g;
  end
until $U_i cg(X_i) \leq f(X_i) \forall i \in [1, t]$;
return X_i;

Algorithm 4: SIMT-parallel rejection sampling requires a loop which terminates stochastically on each of the t threads of a warp.

Understanding the performance of rejection sampling on SIMT architectures can be impactful because rejection sampling is found in myriad applications. Rejection methods are most commonly employed in Monte Carlo methods. Rejection is used nearly ubiquitously to sample from the gamma distribution, following the method prescribed by [91]. The sampling of distributions can be a performance bottleneck for certain problems like deep belief networks where GPUs are used to generate millions of samples as quickly as possible. In addition, rejection sampling may be used in Monte Carlo particle transport simulations, e.g., OpenMC [92] uses rejection to sample outgoing scattering angle distributions for particle simulations where particles take on discrete energies and for sampling the resonance upscattering effect [93] encountered in continuous particle energy simulations. Several recent developments like [94] and [95] extensively employ rejection sampling in their proposed algorithm. In light of this, a theoretical model for the expected decrease in speed of SIMT versus MIMD evaluation of a rejection sampling method would be useful.

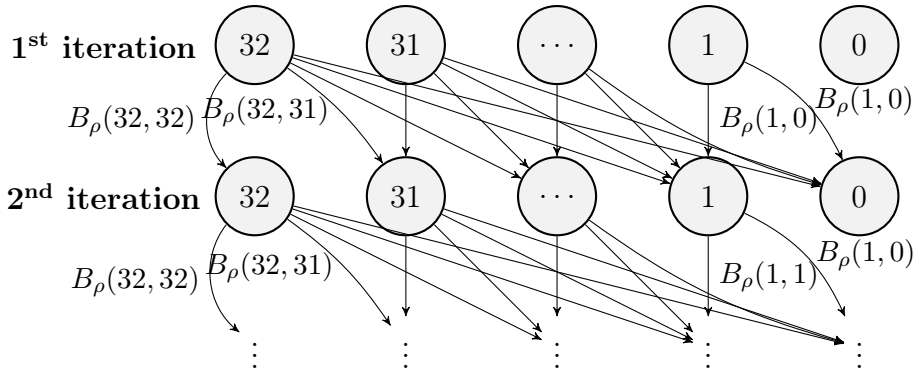


Figure 2.1: Infinite Markov chain representation of transitions between possible amounts of threads awaiting an accepted sample. Two iterations of a thirty-two thread rejection sampling algorithm are depicted. Edges are the probability of transitioning between each state, given by the binomial distribution $B_\rho(i,j)$. The relevant value of the binomial distribution is placed left of a few edges above.

2.1 Derivation of the distribution

Let the rejection probability for a sample be ρ. Then, among t threads, the probability of k threads being rejected to require another sample is given by the binomial distribution:

$$B_\rho(t,k) = \binom{t}{k}\rho^k(1-\rho)^{t-k} \tag{2.2}$$

Using this, we can use induction to predict how the number of threads awaiting an accepted sample changes with each iteration. t is the number of threads initially seeking a sample via rejection, and call $t_1, t_2, \cdots$ the number of threads yet to obtain a sample at each of the successive sampling iterations.

Now, consider the random variables t_n and t_{n+1}. We can see from Fig. 2.1 that:

$$t_{n+1}|t_n \sim B_\rho(t_n, t_{n+1}) \tag{2.3}$$

Moreover, if t_n has the distribution

$$t_n \sim B_{\rho^n}(x, t_n) \tag{2.4}$$

it is then true that

$$t_{n+1} \sim B_{\rho^{n+1}}(x, t_{n+1}) \tag{2.5}$$

From here, it is straightforward to establish the distribution of each t_n inductively. Using

the base case:

$$t_1 \sim B_\rho(t, t_1) \tag{2.6}$$

We then immediately see from the previously stated inductive rule:

$$t_n \sim B_{\rho^n}(t, t_n) \tag{2.7}$$

We can now calculate the probability of termination of the sampling routine in n steps. To be precise, the quantity in question is $P[t_n = 0 | t_{n-1} > 0]$. Computing this by summing all of the independent probabilities in question gives:

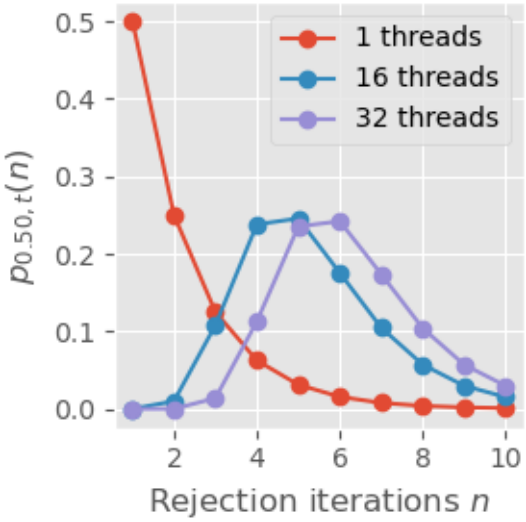
$$\begin{aligned} P[t_n = 0 | t_{n-1} > 0] = p_{\rho,t}(n) &= \sum_{i=1}^{t} (1-\rho)^i B_{\rho^{n-1}}(t, i) \\ &= \sum_{i=0}^{t} \binom{t}{i} \left(\rho^{n-1} - \rho^n\right)^i \left(1 - \rho^{n-1}\right)^{t-1} - \left(1 - \rho^{n-1}\right)^t \end{aligned} \tag{2.8}$$

After applying the binomial theorem once more, this results in the desired distribution for the probability of t SIMT threads taking n iterations to sample some distribution using a rejection method with rejection probability ρ:

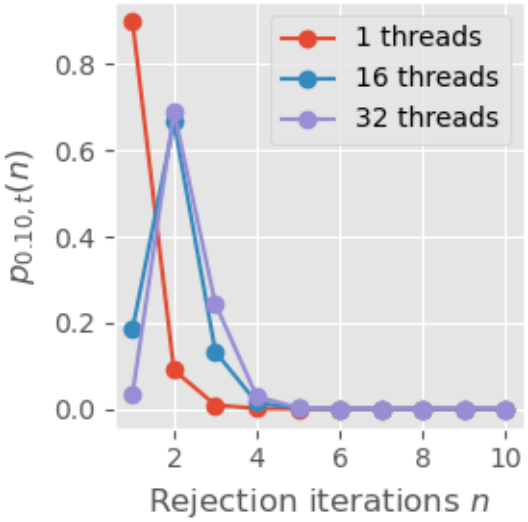
$$\boxed{p_{\rho,t}(n) = (1-\rho^n)^t - (1-\rho^{n-1})^t} \tag{2.9}$$

This discrete distribution has been plotted for a few values of the parameters in Fig. 2.2. Low ρ values are plotted because these are commonly encountered in ziggurat-type algorithms [96]. This distribution has a cumulative distribution function (CDF) of $(1-\rho^n)^t$. This distribution is identical to the one recently proposed by [89], named the exponentiated geometric distribution. This distribution was originally proposed as a modification to the geometric distribution, found simply by exponentiating the CDF of the geometric distribution. To the extent of the authors' knowledge, the exponentiated geometric distribution has thus far only been proposed to fit certain data better than other distributions. This may be the first known case where the exponentiated geometric distribution exactly governs the underlying statistical process.

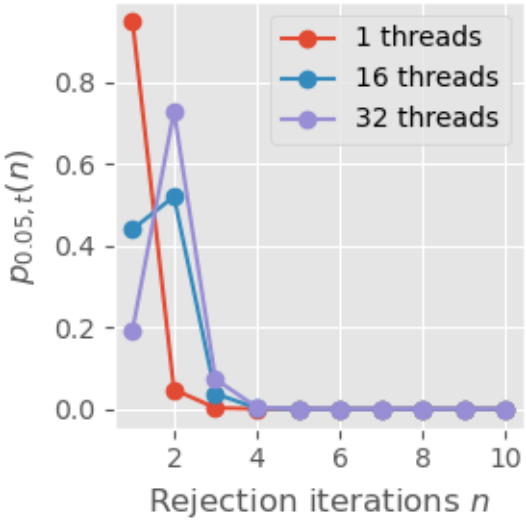
We have verified in numerical experiment that this distribution indeed governs the rejection sampling process on the GPU with a simple CUDA program. A surrogate rejection sampling process has been used, where a rejection loop exits with probability $1 - \rho$. The code for this numerical experiment can be found in Appendix A, and its results compared to the new theoretical result are illustrated by Fig. 2.3.



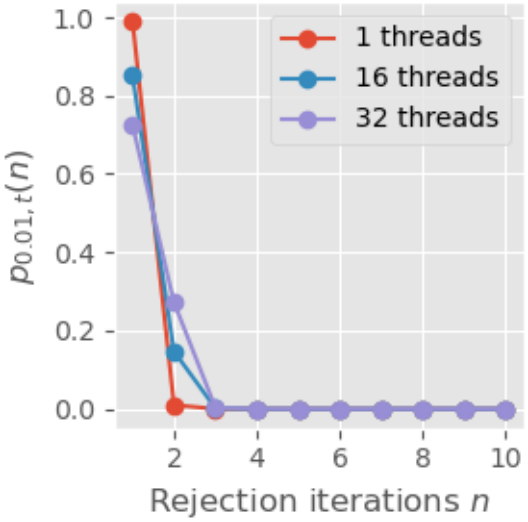
(a) $\rho = 0.5$



(b) $\rho = 0.1$



(c) $\rho = 0.05$



(d) $\rho = 0.01$

Figure 2.2: Probability distributions of completion of the rejection sampling algorithm in a given number of iterations for $\rho = 0.5$ (top left), $\rho = 0.1$ (top right), $\rho = 0.05$ (bottom left), and $\rho = 0.01$ (bottom right).

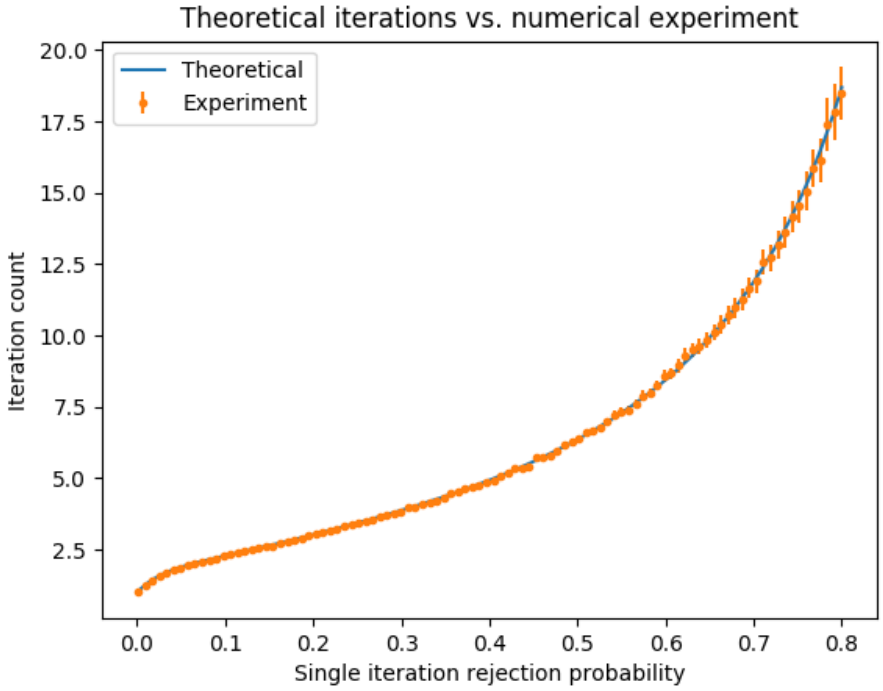


Figure 2.3: The code of Appendix A was used to verify the predicted mean iteration count by Eq. 2.9, with which excellent agreement has been obtained.

2.1.1 Approximate Formula for Mean of The Distribution

The work of [89] provides a convenient numerical expression for all of the moments of the exponentiated distribution. The formula provided contains a summation over infinitely many indices which does not clearly evince any information about the sensitivity of the mean with respect to the distribution parameters. Moreover, we also point out that while finding the mean of this distribution is indeed tractable in closed form, we have verified using computer algebra software for the case of interest $t = 32$ that this takes the form of a three-hundred and twenty-third order rational polynomial function in ρ. Obviously, not much is afforded here in terms of intuitive understanding of the behavior of the mean with respect to ρ and t.

As a result of this, we have derived a simple expression for the mean which can guide intuition about the behavior of the exponentiated geometric distribution upon changing its parameters. This is done in two parts: firstly for small values of ρ as are reminiscent of ziggurat-type algorithms, and secondly for larger values of ρ. This is because the exponentiated geometric transitions from being a monotonically decreasing distribution to a unimodal distribution for $\rho > 2^{1/t} - 1$, as shown by [89]. Two separate formulas can approximate these regions with excellent accuracy.

Approximate Mean in Unimodal Region

It was proved by [97] that the mean of a unimodal distribution always lies within 1.74 standard deviations of the mode. Regarding the distribution discussed here, [89] proved unimodal behavior for $\rho > 2^{1/t} - 1$, so we know the mean could be well-approximated by the mode for $\rho > 2.2\%$ in the 32 thread per warp case which is ubiquitously found in GPUs. Finding a mode here is much easier than finding the mean, which cannot be described in terms of elementary functions for general t, as far as we have found. Despite this inconvenience, Fig. 2.5 shows that the mode approximates the mean quite acceptably. Approximating the mean more accurately has been done, but we found that this results in complicated combinations of special functions like the polylog and hypergeometric functions which lend little insight to the problem at hand, and thus not discussed.

To begin, note that the binomial expression, with t large, is well-approximated by:

$$(1 - \rho^n)^t \approx \exp(-\rho^n t) \tag{2.10}$$

This approximation comes from observing that because $e^x = \lim_{N\to\infty}(1 + \frac{x}{N})^N$, if we define $x' = -\rho^n t$, the expression above is simply $(1 + \frac{x'}{t})^t$. From this definition of the exponential, we can see that this may arbitrarily closely approximate $e^{x'}$ as t grows.

Using this, the exponentiated geometric distribution is well-approximated by the following continuous distribution, so long that t is large:

$$p_{\rho,t}(n) \approx \left(\exp(-\rho^n t) - \exp(-\rho^{n-1} t)\right) \tag{2.11}$$

Then, note that this continuous approximation very nearly preserves normalization:

$$\begin{aligned}\int_0^\infty \left(\exp(-\rho^n t) - \exp(-\rho^{n-1} t)\right) \mathrm{d}n &= \frac{1}{\ln(\rho)} \int_1^0 \left(\exp(-ut) - \exp(-ut/\rho)\right) / u \, \mathrm{d}u \\ &= 1 - \frac{1}{\ln(\rho)} \left(\mathrm{Ei}(-t) - \mathrm{Ei}(-t/\rho)\right)\end{aligned} \tag{2.12}$$

Where the substitution $u = \rho^x$ was used, and $\mathrm{Ei}(\cdot)$ is the exponential integral function, defined in almost any table of integrals, e.g. [98]. Fig. 2.4 clearly demonstrates the adequacy of this approximation for $t > 16$ in terms of its preservation of normalization by plotting the difference between 2.12 and unity.

With this in mind, this value of n which maximizes Eq. 2.11 is expected to approximate

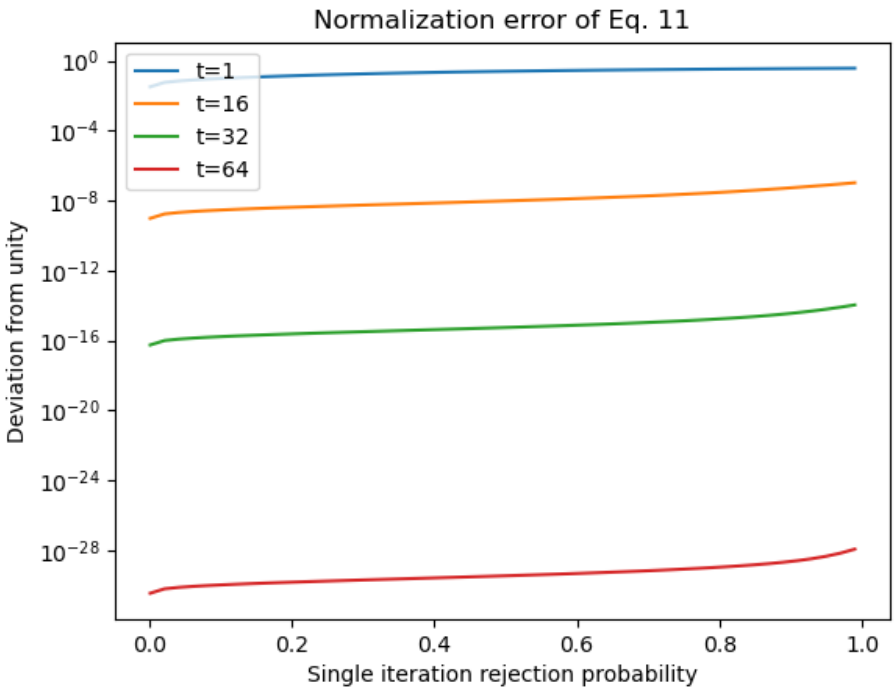


Figure 2.4: This depicts the difference between the integral of Eq. 2.11 over its domain and unity. The preservation of normalization by using the approximation Eq. 2.11 can be seen to be excellent for $t > 16$.

the mean of the discrete distribution in question:

$$\mu \approx \frac{\ln\left(\frac{2(\rho-1)t}{\ln\rho}\right)}{\ln\left(1/\rho\right)} + 1 \tag{2.13}$$

From this, we can see that the expected number of iterations required to exit the parallel rejection sampling loop grows logarithmically with the number of threads for r sufficiently large.

Approximate Mean in Monotically Decreasing Region

In the monotonically decreasing region, i.e., where $\rho < 2^{1/t} - 1$, the derivation of an approximate formula for the mean is much simpler and can be done based on the fact that ρ will be quite small here. The derivation proceeds by considering the exact expression for the mean in terms of an infinite sum of integers times their corresponding probabilities, regrouping some terms in that series, and approximating $\rho^k \approx 0$ for higher order terms in the sum, where k is a small integer. So, the exact mean is:

$$\begin{aligned}\mu = \sum_{i=1}^{\infty} i\left((1-\rho^i)^t - (1-\rho^{i-1})^t\right) = (1-\rho) + \\ 2\left((1-\rho^2) - (1-\rho)\right) + \\ 3\left((1-\rho^3) - (1-\rho^2)\right) + \\ 4\left((1-\rho^4) - (1-\rho^3)\right) + \\ \ldots\end{aligned} \tag{2.14}$$

Note how in the second line, we see $-2(1-\rho)$, which cancels out with the term in the first line. This cancellation can be carried out N times in order to see that the mean is:

$$\mu = \lim_{N\to\infty}\left(N(1-\rho^N)^t - \sum_{i=1}^{N}(1-\rho^i)^t\right) \tag{2.15}$$

The key to obtaining a good approximation is to take k terms of the first sum, and suppose that the remaining terms are close to one since ρ^k is very small. This allows a cancellation with the increasing term on the right, and for $k = 2$ gives:

$$\mu \approx 3 - (1-\rho)^t - (1-\rho^2)^t \tag{2.16}$$

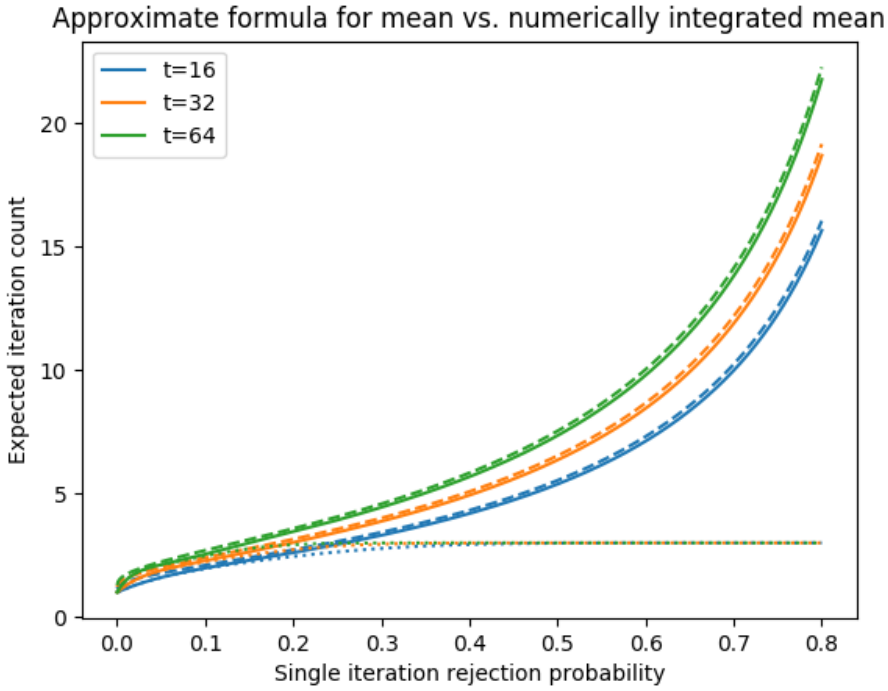


Figure 2.5: The dashed curve is from the large ρ Eq. 2.13, dotted curves are the small ρ Eq. 2.16, and solid curves are the true mean for a few t values.

With that, an accurate piecewise approximation to the mean of this distribution is:

$$\mu \approx \begin{cases} 3 - (1-\rho)^t - (1-\rho^2)^t & \rho < 2^{1/t} - 1 \\ \frac{\ln\left(\frac{2(\rho-1)t}{\ln \rho}\right)}{\ln(1/\rho)} + 1 & \text{else} \end{cases} \tag{2.17}$$

Both formulas have been plotted in Fig. 2.5, with the approximate mean formula compared to the actual numerically calculated mean for a few different values of the thread count. Fig. 2.6 shows these formulas for small values of ρ, where the second formula can be seen to give great accuracy.

2.1.2 A More Efficient Sampling Strategy

Observing from Fig. 2.5 that it is expected to take over seven iterations on average for thirty-two threads to obtain a sample with a 50% rejection probability, which compares unfavorably with the expected two iterations encountered in MIMD rejection sampling algorithms, it is natural to suppose that a more efficient algorithm could perhaps be obtained if, say, only 16 samples are produced by the warp in groups of two threads working to obtain a sample.

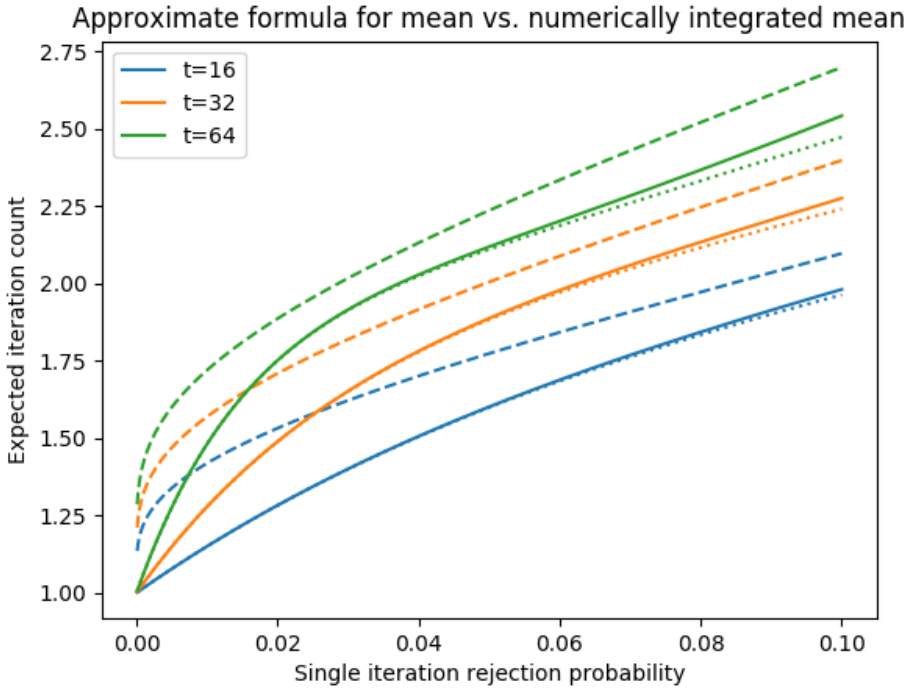


Figure 2.6: This is Fig. 2.5 but with the abscissa zoomed to the $\rho \in [0, 0.1]$ interval to show how the dotted curve Eq. 2.16 is best in this range.

As a further example, consider an extreme case like a 99% rejection probability. It would be expected that dedicating thirty-two threads to each finding a single accepted sample would be more efficient than waiting on thirty-two threads to each obtain their own sample, which, according to the formulas above, would take around 400 iterations. In this case, the probability of all threads rejecting their sample would simply be ρ^{32}, in which case a single sample would be obtained in $1/(1-\rho^{32}) \approx 3.6$ iterations. As such, the samples obtained per rejection sampling iterations grows from 0.08 to 0.28.

Algorithm 5 concisely describes our proposed modified rejection sampling method to be carried out on each thread group of a SIMT device. This section explores when strategies like this are more efficient than the standard algorithm.

Most rejection sampling algorithms are designed such that the proposal distribution yields a low rejection probability, which is most computationally efficient. As such, it remains in question whether the investigation of high rejection probabilities are worthwhile, considering that algorithms such as the ziggurat method [96] typically yields rejection probabilities less than five percent with a properly designed proposal distribution table. Similarly, the commonly employed rejection sampling scheme for generating gamma variates by [91] has a rejection probability less than five percent.

repeat
  if $U_i cg(X_i) \leq f(X_i)$ **then**
    continue;
  end
  else
    foreach *thread* t *in group* n **do**
      sample V_t uniformly from $[0, 1]$;
      sample Y_t with density g;
    end
    if *Any* $T_t cg(Y_t) \leq f(Y_t)$ **then**
      save Y_t as this group's sample
    end
  end
until *all thread groups have a sample*;
return X_i;

Algorithm 5: Our proposed modified rejection sampling algorithm which can produce more samples per rejection sampling iteration. There are n thread groups which each produce a sample.

Some more specialized algorithms, in fact, must cope with higher rejection probabilities. From nuclear engineering, the Doppler broadening rejection correction algorithm [93], which can form a computationally significant portion of a Monte Carlo neutron transport simulation, exhibits rejection probabilities over 99% for some parts of the simulation as shown in [99]. Similarly, [100] presents a problem which exhibits rejection rates over 99.99%, along with a solution to that inefficiency. It is also commonly noted that high dimensional probability distributions typically exhibit high rejection rates, e.g. in [101].

On the ubiquitous thirty-two-thread warp, it would make sense to consider firstly thirty-two threads each attempting to obtain their own sample on each iteration. The next possible configuration would be sixteen groups of two threads where only one sample is asked of each thread pair per iteration. Following that would be eight groups of four threads, sixteen groups of two threads and finally a single group of thirty-two threads looking for a single sample. The formulas above can be used still, but t is divided by the number of groups per warp and ρ is raised to the power of the number of threads per group, representing the probability that each sample from the group was rejected.

The total samples obtained per rejection sampling iteration for each of these configurations have been plotted in Fig. 2.7, where it can be seen that small but perhaps worthwhile performance gains can be made by switching to these alternative sampling setups. Similarly, Fig. 2.8 plots the ratio between the optimal method out of this class of methods compared to the case of attempting to obtain a sample on each thread in the warp. Table 2.1 gives the

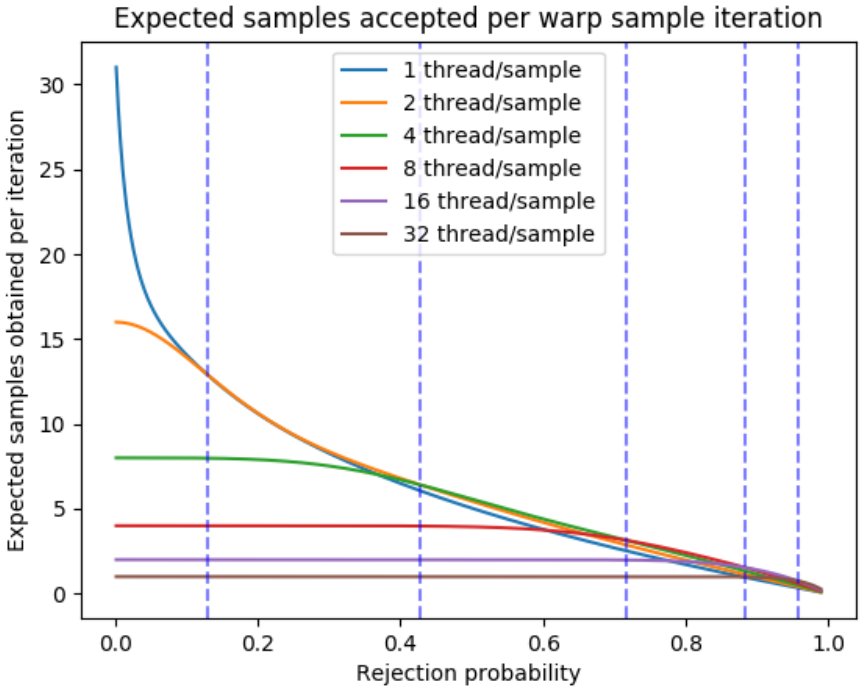


Figure 2.7: Different thread grouping patterns can obtain better sampling performance per unit of GPU work depending on the rejection probability. The dashed vertical lines indicate a location where solid curves intersect, and switching to another grouping is desirable.

specific rejection probabilities where it is optimal to switch to a different layout of thread grouping.

2.2 Discussion

Results have been obtained which predict the performance of rejection sampling in the context of SIMT parallel architectures. While these results do not apply to MIMD parallelism that does not suffer from the necessity of carrying out the same instructions in lockstep, these results do translate to SIMD parallelism where masked instructions are supported. In that case, the number of samples to obtain would correspond to the number of floating point numbers operated on by the SIMD instructions. In the case of the latest Intel AVX-512 instructions [102], which can operate on sixteen single precision numbers at once, this would correspond to the $t = 16$ case. Masked SIMD instructions, which allow SIMD instructions on a subset of the data, could be used to control the exit from the rejection sampling loop. This has not been explored by the authors and could be examined in future work.

The proposed algorithm of this paper is certainly sub-optimal in terms of obtaining the

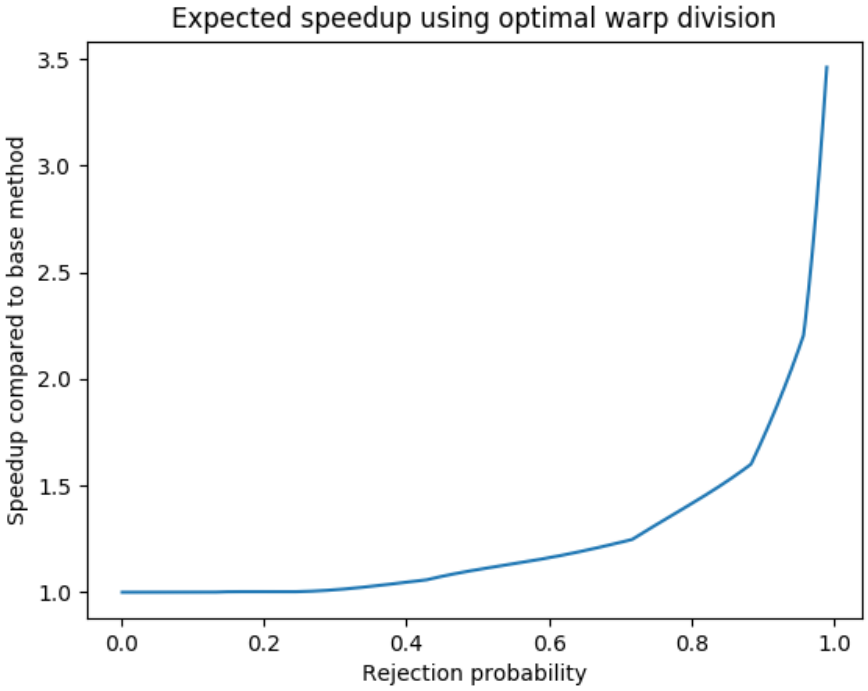


Figure 2.8: As a function of the rejection probability, this plots the ratio between the sampling rate of the best thread grouping of Fig. 2.7 and the base case where each thread is responsible for finding a sample.

Table 2.1: The optimal number of threads per sample to use are given here, in addition to the average expected speedup compared to a single thread per sample for this region, corresponding to divisions by vertical lines in Fig 2.7.

Threads per sample	Maximum efficient ρ (%)	Average Speedup
1	12.88	1
2	42.71	1.01
4	71.70	1.14
8	88.37	1.39
16	95.76	1.84
32	1	2.65

maximum sampling rate for the sampling of one distribution repeatedly on GPU. If this were the desired objective, persistent threading strategies for this task are surely superior. Such methods are detailed by [103] and [34], and have been conclusively shown to improve performance in programs suffering from warp divergence.

The method we present is strong in two distinct ways, however. Firstly, the implementation of this method is very simple, and performance improvements can be made to a GPU Monte Carlo program with little programming effort. Only a few lines of warp-level voting logic and warp-shared memory need be added to implement the method. We expect that statistical software relying on a computationally intensive rejection loop could quickly add this grouped rejection sampling algorithm and easily obtain performance improvements.

Secondly, the proposed method method has been designed to fit into larger Monte Carlo programs which only use rejection sampling as an intermediate but perhaps computationally intensive step. Because different warps may be attending to other parts of the sampling routine of interest, individual warps may be responsible for carrying out a rejection sampling part of the calculation. Because of this, the lauded persistent threading strategies described by works like [103] and [34] were not considered because that very warp may be expected to perform a completely different task in the next stage of the Monte Carlo computation. These are the conditions encountered in Monte Carlo neutron transport simulations that the authors had in mind while developing this method, and we found that similar conditions prevail in large statistical calculations like that presented in [34].

We have shown that the exponentiated geometric distribution [89] exactly governs the statistics of rejection sampling performed by a warp of threads on GPU, where each thread within the warp must perform the same instruction, known as SIMT. The exponentiated geometric distribution has only been shown to fit certain data sets better than other distributions thus far. The findings of this paper suggest that similar phenomena may be identifiable as being distributed according to the exponential geometric.

An approximate formula for the expected number of iterations taken by a thread warp to complete a rejection sampling iteration has been provided, and this formula has been numerically verified to accurately model the expected iteration count for rejection probabilities greater than about 2%. For smaller rejection probabilities, the formula does not work as well, and an alternative formula has been provided.

Lastly, with this theoretical result, we have provided insight on GPU rejection sampling algorithms without resort to numerical experiments. Our result has been verified with a numerical experiment, with which excellent agreement was obtained. This new theoretical result was used to define a more efficient rejection sampling layout on GPU that is particularly effective for high rejection probability. Further work could expand on these results,

particularly regarding the theoretical performance of rejection sampling algorithms that utilize sample caching as in [104]. Knowing the performance impact of rejection sampling on GPUs, we have set the stage for how new algorithms can be designed with GPU execution in mind.

Chapter 3

Bringing OpenMC to GPU

This chapter is a considerably expanded and updated version of the conference paper [105]:

> Ridley, Gavin, and Benoit Forget. "Design and Optimization of GPU Capabilities in OpenMC." ANS Winter Conference, Washington D.C., 2021.

The code discussed in this chapter can be found at https://github.com/gridley/openmc/tree/cuda.

The computational demands of MC to obtain satisfactory solutions for reactor problems involving transients, depletion, and multiphysics can be quite high, and even out of reach for today's computers within reasonable wall times. To solve challenging reactor problems today, we can enable MC applications to run on graphics processing units (GPUs), which tend to attain more FLOPs and memory bandwidth per dollar than conventional central processing units (CPUs). Utilizing this computational firepower for GPU MC, however, has proven nontrivial.

In this chapter, we discuss the development of a GPU-accelerated version of OpenMC. In the beginning, the focus was on offloading the cross-section lookup operation only to GPUs. Cross-section lookup has been shown to yield excellent performance in [57]. However, the overhead of sending microscopic cross-sections back from the GPU to the CPU was shown to incur overhead that would yield unacceptable performance compared to offloading the whole transport process to the GPU, and as part of this chapter we provide an argument as to why offloading cross-section lookup is the wrong approach to GPU MC neutronics. Instead, the full transport process should be implemented on GPU.

To this end, we developed a full-physics implementation of continuous energy GPU Monte Carlo by rewriting or modifying all necessary OpenMC code to be callable in CUDA kernels. In the process, we developed a variety of novel programming techniques which can allow even an individual graduate student to tackle the onerous process of offloading a continuous

energy Monte Carlo neutron transport code to GPU. Relative to the three other continuous energy GPU MC neutronics codes worldwide which have been used to simulate reactors, the performance was competitive.

We believe that the work developed here has the smallest source code delta compared to other efforts to accomplish this task. We show a simple path towards converting complex continuous energy particle transport codes into reasonably but not optimally performant GPU-enabled codes with maximum code reuse and input/output compatibility. Our work explores the efficacy of an accelerated version of OpenMC [92] in the CUDA language. The GPU version has been kept as similar as possible to the CPU version, and returns identical eigenvalue results compared the the CPU version. OpenMC can use event-based [26] tracking in CPU mode, which the GPU implementation presented here is an acceleration of. While this approach does not immediately yield code highly optimized for GPU execution, our guiding strategy is that bottlenecks can be optimized thoroughly, and that the remaining parts of the code can be shared with the CPU version. The two novel resulting benefits are reproducibility of results between CPU and GPU versions and ease of maintenance and development of the GPU code going into the future.

To begin to understand the problem, we produced a call graph of the computationally dominant functions in OpenMC particle tracking using GraphViz [106]. Fig. 3.1 shows how the different particle operations can be grouped into events to run on the GPU; the function calls are mostly separate from each other. In an abstract sense, converting a complex Monte Carlo code to use event-based processing is a graph cutting problem. The events access similar data or tend to take similar amounts of time to execute, or tend to diverge from each other less.

The "embarrassingly parallel" nature of MC methods conceptually makes parallelization of such a code superficially straightforward; one need only run the serial program in parallel but with different random number generator seeds in order to parallelize MC, combined with a simple parallel reduction on tallies [1]. Despite this, most early work on GPU MC yielded disappointing speedups relative to, for example, deep learning problems. Disappointing speedups originate from the nature of the GPU; the GPU performs thousands of simple operations in parallel efficiently, but irregular memory accesses parallelize inefficiently. Conventional formulations of Monte Carlo neutron transport fall into this category. Consequently, research exploring the efficacy of specific techniques for continuous energy GPU MC neutronics can overcome the performance limitations of conventional MC methods for continuous energy neutron transport. Our research specifically restricts its viewpoint

[1] Issues regarding parallel fission banks and tally decomposition may complicate this matter, but for the most part MC is indeed embarrassingly parallel

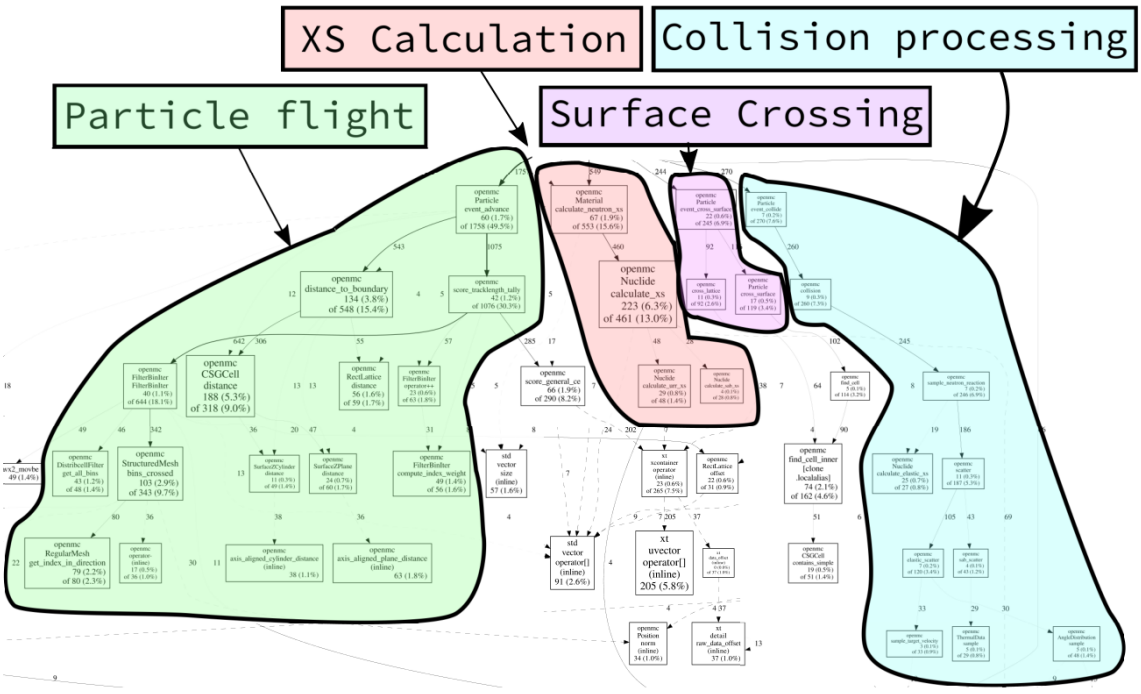


Figure 3.1: Call graph for OpenMC particle tracking, grouped by event type.

to neutron transport calculations in reactors, where, to our knowledge, only three sets of authors have reported speedup relative to CPUs on GPU in the codes WARP, PRAGMA, and Shift, described in respectively [28], [31], [52].

If event MC methods were already so well developed as discussed in the introduction chapter, why should we research new ones? Very generally, computers of the past tended towards being limited by compute throughput. Nowadays, the ratio of memory bandwidth to data processing rate is quite different, implying our algorithms should be designed with this in mind. The same principle especially applies to GPU, where memory accesses from threads are not only relatively slow, but happen in serial if the memory accesses do not fall within the same cache line. As such, revisions of event MC should be developed that place an emphasis on locality of access to particle, geometry, and physics data.

3.1 Design of the New Code

The GPU capabilities within OpenMC in this work have been designed with code maintainability in mind. To our knowledge, we have created the first continuous energy neutron transport code that can share the majority of its codebase with the CPU version of the code. The previous works on PRAGMA [31] and WARP [52] were developed as standalone,

GPU-only code. In the work on Shift [28], a set of GPU code disjoint from the CPU code is employed. The other work on OpenMC [54] involved reworking the code to a state that is clearly less friendly to new developers. In our code, we can share most data structures through the use of unified memory and compile-time configurable data structures. Considering the thousands of person-hours that go into developing continuous energy MC codes, this advantage sets our code apart.

3.1.1 The Advantages and Disadvantages of Unified Memory

Unified memory [107] was introduced in CUDA 6, and can considerably ease developer effort by automatically handling memory copying between the host and GPU. For example, in many examples in the antiquated CUDA book [108], memory on the GPU has to be manually allocated with CUDA-specific functions. The pointers to that memory cannot be used on the host, otherwise a segmentation fault will occur. Data allocated on the host side, for example the nuclear data arrays produced in the initialization phase of OpenMC, has to be copied from host-side pointers to these device side pointers in a contiguous fashion.

Unified memory introduces a small amount of overhead, but obviates the need for manually copying data between the CPU and GPU memory spaces. A special memory allocation function called `cudaMallocManaged` returns pointers that may be dereferenced either on the host or the GPU. While the data may be present on only one side of the PCIE slot at any given time, attempts to dereference the pointer on a device not currently holding a copy of it result in an automatic synchronization to the correct place: either CPU or GPU memory.

It might be obvious then, that in a code like OpenMC with deeply nested data structures, unified memory considerably eases the task of allowing its operation on GPU. In the easiest case when a CPU-based code uses a contiguous array of data, the developer must make only one call to `cudaMemcpy`. In a continuous energy Monte Carlo neutron transport code, the multitude of scattering physics require nested data structures to represent them: a reaction might require numerous sub-arrays to represent the various probability distributions within, and each may be represented by different types of ENDF functions such as polynomials or linear interpolation tables. This heterogeneity of the nuclear data is a challenge in three ways for GPU acceleration.

Firstly, implementation of the GPU code itself is rendered more difficult, as all the nested data must be either flattened to lie contiguously in memory, or specialized code written to handle the allocation and copying of each individual type of data. Secondly, all GPU programming models with the exception of CUDA, to our knowledge, do not support polymorphism.

In the case of [54], all formerly polymorphic data was manually refactored to use `switch/case` statements, which led to at least one hard-to-find bug. From a developer standpoint, this process renders the code less aesthetically appealing and practically maintainable. Thirdly, if the data is not manually flattened to be contiguous in memory, the data structures point to different locations in memory, leading to a strong performance penalty. The nested data structures typically result in accesses to many different locations in memory. This hampers performance because while GPUs do offer massive computational parallelism, random memory accesses separated sufficiently far from each other lead to complete serialization on each thread block. Modern GPUs may even allow threads to perform separate tasks, but accesses to separate, distantly spaced locations in memory bottleneck the work. This effect is *memory divergence*, the opposite of *memory coalescence*.

So, the most performant approach would be to manually reorganize all nuclear data structures into serialized chunks of memory, and even more preferably convert all ENDF function representations to a single modality: perhaps linear interpolation tables, for example. Premature optimization is a waste of programmer time, however. The forthcoming programming techniques make moving the data to the GPU and using it trivial to program; this is the approach with the minimum effort to performance ratio, despite performance being lackluster compared to other methods, as discussed in the conclusion of this chapter.

3.1.2 Nuclear Data Structures in a CE MC Code

A full-physics implementation of continuous energy Monte Carlo requires nearly full support for the range of reaction and function types in ENDF [109]. To our knowledge, no existing work in the modern MC code literature succinctly summarizes the range of data structures and methods that must be able to run on GPUs for a fully offloaded MC code. In this subsection, we will summarize the ENDF Formats Manual's [109] neutron-relevant models to communicate the code complexity of writing a GPU MC code.

Because the ENDF function formats are naturally expressed by polymorphism, developers who wish to move away from polymorphism must switch to a potentially less elegant solution like tagged unions [110]. Figure 3.2 shows the inheritance types for different function types in ENDF used by OpenMC.

Similarly, the data types used to represent energy and correlated angle-energy distributions in OpenMC use polymorphic behavior. Naturally, these classes implement an abstract method that represents sampling a scattering event's outcome. Figure 3.3 shows the types of energy distribution in OpenMC. The `Tabulated1D` objects add yet another step of indirection after the indirection required by storing a pointer to an object guaranteed to be of the

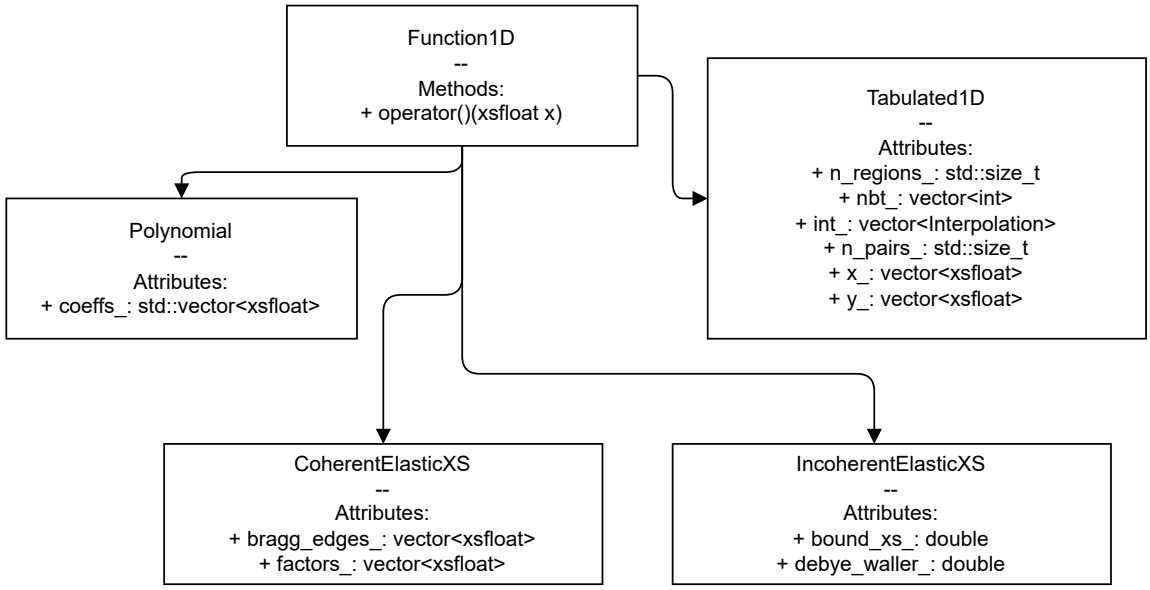


Figure 3.2: Inheritance diagram for ENDF function types in OpenMC.

`EnergyDistribution` base class. These memory accesses are likely to be entirely serialized on a block of threads as a result of sampling potentially different reaction types on each thread.

To expand on this theme, Figure 3.4 shows the inheritance diagram for correlated angle-energy distributions used by OpenMC. In both the `KalbachMann` and `IncoherentInelasticAE` classes, we can see that an array of arrays is required. This adds more indirection to the memory accesses here, particularly if searches over multiple sub-tables are required, as interpolation over the outer variable requires.

To our knowledge, no work has quantified the accuracy impact of neglecting complex double-differential distributions and instead modeling them as more simple distributions or resorting to a purely numerical approximation approach. For collision processing, it may confer a substantial performance benefit to model some cases like n-body phase space distributions or Kalbach-Mann distributions with a simple, general format. Perhaps direct tabulations of the joint energy-angle distribution would yield acceptable accuracy in reactor computations while substantially reducing the complexity and indirection of the code.

3.1.3 Other reasons for the choice of CUDA

Aside from its support for polymorphism that eases the job of translating the code, we chose CUDA over other frameworks for a few more reasons. Best practices guidance for scientific software [53] suggests using the highest level possible language. If we followed this strictly,

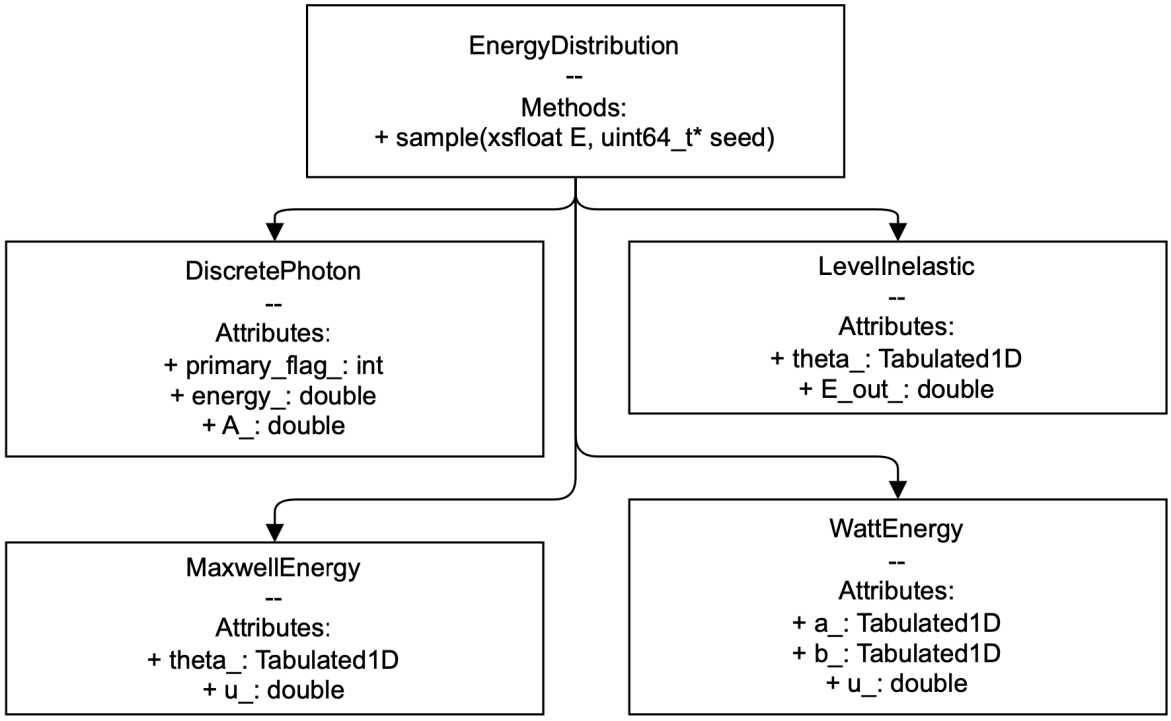


Figure 3.3: Inheritance diagram for energy distributions in OpenMC.

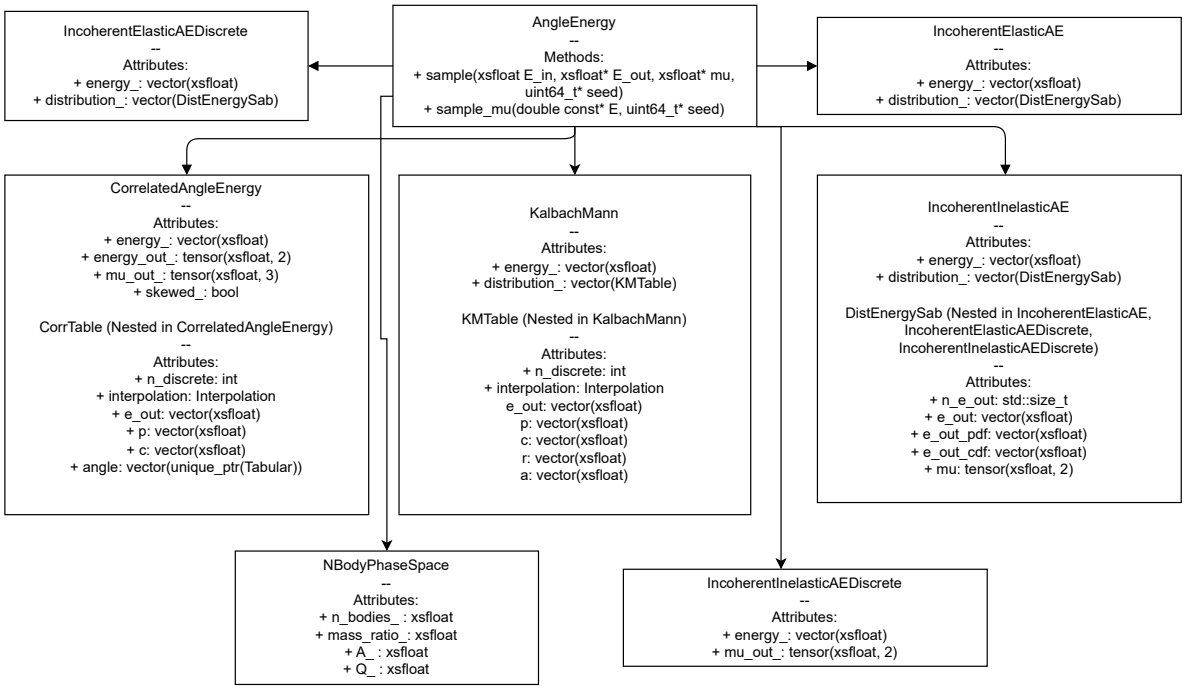


Figure 3.4: Inheritance diagram for correlated angle-energy distributions in OpenMC.

we would have used a performance portability framework like RAJA, Kokkos, Occa, etc. However, the debugging and profiling capabilities of portability frameworks tend to be less mature, and come with a smaller online user community. A few more reasons for choosing CUDA include:

- Extensive, frequently updated documentation
- Conceptually straightforward to convert to AMD HIP language (thus covering most modern GPUs)
- Straightforwardly allows architecture-specific optimizations
- Diverse set of users leads to low compiler bug frequency

We include the last point because as recently as CUDA 10, certain template expressions used by an external library in OpenMC caused the compiler to fail. We suspect that if the CUDA compiler is failing here, any performance portability framework using a CUDA backend would also fail, and this would make finding a workaround more difficult.

While some applications may have a large codebase with the majority of the runtime taken up by smaller computationally intensive kernels, continuous energy neutron transport applications can be characterized as requiring large amounts of code in their kernels. While cross-section lookups do tend to constitute a large fraction of run time, neglecting to write efficient code in collisions or geometry treatment can severely bottleneck performance as we show definitively in a later section. Consequently, the ability to generate device code at link time is essential to the organization of our application.

This violation of sustainable software principles, however, serves a few purposes. Firstly, given the complexity of a continuous energy MC neutron transport code, a well-developed debugging tool will be required. As [53] sets forth, sustainable code should be designed to be debugged. CUDA includes robust debugging, memory checking, and profiling tools used by a diverse online community. If using a general performance portability framework, the programmer is insulated from the more hardware-specific code that will be easier to debug and reason about. Another reason for our choice of CUDA is that translation to AMD's HIP language should be more or less straightforward, and that would extend our code's applicability to most contemporary GPUs.

Two more important reasons for choosing CUDA relate to the nature of the problem being solved. The sustainable scientific software paper directs its attention primarily to scientists writing software to solve new problems. Our problem, though, concerns solving a well-researched problem on a new compute architecture. We may pursue architecture-specific

optimizations in the future, so choosing a high level performance portability framework would be inappropriate. The second important reason is that even CUDA compilers as recent as version 10 failed in some parts of the nearly 50 000 line C++ codebase due to the use of some template libraries in OpenMC. If the CUDA compiler itself could not handle this, portability frameworks would surely not as they often resort to generating CUDA code. Using only the CUDA compiler removes a point of failure in compiling our moderately sized codebase. Using CUDA to compile a GPU version of OpenMC, however, does not work without modification.

Despite the conveniences of CUDA, plenty of challenges remained to bring OpenMC to the GPU. As Salmon [81] pointed out, the C++ standard template library and polymorphism [2] are not obviously supported in CUDA device code. OpenMC extensively uses both. Polymorphic classes can be used on the GPU though, so long as the CUDA object is constructed on the device. Although polymorphism can result in divergent code, adapting it for the device stands as the path of least resistance towards our goal of porting OpenMC to GPU. We use the following technique to support polymorphism on both the CPU and GPU.

3.1.4 Adapting Polymorphic Code for GPUs

In the context of object oriented programming, polymorphism refers to situations where a base class such as `Particle` defines some function like `collide`, which may change behavior in derived classes or not even be defined in the base class. For instance the classes `Photon` and `Neutron` may inherit from `Particle` and change the implementation of the class method `collide`. OpenMC's geometry and nuclear data implementation make extensive use of polymorphism, which fits quite naturally into continuous energy MC codes.

Polymorphic objects in C++ tend to be implemented by the use of a "vtable", or virtual table. This table contains pointers to functions which are implementations for different behaviors. Because this table differs on CPU code than GPU code, one cannot simply copy polymorphic objects to the GPU and expect them to work. The vtable gets set up when a constructor for an object is called. Consequently, the object must be constructed again on the GPU after it has been copied. Alternatively, if the object is in unified memory [23], it may automatically be copied to the GPU or CPU side as needed. This implies that in order to maintain functionality of an object between CPU and GPU throughout the runtime of OpenMC without manually reconstructing objects, one must store separate copies of a polymorphic object on both host and device, and unified memory cannot be used for this scenario.

To refresh the nuclear engineer unfamiliar with polymorphic objects, the following code

[2] There is a typo in Salmon's paper here saying "inheritance" rather than "polymorphism".

listing illustrates a quintessential example. Two classes, `Cat` and `Dog` inherit from a base class called `Animal`. The `Animal` class defines a *virtual* method called `make_noise`. This eases programmer effort by uniting common functionality into base classes: for example in the Monte Carlo neutron transport context a `Reaction` base class could define a virtual method called `scatter` that operates on a particle, and stores pointwise cross-section data as a function of energy. A class called `Elastic` could inherit from it and define elastic scattering physics, and likewise for inelastic scattering and others.

```
vector<unique_ptr<Animal>> animals = {
  make_unique<Dog>(),
  make_unique<Cat>()};
for (animal: animals) {
  animal->make_noise();
}
```

Output:

```
Bark!
Meow
```

Notably, the variable `animals` stores an array of pointers to classes which are known to be subclasses of `Animal`. If pointers were not used, the array would store only `Animal` objects which may not be `Cat` or `Dog` type. By using pointers, objects guaranteed to have `Animal` functionality can be stored together, with each behaving as its own type of animal.

Almost all polymorphic objects in OpenMC are wrapped in C++ unique pointer (`std::unique_ptr`) objects. We implement our own unique pointer for CUDA OpenMC to support seamless polymorphism on both CPU and GPU, which we have included in the `cuda` branch of the repository github.com/openmc-dev/openmc. This must be used with a custom implementation of the `std::make_unique` template function. We use the `openmc` rather than `std` namespace. Our custom allocation function automatically toggles at compile time between replicated and unified memory using the "substitution failure is not an error" SFINAE behavior [111] depending on whether the underlying class is polymorphic.

A minimal example of a `unique_ptr` class is shown below:

```
template<typename T>
class unique_ptr {
private:
  T* ptr;
public:
```

```
  unique_ptr(T* take) : ptr(take) {}
  T* operator->() const {
    return ptr;
  }
  ~unique_ptr() {
    delete ptr;
  }
};
```

If we wish to have polymorphism seamlessly work on GPU, we can employ the below:

```
// Minimal implementation of unique_ptr
template<typename T>
class unique_ptr {
private:
  T* ptr;
  T* ptr_dev; // from cudaMalloc
public:
  __host__ __device__ T* operator->() const {
#ifdef __CUDA_ARCH__
    return ptr_dev;
#else
    return ptr;
#endif
  }
  ~unique_ptr() {
    delete ptr;
    cudaFree(ptr_dev);
  }
};
```

These pointers can be allocated automatically as well:

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
  if (std::is_polymorphic<T>::value)
    return unique_ptr<T>(replicated_new<T>(std::forward<Args>(
       args)...));
```

```
  else
    return unique_ptr<T>(unified_new<T>(std::forward<Args>(args
       )...));
}
```

The vtables on the device can be ensured to be initialized correctly by using this function for memory allocation:

```
template <typename T, typename... Args>
std::pair<T*, T*> replicated_new(Args... args)
{
  T* loc_host = nullptr;
  T* loc_device = nullptr;
  loc_host = static_cast<T*>(malloc(sizeof(T)));

  cudaMalloc(&loc_device, sizeof(T));
  new (loc_host) T(args...);

  cudaMemcpy(loc_device, loc_host, sizeof(T),
     cudaMemcpyHostToDevice);
  run_move_constructor_on_device<<<1,1>>>(loc_device);
  return {loc_host, loc_device};
}
```

The kernel called `run_move_constructor_on_device` takes a while to run, relatively, and launches a single thread on the GPU that runs a move constructor (described in [112]) in place on the object. Move constructors have the semantic value of taking ownership of another object, for example a unique pointer's move constructor removes the pointer held by another unique pointer rather than copying the underlying object. The effect of running a move constructor in place on the GPU usually has the effect of simply constructing the vtables needed for polymorphism to work. One possible implementation of this kernel is

```
// This sets up vtables in device memory
template<typename T>
__global__ void run_move_constructor_on_device(T* __restrict__
   dist)
{
  static_assert(std::is_move_constructible<T>::value,
```

```
    "Polymorphic objects to be put on device must be move
        constructible.");
  char buffer[sizeof(T) + alignof(T)];
  char* aligned_buffer =
    buffer + alignof(T) - reinterpret_cast<intptr_t>(buffer) %
        alignof(T);
  T* tmp = new (aligned_buffer) T(std::move(*dist));
  new (dist) T(std::move(*tmp));
}
```

We can see that the "placement new" operator is used to call the move constructor on place. The kernel also assures, at compile time, that the object in question is capable of being move constructed, which directs the developer to write the relevant move constructors for various polymorphic objects in their Monte Carlo code base as they work through adopting the code for GPUs.

3.1.5 A Few Useful Data Structures and Algorithms for GPUs

In a similar vein, the C++ standard template library will not work on the GPU. In order to address this, we have created custom implementations of a few common standard library containers such as `std::vector` and `std::string` which work as expected on both CPU and GPU. Of course, some intricacies come about from the fact that device code kernels cannot allocate global memory, so methods like `std::vector::push_back` have no obvious meaning on device. Rather, we have extended these containers to include optional thread-safe device operations. Similarly, OpenMC uses the `xtensor` github.com/xtensor-stack/xtensor library for Fortran-like multidimensional arrays. With GPU-compatible versions of these commonly used data structures in common continuous energy codes, developers outside those working on OpenMC can more easily bring their code to GPUs.

We do not necessarily believe that our present approach will provide optimal performance in GPU versions of OpenMC. We aim to show a simple path towards converting complex continuous energy particle transport codes into reasonably performant GPU-enabled codes with maximum code reuse and input/output compatibility.

Our first step to adapting standard library containers to be replaced by GPU-compatible versions was to implement allocators that determine the behavior of memory allocation. One approach to building data structures that use GPU memory straightforwardly is to first ensure that all data structures allocate using the `new` keyword. Secondly, because the `new` keyword can be overloaded, a class called `Unified` can be defined that overrides the

`new` operator with a `cudaMallocManaged` call. Lastly, any data structure which should be used on the GPU can inherit from the `Unified` class, thus automatically allocating and synchronizing to GPU memory where necessary.

Memory Allocator Classes and the Dual Pointer

A more flexible approach to the problem is to define memory allocator classes [112] that handle different potentially desirable memory behaviors on GPUs. Classes behaving analogously to standard library containers like the `vector` then allocate only through the allocator class that they are templated on. For example, an implementation of a unified memory allocator might include methods like:

```
class UnifiedAllocator {
  ...
  inline pointer allocate(size_type n)
  {
    pointer tmp;
    cudaError_t error_code = cudaMallocManaged(&tmp, n * sizeof
       (value_type));
    if (error_code != cudaSuccess)
      CubDebugExit(error_code);
    return tmp;
  }
  inline void deallocate(pointer p, size_type)
  {
    if (p == nullptr) return;
    cudaError_t error_code = cudaFree(p);
    // For the latter, this happens when running deallocate in
       destructors
    // in on of OpenMC's many global variables which destruct
       after main().
    // The CUDA RT API has already handled freeing the memory
       in that case,
    // so that error is safe to ignore.
    if (error_code != cudaSuccess && error_code !=
       cudaErrorCudartUnloading)
      CubDebugExit(error_code);
  }
```

```
};
```

Allocator classes come with a fair amount of other boilerplate code, so we direct the reader to [112] for more information about the requirements of an allocator.

Because unified memory can come with a slight performance overhead, another approach to allocation might be required. In this case, we define the `ReplicatedAllocator` which can allocate two pieces of memory of the same size each on the host and the device, with the user left to manually sychronize the code. Because allocators define the type of the pointer they return, a special pseudo-pointer class that stores both the host and device memory pointers is defined that automatically dereferences the correct choice of host or device pointer based on whether the code runs on the host or device. We named this the `DualPointer`, implemented as:

```
template<typename T>
class DualPointer {
private:
  T* host;
  T* device;

public:
  __host__ __device__ T& operator*()
  {
#ifdef __CUDA_ARCH__
    return *device;
#else
    return *host;
#endif
  }
  __host__ __device__ T const& operator[](unsigned int indx)
     const
  {
#ifdef __CUDA_ARCH__
    return device[indx];
#else
    return host[indx];
#endif
  }
```

```
  __host__ __device__ T& operator[](unsigned int indx)
  {
#ifdef __CUDA_ARCH__
    return device[indx];
#else
    return host[indx];
#endif
  }

  __host__ __device__ T* operator+(unsigned int offset)
  {
#ifdef __CUDA_ARCH__
    return device + offset;
#else
    return host + offset;
#endif
  }
  __host__ __device__ T const* operator+(unsigned int offset)
     const
  {
#ifdef __CUDA_ARCH__
    return device + offset;
#else
    return host + offset;
#endif
  }

  __host__ __device__ DualPointer() : host(nullptr), device(
     nullptr) {}

  // Copy a given pointer into each. This can be used with
     managed memory
  __host__ __device__ DualPointer(T* const& ptr) : host(ptr),
     device(ptr) {}
  __host__ __device__ DualPointer& operator=(T* const& ptr)
  {
```

```
    host = ptr;
    device = ptr;
    return *this;
  }

  // These should exclusively be used for cudaMemcpy calls
  __host__ T* host_pointer() { return host; }
  __host__ T* device_pointer() { return device; }

  DualPointer(DualPointer const&) = default;
  DualPointer& operator=(DualPointer const&) = default;
  DualPointer(DualPointer&&) = default;
  DualPointer& operator=(DualPointer&&) = default;
  ~DualPointer() = default;

  friend class ReplicatedAllocator<T, PinnedAllocationOnHost::
     no>;
  friend class ReplicatedAllocator<T, PinnedAllocationOnHost::
     yes>;
};
```

We can see that the `ReplicatedAllocator` is listed as a friend class, and as mentioned before is responsible for allocating the memory that the `DualPointer` holds. Again we point out that more boilerplate code is required to fully implement an allocator, so only the nontrivial details are shown below.

```
template<typename T,
  PinnedAllocationOnHost UsePinned = PinnedAllocationOnHost::no
     >
class ReplicatedAllocator {
public:
  inline pointer allocate(size_type n)
  {
    pointer tmp;

    if (UsePinned == PinnedAllocationOnHost::yes) {
      cudaError_t error_code =
```

```
      cudaMallocHost(&tmp.host, n * sizeof(value_type));
    if (error_code != cudaSuccess)
      CubDebugExit(error_code);
  } else {
    tmp.host = (T*)malloc(n * sizeof(value_type));
  }

  // Allocate on device
  cudaError_t error_code = cudaMalloc(&tmp.device, n * sizeof
     (value_type));
  if (error_code != cudaSuccess)
    CubDebugExit(error_code);

  return tmp;
}
inline void deallocate(pointer p, size_type)
{
  if (p.host) {
    if (UsePinned == PinnedAllocationOnHost::yes) {
      cudaError_t error_code = cudaFree(p.host);
      if (error_code != cudaSuccess && error_code !=
         cudaErrorCudartUnloading)
        CubDebugExit(error_code);
    } else {
      free(p.host);
    }
  }

  // See comment in UnifiedAllocator on this method
  if (p.device) {
    cudaError_t error_code = cudaFree(p.device);
    if (error_code != cudaSuccess && error_code !=
       cudaErrorCudartUnloading)
      CubDebugExit(error_code);
  }
}
```

```
};
```

The `ReplicatedAllocator` is templated on an execution policy [113] that determines whether host memory is pinned or not. As the book [108] discusses, the `cudaMallocHost` allocates pinned CPU memory. This memory disables the usual memory paging mechanism to keep this memory paged in at all times, which can enhance the data transfer rate between the CPU and the GPU.

With the `UnifiedAllocator` and configurable `ReplicatedAllocator`, data structures in a continuous energy MC code can be parameterized to behave differently depending on the underlying choice of allocator. This functionality is employed in the next section to quickly experiment with a few different means of synchronizing microscopic cross-sections between the GPU and CPU when attempting to offload cross-section lookups to the GPU.

A vector class for GPUs

On top of using parameterizable allocators, independent implementations of nearly standards-conforming data containers should include some additional functionality to facilitate reasonable performance on GPUs. For example, the `push_back` method on `std::vector` does not have a clear interpretation on a GPU, as the behavior here with multiple threads is ambiguous. Should an atomic operation be used to increment the length of the vector? Should additional memory be allocated, a feature possibly not supported in a given GPU programming model?

This ambiguity calls for some specialized routines. In our implementation of `openmc::vector`, we assume that no other threads are attempting to modify the data when calling `push_back`. Because modification from many threads is a common scenario, we implement a separate `thread_safe_append` that uses the CUDA `atomicInc` function to safely increment the index of the end of the array. In both cases, we assume that enough memory has already been allocated from the host using the `reserve` method. If an attempt is made to write beyond the end of the array, the `__trap()` instruction is called to cancel code execution on the GPU and provide an entry point for debugging.

If separate GPU and CPU memory are being maintained by one data structure by use of the aforementioned `DualPointer` type, the user is left to manually synchronize data in either the CPU to GPU direction or vice-versa. We use the SFINAE [111] (substitution failure is not an error) technique to toggle on methods called `syncToDevice` and `syncToHost` that manually synchronize the data in either direction.

Moreover, as will be discussed in the coming section on code optimizations for GPU MC codes, having all threads attempt to simultaneously atomically push back into a vector or

other data structure might result in poor performance due to the contention of the memory transaction. If a collective communication is first carried out on the thread block or warp to decide which threads should write into which indices (typically achieved with a parallel scan operation), the data structure's known end-of-array pointer should be updated manually after the threads decide where to write. Consequently, we provide a method to manually update the end-of-array pointer, `updateIndex`, which must be called to ensure code correctness after a coordinated, collective update like this is made.

An array class for GPUs

In addition to a `std::vector` replacement, we implemented a GPU-compatible `std::array`. This data structure has a fixed size, and requires no memory allocation ever in a reasonable implementation of it. Consequently, the GPU-compatible implementation for a data structure like this looks identical to a typical CPU implementation but with the `__host__ __device__` qualifier on all methods that might run on the GPU. Our implementation of a tensor class similarly required no on-the-fly memory operations, as in a continuous-energy Monte Carlo code the tensors that occur are read-only and relate to nuclear data. However, the tensor class did have one nontrivial aspect to it.

A tensor class for GPUs

We implemented a compile-time loop unfolding technique in our GPU-compatible tensor class. This was accomplished with recursive templates, and allows automatic unrolling of the loop over tensor indices to compute the underlying linear array index. Because the tensor rank is known at compile time, this loop can be fully unfolded without relying on the compiler potentially not unrolling it with template metaprogramming.

The tensor element access operator is defined as:

```
template<typename... Args>
__host__ __device__ T& operator()(Args... args)
{
  constexpr size_type one = 1;
  return begin_[bracket_indx<Rank - 1, Rank>::calc(
    one, std::forward_as_tuple(std::forward<Args>(args)...),
       size_)];
}
```

Where the `bracket_indx` class is defined as:

```
template<int Idx, int Rank>
struct bracket_indx {
  using tuple_type =
    typename expander<size_type, std::make_index_sequence<Rank
       >>::type;
  __host__ __device__ static size_type calc(
    size_type coeff, tuple_type indices, const size_type* size)
  {
    return coeff * std::get<Idx>(indices) +
           bracket_indx<Idx - 1, Rank>::calc(coeff * size[Idx],
               indices, size);
  }
};
template<int Rank>
struct bracket_indx<0, Rank> { // Specialize on base case to
   not recurse
  using tuple_type =
    typename expander<size_type, std::make_index_sequence<Rank
       >>::type;
  __host__ __device__ static size_type calc(
    size_type coeff, tuple_type indices, const size_type* size)
  {
    return coeff * std::get<0>(indices);
  }
};
```

For example, for a 3-tensor of double-precision numbers, denoted in our code as `tensor<double, 3>`, the indexing operation for a variable named `salphabeta` would be `salphabeta(3, 4, 2)` and expands to the generalization of row-major form, `size(1)*size(2)*3 + size(1)*4 + 2`.

A string class for GPUs

Lastly, many classes in OpenMC use `std::string` data to store text such as their name. In order to allow strings to be automatically copied to the GPU, we implemented a string-like class without any optimization that stores character data in the GPU-compatible vector

that automatically uses unified memory described above. A few interfaces that provide interoperability with the standard library string-related were implemented where necessary; details can be found in the file `include/openmc/string.h`.

With these data structures in hand that ease the handling of complex data for GPU-based continuous energy Monte Carlo transport simulations, we proceeded to explore offloading the cross-section lookup operation to GPUs.

3.2 Offloading Cross-Section Lookup

With the means of adapting continuous energy MC codes for GPU laid out, we can explore the first steps to bringing a MC code to GPU. Our first work in attempting GPU-accelerated continuous energy Monte Carlo neutron transport simulations focused on accelerating the cross-section lookup operation only. For most problems of interest in reactor analysis, microscopic cross-sections for each nuclide must be computed. For depletion calculations, the reaction rates in the regions with large numbers of nuclides must be computed, so we assume that the full set of microscopic cross-sections present in a given material has to be looked up on the GPU, then transferred back to the host.

Some other work has been carried out focusing on exclusively [114] offloading the cross-section lookup operation to the GPU. The results showed a speedup factor around one; in other words neither hurting or helping compared to carrying out all operations on the CPU.

3.2.1 Different XS-Lookup Offload Methods

Table 3.1 shows the different approaches to synchronizing microscopic cross-section cache values. In case A, unified memory is used to automatically handle synchronization between the host and GPU. In case B, a replicated memory approach is employed by simply changing the allocator on each particle microscopic cross-section cache to be a `ReplicatedAllocator` rather than a `UnifiedAllocator`. Case C aligns each particle's microscopic cache to take a contiguous block of memory, so only one `cudaMemcpy` call is made rather than many. This increases the effective bandwidth by decreasing latency effects, illustrated in 3.5. Case D pinned the memory pages to effectively increase the bandwidth by reducing latency, and Case E used what we deemed the `transfer queue` in which the event-mode particle array's microscopic cross-section caches were collapsed to a secondary array of reduced size which contiguously held only particles' nuclide-wise microscopic cross-section caches that required a lookup operation. Approach E therefore reduces the amount of useless data transferred to zero (e.g. nuclides in the contiguous cache that did not require a lookup), but requires some

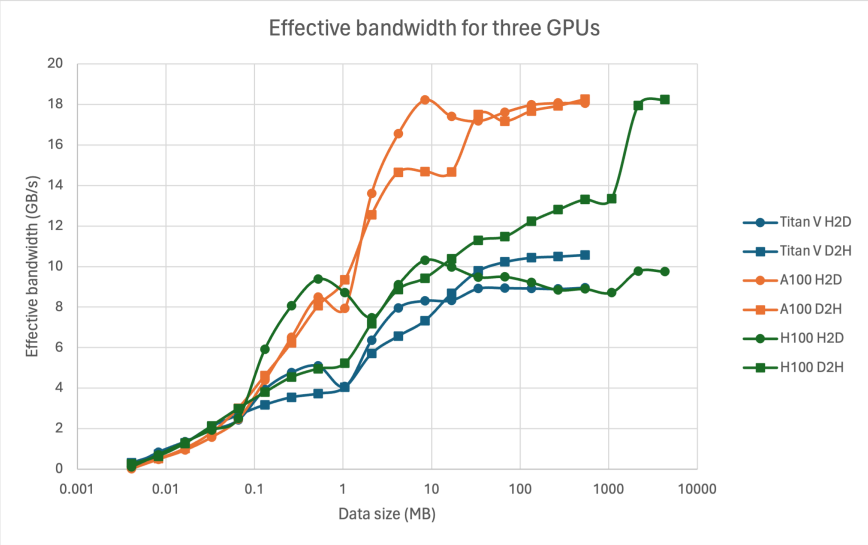


Figure 3.5: Effective bandwidth vs. data transfer size for three GPUs. The Titan V and H100 are connected over PCIE, whereas the A100 is connected over SXM4. The data was measured by sending arrays to and from the GPU; "H2D" means "host to device" and vice-versa.

additional memory copying operations that are not across the CPU/GPU interface (PCIE or SXM).

Fig. 3.6 shows the results for each of these techniques. The different methods A-E are listed by Table 3.1. These cases have to transfer microscopic cross-section data in between the CPU and GPU. In OpenMC, microscopic cross-section cache entries for each nuclide include the total, absorption, fission, fission production, elastic, thermal inelastic, thermal elastic, photon production, energy grid index, nuclide temperature grid index, energy interpolation factor (the fraction of contribution from the right bounding point on the energy grid), whether probability tables were applied to the cross-sections, the energy corresponding to the cache, and the temperature. When the cross-section of a nuclide is requested, data is pulled from the cache entry if the particle's current energy and temperature match the cache's values. Otherwise, all the microscopic cross-sections are recalculated based on a grid lookup or potentially windowed multipole calculation depending on the calculation type and particle energy.

The total size of the microscopic cross-section cache entries for each nuclide is 161 bytes, including all six depletion reactions. For a problem with 250 nuclides (akin to a depleting reactor problem), that means that each particle uses about 40 kB of space for the whole array of microscopic cross-section data. Our measurements shown in 3.5 show that sending individual particle cache arrays results in deeply sub-optimal effective bandwidth.

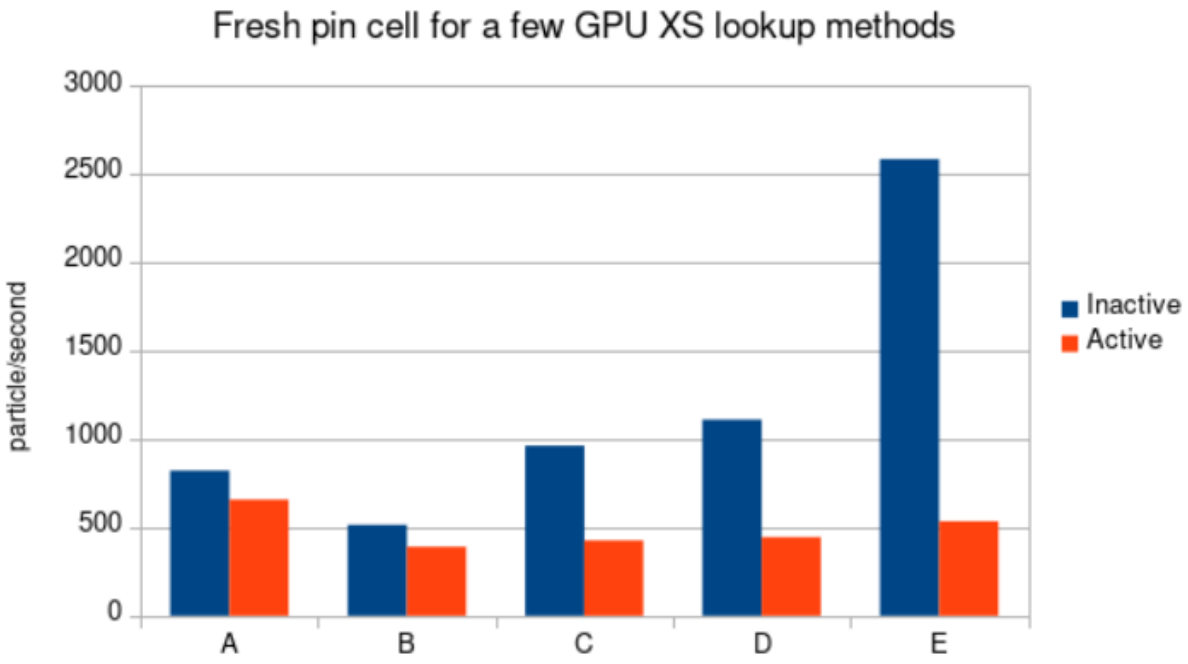


Figure 3.6: Results of offloading the microscopic cross-section lookup operation to the GPU. The speedups are listed relative to one CPU core. We can observe only modest speed gains which do not serve as a compelling basis for using the XS-lookup-only approach to GPU acceleration.

Table 3.1: Cross-section lookup offload synchronization approaches

A	Unified Memory
B	Replicated Memory
C	Replicated, Contiguous Memory
D	Replicated, Pinned, Contiguous Memory
E	Replicated, Pinned, Contiguous Memory w/ transfer queue

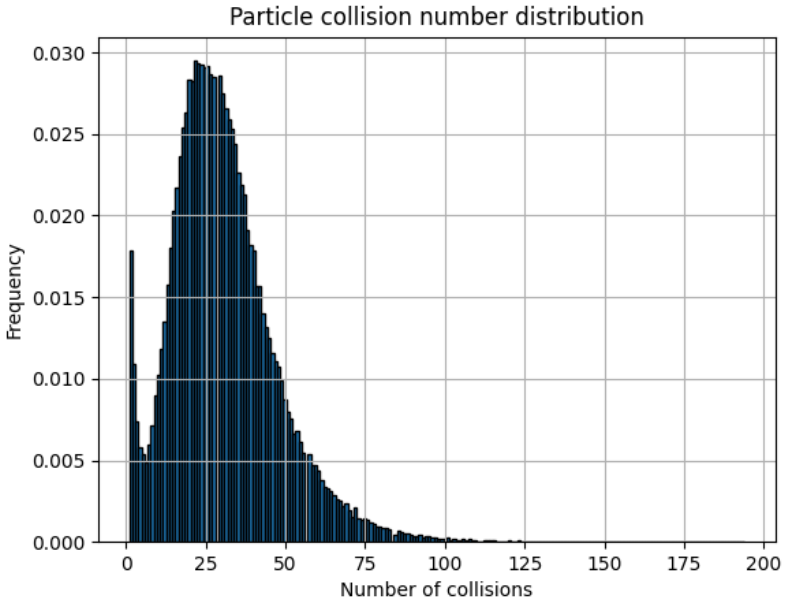


Figure 3.7: The distribution of the number of collisions each particle makes in a typical PWR problem.

3.2.2 The upper bound of offloaded lookup performance

Here, we demonstrate the upper bound of particle processing in a code that offloads cross-section lookups only to the GPU. The upper bound of the tracking rate for a given problem can be estimated by assuming all events to execute instantaneously; the only limiting factor is transferring cross- section cache arrays between the CPU and GPU. The first step here is to collect the distribution of the number of times a particle must have a new set of cross-sections looked up. Assuming that the particle crosses every material in the problem at each new energy, the full set of roughly 250 nuclides must be looked up for each new energy. Again, the problem here represents a typical depleted reactor. Figure 3.7 shows the distribution of the number of collisions a particle for a typical pressurized water reactor unit cell problem makes.

If we assume that fifty collisions per particle adequately represents the long work tail that

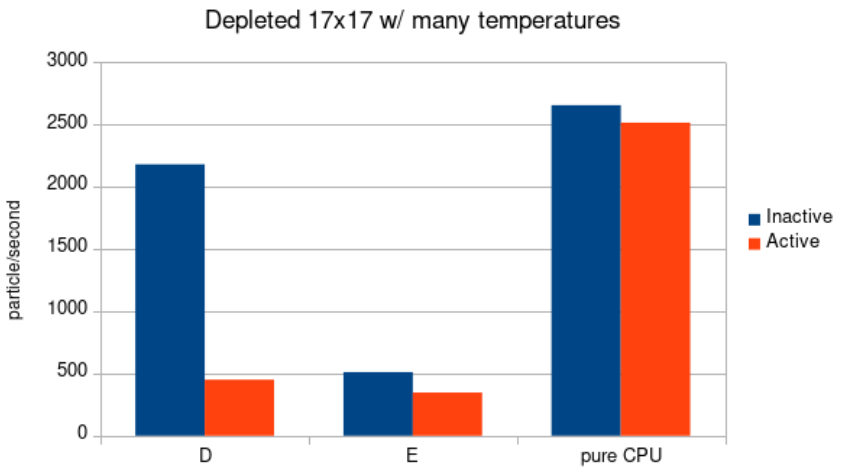


Figure 3.8: The particle processing rate for cross-section offloading relative to CPU performance.

can hamper GPU performance, we find that each particle requires about two megabytes of data to be transferred from the GPU over the course of its life. We can now relate this with GPU bandwidth from Fig. 3.5. If the particle microscopic cross-section cache arrays are transferred individually, a 0.04 MB transfer, we can see that even the most high performance GPUs end up with a 2 GB/s effective bandwidth. This implies that **at most one thousand particles per second** can be handled by the GPU, regardless of how fast the other parts of the code are! If the cache data is amalgamated into a large array, with an infinite number of particles, we can reasonably assume a 10 GB/s effective bandwidth, equating to **at most five thousand particles per second**.

These numbers are in line with the predictions of Fig. 3.6. Now the question stands: how useful is this relative to the tracking rate a typical CPU can attain? We ran a depleted fuel assembly in OpenMC to answer this question, and plotted the result of techniques D and E relative to CPU performance in Fig. 3.8. In line with the results of [114], we see no performance gain relative to pure CPU execution. With the remarkable results obtained by [31] by fully offloading transport in mind, we recommend researchers abandon offloading cross-section lookups only and instead focus on porting the entire transport and tallying process to GPU. PRAGMA can hit tracking rates around one-hundred times as high by fully offloading transport to the GPU.

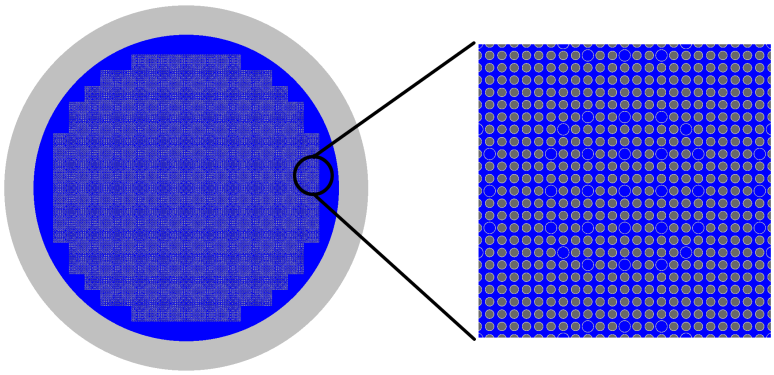


Figure 3.9: The Hoogenboom-Martin benchmark geometry, representing a large PWR.

3.3 Particle Tracking Rate Optimizations

Without a level of performance sufficiently exceeding normal CPU execution, a GPU-based Monte Carlo code is useless. After our initial development of this CUDA-based version of OpenMC, we pursued a variety of changes both of our own design and previously explored by others. Our hope with this section is to guide future developers of GPU MC applications to pursuing the most effective code changes while not wasting time on techniques we found to be marginally or completely fruitless. Specifically, because these tests were carried out on the most recent H100 GPU from Nvidia, the results presented presented here highlight the most promising optimization techniques on modern GPU architectures.

In this section, we quantify the effect of a variety of modifications to our code by testing on the Hoogenboom-Martin benchmark [115], illustrated in Fig. 3.9. The OpenMC input for this problem was prepared and extended by John Tramm and can be found online [116]. His extension includes 255 nuclides in the fuel to mimic the main computational expense in simulating a depleting nuclear reactor–the high computational expense of many cross-section lookups.

Unless noted otherwise, all results presented below were obtained on an NVIDIA H100. The code was compiled using version 12.2 of `nvcc`, all-level caching was used with the `-dlcm=ca` flag, `-O2` level optimization, and double-precision floating point numbers. When results are presented as a function of the number of particles in flight, the total number of particles per generation was chosen as double the maximum number in flight and the in-flight array was continuously refilled per one-hundred events.

In order to measure the effect of each of the below optimizations to the particle tracking rate, we compare a fully optimized baseline version of the code to a version in which a given

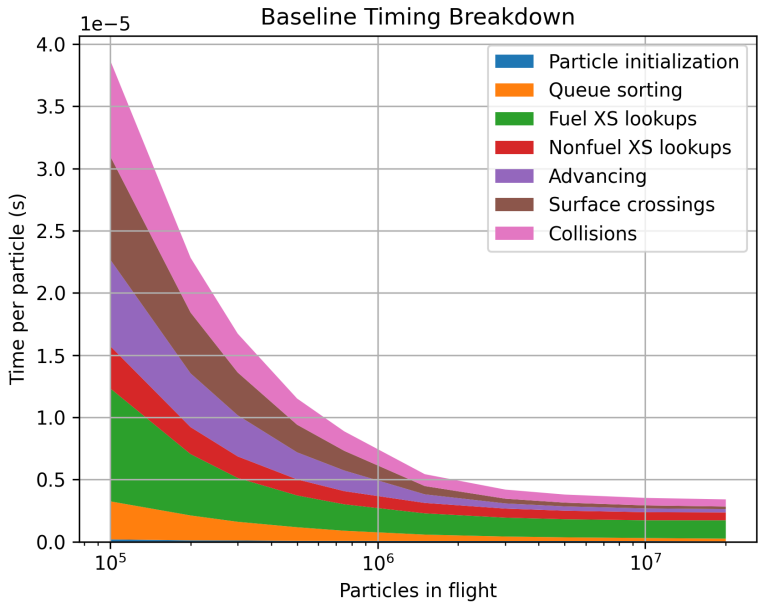


Figure 3.10: The baseline optimized code's time in each type of kernel per source particle on the H100 GPU.

optimization has been removed. The extent that performance falls in each case measures the efficacy of each optimization. We directly compare the overall particle processing rate for each of the below cases to the baseline case. In contrast, the detailed timing breakdown is not presented for direct comparison. To start with, we present the total time spent in each type of CUDA kernel per source particle in Fig. 3.10.

3.3.1 Fast Push Back to the Event Queues

The problem of appending to event queues on GPU is introduced in [28]. We adopted Shift's approach of using separate cross-section lookup event queues for fuel and non-fuel material as a result of the disparate amount of computational work to be done in each material type. This leaves collision, surface crossing, and particle flight as the other main CUDA kernels that comprise the event MC algorithm. In the queue-based event MC algorithm, one allocates an array of particles, and tracks which event should be executed next by queues. There is one queue for each event type, and these queues store indices of particles which need to be operated on. In the case of the cross-section lookup queue, one always knows that all particles should should then be placed into the flight queue. As such, a simple `cudaMemcpyAsync` call for moving the cross-section lookup queue data to the advance queue suffices.

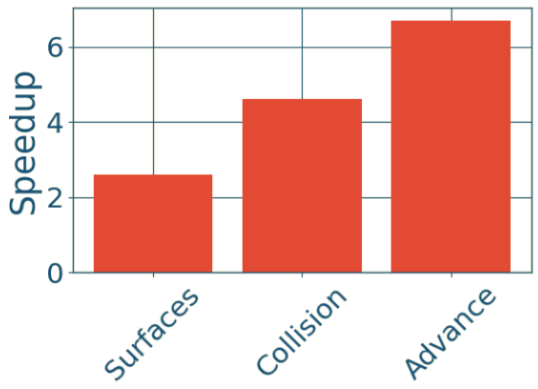


Figure 3.11: Performance gains from using parallel compaction algorithm on particle indices on a fresh PWR pin cell problem.

In contrast, for the other events, such as a collision, it is unknown *a priori* which queue a particle will be transferred to. If a collision happens in the fuel, that particle index must go to the cross-section lookup queue in the fuel, and similarly for non-fuel. The Shift team used an atomic increment (`atomicInc`) function from each thread on the queue index, and use the newly returned value to determine which index in the queue a particle index should be written to. Because many threads on the device call this function at once, an atomic operation can be costly.

In our alternative approach, we propose to assign an integer value of either one or zero to particles which respectively are in the fuel or non-fuel. By performing a block-wide exclusive parallel scan on these values, we can precompute relative indices into the queue, then compute the absolute index with an `atomicAdd` call from the last thread in the block. This reduces the number of atomic operations by two orders of magnitude, and the performance gains in each kernel are substantial. The surface crossing, collision, and advance kernels respectively ran 2.2, 4.2, and 6.6 times faster compared to the original GPU kernels using atomic operations on a fresh PWR pin cell on an NVIDIA Titan V GPU.

Eventually, we pivoted to another more simple and performant approach that accomplishes the same task of reducing contention from an atomic operation. The colloquially named `atomicAggInc` function performs an atomic increment by one of an integer, but does so using collective communication across a warp rather than the block through shared memory [117]. Searching the internet for an implementation of `atomicAggInc` will easily yield an example implementation of it, and we recommend using this approach for appending to the event MC queues.

This optimization has a nearly negligible effect on depleted reactor problems because cross section lookups vastly dominate the computational expense compared to thread contention

over updating the queue index. It also plays an important role when a relatively small number of particles are in flight. We especially recommend this optimization for simulations which do not include hundreds of nuclides per material.

3.3.2 Single precision nuclear data

The second optimization we attempted was using single precision nuclear data rather than 64 bit floats as used by default in OpenMC. Because the uncertainties associated with nuclear data are much higher than the error induced by using 32 bit floats, this change to any GPU MC code is desirable, as we found our tracking rate to rise by about 7% after introducing this change in a depleted PWR fuel pin problem at full saturation. This can be attributed to an increase in the cache hit rate.

Figure 3.12 shows the particle processing rate on the H100 as a function of the number of particles in flight. One data point in the single-precision curve is glaringly missing; this was the result of a chance illegal memory access that rarely happens in single-precision mode calculations. The code was not built for numerical robustness in mind.

Two primary changes were required to enable single-precision nuclear data. Firstly, in the loop over nuclides to sample the collision nuclide, the cumulative probability of each potential collision nuclide is incremented over the course of the loop. After exceeding some uniform random number, that collision nuclide is chosen. Due to roundoff, it is possible for this loop to fail to terminate as a result. One approach to fix this would be defaulting to the last nuclide in the loop when this happens, but instead we used the speculative collision nuclide sampling technique described in Subsection 3.3.9. The speculative collision nuclide sampling technique is far more robust against roundoff errors because it does not rely on computing a cumulative probability.

Secondly, we found that illegal memory accesses also frequently occurred when, during collision processing, a non-elastic reaction had to be sampled. Again, a cumulative probability is accumulated in a loop over the various reaction constituents to the non-elastic cross-section to determine the type of non-elastic collision. The loop would cause an illegal memory access in the event of failure, so we added code to check for out-of-bounds access and fall back to elastic collision sampling in this case. Still, some other rare numerical problem occurred that crashed the missing point in Fig. 3.12. Figure 3.13 shows the kernel timing breakdown per particle for this mode versus the number of particles in flight.

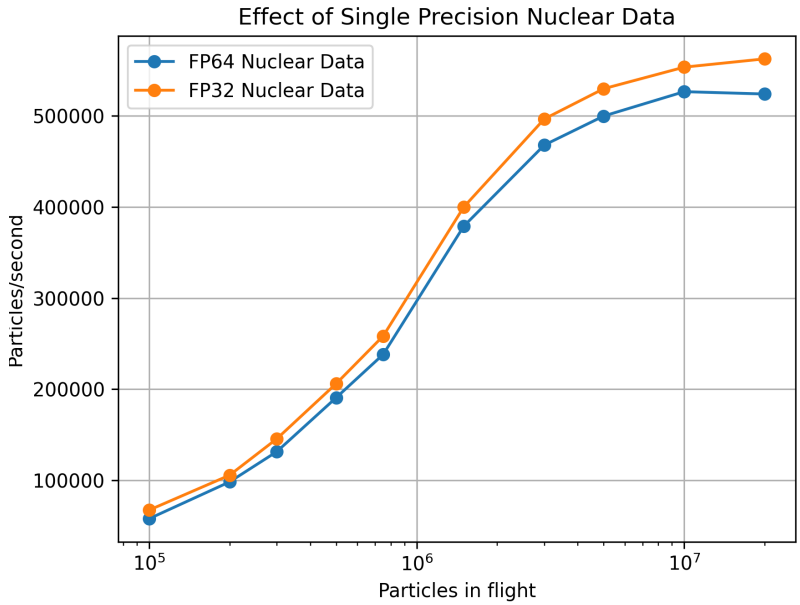


Figure 3.12: The particle tracking rate for the baseline calculation using double-precision nuclear data and a modified version using single-precision data.

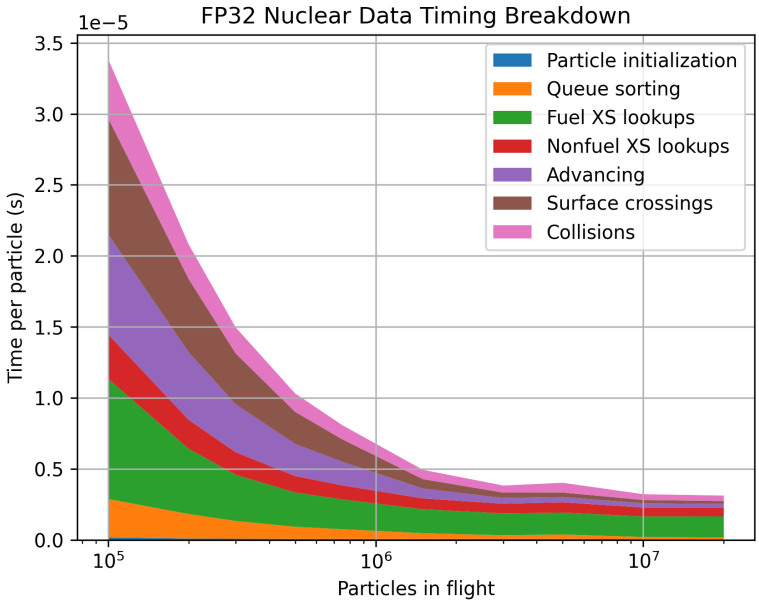


Figure 3.13: The single precision nuclear data code's time in each type of kernel per source particle on the H100 GPU.

3.3.3 Struct of Arrays

The Shift team has explored struct-of-array versus array-of-struct layouts of particle data. We designed our particle class to inherit from a base class that defines particle data layouts in memory and provides accessor methods to them. This allows us to seamlessly switch between a struct-of-array and array-of-struct particle data layout. We found that struct-of-array yields about a XXX% performance improvement in a depleted fuel pin problem.

We present a novel method for easily switching between struct-of-array and array-of-struct approaches within an event MC code. The aim here is to facilitate the sharing of a codebase that can compile for efficient history-based tracking on CPUs or event-based tracking on GPUs. When compiled in GPU mode, the code would switch to struct-of-array mode and vice-versa. The key is to separate responsibilities of representing and accessing data into two separate classes.

The `Particle` class that implements physical operations should only express the underlying data in terms of getter functions: for example using `particle.r()` rather than `particle.r` to denote the particles position in space. The `r()` accessor method should be provided by a base class that is chosen at compile time depending on whether struct-of-array or array-of-struct layouts are desired.

For the latter approach, we implement a class called `ParticleData` that simply stores the relevant particle data and provides accessor functions like `r()`. More interestingly, for struct-of-array mode, we implement a class called `ParticleHandle` which implements the same accessor methods, but instead caches a particle index as well in the class. Methods like `r()` then return values from global arrays that store all particles' positions, for example, based on the cached internal index. By doing so, the `Particle` class is reduced from storing all the particle data to being a simple mask over an integer that provides an interface to the particle data arrays. All physics code, therefore, can be re-used for both struct-of-array mode and normal history MC mode. We direct the reader to the files `include/openmc/particle.h`, `include/openmc/particle_data.h`, and `include/openmc/soa_particle.h` for more details. Figure 3.14 illustrates a simplified picture of the difference in data layout, with lines between boxes indicating their organization as an array.

Figure 3.15 reports the performance difference between the baseline code which uses a structure of arrays to represent particles, whereas the "AOS mode" curve represents an array of particles, similar to what [54] employed. Figure 3.16 shows the impact of AOS mode on each of the kernel timings. Because this configuration of the code results in an entirely different access pattern within every CUDA kernel, comparison against Figure **??** is especially interesting.

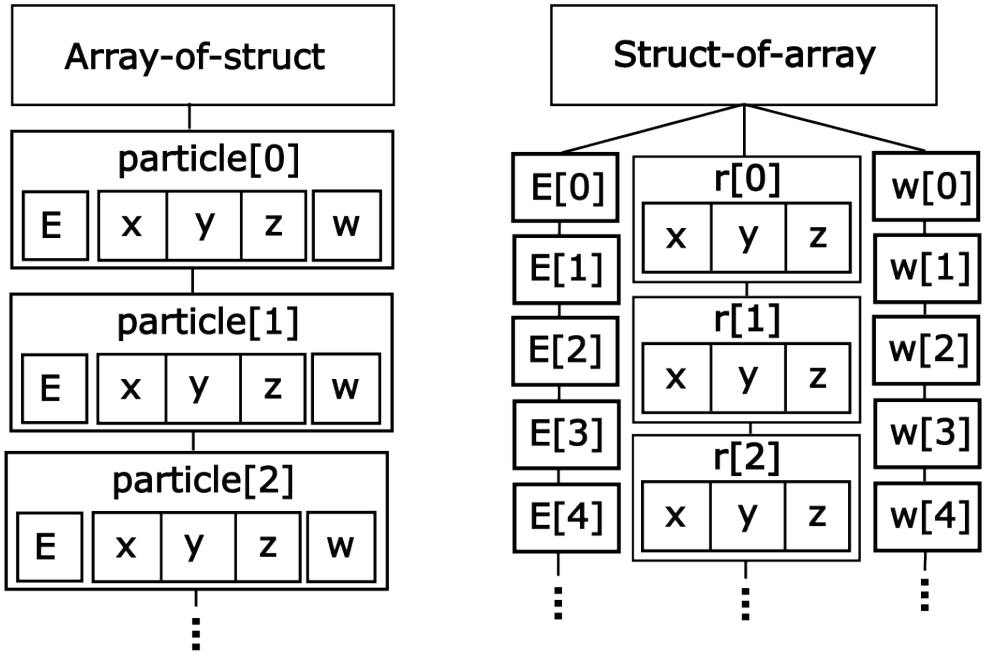


Figure 3.14: The difference between SOA and AOS data layouts visualized for a particle data type with energy, position, and weight items. Notably, the SOA layout may be a struct of arrays of structs, as is shown for the position variable on the right. Whichever layout is best for coalesced memory access should be chosen.

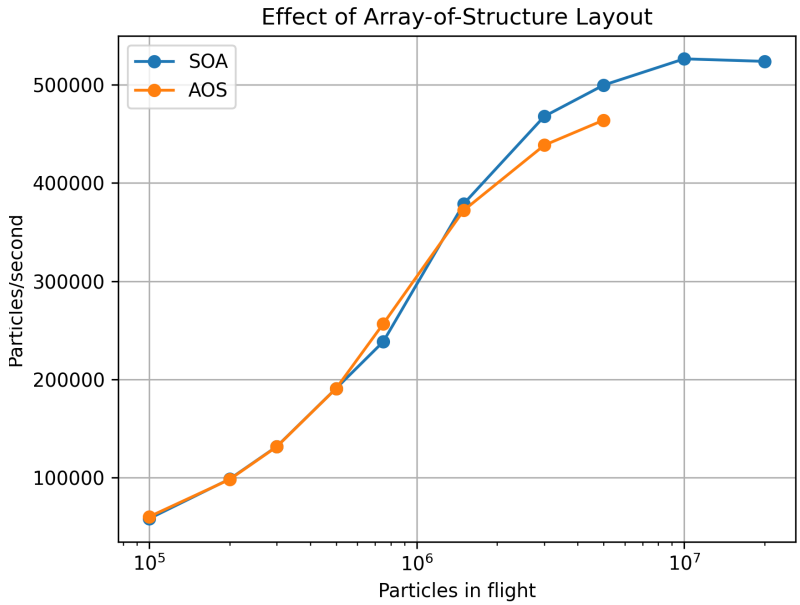


Figure 3.15: The particle tracking rate for the baseline calculation using a structure of arrays to represent particle data versus a modified version of the code using an array of structures.

Overall, we can see that the conventional wisdom about the superiority of structures of arrays on GPUs holds true. The last few missing data points were due to an overly conservative assumption on the maximum number of secondary particles running the code out of memory. Regardless, extrapolating the curve yields the same conclusion and an approximate 10% slowdown incurred by using an array-of-structures data layout.

3.3.4 Linear search beats binary

We use a linear search combined with a hash-based cross-section lookup as suggested by the Shift team [28]. This improved performance by about 4% compared to the binary search (after hashing narrows the window) approach, but only in the limit of large numbers of particles in flight. Figure 3.17 shows the particle processing rate for binary searches on the cross-section grid versus the baseline linear search method. Overall, we can observe that the move from binary to linear search introduces a quite small performance gain compared to binary searches, but considering its simplicity is surely a worthwhile development effort. Lastly, superior hashing techniques exist for tailored for GPUs which provide a constant amount of search work to do for each thread [29] have been developed. The relative merits of binary search and linear search may be different in that case compared to the logarithmic hashing technique [70].

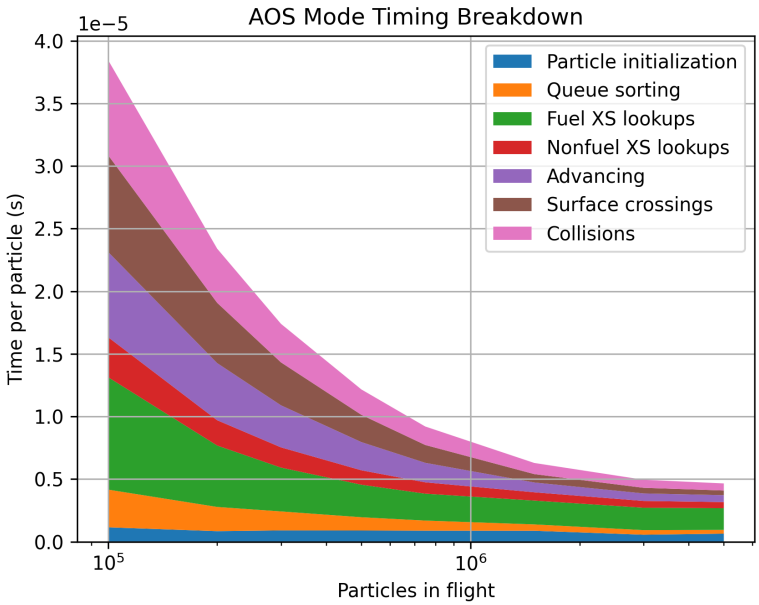


Figure 3.16: The array-of-structures code's time in each type of kernel per source particle on the H100 GPU.

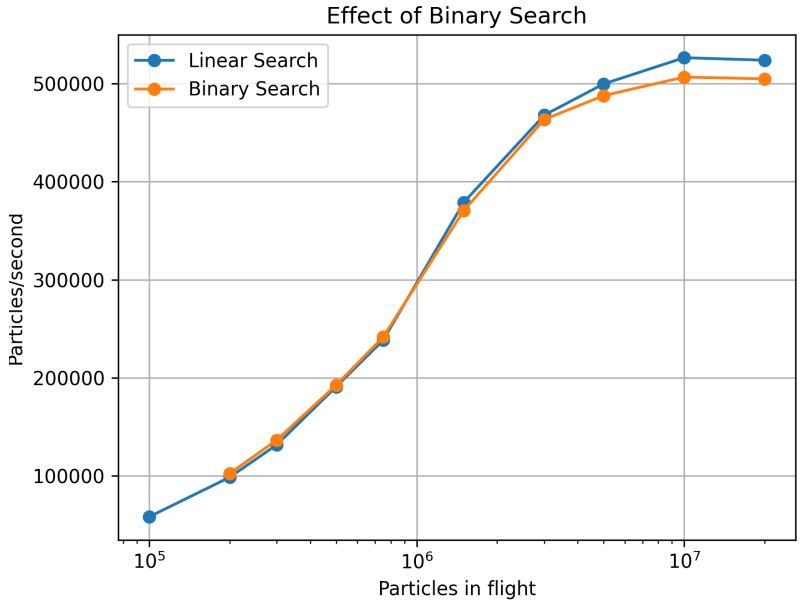


Figure 3.17: The particle tracking rate for the baseline linear search calculation versus binary search for cross-sections on the H100.

3.3.5 Continuous Particle Refill

The last optimization was the continuous refill of in-flight particles from the fission bank as described in the paper [28]. Although the original intent for this technique was for using more particles per generation than will simultaneously fit in GPU memory, we found this can improve the tracking rate by nearly 7%. We suspect this results from dead particle slots filling a smaller fraction of the particle data arrays, which results in more coalesced accesses to particle data.

To implement this, we added an extra event MC operation that handles refilling of the particle array array into locations where particles' lives have completed. First, a kernel scans for locations of particles in the particle array where particles have completed their lives. These free positions are written to a temporary array. A second kernel then handles filling the positions either from the fission bank or sampling the fixed source, which also searches for the current cell containing the particles.

This refilling operation should not be carried out after every event. Instead, we present an extra option which is how frequently refilling should be carried out in terms of the number of events executed. Figure 3.18 shows the performance impact of periodic particle refilling. We direct the reader to the code called within the `process_refill_events` function in our CUDA code. A parallel scan is first carried out to find the indices of dead particles, then a kernel to launch particles from the fission or fixed source runs after creating a temporary array denoting the locations of slots in the particle array which are to be refilled.

From Fig. 3.18, we see that the optimal tracking rate is fairly insensitive to the refill interval. It takes thousands of events for all particles in the array to die out, so this plot highlights the efficacy of continuous refill in enhancing the tracking rate by about 10%. Of course, quite a few more particles per generation than the number in flight have to be used, otherwise no refilling could take place. Here, we used a generation size of 10M particles with 5M in flight.

The performance benefit originates from two effects. Firstly and most importantly, the particle population continuously dwindles as particles traverse their lifetimes. Eventually, the population falls below the limit that fully saturates the device shown in numerous other plots in this chapter as a sigmoidal curve that shows higher tracking rates with more particles in flight. By refilling, the device stays in a state representative of the rightmost part of the curve for longer.

Secondly, continuously refilling particles fills "holes" in the particle array. The mean distance between active particles rises as particles in the array die off, continuously decreasing the probability of cache hits as the particle population falls. By refilling the array, the

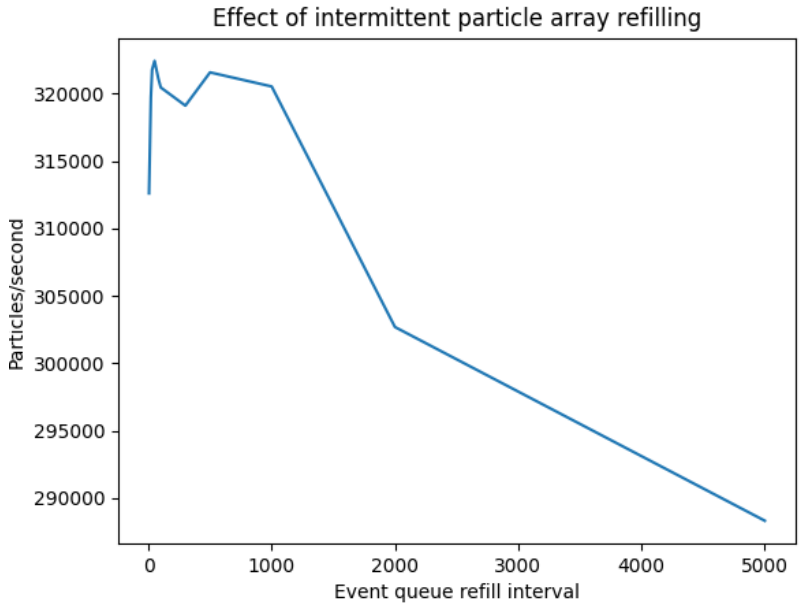


Figure 3.18: The particle tracking rate on an A100 GPU varying with the refill event period.

probability of a cache hit is restored temporarily to its original value.

3.3.6 Using __ldg and __restrict__

The baseline version of the code uses the __ldg and __restrict__ keywords in the cross-section calculation kernel. The __ldg uses loading from global memory through the texture cache, and can only be used for load operations. By using this function to load cross section rather than standard pointer dereferencing, we hint to the code that memory accesses are expected to be encountered here. In the event of having a very large number of particles in flight, this may be beneficial for the cross-section lookup.

Similarly, the __restrict__ qualifier on a pointer is used to indicate that no other pointers reference that memory. By default, without this assumption, certain memory access optimizations cannot be carried out. We also tried applying __restrict__ where possible in our cross-section lookup kernel. Figure 3.19 shows the tracking rate for the baseline calculation which used these keywords versus a version of the code in which all instances of __ldg and __restrict__ were deleted. We can see that the performance difference is negligible. With the `-O2` compiler flag and link-time optimization, the CUDA compiler appears to already apply these optimizations. Therefore, we suggest that developers of GPU MC codes do not spend time adding __ldg and __restrict__ to their code and instead focus on other areas.

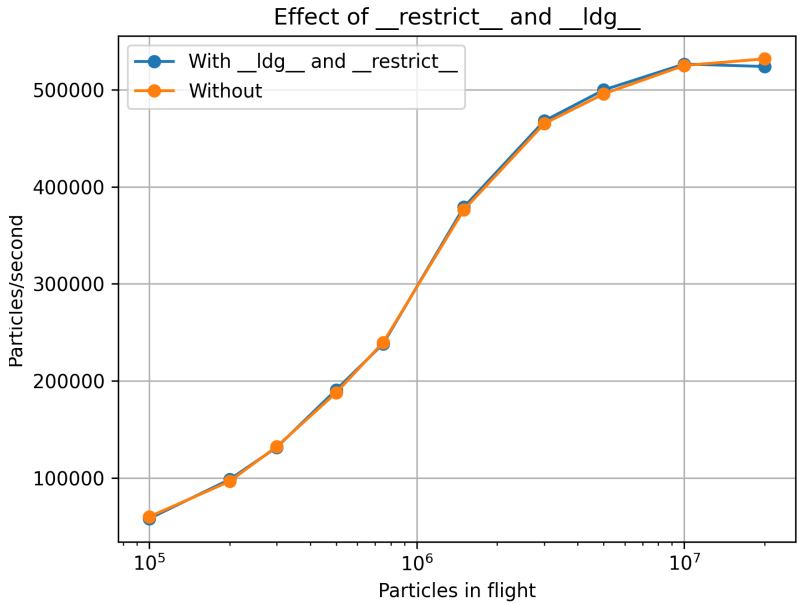


Figure 3.19: The particle tracking rate for the baseline version using __ldg and __restrict__ versus without on the H100.

3.3.7 Sorting the Cross-Section Lookup Queue

Next, we measured the effect of sorting the cross-section lookup queue. As discussed in the introductory chapter, coalesced memory accesses achieve effective memory bandwidth an order of magnitude faster than purely random accesses to global memory. Figure 3.20 illustrates our measurement of this effect on a few GPUs. We can see that GPUs beyond the Volta architecture exhibit a lower penalty for scattered memory accesses due to what we assume are advanced caching algorithms.

It was first noted in [28] that by sorting the cross-section lookup queue by particle energy, the cross-section lookup operation can be made to have a large number of coalesced accesses. After the event queue for cross-section lookup is sorted by energy with a large number of particles, blocks of particles will all be of a similar energy. If they are synchronized to loop over the same nuclides, most accesses to the nuclide's energy grid and cross-section values will be coalesced. It is through this mechanism that substantial acceleration of depleted problems can take place.

Figure **??** compares the performance of a version of the code that does not sort the cross-section lookup queue before running. We point out that we also have sorted the collision queue lexicographically by nuclide in an outer ordering then energy within that, so an increase in the performance with higher numbers of particles in flight can be seen. Interestingly, the

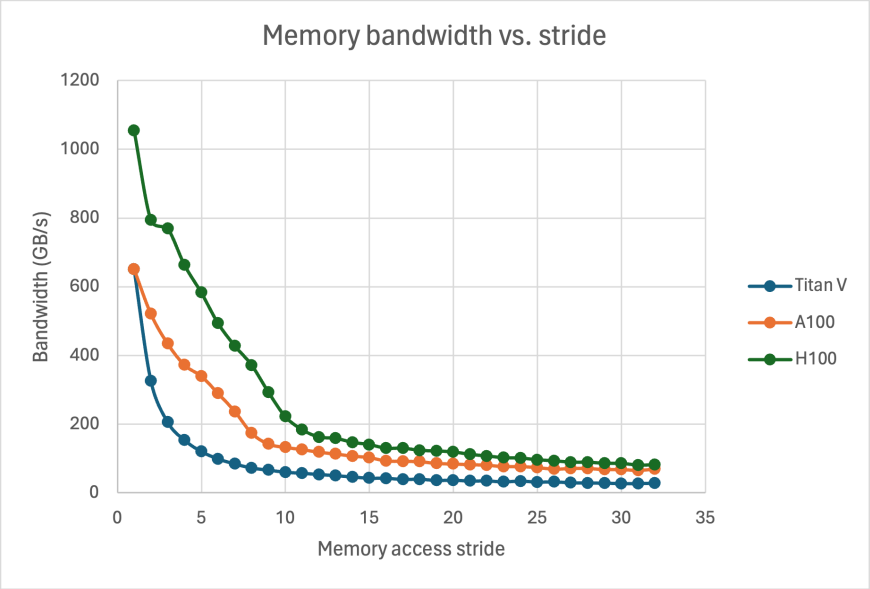


Figure 3.20: Stided memory accesses cause a loss of coalescence. In generations beyond the Titan V, the rate of performance decrease is relatively lower.

performance on the H100 continues to increase substantially as more particles are held in flight, well beyond the number of CUDA cores on this card which is around 17,000.

Figure 3.22 shows the time spent in each type of kernel per particle in flight. We can see that some time is saved compared to Fig. 3.10 by avoiding the sorting operation. However, this benefit is outweighed by the reduced acceleration of the cross-section lookup which comes as a result of less coalesced memory accesses. Overall, for any problem with a substantial number of nuclides, we suggest sorting the cross-section lookup queue.

3.3.8 Cache-Free Cross-Section Lookups

As mentioned in section 3.2, OpenMC caches the microscopic cross-sections for later use. In the event that a particle passes out of a material containing a given nuclide then returns to another with that same nuclide, this eliminates the need to look up the microscopic cross-section again.

It was found that this caching, while it does reduce the overall amount of work to do, tends to result in a large number of divergent memory accesses that thwart the positive effect of sorting the particle queue by energy as mentioned previously. After sorting the cross-section lookup queue by energy, the particles will be ordered more-or-less randomly as a function of their position in the particle array. Therefore, even if the memory accesses to the cross-section grids are coalesced, the subsequent writing of microscopic cross-section values to the cache are fully diverged.

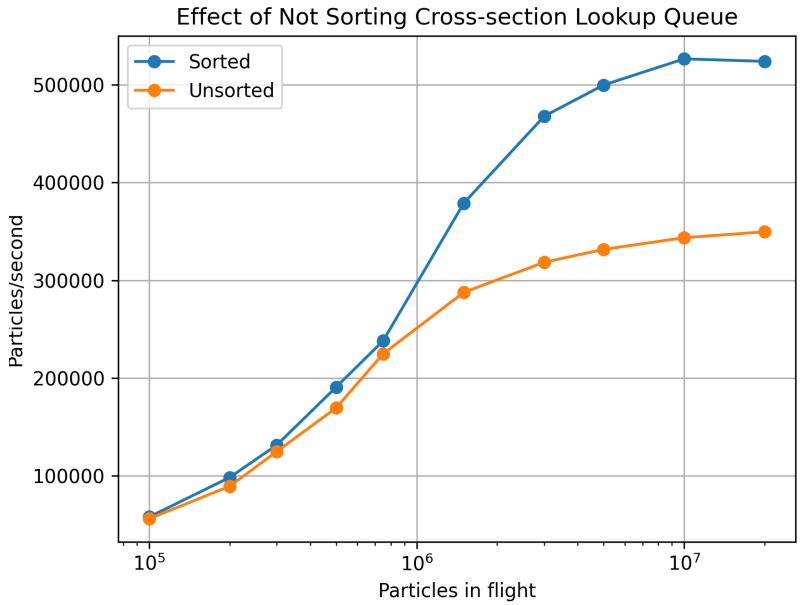


Figure 3.21: The particle tracking rate for the calculation without sorting the cross-section lookup versus the baseline code on the H100.

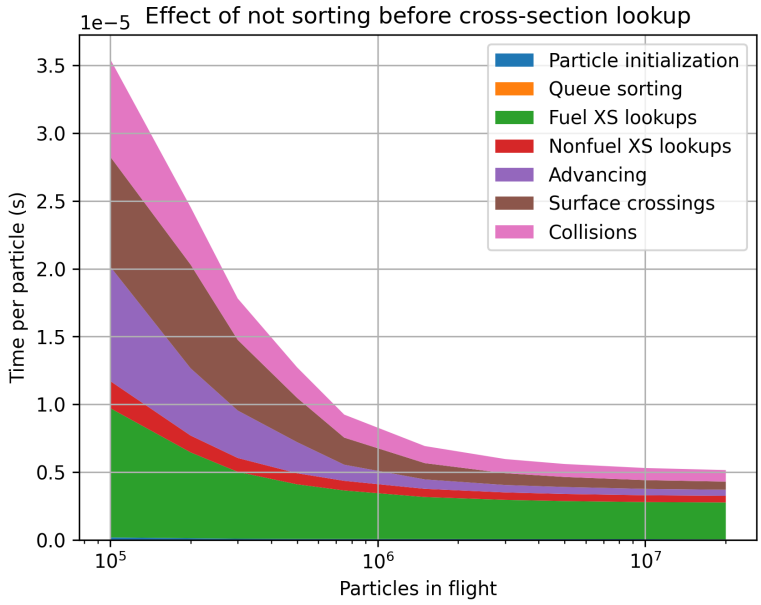


Figure 3.22: The timing breakdown for the calculation without sorting the cross-section lookup queue on the H100.

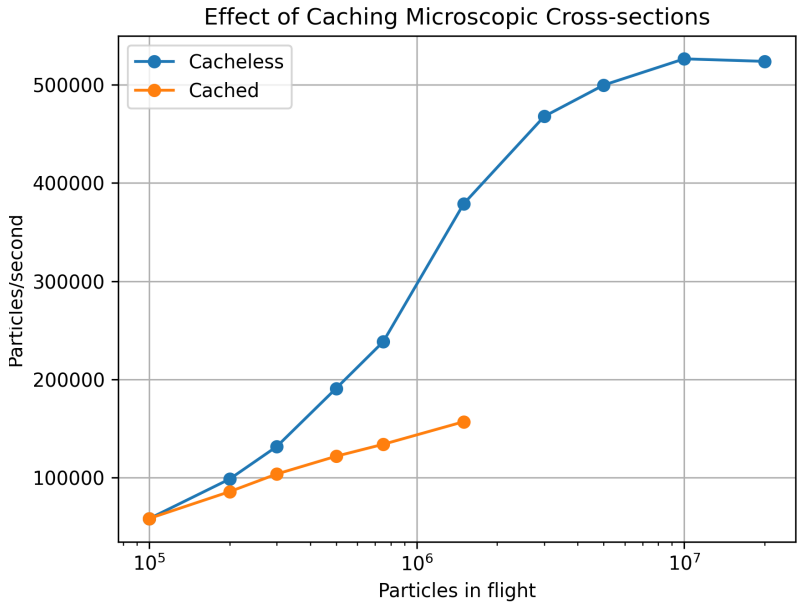


Figure 3.23: The tracking rate for the calculation with microscopic cross-section caching versus the baseline cacheless lookup method on the H100.

The best strategy for a GPU-based code, though, is to not cache these microscopic cross-sections, even if it means more work is required to later sample the collision nuclide. For the purposes of sampling the collision nuclide, the first lookup that precedes particle flight has calculated the normalizing constant to the discrete distribution over the different nuclides. The second pass made at collision time samples the nuclide given the normalizing constant which is the total macroscopic cross-section. In addition to reducing the number of divergent memory accesses, the large amount of cache space used for an array with an enormous number of particles is freed. Figure 3.23 shows the detrimental performance impact explicitly; we stopped the calculation before reaching the highest population counts to not waste further time with this terribly GPU-unfriendly approach.

3.3.9 Speculative Collision Nuclide Sampling

Next, we explored a previoulsy unpublished scheme proposed by Tim Burke for sampling the collision nuclide that reduces the amount of computational work to be done. Subsection 3.3.8 discusses the benefit of not caching the microscopic cross-sections. Without cached total microscopic cross-sections, an additional loop over the nuclide cross-sections has to be carried out to sample the collision nuclide, because the normalizing constant on the discrete distribution is not known ahead of time. The probability of a collision taking place with

nuclide i is proportional to $\sigma_{t,i}N_i$, normalized by the total macroscopic cross-section.

In the baseline implementation of our code, the cross-section lookup kernel is usually used to calculate macroscopic cross-sections. The kernel is templated on a boolean that indicates if the look-up operation is for a collision. Before each collision, a different version of the kernel is ran in which the same loop over nuclides is carried out, but instead the total macroscopic cross section is used as a normalizing constant to compute a discrete cumulative probability of each nuclide. When that cumulative probability exceeds a given pre-sampled standard uniform random number, that nuclide is used to sample the collision. In essence, this is the same method used by the CPU version of OpenMC. The only difference is that microscopic cross-sections are recalculated rather than pulled from a cache.

The "speculative collision nuclide sampling" method can avoid this additional calculation of the microscopic cross-sections before a collision. Is uses a fundamentally different method to sample from a discrete distribution. To state it generally, let $\lambda_1, \lambda_2, \ldots$ be rate constants for independent exponential random variables $X_1, X_2, \ldots$. Then we have that

$$P\left[X_i = \min\left\{X_1, X_2, \ldots\right\}\right] = \frac{\lambda_i}{\lambda_1 + \lambda_2 + \ldots} \tag{3.1}$$

Equivalently, if we were to sample path lengths for neutrons traveling through a medium consisting of each purely of nuclide and find the index of the shortest pathlength, the index of that nuclide would have precisely the correct collision nuclide distribution.

This fact can be taken advantage of to speculatively sample a collision nuclide index in the cross-section lookup operation without any normalizing constant being required. We call this "speculative" because the collision may not take place. It may be that the neutron crosses out of the material without ever using the collision nuclide index. Algorithm 6 describes the method. If a collision ever takes place, the speculatively sampled nuclide index is used to select the collision nuclide. Note that to avoid division by zero in the case of zero-density nuclides, we take the maximum of the inverse pathlength rather than the minimum of the path-length.

Figure 3.24 shows the results of using this method compared to the baseline implementation that did not speculatively sample the collision nuclide and instead carries out an additional cross-section lookup operation before each collision. This method performs slightly better at lower numbers of particles in flight, but falls short of the original method's performance when more particles come into play. This can be understood as being a result of the caching methods being used here. The code implementing this can be found at https://github.com/gridley/openmc/tree/cuda_no_collision_xs_lookup.

Firstly, we can see that in the regimes where most memory accesses for the cross-section

```
Function CrossSectionLookup(E):
    max_inv_pathlength ← 0.0
    collision_nuclide ← -1
    macroscopic_total ← 0.0
    for i_nuclide ← 0 to n_nuclides − 1 do
        sigma_t ← compute_cross_section(i_nuclide)
        macro_contrib ← sigma_t * atom_density[i_nuclide]
        inv_pathlength ← macro_contrib / (-Log(Unif()))
        if inv_pathlength > max_inv_pathlength then
            max_inv_pathlength ← inv_pathlength
            collision_nuclide ← i_nuclide
        end
    end
    store collision_nuclide
    return macroscopic_total
```

Algorithm 6: Cross-section lookup algorithm with speculative collision nuclide sampling

lookup are uncoalesced, the method only slightly out-performs. This can be attributed to the fact that most cross-section lookups are happening for higher energy neutrons which travel over many fuel regions before a collision; therefore the amount of additional cross-section lookups for collision nuclide sampling that has to take place is fairly small. Secondly, because the majority of cross-section lookups in a thermal reactor problem are not used to sample the collision nuclide, the uncoalesced memory access to store the collision nuclide introduces latency that slows down the rest of the loop. At any given point in time, a single thread caching the collision nuclide and its microscopic partial cross-sections might be causing latency for the whole thread block because our code synchronizes threads in the loop over nuclides. Without this, coalesced memory accesses to cross-section data would not be expected. We can anticipate that in fast reactors, this method would under-perform even more compared to the baseline technique because on average more regions will be crossed between collisions.

On the whole, we suggest not using this method for reactor simulations. However, in other applications with smaller numbers of nuclides or more optically thick material cells, this method should be tested as it does reduce the total amount of computation that must take place. Regardless, similar to the removal of the microscopic cross-section cahce, even an infrequent, unfriendly diverged memory access can ruin performance. Considering that for collision estimators of the depletion reaction rates a lookup operation should be carried out before a collision anyway, we again emphasize that this method is not promising to improve performance of depletion simulations of fission reactors.

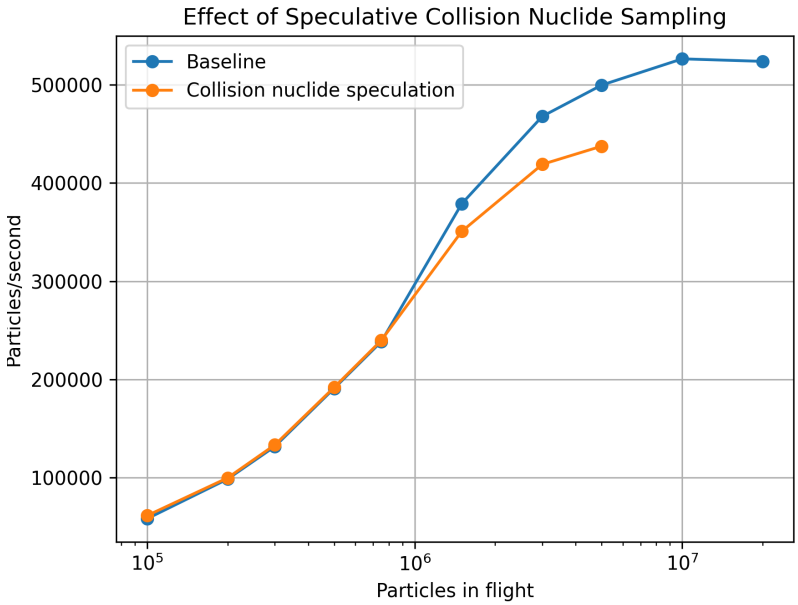


Figure 3.24: The tracking rate for the calculation with speculatively sampled collision nuclides versus the baseline code on the H100.

Modifications to Caching Behavior

In this subsection, we explored the different approaches to caching global memory accesses presented by the CUDA compiler. In [28], the authors point out that they compiled the code using the `-Xptxas -dlcm=ca`. In other words, the PTX assembler was instructed to cache memory accesses at all available levels. Here, we quantify the efficacy of this approach relative to some other caching options available in CUDA, and show that the choice of memory caching method has practically no impact on performance when compiled with the `-O2` flag and link-time optimization. Table **??** summarizes the different possible options. Figure 3.25 shows the performance of each, showing that no practical performance difference can be discerned from each option.

Effects of GPU Code Linking Methods

We next explored two methods of linking CUDA code for our GPU-accelerated version of OpenMC. By default, CUDA kernels cannot link to device code that is outside the translation unit (usually an individual file). Because including device code compiled from other translation units often precludes various types of compiler optimization, this feature tends to degrade performance. However, it enhances the maintainability of code by allowing the huge swaths of physics code to be separated out from the main kernels.

Fortunately, over the course of this research, link-time optimization was developed and released as part of CUDA 11 in 2020. This defers compiler optimizations until the linking phase. While this adds a considerable amount of compile time that cannot be parallelized, the performance gains are clear as shown in Figure 3.26 in which we compare the baseline version of the code to a version compiled using relocatable device code. Interestingly, the

Table 3.2: Summary of data cache load modifier (`dlcm`) options.

Option	Description
`-dlcm=ca`	Cache at all levels. This option caches global memory loads at all levels of the memory hierarchy, including L1 and L2 caches.
`-dlcm=cg`	Cache in global memory only. This option disables L1 cache but allows caching in L2 and other higher levels of cache.
`-dlcm=cs`	Cache streaming. This option hints that the data is not expected to be reused, so it prioritizes L2 caching.
`-dlcm=wb`	Write-back caching. This option ensures that writes to global memory are cached and later written back to global memory.
`-dlcm=wt`	Write-through caching. This option ensures that writes to global memory bypass the cache and are written directly to global memory.

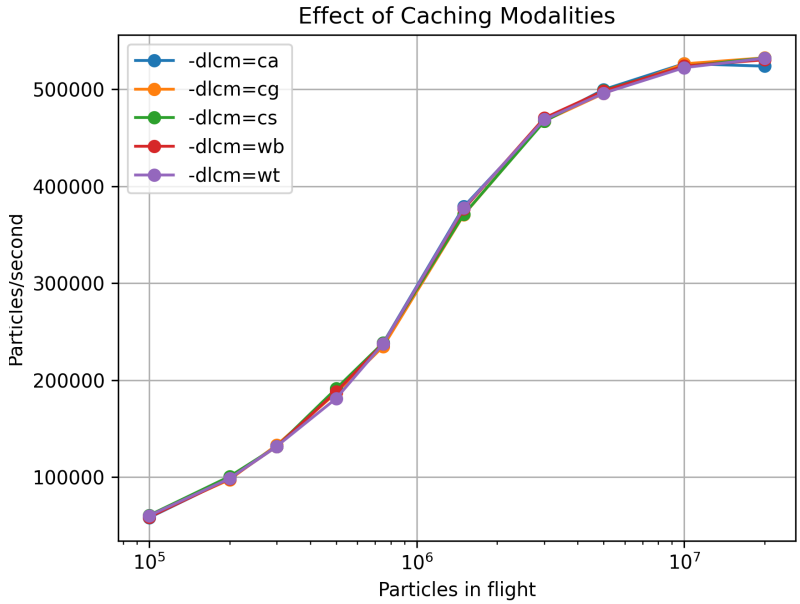


Figure 3.25: The tracking rate for different XPTXAS DLCM options on the H100.

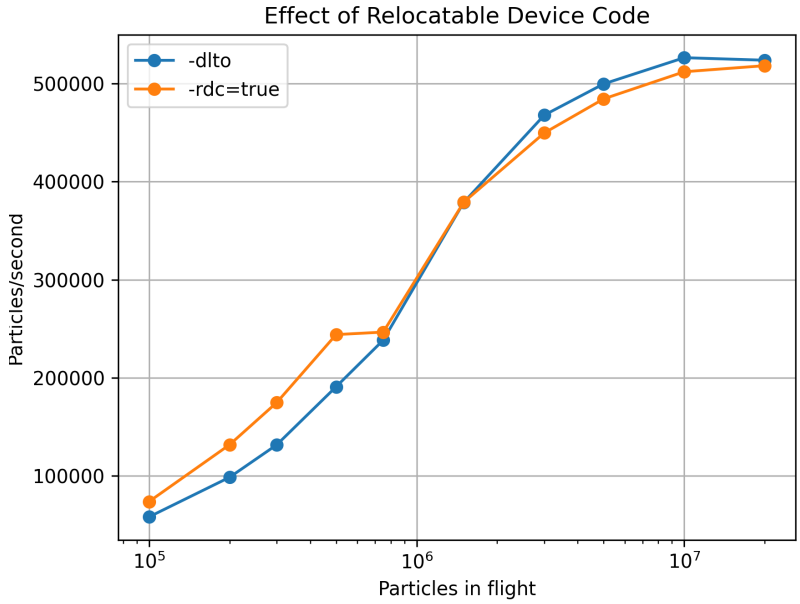


Figure 3.26: The tracking rate for two different GPU code linking methods on the H100.

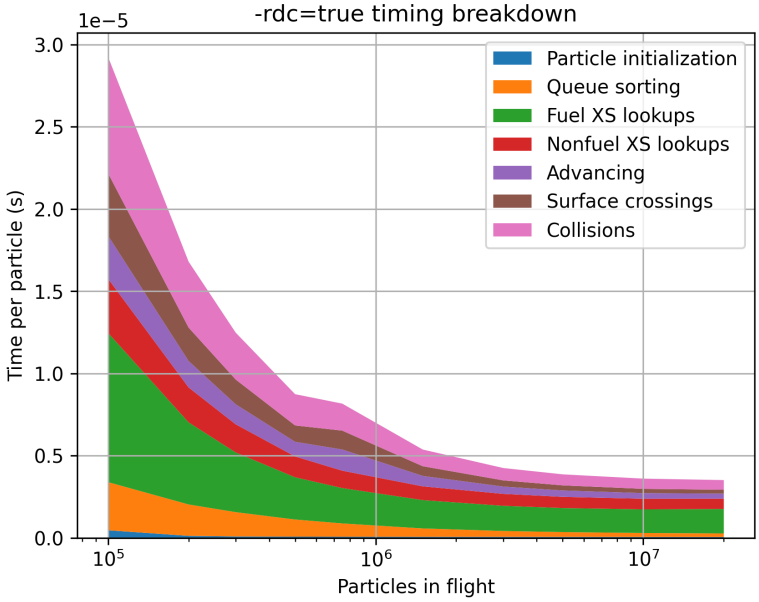


Figure 3.27: Timing breakdown for compiling with relocatable device code rather than link-time optimization.

3.4 Other Aspects of Porting the Code

3.4.1 The Global Block Lock Technique For Neighbor Lists

Neighbor lists are used to accelerate the search for the cell a particle is crossing into in Monte Carlo particle transport codes [118]. Without neighbor lists, a particle which exits a constructive solid geometry cell must search the entire universe of cells and evaluate each surface expression until the containing cell is determined. With a neighbor list, the neighboring cell IDs are cached and stored on each cell. All cells start with an empty neighbor list. If a particle crosses out of the cell, the neighbors' constructive solid geometry definitions are checked. If none are found, a fallback is made to an exhaustive search through the whole universe. After completing the exhaustive search, the new cell ID which was not previously recognized as a neighbor is written to the neighbor list.

Completing this task in a thread-safe way is nontrivial enough to merit its own conference paper [118]. Before this work, neighbor lists were implemented with a mutex thread locking technique in OpenMP. The mutex ensures that duplicate cells are never added to the neighbor list. Because multiple particles are processed concurrently on a multi-threaded CPU, it may be the case that multiple particles simultaneously determine that a given cell must be added to the list. When one thread acquires a mutex, it prevents others from appending that new cell index to the list. The first thread would acquire the mutex and append to the list. The second in this hypothetical situation would then acquire the mutex, but then determine that the first had already inserted that index to the list.

On a GPU, mutexes are not only unavailable but also undesirable. Rather than a few tens of threads executing concurrently, GPUs typically employ a few thousand. The use of mutexes could result in disastrously slow performance as thousands of threads wait to execute only one-at-a-time. A new approach to the problem would preferably be taken.

After considering this possibility for a bit, we determined that designing a new concurrent data structure for this problem was not worth the effort. In a MC neutron or photon transport simulation, only the first few particles have to append to the neighbor lists. As the simulation runs, the fraction of time required for neighbor list operations becomes arbitrarily low. In reactor neutronics computations, only the first one or two generations of particles have to append to the neighbor list. Thousands of generations of particles are required for the most computationally intense problems.

Despite our desire to simply use an off-the-shelf mutex and to accept the performance penalty, as mentioned previously no such functionality is available on a GPU. Instead, we developed a novel mechanism to implement a mutex on GPUs–an unwise choice in most

situations but not this one. Algorithm 7 shows pseudocode for the method. Two separate mechanisms are required to mutually exclude each thread and each thread block.

```
Function neighbor_list::push_back(new_elem)
    for thread_id ← 0 to blockDim.x do
        if threadIdx.x == thread_id then
            while atomicCAS (lock, 0, 1) != 0 do
                ;
            end
            for i ← 0 to current_size do
                if new_elem == buffer[i] then
                    atomicExch (lock, 0);
                    return;
                end
            end
            if current_size == buffer_size then
                trap;
            end
            buffer[current_size++] = new_elem;
            atomicExch (lock, 0);
        end
    end
```

Algorithm 7: Pseudocode for CUDA push_back function

Firstly, when threads call this function, a loop over all threads is carried out. In other words, each thread executes a loop with an inner statement checking the index of the thread. Only that one thread that matches the current loop index is allowed to execute what follows. This excludes the possibility of two threads from the same block trying to lock the neighbor list at the same time. After the lock is acquired, the thread loops over the current elements in the neighbor list to check for the presence of the element it will soon add. The maximum size of the neighbor list has to be allocated in advance for this implementation. Lastly, if the element was not found in the current neighbor list, the current size of the list is advanced and the new element inserted accordingly.

The outer loop over threads solves a dead-locking problem that halts execution of the program. We do not understand the cause of the dead-lock aside from that adding the loop over threads solves it.

3.4.2 Non-recursive Nested Universe Methods

The CUDA compiler is unable to determine the optimal number of registers to assign to each thread when recursive functions are used. At the start of this work, the universe cell-finding code in OpenMC was recursive. It was found in the author's pull request to OpenMC https://github.com/openmc-dev/openmc/pull/1784/files that stack overflows tended to happen in constructive solid geometry definitions with nested universes. This came as no surprise because the CUDA compiler had warned about being unable to determine the correct stack size.

To avoid the need for recursion that potentially leads to stack overflows, we implemented a loop-based approach to nested universes rather than recursion. Algorithm 8 shows what the code that tended to cause stack overflow on GPU looks like, and Algorithm 9 shows the modification required to avoid recursion. In the former, the cell finding routine is called recursively on the new universe that's potentially been found nested inside a cell or lattice. In the latter, a loop over each universe levels is carried out. The nesting index is incremented in each iteration of the loop, which terminates after a cell filled with material is finally found.

3.4.3 On Random Number Reproducibility

In this study, we maintained exact reproducibility between the GPU and CPU versions of the code. This greatly aided debugging because statistical comparisons between the two code versions was not necessary. Many times in the course of code development, slight differences in the eigenvalue returned by a problem immediately alerted us to an issue with the code that may be very difficult to discover otherwise. For this reason, comparisons on the eigenvalues and tallies were not presented in this chapter because they are identical to the results obtained by the CPU version of OpenMC, which has been extensively validated elsewhere.

This required using the same random number generator between the CPU and GPU versions of the code. While CUDA does provide a library for random number generation, it is primarily oriented towards computationally expensive, high quality random number generation. We found that the PCG-LCG [119] random number generation performed excellently on GPU, as the randomness quality needs for generators in MC radiation transport are minimal.

```
Function find_cell_inner(particle):
    found ← false;
    cell_index ← NONE;
    universe ← particle_current_universe(particle);
    cells ← get_cells_in_universe(universe, particle);
    foreach cell in cells do
        cell_index ← cell;
        if cell_universe(cell_index) != particle_current_universe(particle) then
            continue;
        end
        if cell_contains_particle(cell_index, particle) then
            set_particle_cell(particle, cell_index);
            found ← true;
            break;
        end
    end
    if found then
        announce_cell_entry(cell_index);
        cell ← get_cell(cell_index);
        if cell.type == MATERIAL then
            set_material_and_temperature(particle, cell);
            return true;
        end
        else if cell.type == UNIVERSE then
            update_particle_coord_for_universe(particle, cell);
            increment_particle_coord_level(particle);
            return find_cell_inner(particle);
        end
        else if cell.type == LATTICE then
            update_particle_coord_for_lattice(particle, cell);
            increment_particle_coord_level(particle);
            return find_cell_inner(particle);
        end
    end
    return found;
```

Algorithm 8: Recursive Cell Finding

```
Function find_cell_inner(particle):
    found ← false;
    cell_index ← NONE;
    while true do
        if cell_index == NONE then
            universe ← particle_current_universe(particle);
            cells ← get_cells_in_universe(universe, particle);
            foreach cell in cells do
                cell_index ← cell;
                if cell_universe(cell_index) != particle_current_universe(particle)
                 then
                    continue;
                end
                if cell_contains_particle(cell_index, particle) then
                    set_particle_cell(particle, cell_index);
                    found ← true;
                    break;
                end
            end
        end
        if not found then
            return found;
        end
        announce_cell_entry(cell_index);
        cell ← get_cell(cell_index);
        if cell.type == MATERIAL then
            set_material_and_temperature(particle, cell);
            return true;
        end
        else if cell.type == UNIVERSE then
            update_particle_coord_for_universe(particle, cell);
        end
        else if cell.type == LATTICE then
            update_particle_coord_for_lattice(particle, cell);
        end
        cell_index ← NONE;
    end
```

Algorithm 9: Non-Recursive Cell Finding

3.4.4 Re-ordering the nuclide loop

In the CPU implementation of OpenMC, it is perfectly reasonable to loop over nuclides as they appear in a material in computing the cross-sections. This is the default approach to the problem. We identified that the performance of the cross-section lookup operation can be greatly degraded by following this approach and addressed it appropriately.

For example, consider two fuel materials with the nuclides of one being listed in reverse order. The key to obtaining speedups on GPU with large numbers of cross-section lookups is to sort the particles by energy so as to induce coalesced memory accesses to the pointwise cross-section grids. This effect can be fully hampered if different threads are working on different fuel materials–a common scenario in any practical depleted reactor problem.

To address this, we impose a global ordering on nuclides according to the order they were loaded in. When a particle undergoes cross-section lookup, a loop over all nuclides occurring in the problem is carried out. All materials have an additional array allocated matching the length of the global nuclide array. At each position is the index of the nuclide in that material. If the nuclide is not present, a specialized flag value like -1 can be stored. For example, suppose there were two materials, water and zirconium hydride. Globally, three nuclides might be present, ^{1}H, ^{16}O, and ^{90}Zr. In this case, water would have the mapping array [0, 1, -1], and zirconium hydride would have [0, -1, 1]. The nuclide order may be changed in a material: for example if zirconium hydride defined hydrogen second, the array would be [1, -1, 0].

By doing so, a single loop over the global nuclide ordering is carried out on each thread. Only if the nuclide is present, the thread enters an `if` block to carry out cross-section lookup. Otherwise, it idles. At the end of each iteration of the loop, `__syncthreads()` or `__syncwarp()` can be called. We found that on the Titan V GPU, which came from the first generation of GPUs with independent thread execution, this manual synchronization helped the calculation speed. However, in the A100 and H100, it appears that the hardware now plans ro-coalescence of the threads in an intelligent way to obviate the need for this.

By doing so, the performance of a GPU application is made insensitive to changes in the ordering of nuclides of the material. Moreover, we found that on the Titan V GPU, manual synchronization helped to partially alleviate the performance impact of using probability tables. Again, we were unable to measure this benefit on the A100 and H100 GPU.

3.5 Improved Windowed Multipole Method on GPU

It was shown in [120] that the windowed multipole (WMP) method [71] substantially underperformed compared to using traditional pointwise cross-sections. In this section, we show that the WMP method can outperform pointwise cross-sections for reactor simulations in the general case. This relies on one main technological advance for WMP and one removal of an unrealistic modeling assumption for GPU MC computational benchmarking.

Firstly, we will review the results obtained in [120]. The authors reported a tracking rate of 112,000 particles/second on four P100 GPUs when using pointwise continuous energy cross-sections, and 52,000 particles/second using the same setup but with windowed multipole cross-sections. The authors report using a temperature for this depleted reactor problem that allows a single temperature for the pointwise cross-sections which was deemed "conservative". The conservatism in this case is to find the case least favorable to WMP methods, which expected to perform better both in terms of accuracy when pointwise libraries must interpolate between temperatures; its signature advantage is physical temperature handling. At the same time, interpolation in temperature on a GPU can incur a serious computational penalty as a result of the additional memory accesses that have to take place.

Figure 3.28 shows timing breakdown for each kernel type when running in windowed multipole mode for a cold isothermal reactor simulation.

Figure 3.29 shows the tracking rate of pointwise and WMP cross-sections at two temperatures. To refer to the simulation as "WMP" alone is somewhat of a misnomer, because pointwise cross-sections must be used above the unresolved resonance threshold. We present in a later chapter a new method for modeling the unresolved resonance region which can overcome this barrier, but point out that the drawbacks of pointwise cross-sections still come into play when WMP is used just to a lesser extent.

The room temperature case was chosen because it coincides with an evaluation of pointwise nuclear cross-sections. One-thousand and fifty kelvin was chosen carefully: this is the midpoint between the 900 kelvin and 1200 kelvin temperature points in OpenMC's native ENDFB/VIII.0 nuclear data library. As a result, stochastic interpolation leads to memory accesses to two entirely different cross-section grids each half of the time. This degrades performance, as shown by Fig. 3.29. In fact, we can see that WMP *outperforms* pointwise cross-sections. These conclusions fundamentally differ from those presented in [120]. Paired with WMP's superiority in physically accurate temperature interpolation, we suggest not only the use of but also extension of WMP for GPU computations. Much potential research remains along the same vein of moving from linear interpolation tables to physics-based numerical models. Our next chapter advances in precisely that direction.

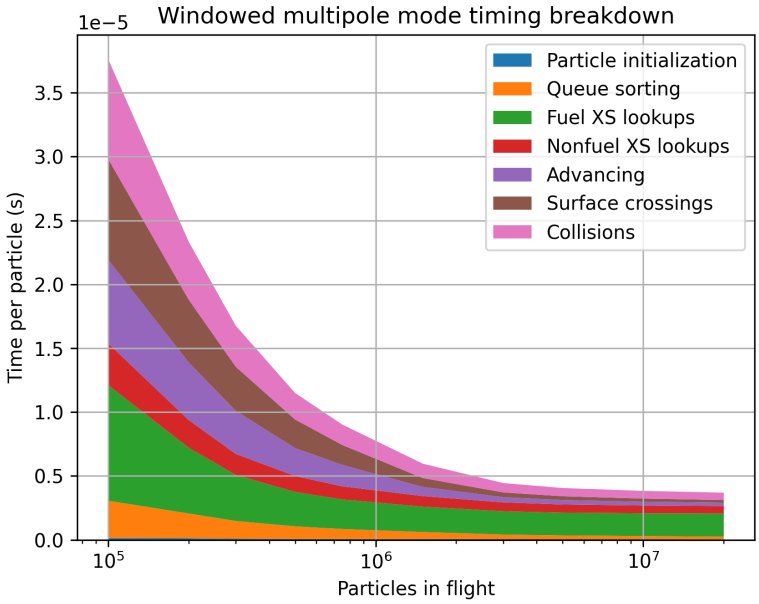


Figure 3.28: Timing breakdown for using windowed multipole cross-sections.

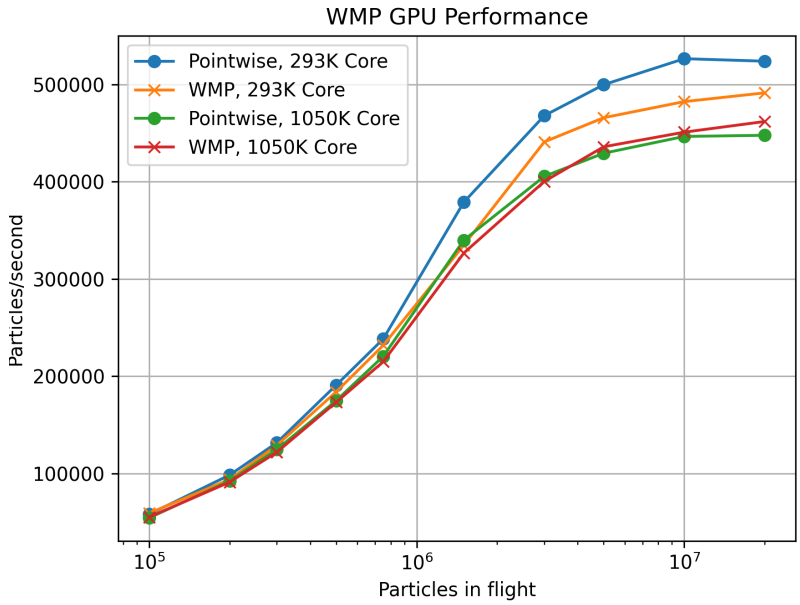


Figure 3.29: Tracking rate for windowed multipole mode versus baseline pointwise mode on the H100 at two reactor temperatures.

Multipole supremacy on GPU was achieved with a single key numerical technique. This started with Dr. Forget's recognition that the numerical methods for the Faddeeva function, a key computational aspect of WMP that enables the analytical handling of Doppler broadening, were bound to under-perform on GPU. The Faddeeva function can be defined for $\Im[z] > 0$ as:

$$w(z) = \frac{i}{\pi} \int_{-\infty}^{\infty} \frac{e^{-t^2}\,\mathrm{d}t}{z - t} \tag{3.2}$$

The mainstream library for calculating the Faddeeva function was also written at MIT [121], and includes numerous branches to handle all possible use cases of the Faddeeva function. The extensive branching yields poor performance on GPUs for reasons discussed in the introductory chapter, and moreover, given the uncertainties inherent to nuclear cross-sections, the usual $w(z)$ implementation's target of a few ulps of precision of a 64-bit number is excessive.

Plenty of research in the past has explored faster ways of calculating the Faddeeva function, one key example from nuclear engineering being developed as part of the classic MC^2 code [122]. More references on Faddeeva function calculation can be found in our paper [123].

In that work, we explored the benefit of a simple approximation to the Faddeeva function for the purposes of the windowed multipole method. It relies on the fact that in the context of windowed multipole, we have that $\Im[z] > 0$ which allows a removal of a branch to handle the $\Im[z] < 0$ that introduces a discontinuous principal value term, requiring a branch in a computer implementation. We can shift the line of integration down to the lower half of the complex plane because it forms a closed loop in the complex plane. Because shifting downward never encloses a pole, a Gauss-Hermite quadrature can be applied to Eq. 3.2 to approximate $w(z)$. It was Humliček that identified the ability of a Gauss-Hermite quadrature to to immediately obtain a complex rational approximation [124]–[126]. The downward shift serves to reduce the impact of nearby poles of $w(z)$ when z has a small imaginary part.

Figure 3.30 shows regions in which the Faddeeva function must be frequently evaluated for a typical problem involved windowed multipole data. A depleted PWR fuel assembly with temperatures ranging from 600-900K was simulated, and whenever $w(z)$ was evaluated, the value of z was saved. The probability density in $\mathbb{C}$ could then be estimated, which guides where $w(z)$ should be approximated best.

It was shown in [123] that the use of an approximation to $w(z)$ yielded accurate results both in terms of the spatial fission distribution and the eigenvalue for a typical fuel assembly problem. Moreover, it was found that even in a CPU-based code, a problem with strongly varying temperatures between fuel pins can be solved more efficiently using windowed mul-

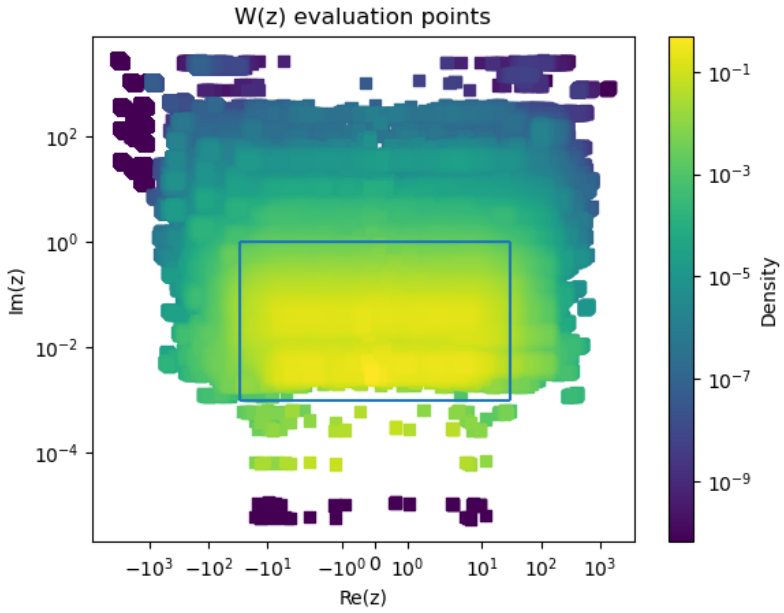


Figure 3.30: Frequency of evaluation of $w(z)$ for a depleted PWR assembly problem.

tipole than pointwise cross-sections due to the high rate of microscopic cross-section cache misses with strongly varying temperature.

The C++ code we wrote using a downward shift value into the lower complex plane is listed in Appendix B for the convenience of others. Notably many coefficients appear with zero real or imaginary parts. In our experience, modern compilers are able to propagate hte zeros through the following expressions to avoid unnecessary arithmetic, but this potentially is not applied at lower optimization levels. That is to say, our efforts algebraically propagate zeros through to cancel many terms in the rational approximation were in vain with modest compiler optimizations enabled. Nonetheless, our CUDA implementation manually expands all terms to avoid multiplication by zero and can be found in the aforementioned Github repository in `include/cuda/calculate_xs_kern.h`.

3.6 Discussion

We have presented some novel software development techniques for enabling easier transition of large continuous energy neutron MC code to GPU execution, and examined the efficacy of some previously employed optimizations to improve the particle tracking rate. We hope other researchers in this area can make use of the code presented in the `cuda` branch of the public repository github.com/openmc-dev/openmc.

Our experience in measuring and optimizing the performance of a continuous energy GPU

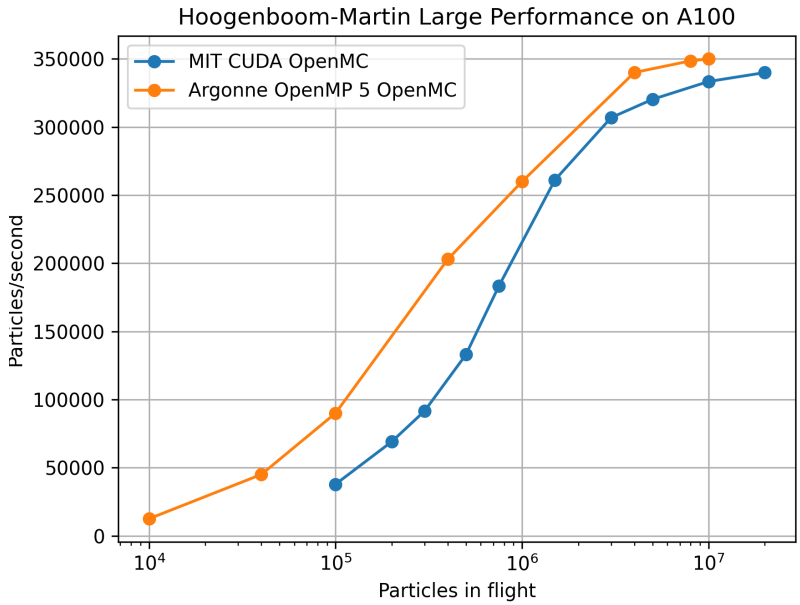


Figure 3.31: Performance comparison of two GPU OpenMC implementations on A100 with Tramm's Hoogenboom-Martin large benchmark.

MC neutronics code came with a few key lessons. Firstly, the maturity of GPU software toolchains is only just reaching the point to obtain practical performance on GPUs. Simply by moving from CUDA 11 to CUDA 12, our implementation's performance increased by nearly a factor of two. This took it from being substantially less performant than the other implementation of OpenMC on GPU that used OpenMP offloading presented in [54] to being on similar footing, as shown by Fig. 3.31. We must point out that later results from Argonne [127] reported a maximum tracking rate of 425,000 particles/second rather than 350,000, but do not present a scaling curve.

The Argonne work [54] also presents a comparison against a typical server CPU, an Intel Xeon 8180M with 56 CPU cores. It obtained a particle tracking rate in history mode around 100,000 particles/second. In other words, both GPU codes performed equivalently to about 196 CPU cores. Argonne's later improved results [127] reported 425,000 particles/second rather than 350,000, or in other words being equivalent to around 240 CPU cores. Our results for the H100 GPU are therefore equivalent to about 280 CPU cores, and Argonne's results in [127] are equivalent to about 360 CPU cores. We can contrast this against the discussion presented in the 2018 work [56] in which the author points out most GPU MC codes were unable to pass about thirty CPU cores' equivalence. Clearly, this barrier has been shattered in the years after 2018, and continues to widen as shown in [127]. This continual advancement of GPU MC codes has likely occurred as a result of advancements made to

improve effective memory bandwidth in scattered accesses as illustrated by Fig. 3.20.

A second important lesson is that conclusions in the optimal approach for Monte Carlo neutron transport can vary based on the number of particles in flight. For example, based on Fig. 3.26, we might conclude that relocatable device code is the optimal choice if only a relatively small number of particles were simulated. On the other hand, link-time optimization wins at higher particle counts. Similarly, structure-of-array wins at higher particle counts but not lower ones as shown by Fig. 3.15.

This code did not implement tallies as a result of lessons learned in the process of writing the CUDA code. It was initially thought at the outset of this project that a novel contribution would be development of a maximally shared code between the CPU and GPU versions of the program. Despite the opportunity for shared code, the paradigm shift is so strong that practically all aspects of the code have to be reconsidered. In the prior sections, we proved that the optimal techniques in a GPU-based MC code almost always differ from the optimal techniques on a CPU. For this reason, tallies were not implemented.

Tallies already introduce considerable overhead to CPU-based Monte Carlo calculations as a result of their intensive memory access pattern. The general architecture of tallies in OpenMC is to push back filter bin index and weight combinations to a per-particle temporary array which is later processed in a scoring routine. On a GPU, the filter bin matching and scoring processes should be united to avoid numerous uncoalesced memory accesses. The scoring system would be less general than the mainline OpenMC tallying system. However, its extreme flexibility is not necessary for practical reactor computations which often need a very specific set of tallies only: power in each cell, depletion reaction rates in each cell, among a few others.

Chapter 4

Fast Resonance Upscatter Sampling with Windowed Multipole Cross Sections

This chapter is based on the following paper:

> Ridley, Gavin, Benoit Forget, and Timothy Burke. "Resonance Scattering Treatment with the Windowed Multipole Formalism." Nuclear Science and Engineering 193, no. 3 (May 25, 2023): 1–25. https://doi.org/10.1080/00295639.2023.2204810.

The code discussed in this chapter can be found at https://github.com/gridley/openmc/tree/mars.

4.1 Introduction

Early continuous energy Monte Carlo neutron transport programs sampled scattering from nuclei in thermal motion assuming that the scattering cross section is effectively constant within the scattering kernel [128]. However, as [129], [130] detail , the resulting scattering kernel implied by the constant cross section approximation may be far from the actual double-differential cross section near a scattering resonance. As shown in [131], [132], the resulting error tends to cause a worst-case 11% underestimation of the Doppler feedback coefficient in a PWR, with even larger discrepancies in HTGR problems.

As exhibited by the PRAGMA project [29], [133], the conventional methods for treating this effect leave something to be desired on graphics processing unit (GPU) architectures, which constitute the majority of computational power on most leading supercomputers. Due to the unique architecture of the GPU, algorithmic modifications to standard Monte Carlo algorithms for neutron tracking can tangibly accelerate computation [29]. In the same direction, we herein present a GPU-friendly method for handling resonance upscatter when

the windowed multipole (WMP) [71] formalism is employed to represent cross sections. In particular, the heuristic for fast GPU code is to avoid rejection sampling and accesses to distantly spaced places in memory, which the new method achieves. To give context to the new method, we first recall some of the conventional methods for modeling resonancekup-scattering.

The Doppler Broadening Rejection Correction (DBRC) [134] was one of the early proposed techniques to treat the effect of strong variations of the interaction cross section within the energetic vicinity of a scattering neutron, whereas $S(\alpha, \beta)$ tables had been used prior [135]. The method has since been implemented in numerous continuous energy Monte Carlo neutron transport programs [134], [136]–[138], and has shown to successfully model the resonance upscatter effect. However, DBRC suffers from rejection probabilities as high as 99.995% [99] for neutron energies in the vicinity of a resonance.

The weight correction method (WCM) [131] also successfully models the effect of resonances on the double-differential cross section of nuclei in thermal motion. This method adjusts the weight of particles to provide numerically correct results even when the constant cross-section double-differential free gas distribution is employed. WCM carries the same benefit of our newly proposed method of not requiring an additional rejection loop or tables; however, adjustments to the particle weights introduce substantial variance to the overall Monte Carlo simulation, thus degrading estimates on quantities of interest [133].

Another technique known as target motion sampling (TMS) [139] can be used to model the resonance upscatter effect. However, this method is only applicable to Monte Carlo neutron transport programs employing the delta tracking technique. Unfortunately, performance of delta tracking appears to be lackluster on GPUs [63].

The relative velocity sampling (RVS) method [99], [140] was created to ameliorate the high rejection rates characteristic to the rejection algorithms used to model resonance upscatter. These schemes, in essence, sample probability distributions proportional to $f(x)g(x)$, where $f(x)$ is a distribution and $g(x) \in [0, 1]$. One then samples from $f(x)$ and accepts the sample with probability $g(x)$. The RVS method moves the direct sampling from the thermal motion term to the cross section term, thus worsening the average case rejection rate but massively improving the worst-case rejection rate.

Despite this improvement, even moderate rejection rates harm performance on GPUs in an outsized fashion. GPUs offer promise for extremely high performance Monte Carlo transport calculations, e.g. full-core PWR cycle depletion calculations on small computer clusters [29]. This power comes with limitations of the GPU computing paradigm. The single instruction multiple thread (SIMT) parallelism strategy employed by GPUs offers power and computational efficiency advantages, but requires special attention to designing the tracking

algorithm. For example, if there are t threads executing in parallel, it was shown in Chapter 2 that the expected number of rejection loop iterations with rejection rate ρ grows from $1/(1-\rho)$ in the single thread case to:

$$(\text{mean samples before acceptance}) \approx \frac{\log\left(2(\rho-1)t(\log\rho)^{-1}\right)}{\log(\rho^{-1})} + 1 \tag{4.1}$$

where t is the number of threads in the execution unit, depending on the synchronization and kernel launch settings of the program. Since t is often around 256, this would imply that a rejection loop with a 99% rejection probability would grow from requiring around 100 iterations to around 620 iterations.

The relative speed tabulation (RST) method [133] was developed to address the rejection sampling performance impact on GPUs. RST is the first resonance upscatter modeling technique not requiring a rejection loop or bivariate scattering distribution tables. The key observation underlying RST is that the target velocity distribution can be factorized into a marginal relative speed distribution encapsulating the information about the resonances and a simple distribution of the target polar angle conditioned on the target relative speed. Consequently, the univariate relative speed cumulative distribution can be tabulated in select areas of the pointwise cross-section, and the conditional polar angle distribution is then directly sampled without requiring any additional data or rejection step.

Despite its simplicity and efficacy on GPUs, the RST method comes with some clear disadvantages. Gigabytes of additional memory are used in storing the relative speed cumulative distributions (CDFs) at each energy point and temperature on a pointwise cross section representation, if all nuclides have the resonance upscatter effect treated. The resonance influence on the double differential cross section is thus restricted for practical reasons to a select few nuclides in the problem to avoid extreme memory usage. Moreover, the method introduces some discretization error in temperature, although this was shown to be a reasonable approximation. Lastly, the ultimate flaw of RST stems from its fundamental incompatibility with the windowed multipole formalism, an efficient, physics-based format for storing nuclear cross sections. While RST naturally shoehorns into a code employing pointwise cross sections, its is of minimal applicability to windowed multipole based calculations.

If one could avoid pre-tabulated relative speed distributions, this could substantially reduce the memory needs and avoid costly divergent memory accesses. Reducing serialized memory accesses on GPUs typically leads to significant speedups.

We propose a new method which does just that, and achieves this using the WMP cross section representation [71]. Our new method introduces a novel special function we have

deemed the incomplete Faddeeva function which encodes the behavior of the temperature-dependent influence of resonances on the double differential scattering distribution. It relies on the numerical inversion of the analytic representation of the relative speed distribution under the WMP formalism, and polar angle sampling in the same manner as the RST method, but without any precomputed tables.

This contrasts the WMP-based target velocity sampling technique presented in [141], which is similar in nature to the newly presented method in this work. However, [141]'s method relies on the separation of the target velocity distribution into a zero kelvin cross section component and a Maxwell-Boltzmann component. This method thus requires a numerical inversion step of the integrated scattering cross section function wrapped in a rejection loop, similar to [99] but instead using a functional representation of the integrated cross section rather than tabular.

The algorithm presented in [142] shows how the relative speed can be sampled in the windowed multipole framework similarly to our work. However, [142] relies on fitting a sum of Gaussians to replace the poles in Eq. 4.5, thus representing the zero kelvin scattering cross section in the form:

$$\sigma(E) = \frac{1}{E} \sum_{k} \sum_{j} \left[h_{s,k,j} e^{\left(\sqrt{E}-u_k\right)^2 / w_{s,k,j}^2} + h_{a,k,j} e^{\left(\sqrt{E}-u_k\right)^2 / w_{a,k,j}^2} \right] \tag{4.2}$$

While it remains to be seen that Gaussians can be used to approximate all poles appearing in a windowed multipole library, this proposed approximation introduces a considerable number of degrees of freedom to the problem, with eighteen unknowns for each pole. The optimization problem thus encountered is highly nonlinear and nonconvex leading to fitting difficulties. Additionally, scattering kernels in [142]'s formalism cannot be straightforwardly differentiated with respect to windowed multipole parameters which would be needed to perform sensitivity analysis.

Our new method, multipole analytic resonance scattering (MARS), only requires the same windowed multipole data as would be used in a calculation without any treatment of the resonance upscatter effect, avoiding the need for additional tables or fitting steps. We demonstrate the new method's negligible performance overhead compared to other methods for sampling the resonance upscatter effect on both CPU and GPU architectures.

4.2 Theory

Rothenstein et. al. [129] expounds the rigorous Doppler broadened double-differential cross section. Since the exact expression for the lab frame scattering distribution is quite com-

plicated, authors presenting algorithms to model it typically choose to forgo the lab frame expression and instead reason in terms of joint distributions of the target velocity and direction cosine relative to the direction of projectile motion. After sampling the target speed and direction cosine, standard two-body collision kinematics for elastic scattering are employed, where the center of mass angular distribution comes from the nuclear data file. The resulting physics matches the complicated expressions of [129].

Similarly considering the distribution of collision target velocities, this joint distribution is:

$$f(V,\mu) = Cv_r\sigma(v_r)M(T,V)f'(\mu) \tag{4.3}$$

where M is the Maxwell-Boltzmann distribution of target speeds:

$$M(T,V) = \frac{4}{\sqrt{\pi}}\beta^3 V^2 e^{-\beta^2 V^3} \tag{4.4}$$

and the variable with units of inverse velocity $\beta = \sqrt{\frac{A}{2kT}}$ parameterizes the target velocities, and the distribution $f'(\mu)$ is a uniform distribution between -1 and 1. The other variables are C, the distribution's normalizing constant; v_r, the relative speed of the neutron with respect to the target; σ, the zero kelvin scattering cross section; A, the target mass; k, the Boltzmann constant; T, the absolute temperature; and V, the target speed.

Classically, the approximation that $\sigma(v_r)$ is constant has been employed. However, it was shown in [130], [131] that this approximation is incorrect in the vicinity of resonances, where $\sigma(v_r)$ varies over a few orders of magnitude, preferentially causing scattering with targets of relative velocity more closely matching the scattering resonance peaks. The various aforementioned techniques are all just methods for sampling from the distribution of Eq. 4.3 with arbitrary forms of the function $\sigma(v_r)$.

More specific knowledge about the form of $\sigma(v_r)$ can be employed. It has been shown extensively [143], [144] at this point that the cross section is accurately represented as a sum of poles in addition to a low order Laurent expansion ($N \approx 7$), vis:

$$\sigma(E) = \frac{1}{E}\left(\Re\left[\sum_j \frac{r_j}{p_j - \sqrt{E}}\right] + \sum_{n=0}^{N} a_n E^{n/2}\right) \tag{4.5}$$

In fact, for the purposes of sampling the resonance upscattering effect, we claim and later numerically demonstrate that the narrow range of attainable v_r leads the zero-kelvin cross section to be accurately represented as a single pole and a linear term, over a sufficiently

narrow range of energies:

$$\sigma(E) = \frac{1}{E}\Re\left[\frac{r_j}{p_j - \sqrt{E}}\right] + \sigma_0 + \sigma_1\sqrt{E} \tag{4.6}$$

The σ_0 and σ_1 terms are calculated through a linearization process to avoid some complexities with higher order polynomial fitting. An algorithm for finding the best values of σ_0 and σ_1 is presented in section 4.2.4.

With this approximation, we use the technique developed in [133]: rather than attempting to sample the target speed (V) and direction cosine (μ), one instead samples first the relative velocity v_r, and then samples μ from the distribution of μ conditioned on v_r. The distribution in this form for projectile speed v is thus [133]:

$$P(v_r|v) = \left(e^{-\beta^2(v-v_r)^2} - e^{-\beta^2(v+v_r)^2}\right) v_r^2\sigma_0(v_r) \tag{4.7}$$

Next, as previously shown for Doppler-broadening of the Windowed Multipole format [71] and the original full multipole format [145], we note the term $e^{-\beta^2(v+v_r)}$ to be negligible and use the following:

$$P(v_r|v) \approx e^{-\beta^2(v-v_r)^2} v_r^2\sigma_0(v_r) \tag{4.8}$$

At this point, we write the multipole cross section in terms of the relative velocity rather than in terms of energy. The nonrelativistic approach is quite accurate for neutrons in the epithermal range. One must mind the fact that resonances tend to be of the same width at high energy, so the resonance upscatter effect still exists at high energy where relativistic formulae should be employed. However, two effects cause resonance scattering at high energies to be negligible. Firstly, the scattering kernel is wide compared to the widths of resonances. Secondly, due to the $1/E$ spectrum encountered in thermal reactors where the resonance upscatter effect has a tangible influence, less neutrons make collisions at these energies in the first place. Therefore, high energy resonance-influenced scattering can be entirely neglected, as it has been over 1 keV in the past [99] and given excellent results. That being said, the formula to use is:

$$\sigma(v_r) = \frac{2}{m_n v_r^2}\Re\left[\frac{r_j}{p_j - \sqrt{2m_n}v_r}\right] + \sigma_0 + \sigma_1\sqrt{2m_n}v_r \tag{4.9}$$

If we define the auxiliary variables $x = \beta(v_r - v)$ and $y = \beta v$, and insert Eq. 4.9 in the marginal target collision rate distribution in terms of relative speed, Eq. 4.8, some algebra

reveals that:

$$p(x|y) = e^{-x^2}\left(\Re\left[\frac{\beta r_j}{z-x}\right] + \beta^{-2}(x+y)^2\left(\sigma_0 + \sigma_1 x\right)\right) \tag{4.10}$$

where $z = \beta p_j - y$, which represents a dimensionless measure of the energy gap between the center of the Maxwell-Boltzmann distribution and the location of the resonance.

At this point, we can consider the CDF for the random variable x (dimensionless target velocity) conditioned on y (dimensionless projectile velocity). Integrating Eq. 4.10 yields:

$$CP(x|y) = \int_{-\infty}^{x} e^{-x'^2}\left(\Re\left[\frac{\beta r_j}{z-x'}\right] + \beta^{-2}(x'+y)^2\left(\sigma_0 + \sigma_1 x'\right)\right)\mathrm{d}x' \tag{4.11}$$

where C is the normalizing constant. After distributing the integral and interchanging the $\Re$ operator with integration, we obtain:

$$CP(x|y) = \Re\left[\frac{r_j\pi}{i\beta^{-1}}w(z,x)\right] + \frac{\beta^{-2}\sigma_0}{4}(-2e^{-x^2}(x+2y) + \sqrt{\pi}(1+2y^2)(1+\mathrm{erf}(x))) + \frac{1}{2\beta^2}\sigma_1 e^{-x^2}\left(1 + (x+y)^2 + \sqrt{\pi}y(1+\mathrm{erf}(x))\right) \tag{4.12}$$

where the normalizing constant for the distribution is:

$$C = \Re\left[\frac{r_j\pi}{i\beta^{-1}}w(z)\right] + \frac{\sqrt{\pi}}{2\beta^2}\left(\sigma_0(1+2y^2) + \sigma_1 y\right) \quad , \tag{4.13}$$

which we point out is nothing more than the Doppler-broadened scattering cross section at temperature T under the single pole approximation.

The new special function we deem the "incomplete Faddeeva function" is defined as:

$$w(z,x) = \frac{i}{\pi}\int_{-\infty}^{x}\frac{e^{-t^2}}{z-t}\,\mathrm{d}t \tag{4.14}$$

And it can easily be seen that $w(z,\infty) = w(z)$ as per the definition of the Faddeeva function for $\Im[z] > 0$:

$$w(z) = \frac{i}{\pi}\int_{-\infty}^{\infty}\frac{e^{-t^2}}{z-t}\,\mathrm{d}t \tag{4.15}$$

Indeed, for this application, $\Im[z] > 0$ and we maintain this assumption going forward. A specialized root finder has been developed to quickly invert this CDF and consequently sample the relative velocity. This thus constitutes a method to sample target velocities without rejection sampling or extensive tables.

The novelty in our approach lies entirely in the treatment of the zero kelvin cross section

and analytical representation of the relative speed cumulative distribution. After sampling from the relative speed distribution, we must sample the target polar angle distribution conditioned on the relative speed as done in [133]. For completeness, we conclude with the CDF of the target speed:

$$C(V|v_r) \propto \begin{cases} 0 & V \leq |v_r - v| \\ 1 - e^{-\beta^2 V^2} & |v_r - v| < V < v_r + v \\ 1 & v_r + v \leq V \end{cases} \tag{4.16}$$

for which [133] provides a straightforward sampling technique. The remaining work is purely numerical, particularly in requiring an efficient, reasonably accurate algorithm for the incomplete Faddeeva function $w(z, x)$.

4.2.1 The Incomplete Faddeeva Function

The forthcoming discussion explores the properties of the incomplete Faddeeva function, with a particular focus on properties which can be leveraged to obtain efficient numerical approximations to it. We advise the reader that absorbing this section in depth is not necessary to grasp the claims and algorithms made herein. Rather, this section stands as a necessary mathematical building block that constitutes our new method and can be viewed as a black box by the uninterested reader.

The incomplete Faddeeva function, as defined by Eq. 4.14 is $w : \mathbb{C} \times \mathbb{R} \to \mathbb{C}$. This section attempts to build some intuition as to how this function behaves as z and x individually vary. In resonance upscatter treatment, z parametrizes the location, height, and width of the resonance with respect to the Maxwell-Boltzmann distributed velocities. Values of $\Re[z] = 0$ correspond to scattering resonances exactly situated at the mean component of relative speed along the neutron's line of flight predicted by the Maxwell-Boltzmann distribution. Values of $\Re[z] > 0$ correspond to resonances at higher energies than the particle's energy, and therefore induce preferential scattering with relative velocities higher than the incident velocity, and vice-versa for $\Re[z] < 0$. x parametrizes the dimensionless target relative speed. Small values of $\Im[z]$ imply tall, narrow resonances, with a limit of $\Im[z] = 0$ being a singularity representing a resonance of infinite cross section. Large values of $\Im[z]$ model wider, weaker resonances.

Fig. 4.1 plots the incomplete Faddeeva function as a function of z for a few values of x, and Fig. 4.2 shows the behavior of its real part in x for a few values of z. Both illustrate the rapidly varying behavior of this function for small values of $\Im[z]$ when $\Re[z] \approx x$, where a sharp peak follows the value of x near the real line. Fig. 4.1 particularly shows the

approach to the familiarly shaped $w(z)$ as $x \to \infty$. The figures also show that $\Re[w(z,x)]$ is an increasing function in x, as would be expected of a cumulative distribution function. This behavior is explained by the following asymptotic analysis.

Humlíček [146] provides the following asymptotic formula for the Faddeeva function without derivation:

$$w(z) \approx \frac{i}{\sqrt{\pi} z} \quad . \tag{4.17}$$

Considering the definition of the Faddeeva function once more in Eq. 4.15, This approximation results from supposing that if z is large, and that only values of t near zero contribute to the integral, we can approximate $w(z)$ as:

$$w(z) \approx \frac{i}{\pi} \int_{-\infty}^{\infty} \frac{e^{-t^2}}{z} \, \mathrm{d}t \tag{4.18}$$

which immediately yields the asymptotic estimate of Eq. 4.17. Proceeding with the same approximation in the context of the incomplete Faddeeva function, we thus obtain:

$$w(z,x) \approx \frac{i}{2\sqrt{\pi} z} \left(\operatorname{erf}(x) + 1\right) \qquad \text{if} \quad |z| \gg 1 \tag{4.19}$$

Suggesting a close connection between the incomplete Faddeeva function's behavior in x and the error function. In fact, this hunch is confirmed by the following identity which connects the incomplete Faddeeva to the standard Faddeeva function:

$$w(z,x) = \frac{1}{2}(1 + \operatorname{erf}(x)) w(z) + \frac{i e^{-x^2}}{\pi} \int_0^{\infty} \frac{e^{-t^2} e^{2itz} \, \mathrm{d}t}{i(x - z) + t} \tag{4.20}$$

Proof of this relation is provided in Appendix A.

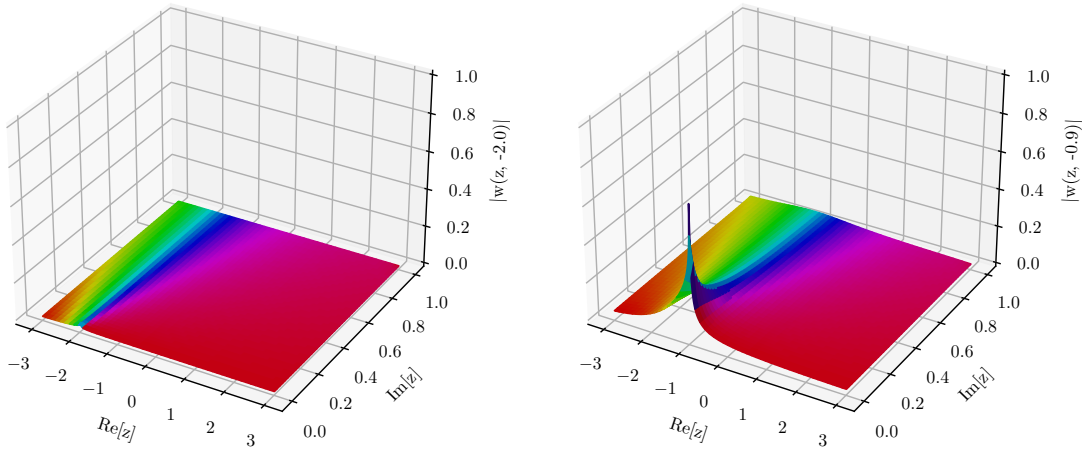
An even more accurate asymptotic estimate can be obtained for large $\Re[z]$, approximating the pole term as

$$\frac{1}{z - t} \approx \frac{1}{2z} \left(1 + e^{2t/z}\right) \quad . \tag{4.21}$$

This matches the value, slope, and curvature with respect to t of the pole about $t = 0$. Substituting this back to Eq. 4.14 and adjusting the expression such that $\Re[w(z,x)]$ is strictly increasing (as suggested by Eq. 4.19), and matching the asymptotic value of $w(z)$ as suggested by Eq. 4.20, results in

$$w(z,x) \approx \frac{1}{2} \left(1 + \operatorname{erf}\left(x - \Re[z]^{-1}\right)\right) w(z) \quad . \tag{4.22}$$

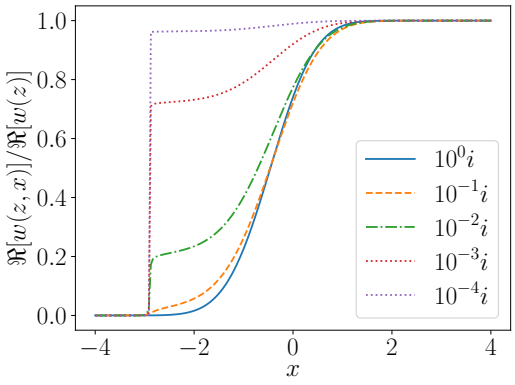
Eq. 4.22 is sufficiently accurate to be used in practical computations, as shown by Fig. 4.3.



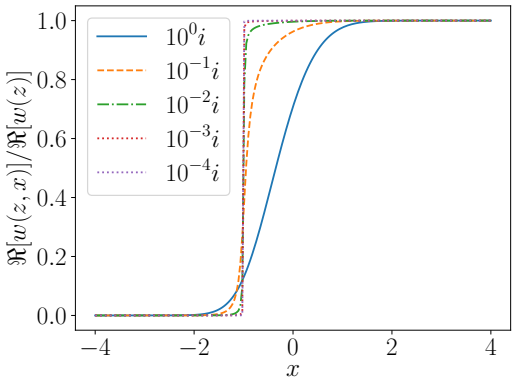
(a) $w(z, -2.0)$ (b) $w(z, -0.9)$

(c) $w(z, 0.5)$ (d) $w(z, 2.0)$

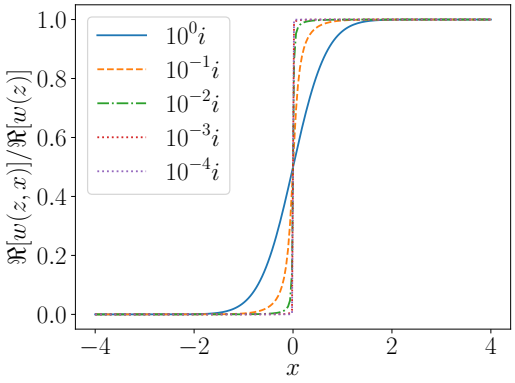
Figure 4.1: $w(z, x)$ for a few values of x. The height represents the magnitude, and coloring is done by phase.



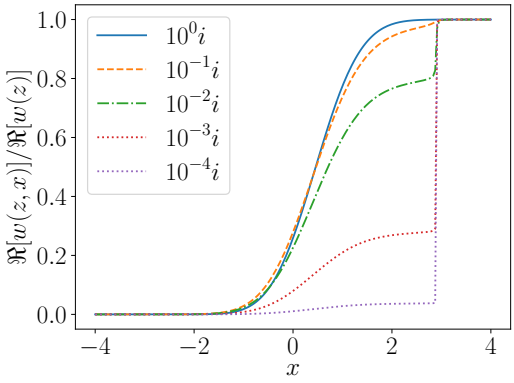
(a) $\Re[z] = -2.9$



(b) $\Re[z] = -1.0$



(c) $\Re[z] = 0.0$



(d) $\Re[z] = 2.9$

Figure 4.2: $\Re[w(z,x)]$ for a few values of z. The legend is the imaginary number added to the real part specified in each figure's caption. The plotted quantity is normalized by $\Re[w(z)]$ so all lines tend to unity as x grows.

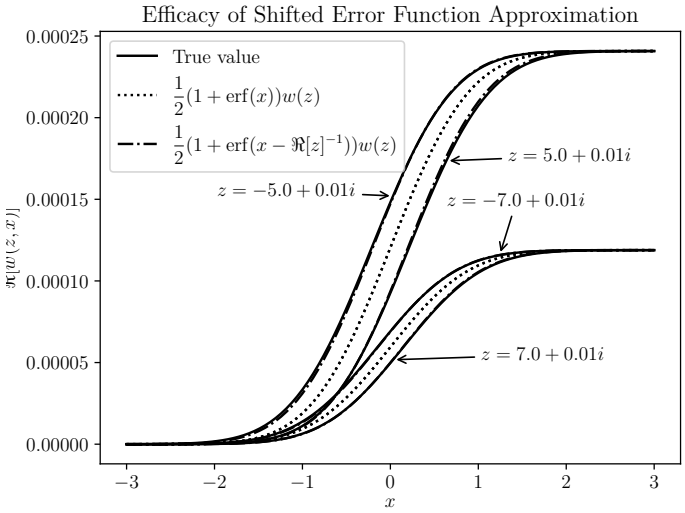


Figure 4.3: Accuracy of the $w(z,x)$ approximation for $|\Re[z]| > 5$. An unshifted error function is shown for comparison, representing a more naive asymptotic approximation to $w(z,x)$.

In the context of resonance scattering, Eq. 4.22 shows that the relative speed distribution gets shifted forward by a nondimensionalized factor of $1/\Re[z]$ with its influence scaling by $\Re[r_j w(z)]$ times the resonances residue as suggested by Eq. 4.12.

Another useful property of the incomplete Faddeeva function is a simple connection between its derivative in the complex plane and its value. This is similar in nature to the derivative of the Faddeeva function [98]:

$$\frac{\mathrm{d}w(z)}{\mathrm{d}z} = \frac{2i}{\sqrt{\pi}} - 2zw(z) \tag{4.23}$$

The relation we have obtained generalizes this as:

$$\frac{\mathrm{d}w(z,x)}{\mathrm{d}z} = \frac{i}{\sqrt{\pi}}\left(1 + \mathrm{erf}(x) + \frac{e^{-x^2}}{\sqrt{\pi}(x-z)}\right) - 2zw(z,x) \tag{4.24}$$

which clearly maintains consistency with Eq. 4.23 as $x \to \infty$. This fairly simple connection between the derivative of $w(z,x)$ and its value can be utilized for efficient sensitivity analysis of the scattering kernel with respect to windowed multipole parameters.

The forthcoming discussion presents some further concepts in the direction of efficient numerical evaluation of the incomplete Faddeeva function. Going forward, we denote the

second integral appearing in Eq. 4.20 as:

$$I(z,m) = \int_0^\infty \frac{e^{-t^2} e^{2itz}\,\mathrm{d}t}{im+t} \tag{4.25}$$

where $\mathbb{R} \ni m = x - z$. While it may seem that a half-range Gauss-Hermite quadrature may work well to efficiently approximate Eq. 4.25, this is not the case. Firstly, complex exponentials would have to be calculated at each quadrature point. Secondly, as the real part of z grows, the integrand oscillates more. In practice, values of $\Re[z] > 10$ are frequently encountered, and low degree quadratures would not capture the oscillation. Additionally, given that the value of $\Im[z]$ is small, the denominator becomes nearly singular when $x \approx \Re[z]$. In fact, when $\Im[z] = 0$, $I(z,m)$ becomes a discontinuous function in x when interpreted as a principal value integral.

Eq. 4.25 is equivalent to the *incomplete Goodwin-Staton integral*, referred to in [147]. However, to our knowledge, only asymptotic analysis has been performed on this type of integral before, without any development of numerical routines. Recent work in the field of finance [148] presents results for computing what the authors define as the *extended incomplete Goodwin-Staton integral*, for which the $\nu = 1$ case is of interest in the present discussion. Unfortunately, the authors' numerical method works for all cases except $\nu = 1$, suggesting Eq. 4.25 to be of a fundamentally different nature.

In our experience, the difficulty with large $\Re[z]$ cannot be ameliorated by a stationary phase technique [149], as these tend to accentuate the pole behavior and remain of similar difficulty for half range Gauss-Hermite quadrature.

$I(z,m)$ satisfies a scaling property:

$$I(z,m) = I(az, m/a) \tag{4.26}$$

where $a \in \mathbb{R}$.

In the windowed multipole method, $\Im[z]$ is near zero, as shown in Fig. 4.4. Therefore, we can expect to frequently encounter nearly singular integrands in Eq. 4.25. An efficient numerical technique which explicitly treats this behavior can be devised by first noticing that $I(z,m)$ satisfies this differential equation in the complex plane:

$$\frac{\mathrm{d}I}{\mathrm{d}z} + 2zI = \frac{1}{x-z} \tag{4.27}$$

This can be used to connect the value $I(z_0, x)$ at a point z_0 to another point z_1. The

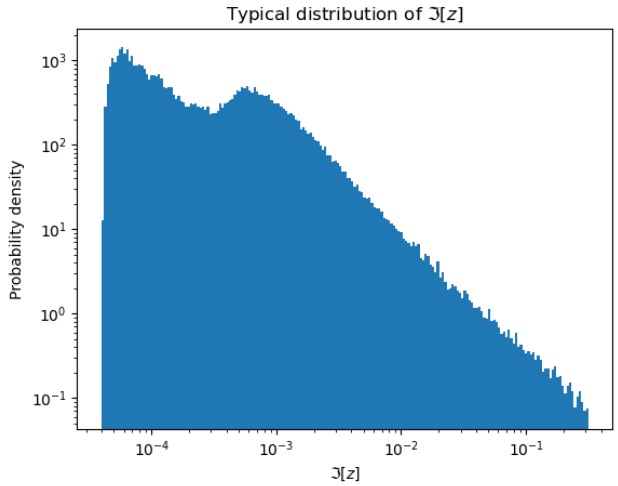


Figure 4.4: Distribution of imaginary part of the poles in the windowed multipole method. These were collected from every nuclide in OpenMC's regression testing dataset, based on ENDFVII.1.

integrating factor technique shows that:

$$I(z_1, x) = \int_{z_0}^{z_1} \frac{e^{t^2 - z_1^2}\, \mathrm{d}t}{x - t} + e^{z_0^2 - z_1^2} I(z_0, x) \tag{4.28}$$

If the distance between z_0 and z_1 is small, the exponential in the numerator of Eq. 4.28 can be Taylor expanded about z_0 to yield an efficient numerical scheme.

We have found that the imaginary part of $I(\Re[z], x)$ can be calculated with a closed-form, discontinuous-in x formula. Because the nearly discontinuous behavior of $I(z, x)$ in x is the main source of difficulty here, Eq. 4.28 can be used to resolve this behavior accurately after calculating the value of $I(\Re[z], x)$. The real part of $I(\Re[z], x)$ is continuous in x but not available in formula in terms of elementary functions; however, it is readily amenable to numerical approximation. In conclusion, the real and imaginary part of $I(\Re[z], x)$ contrast each other: the former is readily numerically approximated by series or similar methods, whereas the latter is discontinuous and therefore not amenable to series or rational function approximation, and fortunately has an exact formula.

The first goal at hand is to calculate $\Im[I(z, m)]$ for real values of z. This can be written as:

$$\Im[I(z, m)] = \int_0^{\infty} \frac{e^{-t^2} \left(t \sin(2zt) - m \cos(2zt)\right) \mathrm{d}t}{t^2 + m^2} \quad . \tag{4.29}$$

The linearity of integration can be distributed over both trigonometric terms. Each of the resulting integrals can be found as respective sine and cosine transform integrals, which are

listed in [150]. Combining results from the sine and cosine transform tables yields:

$$\Im[I(z,m)] = \pi e^{-z^2+(m+z)^2}\left(\frac{1}{2}\text{erf}(m+z) - \text{sign}(m)\right) \quad z \in \mathbb{R} \tag{4.30}$$

Next, we can find an approximation for the real part of $I(z,m)$:

$$\Re[I(z,m)] = \int_0^\infty \frac{e^{-t^2}\left(t\cos(2zt) + m\sin(2zt)\right)\mathrm{d}t}{t^2+m^2} \quad . \tag{4.31}$$

This expression can neither be written terms of elementary nor special functions, to our knowledge. In order to manipulate it to obtain a numerical expression, consider the auxiliary function:

$$R(z,m) = \int_0^\infty \frac{e^{-t^2}\sin(2zt)\,\mathrm{d}t}{t^2+m^2} \tag{4.32}$$

which again, of course, cannot be represented in terms of elementary or special functions. This pinpoints the difficulty in calculating the real part of $I(z,m)$ because:

$$\Re[I(z,m)] = mR(z,m) + \frac{1}{2}\frac{\partial R}{\partial z} \quad . \tag{4.33}$$

We thus seek a straightforwardly differentiable approximation to $R(z,m)$. The intuition behind our forthcoming numerical approximation to $R(m,z)$ comes from the fact that for $m \gg 1$, t is negligible in the denominator compared to m over the range where the e^{-t^2} weighting is large, hence

$$R(m,z) \approx \frac{1}{m^2}\int_0^\infty e^{-t^2}\sin(2zt)\,\mathrm{d}t = \frac{1}{m^2}F(z) \tag{4.34}$$

where $F(z)$ is the Dawson F function [98]. By standard asymptotic analysis, matching the z derivative at $z=0$ and the $z \gg 1$ asymptote for the Dawson F function results in the improved estimate:

$$R(m,z) \approx \frac{1}{m}\sqrt{e^{m^2}E_1(m^2)}F\left(mz\sqrt{e^{m^2}E_1(m^2)}\right) \tag{4.35}$$

which is accurate to within about 5% across the full range of m values. However, this level of accuracy is not appropriate for engineering calculations.

This led us to pursue approximations to $R(m,z)$ of the form:

$$R(m,z) \approx \frac{1}{m}g(m,z)F\left(mzg(m,z)\right) \tag{4.36}$$

which guarantees the correct large z behavior and small z behavior when $g(m,0) = \sqrt{e^{m^2} E_1(m^2)}$. However, we were not able to find success in this direction.

To build more intuition for $R(m,z)$, its close relation to the Dawson F function is evinced by considering the Maclaurin series in z for both:

$$F(z) = z - \frac{2}{3}z^3 + \frac{4}{15}z^5 - \frac{8}{105}z^7 + \cdots \quad , \tag{4.37}$$

$$R(m,z) = e^{m^2}\left(E_1(m^2)z - \frac{2}{3}E_2(m^2)z^3 + \frac{4}{15}E_3(m^2)z^5 - \frac{8}{105}E_4(m^2)z^7 + \cdots\right) \tag{4.38}$$

which further highlights the utility of maintaining consistency of $R(m,z)$ with its asymptotic sister $F(z)$. The series are the same, but with the addition of exponential integral multiplying factors in $R(m,z)$.

In order to find such a numerical relation which maintains this consistency in the asymptotic case, Eq. 4.32 can be transformed via interchange of differentiation and integration. Consider the generalized function:

$$R(m,z,\alpha) = e^{m^2}\int_0^\infty \frac{e^{-\alpha(t^2+m^2)}\sin(2zt)\,\mathrm{d}t}{t^2+m^2} \tag{4.39}$$

Differentiating reveals that:

$$\frac{\partial R(m,z,\alpha)}{\partial \alpha} = -e^{m^2}\int_0^\infty e^{-\alpha(t^2+m^2)}\sin(2zt)\,\mathrm{d}t = -\frac{e^{-m^2(\alpha-1)}}{\sqrt{\alpha}}F\left(\frac{z}{\sqrt{\alpha}}\right) \tag{4.40}$$

The fundamental theorem of calculus then applies:

$$R(m,z,\infty) - R(m,z,1) = e^{m^2}\int_1^\infty \frac{1}{\sqrt{\alpha}}e^{-m^2\alpha}F\left(\frac{z}{\sqrt{\alpha}}\right)\mathrm{d}\alpha \tag{4.41}$$

Using the fact that $R(m,z,\infty) = 0$ and doing a change of variables, Eq. 4.32 becomes:

$$R(m,z) = 2e^{m^2}\int_1^\infty e^{-m^2t^2}F(z/t)\,\mathrm{d}t \tag{4.42}$$

which confirms our hunch about the close relation of $F(z)$ to $R(m,z)$; it is an infinite superposition of stretched and scaled Dawson F functions.

This formulation yields useful numerical results by applying Gauss-Laguerre quadrature to Eq. 4.41, amounting to approximation by a sum of Dawson F function. However, this approach fails for small values of m. Instead, we have found success in inserting an approximation for $F(z)$ to Eq. 4.42. Well-known approximations to $F(z)$ based on rational

expressions and other elementary functions [151], [152] do not result in numerically useful expressions. However, noting that:

$$\int_1^\infty e^{-m^2t^2}(z/t)^{(2k+1)}\,\mathrm{d}t = \frac{1}{2}z^{1+2k}E_{1+k}(m^2) \tag{4.43}$$

it can be seen that approximations to $F(z)$ in the form a power series can be computationally efficient in light of the recursion relation for exponential integrals:

$$ne^{m^2}E_{n+1}(m^2) = (1 - m^2e^{m^2}E_n(m^2)) \tag{4.44}$$

Careful attention must be paid to the floating point properties of this relation [153]. The magnification of computational errors grows arbitrarily large, and we later present a specialized numerical algorithm guaranteeing floating point stability.

In order to thus obtain a simple, efficient approximation to $R(m, z)$, we employ the Chebyshev expansion valid for $z \in [-5, 5]$ presented in [154]. The results of our method could be improved by using a finer piecewise division for the Chebyshev expansion of $F(z)$ as in [155], but we have used the present approach for simplicity of implementation. Thus, if $F(z)$'s truncated Chebyshev expansion is converted to the power series basis:

$$F(z) \approx \sum_{i=0}^{n} c_n(z/5)^{2*i+1} \tag{4.45}$$

we obtain approximations of the form

$$R(m, z) \approx \sum_{i=0}^{n} \frac{c_n}{2}E_{1+i}(m^2)(z/5)^{2*i+1} \quad . \tag{4.46}$$

For values $|z| > 5$, the integration can be split into two ranges. In the first $t \in [1, z/5]$, the asymptotic formula for $F(z)$ is employed:

$$F(z) \approx \frac{1}{2z} - \frac{1}{4z^3} + \frac{3}{8z^5} + \cdots \quad . \tag{4.47}$$

For the remaining range of integration $t \in [z/5, \infty)$, the truncated Chebyshev expansion in a power series basis is again employed. However, we have found the simplicity and accuracy of Eq. 4.22 to be appropriate for resonance upscatter applications.

4.2.2 Numerical Implementation

Stable, Efficient Calculation of an E_n Sequence

The magnification of error in the forward recurrence relation for exponential integrals from [153] is:

$$|\rho_n| = \frac{x^n E_1(x)}{n! E_{n+1}(x)} \tag{4.48}$$

Notably, the error magnification of the reverse recurrence relation is the reciprocal of this quantity. Moreover, $|\rho_n|$ is a function increasing from 1, reaching a maximum, and monotonically descending below 1 [153]. As a consequence, a critical index n^* exists such that iterating outward from it results in a numerically stable recursion algorithm. In terms of evaluating Eq. 4.46, this means splitting the polynomial in z into parts above the index n^* and those below. After computing $e^{m^2} E_{n^*}(m^2)$, Horner's method is used in the reverse recurring relation down to the term of order z, and forward recursion is employed to evaluate the polynomial of degree leading from $2n^* + 1$ up to $2n + 1$.

A simple result we have obtained is that the smallest value of n^* such that:

$$\frac{x^{n^*} E_1(x)}{n^*! E_{n^*+1}(x)} < 1 \tag{4.49}$$

is well approximated by:

$$n^* \approx ex - \frac{1}{2}\log \pi \quad . \tag{4.50}$$

Appendix D shows how this can be obtained. Fig. 4.5 illustrates the accuracy of Eq. 4.50. Using this information, algorithm 10 explicitly states the procedure to calculate $R(m, z)$ and its derivative with respect to z. While the use of a power basis polynomial is sub-optimal, numerically speaking, the main source of numerical error in this scenario originates from the recursive exponential integral formula. The specification of the algorithm assumes that an accurate method for computing $E_n(x)e^x$ has been provided, which is well documented in many other works. We have employed a C++ adaptation of the continued fraction approximation employed by the Cephes library [156], which is documented in [98].

The Jump Integral

After computing the value of $I(\Re[z], m)$, the differential equation of Eq. 4.27 which I follows can be used to calculate $I(z, m)$. We calculate $I(z, m)$ in this manner due to the nearly discontinuous behavior of $I(z, m)$; it has a jump discontinuity about $m = 0$ if $z \in \mathbb{R}$. Because the imaginary part of z is small in windowed multipole libraries, the resulting behavior is

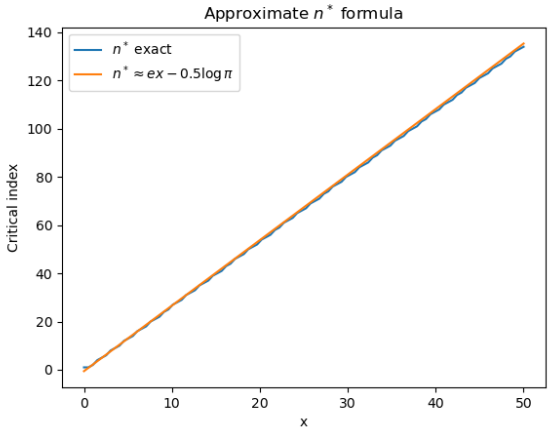


Figure 4.5: Approximate solution to finding the first value of n such that $\frac{x^n E_1(x)}{n! E_{n+1}(x)} < 1$.

nearly discontinuous and hence is not captured efficiently by general approximation techniques; finely resolved tables or high polynomial orders would be required. Our approach thus resolves the discontinuous component exactly with the piecewise function Eq. 4.30. The nontrivial part of Eq. 4.28 is the transcendental integral:

$$J(z, x) = e^{-\Re[z]^2} \int_{\Re[z]}^{z} \frac{e^{t^2}\, \mathrm{d}t}{x - t} \tag{4.51}$$

We have deemed this term the *jump integral* because it allows jumping from values of $I(z, m)$ on the real line to values above the real line in the complex plane. While it seems that our issue of approximating the transcendental integral $w(z, x)$ has seemingly not been heretofore ameliorated due to the appearance of yet another transcendental integral Eq. 4.51, a change of variables puts it into a form suitable for numerical approximation:

$$J(z, x) = \int_{0}^{i\Im[z]} \frac{e^{u^2 + 2u\Re[z]}\, \mathrm{d}u}{m - u} \tag{4.52}$$

where again, $m = \Re[z] - x$. Because $\Im[z]$ is small as shown by Fig. 4.4, the argument to the exponential term is similarly small. Where this integral is well-defined ($m \neq 0$), the exponential term can be expanded in its Maclaurin series and integrated term by term:

$$e^{u^2 + 2u\Re[z]} = \sum_{n=0}^{\infty} \frac{a_n}{n!} u^n \quad . \tag{4.53}$$

It is verified that the coefficients a_n satisfy the two-term recurrence:

$$a_{n+1} = 2\Re[z]a_n + 2(n-1)a_{n-1}; \quad a_0 = 1; \quad a_1 = 2\Re[z] \tag{4.54}$$

Next, the term-by-term integrals appear in the form:

$$\int_0^{i\Im[z]} \frac{u^n \, \mathrm{d}u}{m-u} = m^n B_{i\Im[z]/m}(1+n, 0) \tag{4.55}$$

Where $B_x(\cdot, \cdot)$ is the incomplete beta function, defined as:

$$B_x(a, b) = \int_0^z t^{a-1}(1-t)^{b-1} \, \mathrm{d}t \quad . \tag{4.56}$$

A recursion formula derived as a special case of formulas in [98] efficiently calculates these incomplete beta function values of higher n in sequence:

$$B_x(n+1, 0) = B_x(n, 0) - \frac{x^n}{n} \tag{4.57}$$

In combination with the fact that:

$$B_x(1, 0) = -\log(1-x) \quad , \tag{4.58}$$

this yields an efficient numerical scheme for evaluating an integral of the truncated Maclaurin series of the exponential of Eq. 4.52. Algorithm 11 details the combination of all of these facts for an efficient approximation to $J(m, z)$. This approximation works very well for problems with $|\Re[z]| \leq 5$, which easily covers the range of scattering events where resonances appreciably affect the double differential at temperature. Outside of that range, the integral becomes increasingly oscillatory, so an asymptotic approximation is employed for $|z| > 5$. This approximation is documented in Appendix F.

Lastly, Algorithm 12 gives the overall algorithm to compute $w(z, x)$ efficiently. It relies on access to some implementation of calculating $w(z)$, e.g. the permissively licensed [121] which implements a variety of approximations to achieve high accuracy, or one of the various rational approximations [146], [157], [158] when higher error is permitted. Regardless of the chosen $w(z)$ implementation, our algorithm maintains asymptotic consistency such that $\lim_{x\to\infty} w(z, x) = w(z)$. This work leverages a recent approximation tailored for WMP [123].

Input : $m \in \mathbb{R}$, $z \in \mathbb{R}$
Output: $R(m, z)$ from Eq. 4.32 and $\frac{\partial R}{\partial z}$
Data: c_n coefficients of Eq. 4.45, $1 \leq n \leq n_{max}$
$n^* \leftarrow \min(\max(em^2 - 0.57, 1), n_{max})$;
expnexp $\leftarrow E_{n^*}(m^2)e^{m^2}$;
expnexp_orig $\leftarrow$ expnexp;
result $\leftarrow 0$;
derivative $\leftarrow 0$;
`// Backward recurse with Horner scheme`
for $n \leftarrow n^* - 1$ **to** 1 **do**
  expnexp $\leftarrow (1 - n \cdot \text{expnexp})\,/m^2$;
  derivative $\leftarrow$ derivative $\cdot$ pow$(z/5, 2)+$ result;
  result $\leftarrow$ (result)pow$(z/5, 2) +$ expnexp $\cdot\, c_{n-1}$;
end
derivative $\leftarrow$ 2derivative $\cdot$ pow$(z/5, 2)$;
derivative $\leftarrow$ derivative $+$ result ;
result $\leftarrow$ result $\cdot\, z/5$;
`// Forward recursing polynomial evaluation`
x2 $\leftarrow$ pow$(z/5, 2n^* - 1)$;
x1 $\leftarrow$ pow$(z/5, 2(n^* - 1))$;
expnexp $\leftarrow$ expnexp_orig;
for $n \leftarrow n^*$ **to** n_{max} **do**
  result $\leftarrow c_{n-1}$·expnexp $\cdot$ x2;
  derivative $\leftarrow c_{n-1}$·expnexp $\cdot$ x1 $\cdot(2n - 1)$;
  x2 $\leftarrow x2 \cdot$ pow$(z/5, 2)$;
  x1 $\leftarrow x1 \cdot$ pow$(z/5, 2)$;
  expnexp $\leftarrow (1 - m^2 \cdot \text{expnexp})/n$;
end
return (result, derivative);

Algorithm 10: Stable $R(m, z)$ approximation for $|z| \leq 5$. Appendix E is used for $|z| > 5$.

Input : $x \in \mathbb{R}$, $z \in \mathbb{C}$
Output: $J(m, z)$ from Eq. 4.51
result $\leftarrow 0$;
a0 $\leftarrow 1.0$;
a1 $\leftarrow 2\Re[z]$;
result $\leftarrow$ b1 $\cdot$ a0;
b1 $\leftarrow b1 - i\Im[z]/(x - \Re[x])$;
// n_{max} adjusts the number of truncated series terms
for $n \leftarrow 2$ **to** n_{max} **do**
  tmp $\leftarrow 2\Re[z] \cdot$ a1 $+ 2(n-1) \cdot$ a0;
  a0 $\leftarrow$ a1;
  a1 $\leftarrow$ tmp;
  result $\leftarrow$ result $+$ b1 $\cdot$ a1$\frac{m^n}{n!}$;
  b1 $\leftarrow$ b1 $- \frac{1}{n+1}\left(\frac{i\Im[z]}{(x-\Re[x])}\right)^n$;
end
return result;

Algorithm 11: Efficient $J(m, z)$ Approximation for $\Im[z] < 1$ and $|\Re[z]| < 5$. Appendix F describes the approximation for $|\Re[z]| > 5$.

Input : $z \in \mathbb{C}$, $x \in \mathbb{R}$
Output: $w(z, x)$ from Eq. 4.14
m $\leftarrow x - \Re[z]$;
// $R(m, z)$ integral and its derivative
rmz, drmzdz $\leftarrow$ call(Alg. 10);
// $I(m, z)$ integral
imz $\leftarrow m \cdot$ rmz $+ \frac{1}{2}$drmzdz $+ \frac{i\pi}{2} e^{-\Re[z]^2 + x^2}\left(\text{erf}(x) - \text{sign}(m)\right)$;
// $J(m, z)$ integral
ji $\leftarrow$ call(Alg. 11);
imz $\leftarrow$ imz$+$ ji;
imz $\leftarrow$ imz$\cdot e^{\Re[z]^2 - z^2}$;
result $\leftarrow \frac{i}{\pi} e^{-x^2} \cdot$ imz;
result $\leftarrow$ result $+ \frac{1}{2}\left(\text{erf}(x) + 1\right) w(z)$;
return result;

Algorithm 12: Efficient $w(z, x)$ approximation for $\Im[z] << 1$.

4.2.3 The Pole Sampling Approximation

A key approximation of our technique that enables its computational efficiency is viewing the multipole cross section in the relative speed distribution as a mixture distribution. The theoretical justification is that if poles are present and sufficiently close to the incident neutron energy ($|z| < 20$ specifically), the relative speed PDF is well-approximated by ignoring the polynomial contribution:

$$P(x|y) \approx e^{-x^2} \Re \left[\sum_{j \in W(\beta^{-2} y^2)} \frac{\beta r_j}{z_j - x} \right] \tag{4.59}$$

This expression is not employed to actually sample the scattering distribution. Rather, it is best viewed as a mixture distribution in which each pole contributes a probability proportional to:

$$\mathbb{P}[\sigma_s(x) = \frac{\beta r_j}{z - x} + \sigma_{0,j} + \sigma_{1,j} x] \propto \Re \left[r_j w(z_j) \right] \tag{4.60}$$

which defines a discrete distribution. In order to avoid the need for auxiliary storage and the calculation of a normalizing constant to this distribution, we recommend finding the maximum of $-|r_j w(z_j)| / \log(\xi_j)$ where ξ_j are uniform random numbers differing for each pole. The j corresponding to the maximum of this expression follows the desired discrete distribution. We also note that the quantity $\Re \left[r_j w(z_j) \right]$ is exactly the Doppler broadened contribution to the integrated scattering cross section, so this sampling procedure incurs no additional Faddeeva function evaluation overhead if this is done in tandem with a WMP cross section lookup operation.

Finally, we emphasize that the pole sampling approximation is not precisely consistent with the original multipole cross section representation. Instead, it uses the fact that polynomial contributions to the cross section negligibly affect the scattering kernel, while poles do so substantially. As a concluding remark, it must be emphasized here that the approximations made behind pole sampling are at the moment not mathematically rigorous. This stems from the neglect of the polynomial part of the cross section. Unfortunately, an exact treatment of the mixture distribution defined by the poles would require the piecewise integration of multiple energy windows, which would be computationally expensive. In particular, considering how the polynomial part of the cross section is given as a linearization in the selected resonance trough, the consistency of the cross section for various poles in our method is difficult to interpret. In the future, further theoretical work could likely justify the choices made here.

4.2.4 Finding the values of σ_0 and σ_1

While it may seem that the polynomial contribution to Eq. 4.6 could come as the first terms from the polynomials defined within the windows, as [142] used, we have found in practice that this choice is inconsistent with the approximation of the pole sampling technique and leads to negative cross section values.

To remedy this issue, we use the heuristic that the relative error of the cross section's local approximation is minimized by matching polynomial values in the vicinity of the dip of the resonance. The location of the scattering resonance trough is calculated as:

$$\sqrt{E}_{\text{trough}} = \frac{-b + \sqrt{b^2 - abc + a^2 d}}{a} \tag{4.61}$$

where

$$a = -\Im[\left(r_j^* - \bar{r_j^*}\right)] \tag{4.62}$$

$$b = -\Im[\left(\bar{r_j^*} p_j^* - r_j^* \bar{p_j^*}\right)] \tag{4.63}$$

$$c = -\Re[\left(\bar{p_j^*} + p_j^*\right)] \tag{4.64}$$

$$d = |p_j^*|^2 \tag{4.65}$$

$$\tag{4.66}$$

At this point, the window index of $\sqrt{E}_{\text{trough}}$ is calculated [1] . This may be a different window from the incident neutron energy's window. The windowed multipole cross section of Eq. 4.5 is then evaluated at $\sqrt{E}_{\text{trough}}$ but *excluding* the sampled pole p_j, i.e.

$$\sigma_0 = \frac{1}{E_{\text{trough}}} \Re \left[\sum_{j^* \neq j \in W(E_{\text{trough}})} \frac{r_j}{p_j - \sqrt{E_{\text{trough}}}} \right] + \sum_{n=0}^{N} a_n E_{\text{trough}}^{n/2} \tag{4.67}$$

From there, $\sigma_1 \approx \frac{\partial \sigma_s(E)}{\partial \sqrt{E}}$ is calculated at the same point, this time *only* including contributions from the polynomial expansion but not from any poles. This linearization technique has been found to improve the accuracy of our method when considering nuclides with tightly spaced

[1] The term under the square root in Eq. 4.61 can sometimes be negative in the vicinity of nonphysical poles which are artifacts of the fitting process, and dealing with imaginary quantities in this case is undesirable. Therefore, our calculation uses a linearization of the non-pole cross section at the incident energy instead in that case.

resonances such as ^{235}U. For complete clarity, the resulting expression is:

$$\sigma_1 = \sum_{n=0}^{N} a_n \frac{n-2}{2} E_{\text{trough}}^{n/2-2} \tag{4.68}$$

Finally, a linearization of the cross section in $\sqrt{E}$ space has been obtained. For use with the root finder, the nondimensional variable $\beta(\sqrt{E} - \sqrt{E_{\text{incident}}})$ is preferable, so σ_0 is appropriately shifted and σ_1 appropriately scaled.

4.2.5 Inverting the relative speed CDF

To sample from the relative speed PDF, we employ the CDF inversion technique. A naive attempt at this would be a few bisection root finding steps followed by a handful of Newton-like iterations. In practice, we've found that five bisection iterations followed by three Halley-Newton iterations resolves the root to within acceptable tolerance; however, a far more efficient root finder has been developed which takes a maximum of four iterations total, only requiring more work for unusual edge cases.

Inverting the CDF within a number of steps which is random but with limited upper bound is essential to obtaining acceptable performance on GPU, which is partly the goal of this method. A root finding technique which takes many iterations 1% of the time would be just as lackluster on GPU as rejection sampling. Hence, the *bootstrapping* step, as we call it, is essential to an efficient implementation of MARS. The bootstrapping step cheaply obtains an initial guess to the solution of the CDF inversion problem, from which a small number of Newton-like iterations improve the solution.

The key to doing so lies in finding a cheap approximation to the inverse of the CDF with general pole parameters. In order to do so, we first move from the root finding space of $x \in (-\infty, \infty)$ to the nonlinearly mapped variable $\tilde{x} = \frac{1}{2}(1 + \text{erf}x)$. The intuition behind using this modified space is that as the resonances become weak and the incident neutron energy becomes high, it can be shown that the CDF is simply equal to $\tilde{x}$ which ranges between zero and one. Resonances and low energy free gas effects simply act as perturbations to this linear function, which enables a good starting point for approximating the root location.

The next step in improving the CDF model in the mapped space is to observe that the contribution of collision probability from the resonance largely does not depend on its imaginary part. Increasing the imaginary part of the resonance broadens it and decreases its width. Therefore, the magnitude of the jump in $w(z, x)$ when x is near $\Re[z]$ quantifies the probability that the neutron experiences a collision near the peak of the resonance. The

jump in $w(z,x)$ for small $\Im[z]$ about $\Re[z]$ is approximately $e^{-\Re[z]^2}$,

$$\lim_{\epsilon\to 0}\frac{i}{\pi}\int_{\Re[z]-\epsilon}^{\Re[z]+\epsilon}\frac{e^{-t^2}\,\mathrm{d}t}{\Re[z]+i\epsilon-t}=\frac{1}{2}e^{-\Re[z]^2} \quad , \tag{4.69}$$

and therefore the probability contribution due to the resonance is approximately:

$$p_{\text{jump}} = ¶[x \approx \Re[z]] \approx \beta r_{j^*}\pi e^{-\Re[z]^2}/C \tag{4.70}$$

where C is the normalizing constant given by Eq. 4.13. Because this is an approximation, the probability of Eq. 4.70 may not be bounded between zero and one, so we threshold it to that range. In practice, the estimate provided here is accurate. We have found that this probability tends to be added into the CDF about $\Re[z]$ over the interval $[\Re[z]-\frac{3}{2}\Im[z], \Re[z]+\frac{3}{2}\Im[z]]$. This estimate could obviously be tuned for greater accuracy.

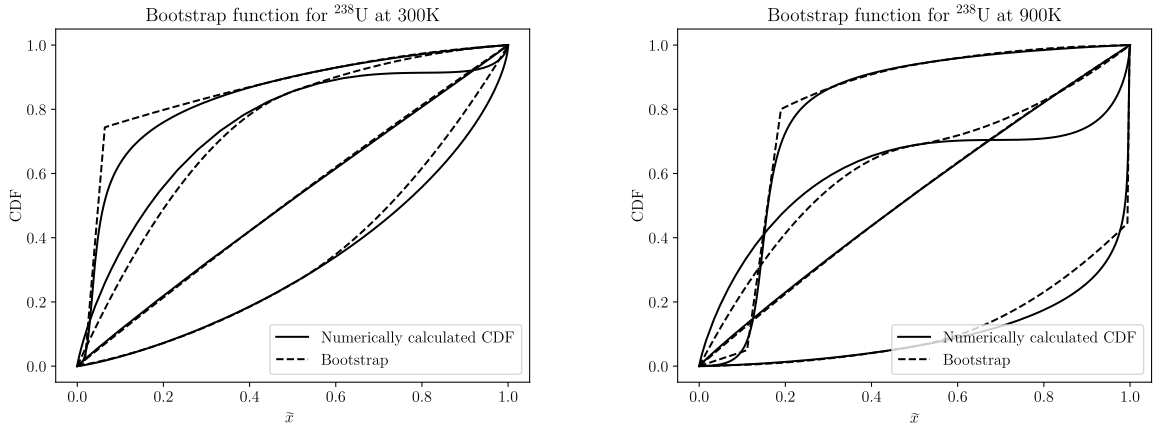
One final tool we employ to bootstrap the root finding process pertains to the values of the CDF about $x = 0$. In this case, numerous instances of functions occuring in its expression such as $\text{erf}(x)$ and e^{-x^2} take on easily calculated values. On top of that, the incomplete Faddeeva function has a closed form expression when $x = 0$:

$$w(z,0) = \frac{1}{2}w(z) + \frac{i}{2\pi}e^{-z^2}E_1(-z^2) \quad . \tag{4.71}$$

Because e^{-z^2} is already computed and cached for the CDF inversion, the calculation of a complex exponential integral is the only difficulty. This is much easier and computationally cheaper to do than the more involved $w(z,x)$ evaluation, so any off-the-shelf approximation of $E_1(-z^2)$ can be employed here.

Addtionally, the derivatives of the CDF with respect to x about $x = 0$ are also easily obtainable, which we use to further improve our rootfinding guess. So far, we have only incorporated information from the first derivative which has proven sufficient.

This leaves us with the following pieces of information from which the root estimate is extracted: the probability due to the resonance, its width, the value and slope of the CDF about $x = 0$ i.e. $\tilde{x} = 1/2$, and the known endpoint values of the CDF at 0 and 1. We therefore construct a function which is piecewise quadratic on the left and right of $\tilde{x} = 1/2$. This quadratic interval ranges to either the endpoints $\tilde{x} = 0$ or $\tilde{x} = 1$, or the resonance's upper or lower range of probability gain, estimated here as x$\in [\Re[z] - \frac{3}{2}\Im[z], \Re[z] + \frac{3}{2}\Im[z]]$. Note that this interval has to be mapped to an interval in $\tilde{x}$ space. Because the interval of the resonance is small, the Jacobian of the transformation $x \mapsto \tilde{x}$ which is proportional to $e^{-\Re[z]^2}$ (a quantity already computed) can be used to calculate the range in $\tilde{x}$ space.



(a) $T = 300$ kelvin

(b) $T = 900$ kelvin

Figure 4.6: The bootstrapping CDF provides a fairly accurate, easily invertible approximation to the true relative speed CDF to kickstart the root finding process. The pairs of lines, moving from top to bottom, represent 35.25, 36.25, 38.25, and 66.25 eV incident neutron energies.

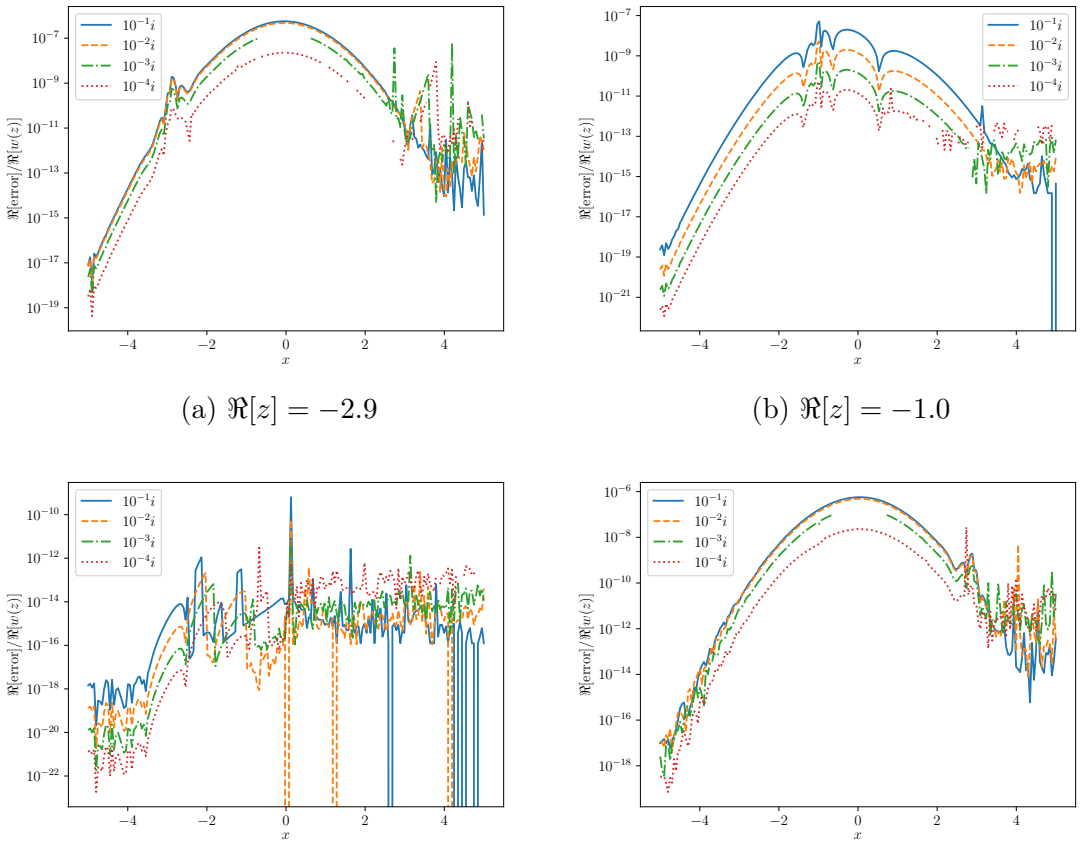
With this knowledge, the CDF can be approximated somewhat accurately in $\tilde{x}$ space. Despite the apparent complexity of what was just described, the inversion of the previous paragraph's function can be done using simple branching logic and, at worst, the solution of a quadratic equation. Because translating the inverse of the above function into code can take nontrivial effort, C++ code to achieve this has been provisioned in Appendix G. Figure 4.6 shows two examples of how this can be a quite satisfactory approximation of the CDF in $\tilde{x}$ space when resonances are influencing the scattering distribution.

After approximating the root we wish to find by the above procedure, we correct it with two Newton-Halley iterations. The Halley factor is damped according to the formula provided in *Numerical Recipes* [159].

4.3 Results

4.3.1 Calculation of $w(z, x)$

In order to test the accuracy of Alg. 12, we haved computed reference values of $w(z, x)$ using `scipy`'s [160] adaptive quadrature routine, `scipy.integrate.quad`, to evaluate the integral formulation Eq. 4.14. In approximation of the jump integral Eq. 4.51, only the first five terms in the series are retained. Where functions such as $\log(x)$ or e^x appear, C++ standard library implementations have been employed. The implementation of $w(z)$ from



(a) $\Re[z] = -2.9$

(b) $\Re[z] = -1.0$

(c) $\Re[z] = 0.0$

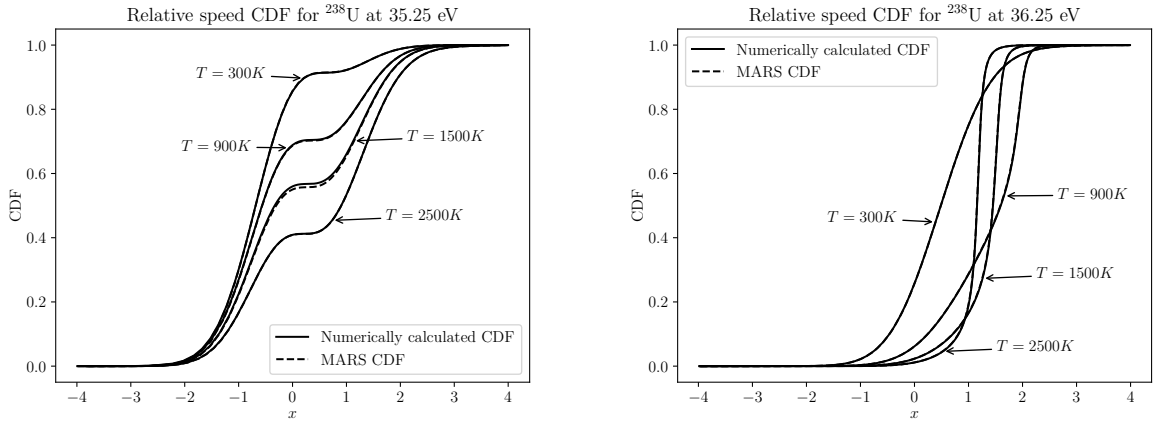
(d) $\Re[z] = 2.9$

Figure 4.7: Error of $\Re[w(z,x)]$ for a few values of z. The legend is the imaginary number added to the real part specified in each figure's caption. The plotted error, $w_{\text{approx}}(z,x) - w(z,x)$ is normalized by $\Re[w(z)]$ to match the scaling of Fig. 4.2. Where the `scipy` numerical integration routine returned NaNs, the lines disappear.

[121] has been employed. This results in the error profiles exhibited by Fig. 4.7, where we have plotted the real part of $\left(w_{\text{approx}}(z,x) - w(z,x)\right)/w(z)$. Because only the real part is of interest in resonance upscatter calculations, results on the imaginary component's error are omitted.

4.3.2 Single Energy Testing

We first present in Fig. 4.8 the relative speed distribution of ^{238}U for two different energies and a few temperatures as calculated both by numerical integration and the MARS analytic CDF. The energies correspond to being in the trough and near the peak of a scattering resonance. These plots clearly show the influence of the resonances on the double differential cross section; a nuclide with constant cross section has a relative speed distribution which



(a) 35.25 eV

(b) 36.25 eV

Figure 4.8: The MARS analytic CDF matches numerically integrated relative speed cumulative distributions. Some error can be observed for the 1500K case at 35.25 eV; scattering in the resonance dip is fortunately an extremely rare event.

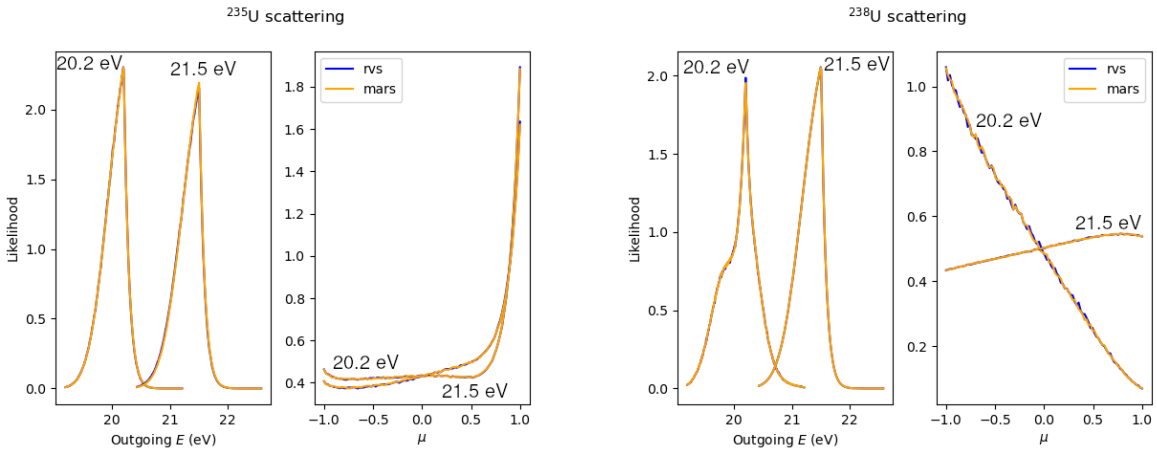
is very nearly an error function at epithermal energies. The relative speed distribution near resonances has a jumping effect which is governed by $w(z, x)$. They bear a resemblance to $w(z, x)$ behavior depicted by Fig. 4.2.

If the relative speed distribution is correct, the resultant double-differential scattering distribution is also correct. Fig. 4.9 shows this is the case for our method when compared to the RVS method of [99]. These results were obtained from our modified version of OpenMC, available at `github.com/gridley/openmc/tree/mars`. It also shows that the pole sampling technique successfully works for ^{235}U and its tightly spaced resonances.

4.3.3 Pin Cell Reactivity Feedback

The 2.4% enriched PWR pin cell example from OpenMC's suite of example problems was used to calculate Doppler reactivity feedback effects with four different models. The first and second used pointwise cross sections that were interpolated between 300, 600, 900, 1200, and 2500 kelvin. The model was run at temperatures ranging from 300 to 1800 kelvin in increments of 20 kelvin. Of the two using pointwise cross sections, one used the historical constant cross section free gas scattering approximation, and the second used the RVS method. The second two cases both used windowed multipole cross sections, one using RVS and the second MARS. The ENDFB-VII.1 nuclear dataset was employed. Figure 4.10 shows how these cases compare. Two hundred cycles with ten inactive were employed, using 200,000 particles per cycle. k_{eff} was thus converged to 20 pcm for each case.

It can be seen that the pointwise cross section representation incurs some interpolation



(a) ^{235}U (b) ^{238}U

Figure 4.9: Scattering at 1200K matches results from RVS method well at two different energies. These energies interact with resonances for both nuclides.

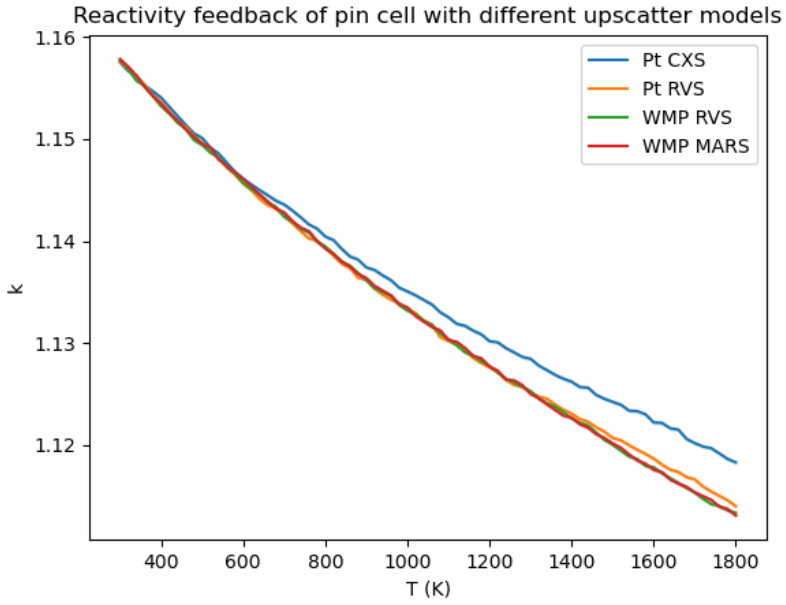


Figure 4.10: MARS matches the k eigenvalue of the RVS method on a 2.4% enriched fresh PWR pin cell problem. Line width represents estimated standard deviation of the mean.

Method	Inactive	Active
CXS	60.5	11.4
DBRC	57.0	11.1
RVS	58.3	11.3
MARS	60.3	11.1

Table 4.1: Tracking rate in thousand particles per second obtained by the constant cross section treatment, and three resonance upscatter models. MARS is comparable in speed to widely accepted techniques.

error between 1200 and 1800 kelvin. The MARS method matches the RVS results where multipole cross sections were employed. We can thus conclude that the new method works correctly across the range of energies where resonances influence the double-differential cross section at temperature for nuclides of both strong, distantly spaced resonances (^{238}U) and closely spaced weak resonances (^{235}U).

4.3.4 Influence on Tracking Rate

Finally, in order to determine the computational efficiency of the new method, tracking rate comparisons were carried out on the same PWR pin cell example problem. The computational performance of both inactive and active cycles was assessed. For the active cycles, a 100x100 Cartesian mesh tallied flux, fission rates, and neutron production rates using track-length estimators. In addition, a spatially homogenized energy spectrum tally consisting of 500 equal lethargy bins was applied.

An Intel Xeon W-2133 with six physical cores carried out the calculations, and obtained the results depicted in Table 4.1. This clearly demonstrates the computational efficiency of MARS along with the RVS and DBRC methods–it is on par with the finest contenders for modeling this type of problem.

The computational expense incurred by tallying tends to render the performance impact of our new method particularly negligible. Collision estimators could be used on the mesh tally to improve the tracking rate, but we arbitrarily opted for track length estimators. Due to subtle hardware-related effects such as cache utilization or branch prediction, the tracking rates of the three resonance upscatter handling methods have different relative performances when comparing active and inactive cycles. Future work will explore detailed performance results on a variety of architectures.

4.4 GPU Performance

We implemented the MARS method on a branch of our GPU version of OpenMC described in Chapter 3. The code can be found at https://github.com/gridley/openmc/tree/cuda_mars. From an implementation point of view, we found conversion of the CPU code to require hardly any effort. `std::complex` numbers had to be replaced with `thrust::complex` numbers for GPU compatibility, and some functions like `std::exp` had to be changed to `thrust::exp` to compile for the GPU. Similarly some `xt::xtensor` objects had to be converted to our GPU-compatible implementation, `openmc::tensor`. This shows the benefit of our approach compared to that carried out at Argonne, where all tensors, vectors, and more were converted to raw pointers.

To illustrate the performance of hte new method on GPUs, we first examine the GPU performance of a WMP-based calculation using the (incorrect) constant cross-section approximation, the RVS method, and the DBRC method. For this run, we used results obtained on the 293 kelvin Hoogenboom-Martin large benchmark from [116]. Figure 4.11 shows those results. We can see that as predicted in Chapter 2, the DBRC method is practically unusable, and processed around 500 particles per second. This plot also identifies that the resonance upscatter sampling when using RVS takes a negligible amount of time compared to the rest of the collision processing kernel.

To gauge the greatest possible advantage of our newly proposed method over the state-of-the-art RVS method, we consider the same computational benchmark but using 1050K fuel rather than 293K fuel. This represents the case with a relatively higher rejection in the RVS loop, since the performance of RVS was shown to degrade slightly at higher temperatures in [99]. Figure **??** shows this result. As for the CPU-based results in Table 4.1, we can see that the choice of resonance upscatter numerical algorithm makes no practical difference as long as rare cases with extremely high rejection rates are not encountered, as DBRC does. The MARS algorithm outperforms slightly compared to RVS, but because the importance of resonance upscattering modeling is low in the first place, the performance gain is limited.

4.5 Discussion

The multipole formalism carries a variety of advantages compared to pointwise cross sections. Aside from its potential gains in computational efficiency on modern compute architectures, it enables accurate Doppler broadening without a library size tradeoff [71], elegant sensitivity quantification, and narrows the gap between R matrix theory and the cross section representation [144]. This work develops yet another advantage to the windowed multipole

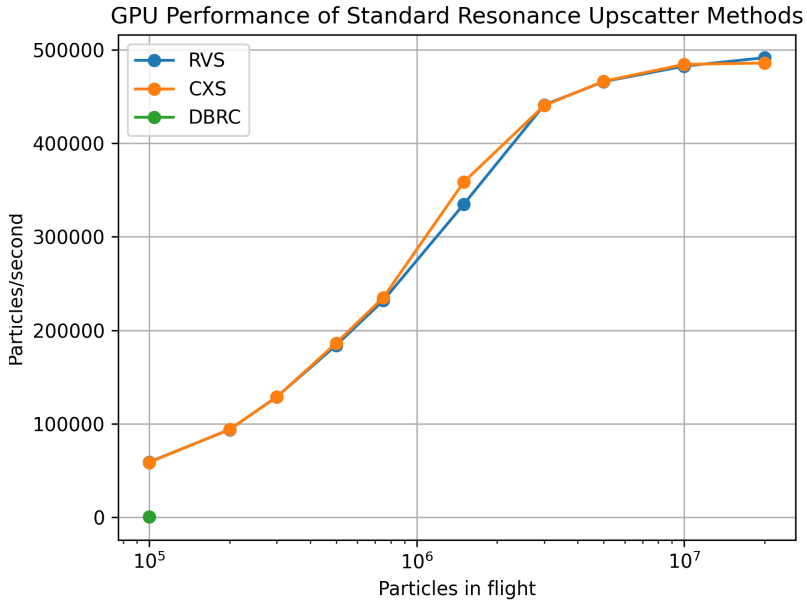


Figure 4.11: The speed of the standard resonance upscatter methods on GPU: neglecting it with CXS (constant cross-section), RVS (relative velocity sampling), or DBRC (Doppler broadening rejection correction).

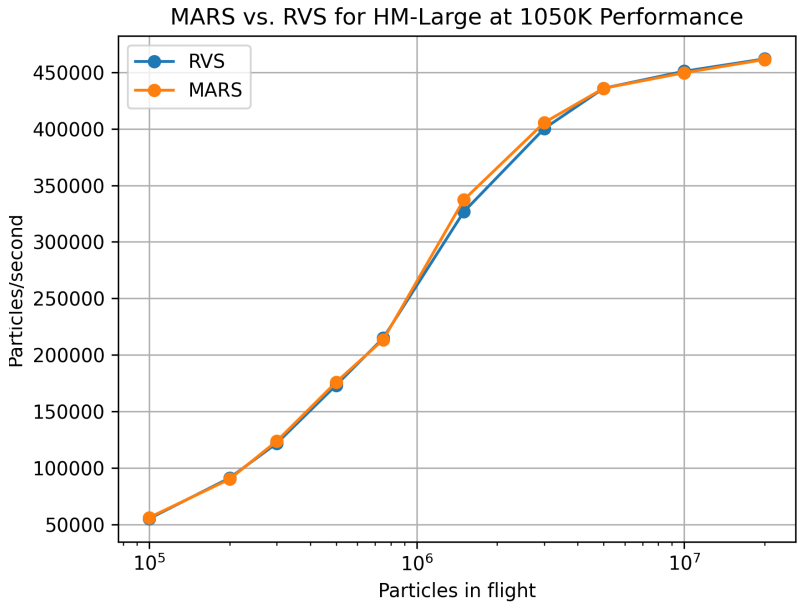


Figure 4.12: The MARS method's performance on the HM large benchmark with 1050K fuel relative to RVS.

formalism: closed-form resonance upscatter treatment.

We have demonstrated that the new method matches the results obtained by other resonance upscatter techniques. To achieve this, we derived an expression for the target relative speed distribution, and identified a novel special function which universally arises in this application. Novel numerical techniques that balance efficiency and accuracy were derived, implemented, and tested. The overall scheme was shown to achieve the same tracking rate as other resonance upscatter modeling methods.

In Figure 4.12, it was shown that the new method outperforms the RVS method for moderate particle counts by a few percent, but RVS and MARS become nearly indistinguishhable in performance when the device becomes saturated with work. This originates from increasingly coalesced memory accesses in the RVS method; our collision processing kernel lexicographically sorts by collision nuclide index in an outer loop and energy index in an inner loop. The only potentially diverged memory accesses in RVS within a rejection loop eventually become fully coalesced. Moreover, the expected waiting time for the last thread in a thread block to complete the sampling process is not terribly high.

We can apply Eq. 2.13 from Chapter 2 to approximately analyze the penalty due to waiting on many threads to complete a rejection loop. Unfortunately, our developments in Chapter 2 did not consider the case of a variable rejection rate across threads. However, the worst case rejection rate likely presents a representative result. We can take the worst-case representative single-thread expected number of samples to be around 50,000 and 1,000 for DBRC and RVS respectively, judging from results in [99]. Generally, RVS has a higher average number of iterations but much smaller maximum number of iterations. We used 128 threads per block in this case, Eq. 2.13 implies that RVS will on average take 5,500 loop iterations to complete, but DBRC will take around 280,000. If this increased number of iterations ends up producing uncoalesced memory accesses, the performance will be more than three to four magnitudes lower. Indeed, this conclusion matches the single point reporting DBRC performance in Fig. **??**. We see the delicate tightrope that must be walked when rejection sampling on the GPU: a moderately high rejection rate will degrade performance some, but very high peak rejection rates along with potentially diverged memory access will fully bottleneck the simulation and crash performance.

Figure 4.12 showed that the overall tracking rate of the MARS method compared to RVS confers no significant advantage on a work-saturated GPU. Despite this, there are a variety of reasons to use over other methods. For one, we can compare to the RST method [133], which was shown to also not induce a slowdown on GPU compared to constant cross-section sampling. This method requires an enormous amount of memory; a univariate relative speed distribution must be tabulated at every point of the pointwise grid in the relevant energy

range, leading the authors to neglect this resonant scattering in all nuclides but ^{238}U. Even so, the method took around 1 GB of extra storage, and requires a temperature interpolation process which has been validated to a limited extent.

Similarly, the RVS method requires the storage of the pointwise zero kelvin scattering cross-sections's cumulative integral in energy space. This requires a tabulation of energy points at zero kelvin, which requires more points than other temperatures due to the unbroadened resonance shape. This amounts to about 150 MB in ENDF/B-VIII.0. Far more importantly, we envision a future for the windowed multipole method in which pointwise cross-sections have been fully dealt away with. This would straightforwardly allow integrated uncertainty quantification purely via windowed multipole parameters, which are likely more closely linked to actual uncertainty parameters on resonances.

One reason that the MARS method did not exhibit a substantial performance gain relative to RVS originates from problems outside the domain of resonance upscatter sampling. Indeed, this may become a computationally important part of the cross-section lookup kernel. Currently, our CUDA code exhibits far from optimal performance in the collision kernel. As pointed out in Chapter 3, the collision formalisms entail substantial branching logic and nested data structures. This leads to an already dire degree of memory and logical divergence, drowning out the influence of the relatively benign resonant scattering sampling step in all but the most poorly behaved algorithms like DBRC. In fact, this highly sub-optimal collision kernel may explain why our code takes more particles to saturate the device relative to Argonne's code as shown by Fig. 3.31. Because our code sorts the collision kernel lexicographically first by nuclide then energy, the expense of these highly diverged memory acccesses likely becomes amortized with a tremendous number of particles in flight.

The new method called multipole analytic resonance scattering (MARS) overcomes the storage requirements of relative speed tabulation [29], and avoids rejection sampling as employed by other common approaches. Without a need to access intermediate storage, the accesses to global memory can be reduced on GPU architectures. On top of that, the work discrepancy between threads incurred by rejection sampling on GPUs is similarly overcome. The RVS method indeed exhibits these deficiencies which might superficially judged to be consequential to GPU performance; however, the fairly un-optimized collision kernels employed in both the Argonne and MIT GPU MC codes clearly are not bottle-necked by resonant scattering modeling. Only in the future when all the other types of collision have been optimized thoroughly–for example sampling the center-of-mass angular distributions–can these methods be compared in a manner that illustrates a non-negligible benefit of one over the other.

Chapter 5

Beyond Probability Tables for the Unresolved Resonance Region

5.1 Introduction

5.1.1 The Unresolved Resonance Region

We review the requirements of Monte Carlo neutron transport codes to accurately model the phenomena associated with the unresolved resonance regime, present the first rigorous justification for the modeling approach employed by probability tables for unresolved resonances, and present novel theoretical insight to the nature of the unresolved resonance cross section distribution. Using the new insights, we present a new method named AURA (analytic unresolved resonance algorithm) for modeling unresolved resonance cross sections with the normal inverse Gaussian distribution. We show that the new model accurately models temperature dependence in the URR region, requires less data than previous methods, and outperforms the conventional URR treatment on GPUs.

To verify the performance impact of probability tables, first published in [54], we ran the Hoogenboom-Martin large benchmark from Chapter 3 both with and without probability tables present. Figure 5.1 shows the results, which matches the previously observed extent of performance degradation from URR modeling. Our goal in this chapter is to devise a new method for modeling URR effects that exhibits negligible performance penalty, or as we show later, a performance increase is even possible.

The behavior of neutron cross sections in the energy range relevant to reactor physics is typically divided into a few distinct regimes: thermal, resolved resonance region (RRR), unresolved resonance region (URR), and fast. The dominant influence of molecular binding and thermal motion characterizes the thermal region, less than a few eV when considering room

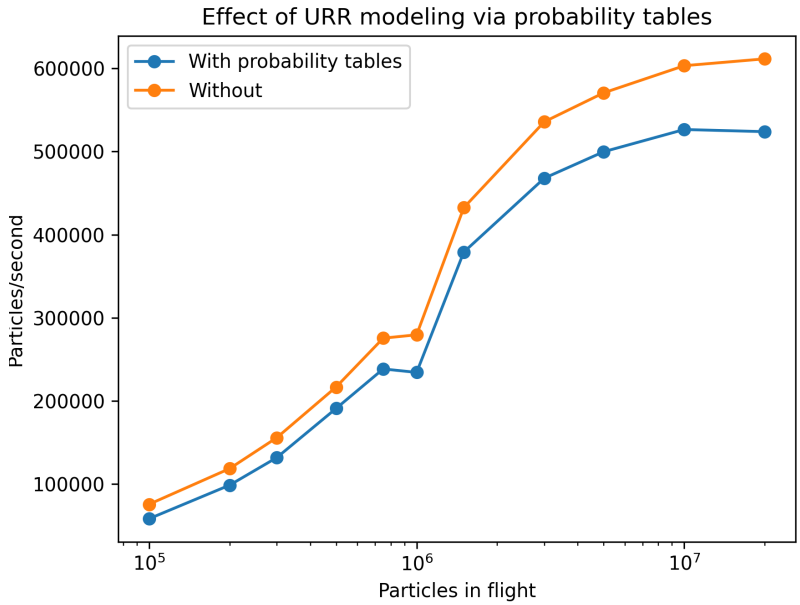


Figure 5.1: Probability tables incur a serious processing rate penalty on the H100 on the Hoogenboom-Martin benchmark.

temperature problems. Next, in the resolved resonance region, experiments have sufficient energy resolution to map individual resonances. After that comes the unresolved resonance region, where distinct resonances in the neutron cross section exist, but are impractical to measure as a consequence of their small width or spacing relative to experimental resolution. Due to differences in the resonant structure of different nuclides, the unresolved region can begin at low or high energies. For example, ^{238}U's unresolved region begins at 20 keV, but ^{239}Pu's URR begins at 2.5 keV in ENDF/B-VIII.0 [161]. After the URR region is the fast region, typically considered as ranging from 0.5 MeV upward. There is no physical distinction between the URR region and the fast region; only a modeling distinction that variability due to unresolved resonances is not taken into account. This work focuses specifically on improving the modeling of cross section variability in the URR.

In the URR, evaluated nuclear data present statistics on the resonances in the URR regime, from which "probability tables" (PT) [162] of the neutron cross section can be created for use in Monte Carlo codes, or dilution tables [163] for use in deterministic codes. Specifically, NJOY's [164] respective PURR and UNRESR modules carry out these tasks. More generally, deterministic codes employing the subgroup approach may also employ PT, as described by [165] and implemented in their CALENDF code. Using PT enhances the accuracy of continuous energy Monte Carlo calculations to eliminate biases typically around 100-200 pcm in fast spectrum problems that occur when neglecting oscillation of the cross

section in the URR range, as shown in [166], [167] among many other works.

Codes used to generate PT often employ the approach of PURR. The ENDF file provides statistical information about the nature of the resonances in the unresolved resonance region at a variety of energy points. This enables calculation of the distribution of the cross section in a small energetic vicinity about these points. The approximation made in doing so is that the resonant structure happens over very fine energy scales compared to the spacing between points. In ENDF/B-VIII.0, three nuclides are exceptional and use a single set of resonance parameters for all energies in the unresolved region : ^{58}Fe, ^{140}Ba, and ^{167}Er.

The resonance statistics presented are the mean spacing between resonances, mean resonance widths for use in single level Breit-Wigner (SLBW) resonance formulas, the number of χ^2 degrees of freedom used to model each partial resonance width, and information about various spin states. The various possible spin configurations of the neutron incident on the target lead to statistically independent sets of resonances which are summed together to represent the total cross section. Two quantum numbers are required to represent each spin configuration: L, the orbital angular momentum of the incident particle relative to the nucleus, and J, the total angular momentum of the compound state, equivalently the total angular momentum of the initial constellation.

The states corresponding to $L = 0$ are the s-wave resonances, $L = 1$ are p-wave resonances, and so on under standard spectroscopic notation. In ENDF/B-VIII.0, the file for ^{235}U has for $L = 0$, $J \in \{3, 4\}$, or for $L = 1$, $J \in \{2, 3, 4, 5\}$. The d-wave and higher resonances are negligible. For ^{238}U, parameters for $L = 0$ with $J \in \{0.5\}$, $L = 1$ with $J \in \{0.5, 1.5\}$, and $L = 2$ with $J \in \{1.5, 2.5\}$ are all listed. ^{58}Fe is the lone nuclide for which f-wave resonances are listed.

The mean reduced scattering width, $\langle \Gamma_{n,0} \rangle$ is given, along with the mean capture width $\langle \Gamma_\gamma \rangle$ and mean fission width $\langle \Gamma_f \rangle$ are given. The ENDF file also may include an arbitrary additional channel width, $\langle \Gamma_x \rangle$ called the competitive component. It should be noted that the channel may occur only after a certain energy threshold, so the cross section might change abruptly past the threshold for the competitive channel.

Of course, only knowing the average resonance widths does not allow their sampling. From Porter and Thomas's work [168], physical reasoning implies that these widths follow χ^2_a distributions with the number of degrees of freedom, a, corresponding to the number of ways the compound nucleus can decay. Fission is a multi-channel process, so two or three degrees of freedom are commonly encountered there. In heavy nuclei, the capture reaction often has over 50 channels available [169]. Noting that $\mathrm{Var}(Q/a) \to 0$ with $Q \sim \chi^2_a$ as $a \to \infty$, ENDF's choice of a single constant value of the capture width is well-justified. For scattering, again only one or two degrees or freedom are typically encountered.

To restate the summed cross section across all spin states and resonances under the SLBW formalism, we have that the elastic scattering cross section is:

$$\sigma_n(E) = \sigma_{\text{pot}} + \sum_{l=0}^{N_l-1} \sum_{j=1}^{N_j(l)} \sum_{\lambda=1}^{N_\lambda} \sigma_\lambda \left(\psi(\theta_\lambda, x_\lambda(E)) \left[\cos(2\phi_l) - \left(1 - \frac{\Gamma_{n,\lambda}}{\Gamma_\lambda} \right) \right] + \chi(\theta_\lambda, x_\lambda(E)) \sin(2\phi_l) \right) \quad , \tag{5.1}$$

where σ_λ, the approximate maximum value of the cross section, is:

$$\sigma_\lambda = \frac{4\pi g_J \Gamma_{\lambda,n}}{k^2 \Gamma_\lambda} \quad . \tag{5.2}$$

The cross section for fission, absorption, or any competitive reaction is:

$$\sigma_\gamma(E) = \sum_{l=0}^{N_l-1} \sum_{j=1}^{N_j(l)} \sum_{\lambda=1}^{N_\lambda} \sigma_\lambda \frac{\Gamma_{\gamma,\lambda}}{\Gamma_\lambda} \psi(\theta, x_\lambda) \quad , \tag{5.3}$$

Fission or the competitive reaction would be the same equation as the above but with"γ" subscripts replaced by the above equation by the respective symbol to denote it, e.g. "f" in the case of the fission cross section.

The dimensionless temperature factor θ is:

$$\theta_\lambda = \frac{\Gamma_\lambda \sqrt{A}}{2\sqrt{k_B T E}} \quad . \tag{5.4}$$

A denotes the ratio of the target mass to that of the neutron, k_B the Boltzmann constant, T the material absolute temperature, and E the incident neutron energy.

The potential scattering cross section, taken to be constant on the fine scale of resonant variations in energy, is:

$$\sigma_{\text{pot}} = \frac{4\pi}{k^2} \sum_{l=0}^{N_l-1} (2l+1) \sin(\phi_l) \quad . \tag{5.5}$$

The phase shifts ϕ_l depend on the incident neutron energy, and can be found in [170].

The ψ,χ functions describe Doppler broadening of SLBW resonances [171]. They are:

$$\psi(\theta, x) = \frac{\sqrt{\pi}\theta}{2} \Re w \left(\frac{\theta x}{2} + i\frac{\theta}{2} \right) \tag{5.6}$$

$$\chi(\theta, x) = \frac{\sqrt{\pi}\theta}{2} \Im w \left(\frac{\theta x}{2} + i\frac{\theta}{2} \right) \tag{5.7}$$

and the Faddeeva function can be defined as (because $\theta > 0$):

$$w(z) = \frac{i}{\pi} \int_{-\infty}^{\infty} \frac{e^{-t^2}\,\mathrm{d}t}{z-t} \quad . \tag{5.8}$$

The average neutron width can be obtained from the reduced neutron width as

$$\langle \Gamma_n \rangle = \frac{P_l}{\rho} \sqrt{E} \langle \Gamma_{n,0} \rangle \quad . \tag{5.9}$$

Reaction widths for individual resonances are sampled by multiplying the mean widths by random variables χ^2_n/n. For completeness, we restate the density function for χ^2 random variables as:

$$p_{\chi^2_n}(y) = \frac{y^{n/2-1}}{2^{n/2}\Gamma(n/2)} e^{-y/2} \quad , \tag{5.10}$$

where Γ in this context refers to the gamma function.

Notably, the PURR module of NJOY generates χ^2 random variables in an unconventional way. Twenty samples from χ^2 distributions with one, two, three, and four degrees of freedom are hardcoded into NJOY. A random integer in $[1, 20]$ is generated, and the value corresponding to that index is chosen as representative of a sample. For this reason, matching the results from NJOY exactly seems undesirable. It incorporates numerous approximations and tricks rendered unnecessary with modern computing, so the slight mismatch against NJOY encountered in [172] and our results is to be expected.

The values x_λ are nondimensional distances from the center of each resonance:

$$x = \frac{2(E' - E_\lambda)}{\Gamma_\lambda} \quad , \tag{5.11}$$

The spin statistical factor is:

$$g_J = \frac{2J+1}{4I+2} \quad . \tag{5.12}$$

The shifted energies are:

$$E'_\lambda = E_\lambda + \Gamma_{n,\lambda} \frac{S_l(|E_\lambda|) - S_l(E_n)}{2P_l(|E_\lambda|)} \quad . \tag{5.13}$$

In the unresolved region, the resonance energy shifts become negligibly small, so the approximation that $E'_\lambda \approx E_\lambda$ is applied. To our knowledge, all URR processing tools employ this accurate approximation. The penetrability P_l and shift factor S_l have different dependencies on ρ for each value of l. We refer the interested reader to [170] for their expressions.

The resonance energies are the eigenvalues of a random Hermitian matrix, so may be

obtained either using a random matrix theory approach or using the asymptotic Wigner distribution [172]. If we denote the spacing as $\Delta E = \langle \Delta E \rangle S$, then Wigner's asymptotic result is that

$$S \sim \frac{\pi x}{2} e^{-\pi x^2/4} \quad . \tag{5.14}$$

The total cross section is found for any partial cross section type by summing the elastic, absorption, and fission cross sections. Notably, NJOY does not include the optional competitive reaction. While in principle a large number of reaction types could be included, only data for the elastic, absorption, fission, and an optional fourth channel are given in ENDF [170]. The fourth channel acts to change the distribution of the Γ_λ terms only, since it is not added in to the total cross section. The fourth channel is called the competitive channel, and, for instance represents inelastic scattering that may open above some threshold energy in the unresolved region. This behavior can be observed in ^{232}Th.

The accurate approximation used in generating the unresolved resonance cross section distribution is that energy-dependent parameters in these equations that are non-resonant are fixed about the energy point of interest, whereas the fine structure granted by the resonance is taken to vary on a fine scale, over which we assume the incident energy distribution is locally uniform. Energy dependence of the width statistics, spacing distribution, wavenumber, and phase shifts give a distribution of the cross section due to resonant fluctuations at each tabulated energy.

In order to calculate the expectation value of the total cross section and its distribution at intermediate energies, the resonance parameters must be interpolated, rather than the cross section values. This historically led to disagreement [173] between the PREPRO [174] and NJOY codes, and has been resolved today. In order to eliminate the impact of the methods of generating the PT on the infinite-dilute cross section, the `lssf` flag was introduced in NJOY. This flag specifies that the cross section, when sampled from a PT, should be multiplied by the pointwise cross section value on the file. In other words, the cross sections on the PT are normalized by the infinite-dilute cross section.

5.1.2 Related Works

The work [175] demonstrates on-the-fly temperature interpolation of probability bands for modeling the unresolved resonance region. Their work confirms the 250 pcm effect of probability tables on the ZEBRA-8H benchmark of [176].

Another work [177] demonstrates the implementation of the ladder sampling for a new unresolved region processing module called PURC. The innovations in PURC are a shared memory parallelism approach to sampling resonance ladders and the use of an $\mathcal{O}(n \log n)$

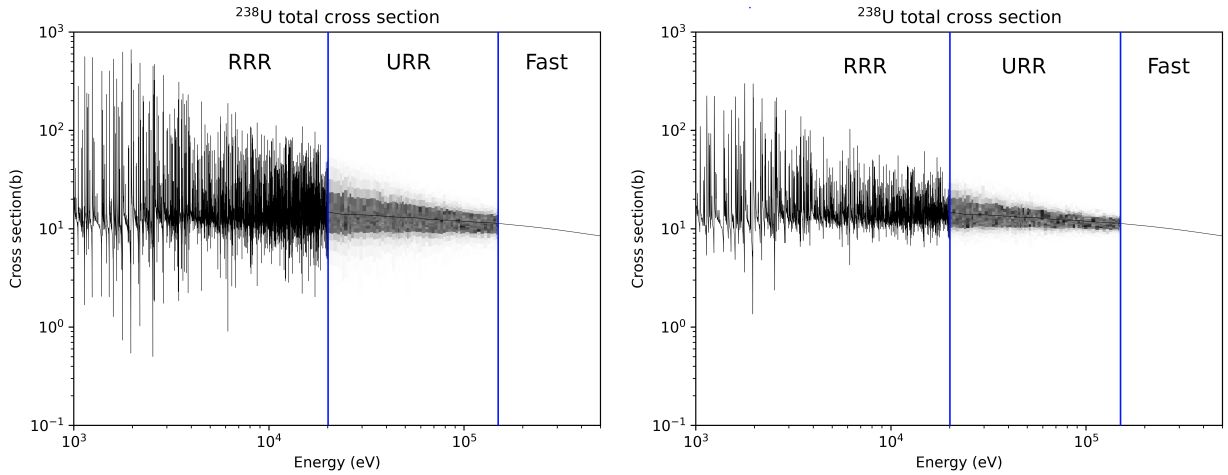
sorting algorithm for structuring the uniformly sampled energy grid on which cross sections are reconstructed.

Some novel techniques have arisen over the past two decades for more accurately generating PTs. Dunn and Leal [178] describe a more physically rigorous method than that employed by PURR. Their technique rigorously deals with the fact that naive use of the Wigner distribution for the level spacing distribution is nonphysical. In fact, the levels are repelled from each other as a consequence of them being eigenvalues of Hamiltonian matrices [179]. Dunn and Leal first sample energy level differences from the Wigner distribution with a standard PURR ladder technique, then reject based on the Dyson-Mehta Δ_3 statistic [180]. This rejection step ensures physically realistic level repulsion, in contrast to NJOY's PURR. Similarly, [172] explores directly obtaining energy levels as the eigenvalues of a Gaussian orthogonal ensemble (GOE). The results obtained were shown to provide the expected level repulsion effects and exhibit roughly the Wigner spacing distribution.

In a similar vein, some other codes have explored more realistic cross section formulas, as opposed to more realistic level spacings as described above. Walsh [181] has explored the use of a multilevel Breit-Wigner (MLBW) formula for the cross section, and determined it to have a negligible impact on criticality calculations for a few benchmarks compared to the standard SLBW formulas. Similarly, [182] explored the use of the R-matrix limited (RML) formalism for the computation of the cross section after sampling widths and energy levels. To our knowledge, no code has yet both examined the results of using a more sophisticated formula than SLBW and a realistic model of level spacings simultaneously.

The SLBW formulas are most inaccurate where resonances are closely spaced, and the enhanced level spacing techniques tend to separate resonances from each other more. Consequently, we can expect that while both MLBW/RML and GOE approaches on their own may influence the results of a URR sampling calculation a bit, the energy spacing repulsion effect will likely lessen the impact of deficiencies of the SLBW formula. This may explain why the traditional PURR technique works well when rejecting unphysical negative cross sections.

Wu et al. [183] describe a new module in the Ruler code system called `NURD` that generates PTs using the ladder method. It uses a multithreaded algorithm and works substantially faster than NJOY while matching its results. Hu et al. [184] describe the development of the AXSP nuclear data processing code, which uses an improved energy grid sort, noting that NJOY employs the $\mathcal{O}(n^2)$ bubble sort algorithm. [185] describes `mcres.py`, a tool for generating resonance sequences written using modern programming practices. URRPACK [186] also is another code that can calculate the dilution tables of nuclides in the unresolved resonance region.



(a) 294K (b) 2500K

Figure 5.2: The unresolved resonance region for ^{238}U begins around 20 keV and continues to about 500 keV. The influence of temperature in the URR can be observed here; Doppler broadening of resonances results in narrowing of the total cross section distribution. Darker grey indicates a higher probability of encountering that value of σ_t.

Even if such an experiment were available to map the resonances into high energies, the gain in reactor physics modeling capability would be marginal at best. Because the neutron flux tends to scale as $1/E$ between the thermal and fast neutron regions in thermal reactors, the capture rate in high-energy resonances is much lower than at lower energies. By modeling individual sequences of statistically feasible resonances and performing independent Monte Carlo calculations for each sequence, Walsh [181] has shown that the uncertainty due to the lack of knowledge of the specific locations of resonances is around 50 pcm for a PWR pin cell, 100 pcm for the ZEBRA experiment, and 10 pcm for Godiva, to name a few.

The typical behavior of the cross section in the unresolved resonance region has been depicted in Fig. 5.2. The data used to generate this figure originates from ENDF/B-VIII.0 [161], and we used OpenMC's nuclear data interface [187] for straightforward plotting. We can see that the probabilistic interpretation of the cross section abruptly ends around 500 keV. It has been shown by Walsh [181] that the historical neglect of random variations in the cross section at these higher energies (139-500 keV) introduces systematic discrepancy between the Big Ten experiment and simulations. In fact, fine resonance structure extends far into the fast neutron region.

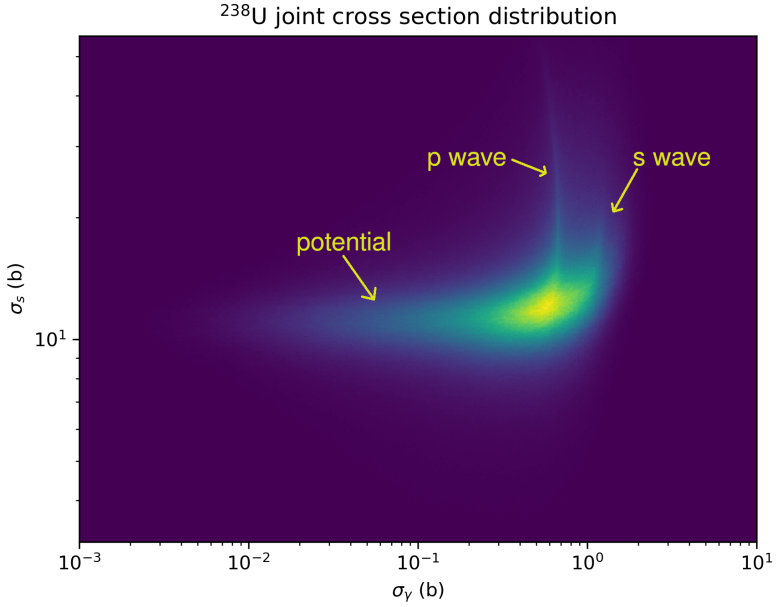


Figure 5.3: Example of the joint distribution between σ_γ and σ_s for ^{238}U at 294K at 23 keV.

5.1.3 The Joint Partial Cross Section Distribution

Considering the clearly non-independent nature of the partial cross section components appearing in Fig. 5.3, before embarking on a new method to represent the full joint distribution, we analyze the extent that results of steady-state transport calculations with unresolved resonances are influenced by joint effects. Conventionally, the joint effect has been modeled via tabulation of the marginal distribution of the total cross section with accompanying conditional expectations of the absorption, scattering, and fission components [164].

To address the question as to whether a new model for cross sections should incorporate the full joint distribution of the component-wise cross sections, we below show that expectations conditioned on the total cross section of partial cross sections suffice to produce the mean flux at each point in space, energy, and angle under a few simple assumptions. The full joint density need not be stored. While this has been shown intuitively in [188], we present a rigorous argument below which precisely pins down the assumptions.

Consider the fixed source transport equation with random cross sections.

$$\hat{\Omega}\cdot\hat{\varphi}(r,E,\hat{\Omega})+\left(\sum_j N_j(r)\sigma_{t,j}(E)\right)\hat{\varphi}(r,E,\hat{\Omega})= S(r,E,\hat{\Omega})+\int_0^\infty \mathrm{d}E'\int_{4\pi}\mathrm{d}\hat{\Omega}'\left(\sum_j N_j(r)\sigma_{s,j}(\hat{\Omega}'\to\hat{\Omega},E'\to E\right)\hat{\varphi}(r,E',\hat{\Omega}') \quad , \quad (5.15)$$

where N_j denotes the spatially dependent densities of each nuclide. For each E, we assume that individual cross section components σ_t, σ_s are distributed jointly. As a consequence, the angular flux in this model is a random variable with some fixed distribution at each point in r, E, $\hat{\Omega}$ space, and this flux is correlated against the cross section variables.

The objective is to find the expectation of the flux, representative of the properly self-shielded flux due to fine variations in cross section at a given energy point E. We will take the expectation value of each side of the transport equation. To handle the left side,

$$\mathbb{E}\left[\hat{\Omega}\cdot\hat{\varphi}(r,E,\hat{\Omega})+\left(\sum_j N_j(r)\sigma_{t,j}(E)\right)\hat{\varphi}(r,E,\hat{\Omega})\right]= \quad (5.16)$$

$$\hat{\Omega}\cdot\mathbb{E}[\hat{\varphi}(r,E,\hat{\Omega})]+\mathbb{E}\left[\left(\sum_j N_j(r)\sigma_{t,j}(E)\right)\hat{\varphi}(r,E,\hat{\Omega})\right] \quad (5.17)$$

The first term on the left now contains the object of our interest, the properly self-shielded flux. However, the term on the right does not allow distribution of the expectation due to the correlation between fluxes and cross sections. To deal with this, recall that for two random variables X, Y,

$$\mathbb{E}_{X,Y}[f(X,Y)]=\mathbb{E}_Y[\mathbb{E}_X[f(X,y)|Y=y]] \quad . \quad (5.18)$$

We apply this fact to the absorption term, conditioning on all the microscopic total cross sections. The second term in Eq. 5.17 thus becomes:

$$\mathbb{E}\left[\left(\sum_j N_j(r)\sigma_{t,j}(E)\right)\hat{\varphi}(r,E,\hat{\Omega})\right]=\mathbb{E}_{\bar{\sigma}_t(E)}\left[\left(\sum_j N_j(r)\sigma_{t,j}(E)\right)\mathbb{E}\left[\hat{\varphi}(r,E,\hat{\Omega})|\bar{\sigma}_t(E)\right]\right] \quad , \quad (5.19)$$

where $\bar{\sigma}_t(E)$ is the vector of all random microscopic total cross sections at this energy, and

the linearity of expecation has been employed after the total cross sections are considered as deterministic random variables due to conditioning. This step requires the assumption that the individual nuclide-wise random cross sections are not correlated with each other.

The source term is not random, so the expectation operator does not affect it. For the scattering term, we suppose that expectation can distribute through the integrals over energy and angle, and then consider:

$$\mathbb{E}\left[\left(\sum_j N_j(r)\sigma_{s,j}(\hat{\Omega}'\to\hat{\Omega}, E'\to E\right)\hat{\varphi}(r,E',\hat{\Omega}')\right] = \quad (5.20)$$

$$\mathbb{E}_{\bar{\sigma}_t(E')}\left[\mathbb{E}\left[\left(\sum_j N_j(r)\sigma_{s,j}(\hat{\Omega}'\to\hat{\Omega}, E'\to E\right)\hat{\varphi}(r,E',\hat{\Omega}')|\bar{\sigma}_t(E')\right]\right] = \quad (5.21)$$

$$\mathbb{E}_{\bar{\sigma}_t(E')}\left[\left(\sum_j N_j(r)\mathbb{E}\left[\sigma_{s,j}(\hat{\Omega}'\to\hat{\Omega}, E'\to E\right)|\bar{\sigma}_t(E')\right]\mathbb{E}\left[\hat{\varphi}(r,E',\hat{\Omega}')|\bar{\sigma}_t(E')\right]\right] = \quad (5.22)$$

The first expectation is over all random variables: $\sigma_t(E)$ and $\sigma_s(E)$ for all values of E. The second equality comes from the modeling assumption that cross section distributions between different energies are independent. Of course, for energy differences on the scale of the size of a resonance, they are in fact tightly correlated. The probabilistic approach to unresolved resonances therefore can only be expected to be accurate at sufficiently high energies where the scattering energy losses are much greater than the spacings between resonances.

The final equality of Eq. 5.22 comes from assuming that given a fixed set of microscopic total cross sections, varying the scattering cross section at energy E' does not influence the angular fluxes at E. This assumption is exactly true if the scattering operator only exhibits downscattering. With upscattering back to the energy of interest or within-group scattering in a multigroup approach, variation in the scattering cross section can change the fluxes at energy E', albeit to a much smaller extent.

For simplicity, this discussion did not include temperature dependence. Correlation between temperatures certainly exists: to simulate a neutron crossing from temperature T to another material of the same composition at temperature $T' \neq T$ then back to the same composition with temperature T, the same random number must be recycled. Even though the resonances are unresolved, in reality, their location is still fixed. On top of that, the cross section should depend continuously on temperature without full resampling of the underlying random state when passing from one temperature to another, since Doppler broadening

continuously modifies the cross section with respect to temperature.

With each term analyzed, we can equate the expectations of each term in the transport equation to find the self-shielded flux. In practice, this procedure amounts to the technique used in any Monte Carlo code treating unresolved resonances: a total cross section is sampled using a fixed uniform $[0, 1)$ random number ξ_j corresponding to each nuclide of index j while the particle is at the same energy. The partial cross sections are conditional expectations conditioned on the sampled value of the total cross section. Only when scattering to a new energy do the values of ξ_j change, which correctly reflects the assumptions detailed below required to arrive at the expression for the energetically self-shielded flux:

- Negligible correlation between nuclide-wise cross section components at a given energy
- Negligible correlation of cross section for one nuclide between arbitrary $E' \neq E$
- Neutrons cannot scatter from energy E to some other state then back to E

5.1.4 `urrtools.py`

Our python module `urrtools.py` uses the same basic methods as NJOY, but offloads reconstruction of cross sections at each sampled energy point to a GPU via pyCUDA [189]. We carry out the reduction from partial cross sections at each energy point to a histogram in total cross section space with a specialized CUDA kernel that simultaneously tabulates the conditional expectations of the partial cross sections for each σ_t bin.

The calculation of ψ and χ differs from NJOY; We employ the 16th order rational approximation to the Faddeeva function $w(z)$ presented in [123]. Because the majority of resonances are far away from each energy point, we enhance the computational efficiency of our program by falling back to the asymptotic approximation $w(z) \approx \frac{i}{\sqrt{\pi}z}$ for $|\Re[z]| > 20$.

Our code does not match the results of NJOY for generating probability tables. Similarly, [172] presents an independent implementation of unresolved resonance sampling which fails to reproduce results from NJOY exactly. NJOY samples χ^2 random variables in a consistently biased way [1]. Because the χ^2 random variables feed into the resonance widths, NJOY biases resonance widths consistently.

The file `purr.f90` reveals an array named `chisq` defined as shown in Tab. 5.1. To sample a χ^2 random variable with n degrees of freedom, a random row from the nth column is employed. Unfortunately, this pretabulated technique to quickly generate approximately χ^2 distributed random variables suffers from severely biased means. The means of each column are respectively 1.86, 2.18, 2.59, and 3.36, but should equal 1, 2, 3, 4. Results

[1] at least as of commit `9adfc0898848b6864a71d3e7920d255ce45f7901`

Table 5.1: The samples of χ^2_n random variables employed by NJOY, yielding incorrect average samples.

χ^2_1	χ^2_2	χ^2_3	χ^2_4
1.31003e-3	9.19501e-3	.0250905e0	.049254e0
.0820892e0	.124169e0	.176268e0	.239417e0
.314977e0	.404749e0	.511145e0	.637461e0
.788315e0	.970419e0	1.194e0	1.47573e0
1.84547e0	2.36522e0	3.20371e0	5.58201e0
.0508548e0	.156167e0	.267335e0	.38505e0
.510131e0	.643564e0	.786543e0	.940541e0
1.1074e0	1.28947e0	1.48981e0	1.71249e0
1.96314e0	2.24984e0	2.58473e0	2.98744e0
3.49278e0	4.17238e0	5.21888e0	7.99146e0
.206832e0	.470719e0	.691933e0	.901674e0
1.10868e0	1.31765e0	1.53193e0	1.75444e0
1.98812e0	2.23621e0	2.50257e0	2.79213e0
3.11143e0	3.46967e0	3.88053e0	4.36586e0
4.96417e0	5.75423e0	6.94646e0	10.0048e0
.459462e0	.893735e0	1.21753e0	1.50872e0
1.78605e0	2.05854e0	2.33194e0	2.61069e0
2.89878e0	3.20032e0	3.51995e0	3.86331e0
4.23776e0	4.65345e0	5.12533e0	5.67712e0
6.35044e0	7.22996e0	8.541e0	11.8359e0

from NJOY's PURR module should be interpreted cautiously despite decades of validation studies.

Sampling of the cross section correctly handles temperature correlation effects during sampling. Namely, the same set of resonance positions should be used for each temperature to evaluate the cross section at in order to ensure correct temperature dependence of an individual ladder sample.

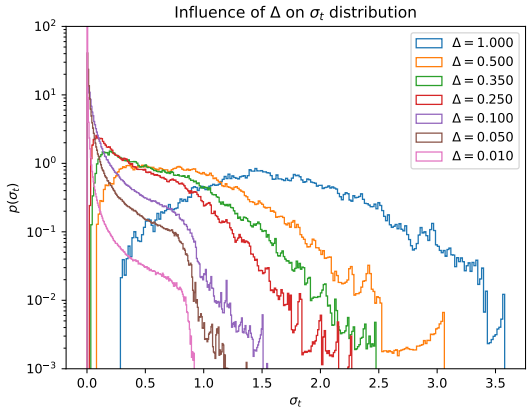
5.1.5 Numerical study of the σ_t distributions in a few conditions

We present some numerical results here that exhibit the variety of shapes exhibited by the σ_t distribution. We consider a single spin state with SLBW resonances as described by Eq. 5.23. Table 5.2 describes the cases covered by our parameter sweep. No competitive cross section variation has been included due to its effect being the same as a fission component. The absorption width was held fixed at 0.035, where the units here are arbitrary. Figures 5.4 and 5.5 show the results.

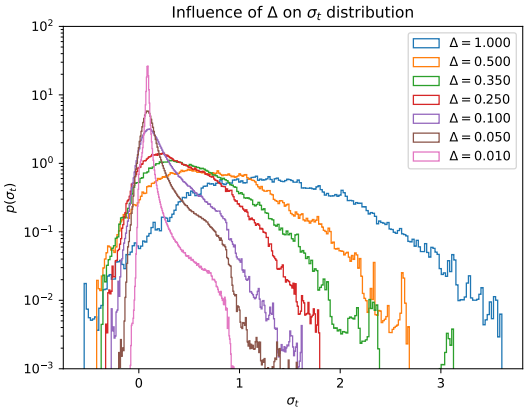
We can identify a few distinct behaviors here. For the case when Γ_t is comparable to

Table 5.2: Parameters swept to explore σ_t distribution at zero kelvin for a single spin state.

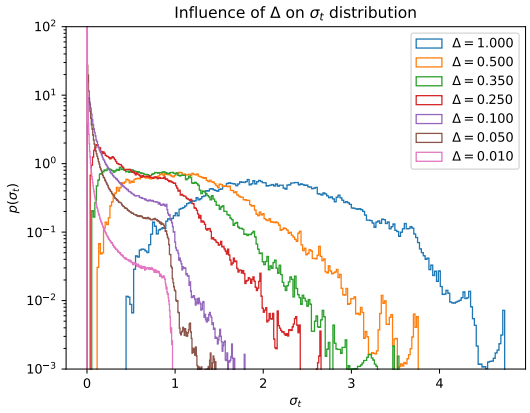
Case	ϕ	$\bar{\Gamma}_n$ (DOF)	$\bar{\Gamma}_f$ (DOF)
1	0.0	0.07 (1)	0.0
2	0.3	0.07 (1)	0.0
3	0.0	0.14 (1)	0.0
4	0.3	0.14 (1)	0.0
5	0.0	0.07 (1)	0.035 (3)
6	0.3	0.07 (1)	0.035 (3)
7	0.0	0.07 (2)	0.0
8	0.3	0.07 (2)	0.0



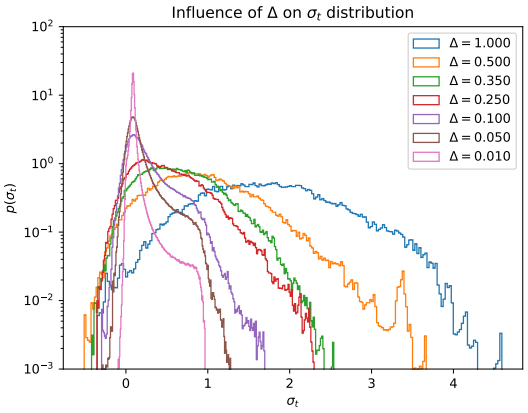
(a) Case 1



(b) Case 2

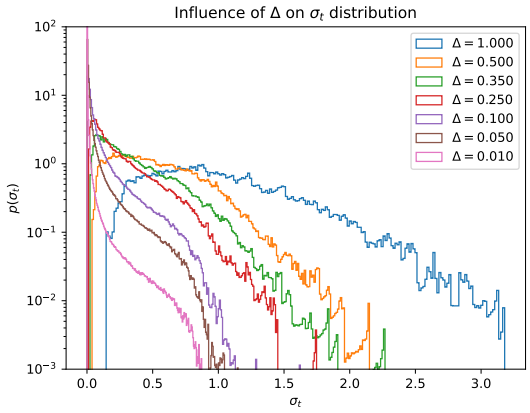


(c) Case 3

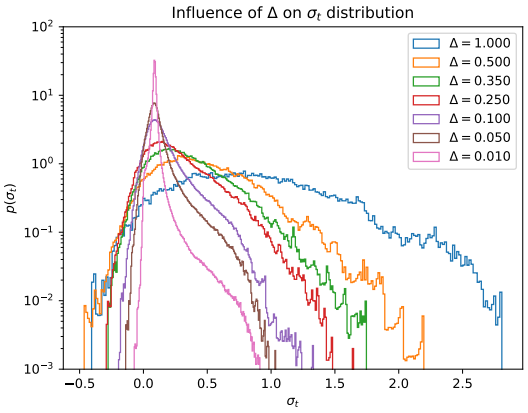


(d) Case 4

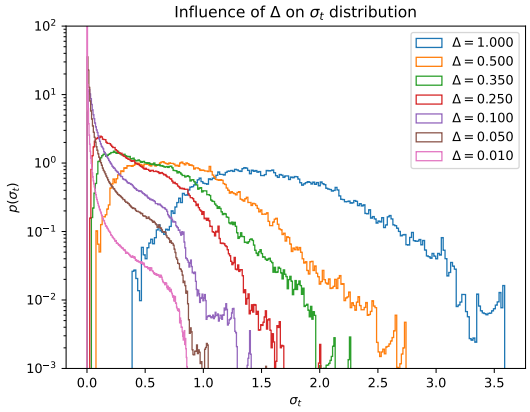
Figure 5.4: Cases 1-4 in the single spin state parameter sweep.



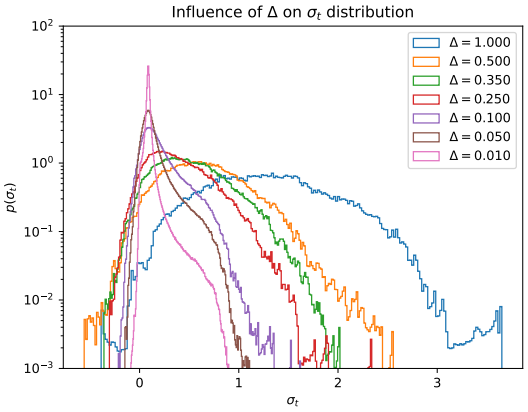
(a) Case 5



(b) Case 6



(c) Case 7



(d) Case 8

Figure 5.5: Cases 5-8 in the single spin state parameter sweep.

$\langle \Delta E \rangle$, i.e. $\Delta \approx 1$, a slightly skewed universal parabolic behavior can be observed in log probability density space. Therefore, a skew Gaussian may model this regime well. For the intermediate values of Δ, a straight lines for the tails suggest exponentially distributed behavior, as once observed [190]. In fact, Hwang in that work had suggested fitting distributions with exponential tails to the cross section distribution and demonstrated success for some actinides, but it seems this approach never materialized to a useful scheme.

As $\Delta \to 0$, the distribution limits to another shape. We show in the next section that the small Δ behavior can be very accurately described as a Cauchy distribution truncated to the range of allowable σ_t values.

5.2 An Analytic Model to the Cross Section Distribution

We are aware of only three works which sought analytical models of the cross section distribution in the unresolved region. Recently [191] sought such a representation, but no results on the probability distribution exhibited by the cross section were presented.

Starting from random R matrix theory and assuming uniformly spaced resonances, Jordanov et al. [192] were able to analytically compute the attenuation of neutrons with a random cross section. Unfortunately, this approach does not satisfy the needs of Monte Carlo codes which seek to sample values of the cross section. In other words, no results regarding the probability distribution of the cross section were presented. However, letting the cross section be a random variable as a result of the R matrix being random, Jordanov was able to obtain the expected attenuation. Notably, the attenuation function in the case of random cross sections deviates from the standard exponential behavior. The portion of neutrons with energies matching resonances attenuate out first, thus causing non-exponential attenuation.

Crucially, Jordanov's work had to assume uniformly spaced resonances to obtain these results. Seeking an analytical model of the full unresolved resonance system including variable resonance widths and spacings summed over a variety of spin states is a mathematical sword in the stone. In order to study this problem, we first consider the distribution of the total cross section taken on by a single spin state with evenly spaced resonances of equal width, assuming a randomly sampled incident energy.

Kumar [193] presents some of the most impressive results regarding random cross sections. In this case, the "off-diagonal" component of scattering is studied. In other words, the work focuses on the (p, α) cross section. Kumar et al. assume randomness in the R matrix and rigorously show that this reaction cross section takes on an exponential distribution. Unfortunately, for the total neutron cross section under study here, the behavior is quite

clearly differing from an exponential. Rigorous analytical tractability seems impossible.

Hwang's last paper [194], published posthumously, presents remarkable work which we did not discover until the conclusion of this study. In fact, according to Google Scholar, no other works have cited this paper. Hwang's last work publishes the only analytical descriptions of the cross-section distribution that are potentially practically useful for nuclear engineering analysis. He provides an intuitive argument that under the narrow resonance flux approximation, only the distribution of σ_t and the conditional expectations of partial cross-sections $\mathbb{E}[\sigma_x|\sigma_t]$ are required to compute properly self-shielded reaction rates. Notably, our rigorous argument above removes the need to assume a narrow resonance flux, and therefore provides an important theoretical contribution in that regard justifying the conventional practice of neglecting joint cross-section correlation effects.

The work [190] provides analytical expressions for the distribution of the cross-section in the case of a single isolated Breit-Wigner resonance at zero kelvin. In contrast, our coming derivation is for an infinite sequence of resonances of equal spacing, which presents slightly different results. Referring to that work after this one, the similarity of the resulting expressions is clear. However, on top of only applying to a single resonance, Hwang's result of course only corresponds to a single spin state. The actual cross-section is composed of many summed spin states. Correspondingly, we use these results as only one piece of the puzzle to inform a practically useful form of the cross-section distribution. Importantly, this work also contributes proposed analytical forms for the conditional means of the partial cross-sections conditioned on the total in the single spin state, isolated Breit-Wigner resonance case. These results inspire our approach to that problem based on rational polynomials.

Studying this system which resembles the unresolved resonance sampling problem may yield insight or can be adjusted to model the fully complex sampling problem. We are able to obtain an approximation to the cross section as a limiting case for widely spaced resonances at zero temperature that matches real ENDF/BVIII.0 data quite well. To our knowledge such a result is first-of-a-kind.

We first consider the case of constant resonance widths and spacings, with randomly placed energies, with a single spin state present and at zero temperature. The zero kelvin cross section can be written as:

$$\sigma_t(E) = \sigma_{\text{pot}} + \frac{4\pi\Gamma_n g_J}{\Gamma_t k^2} \sum_{j=-\infty}^{\infty} \frac{\cos(2\phi) + \sin(2\phi)\frac{2(E-E_i)}{\Gamma_t}}{1 + \left(\frac{2(E-E_i)}{\Gamma_t}\right)^2} \quad , \tag{5.23}$$

which can be found from summing Eq. 5.1 and all reactions of Eq. 5.3. The above results

from noting that the term on the left of the next equation can be rewritten:

$$1 - \Gamma_n/\Gamma_t = \frac{\sum_{x\in\{\gamma,f,\text{compet}\}} \Gamma_x}{\Gamma_t} \quad , \tag{5.24}$$

and noting that this quantity is subtracted out in Eq. 5.1. This leads to the simplification that the total cross section only depends on the other reaction resonance widths through the total width term, with a direct dependence on the scattering width term.

We now define the mean nondimensional resonance width as:

$$\Delta = \frac{\Gamma_t}{2\langle \Delta E \rangle} \quad . \tag{5.25}$$

In our model problem with equidistant resonances, $E_j = j\langle \Delta E \rangle$, which allows us to rewrite Eq. 5.23,

$$\frac{(\sigma_t(E) - \sigma_{\text{pot}})\Gamma_t k^2}{4\pi\Gamma_n g_J} = \Delta \sum_{j=-\infty}^{\infty} \frac{\Delta\cos(2\phi) + \sin(2\phi)\left(\frac{E}{\langle \Delta E\rangle} - j\right)}{\Delta^2 + \left(\frac{E}{\langle E\rangle} - j\right)^2} \tag{5.26}$$

We now observe that the term on the right resembles taking the real part of a complex number's reciprocal $1/z$, found as $\Re\bar{z}/(\bar{z}z)$. In this case we pick $z = E/\langle \Delta E\rangle - j - i\Delta$, resulting in:

$$\frac{\Delta\cos(2\phi) + \sin(2\phi)\left(\frac{E}{\langle \Delta E\rangle} - j\right)}{\Delta^2 + \left(\frac{E}{\langle E\rangle} - j\right)^2} = \Re\left[\frac{\sin(2\phi) + i\cos(2\phi)}{E/\langle \Delta E\rangle + i\Delta + j}\right] \tag{5.27}$$

The linearity of the $\Re$ operator lets us factor it out of the summation. We then apply "the most interesting formula involving elementary functions" [195]:

$$\pi\cot(\pi x) = \lim_{N\to\infty} \sum_{j=-N}^{N} \frac{1}{x+j} \quad \forall x \in \mathbb{R}\setminus\mathbb{Z} \tag{5.28}$$

Assuming this holds in the upper complex plane (which appears numerically to be the case), this leads to a fascinating, useful result about equidistant, equi-width resonance ladders:

$$\frac{(\sigma_t(E) - \sigma_{\text{pot}})\Gamma_t k^2}{4\pi\Gamma_n g_J} = \Re\left[\left(\sin(2\phi) + i\cos(2\phi)\right)\pi\cot\left(\pi\left(E/\langle \Delta E\rangle + i\Delta\right)\right)\right] \tag{5.29}$$

We then apply the identity:

$$\Re[\cot(a+bi)] = \frac{\cot a \coth^2 b - \cot a}{\cot^2 a + \coth^2 b} \tag{5.30}$$

which leads to the expression:

$$\frac{(\sigma_t(E)-\sigma_{\text{pot}})\Gamma_t k^2}{4\pi\Gamma_n g_J} = \frac{\cos(2\phi)\coth(\pi\Delta)\left(\cot^2(\frac{\pi E}{\langle\Delta E\rangle})+1\right)+\sin(2\phi)\coth^2(\pi\Delta)\cot^2(\frac{\pi E}{\langle\Delta E\rangle})}{\cot^2(\frac{\pi E}{\langle\Delta E\rangle})+\coth^2(\pi\Delta)} \quad . \tag{5.31}$$

To make the notation more compact, we define the nondimensional total cross section as:

$$s(E) = \frac{(\sigma_t(E)-\sigma_{\text{pot}})\Gamma_t k^2}{4\pi\Gamma_n g_J} \quad . \tag{5.32}$$

Eq. 5.31 can lead to a few asymptotic regimes depending on Δ and ϕ. For p-wave and higher resonances, ϕ tends to become quite small. Some nuclides have a relatively high resonance spacing to width ratio, and we derive the distribution of this case, because it represents nuclides with the highest variance in the cross section.

5.2.1 Large values of Δ, e.g. ^{238}U's s wave

In this case, we have that $\coth^2(\pi\Delta) \gg \coth(\pi\Delta)$, because Δ is small. As a consequence, we neglect the first term in the numerator of Eq. 5.31, leading to the expression:

$$\frac{(\sigma_t(E)-\sigma_{\text{pot}})\Gamma_t k^2}{4\pi\Gamma_n g_J} = \frac{\sin(2\phi)}{\frac{\cot\left(\frac{\pi E}{\langle\Delta E\rangle}\right)^2}{\coth(\pi\Delta)^2}+\tan\left(\frac{\pi E}{\langle\Delta E\rangle}\right)^2} \quad . \tag{5.33}$$

Finally, we have arrived at an expression that can be solved for E in closed form as two roots of a quadratic equation in $\tan\left(\frac{\pi E}{\langle\Delta E\rangle}\right)$. Of course, there are an infinite number of solutions to the equation $\tan x = c$ function for any given $c \in \mathbb{R}$. This is both physical and beneficial. Because we assume E to be uniformly distributed over the whole real line, it lets us restrict our attention to a single period of the tangent function. With this, we can solve the quadratic and find two possible solutions:

$$\tan\left(\frac{\pi E}{\langle\Delta E\rangle}\right) = \frac{1}{2}s^{-1}\sin(2\phi) + \sqrt{\frac{1}{4}s^{-2}\sin^2(2\phi)-\coth(\pi\Delta)^{-2}} \quad , \tag{5.34}$$

or

$$\tan\left(\frac{\pi E}{\langle\Delta E\rangle}\right) = \frac{1}{2}s^{-1}\sin(2\phi) - \sqrt{\frac{1}{4}s^{-2}\sin^2(2\phi)-\coth(\pi\Delta)^{-2}} \quad . \tag{5.35}$$

Fig. 5.6 shows how these equations apply to different parts of a scattering-dominated resonance in the total cross section. Recalling the fact that if $f(x)$ is a monotonic function, and

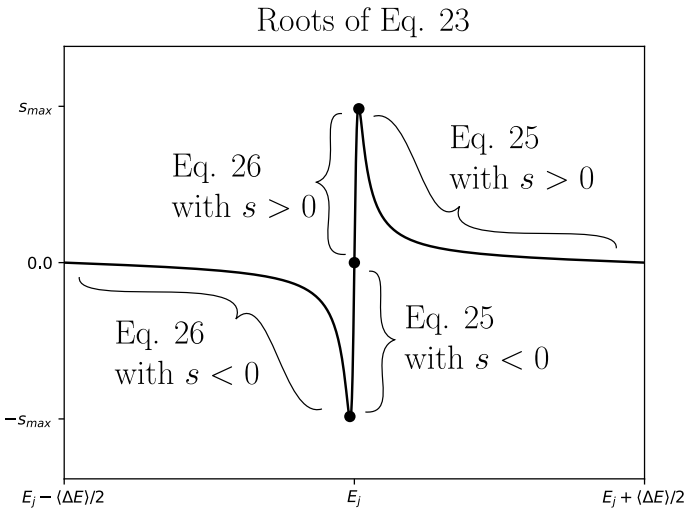


Figure 5.6: The quadratic tangent approximation can be analytically inverted to find its probability density function.

X has the probability distribution $p_X(x)$, it follows that $Y = f(X)$ has the distribution

$$p_Y(y) = \left|\frac{\mathrm{d}f^{-1}(y)}{\mathrm{d}y}\right| p_X(f^{-1}(y)) \quad . \tag{5.36}$$

Of course, $\sigma_t(E)$ is far from a monotonic function. We claim, though, that it can be broken into an increasing part and a decreasing part over its period. Eq. 5.37 can be generalized by noting that if f increases and decreases monotonically over N intervals I_j, we have that Y follows a mixture distribution:

$$p_Y(y) = \sum_{j=1}^{N} \left|\frac{\mathrm{d}f_j^{-1}(y)}{\mathrm{d}y}\right| p_X(f_j^{-1}(y))\mathbb{P}[x \in I_j] \quad . \tag{5.37}$$

Because the function f^{-1} is not unique, f_j^{-1} refers to the inverse over the interval I_j where it is monotonic. We also point out that if for a given value of y, $y \notin \mathrm{Dom}(f_j^{-1})$ implies that term of the summation needn't be considered.

With this in mind, we point out that the cross section distribution will rigorously take the form:

$$p_{\sigma_t}(\sigma_t) = \sum_{j=-\infty}^{\infty} \mathbb{P}\left[E \in [E_j - \langle\Delta E\rangle/2, E_j + \langle\Delta E\rangle/2]\right] \sum_{k=1}^{2} \frac{1}{\Delta E_k} \left|\frac{\mathrm{d}E_{j,k}}{\mathrm{d}s}\frac{\mathrm{d}s}{\mathrm{d}\sigma_t}\right| \quad . \tag{5.38}$$

Where the j index indicates the infinity of possible intervals resulting from the inversion of the tangent function of Eq. 5.34 or 5.35. The k index indicates taking within the j possible

values one of the two branches given by respectively Eq. 5.34 and 5.35. Because all the resonances are identical in this simplified model, and we take E to be uniform over the real line, we have that

$$\sum_{j=-\infty}^{\infty} \mathbb{P}\left[E \in [E_j - \langle \Delta E \rangle/2, E_j + \langle \Delta E \rangle/2]\right] = 1 \tag{5.39}$$

and that the $\frac{\mathrm{d}E_j}{\mathrm{d}s}$ terms are equal within each interval. Moreover, we note that in Fig. 5.6 the two of four inner sections contribute negligible probability in this case of small Δ. On top of that, the outer two branches are symmetric about $s = 0$. Consequently, we need only derive the behavior of the distribution from one branch for $s > 0$ and, if symmetric about $s = 0$, will remain valid for the $s < 0$ region too.

Putting this into symbols, it implies that:

$$p_{\sigma_t} \approx \frac{-2}{\langle \Delta E \rangle} \frac{\mathrm{d}s}{\mathrm{d}\sigma_t} \frac{\mathrm{d}}{\mathrm{d}s} \left(\frac{\langle \Delta E \rangle}{\pi} \arctan\left(\frac{1}{2} s^{-1} \sin(2\phi) + \sqrt{\frac{1}{4} s^{-2} \sin^2(2\phi) - \coth(\pi\Delta)^{-2}} \right) \right) \tag{5.40}$$

$$= \frac{1}{\pi} \frac{\mathrm{d}s}{\mathrm{d}\sigma_t} \frac{\sin(2\phi) s^{-2} + \frac{\sin(2\phi)}{s^3 \sqrt{s^{-2}\sin^2(2\phi) - 4\coth(\pi\Delta)^{-2}}}}{1 + \left(s^{-1}\sin(2\phi) + \sqrt{s^{-2}\sin(2\phi)^2 - 4\coth(\pi\Delta)^{-2}} \right)^2} \quad . \tag{5.41}$$

This is not an insightful expression, and deserves approximation. We multiply the top and bottom by s^2 and find the second numerator term negligible in this case.

$$p_{\sigma_t}(\sigma_t) = \frac{\pi^{-1} \frac{\mathrm{d}s}{\mathrm{d}\sigma_t}}{s^2 \sin(2\phi)^{-1} \left(1 + \tanh(\pi\Delta)^2\right) + \sqrt{\frac{1}{4}\sin(2\phi)^2 - s^2 \tanh(\pi\Delta)^2} + \frac{1}{2}\sin(2\phi)} \tag{5.42}$$

In order to put this into a form of a well-known distribution, we approximate the radical term by a parabola about $s = 0$, leading us to a simple model of unresolved resonances when the spacing is wide:

$$\sigma_t \sim \text{Cauchy}\left(\sigma_p, \frac{\pi^2 \sin(2\phi) g_J \langle \Gamma_n \rangle}{4 \langle \Delta E \rangle k^2} \right) \tag{5.43}$$

To test the validity of Eq. 5.43, we simulated the cross section distribution for the s-wave contribution to the ^{238}U total cross section at a 20 keV incident neutron energy on zero kelvin targets. The s-wave potential scattering contribution was also included in addition to the resonances. Parameters from the nuclear data file were used to calculate the parameters

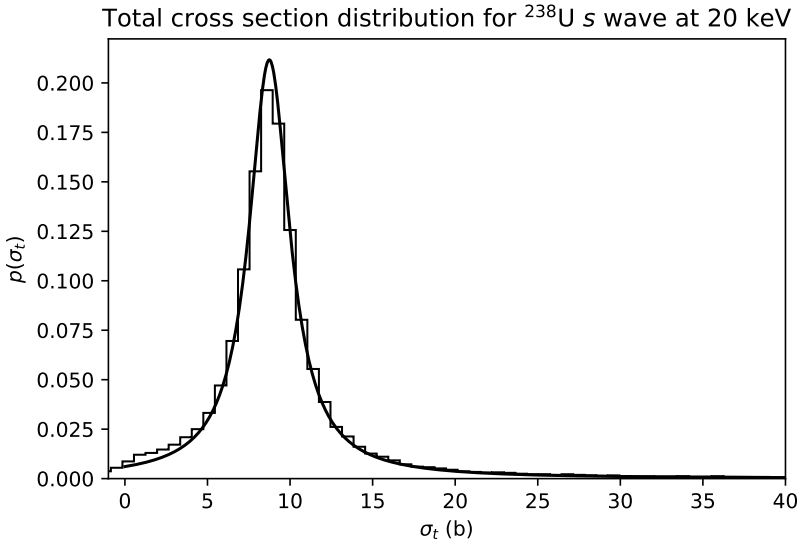


Figure 5.7: Eq. 5.43 can accurately model the total cross section distribution when the phase shift angle is low and the resonances are widely spaced.

appearing in Eq. 5.43. We emphasize that no curve fitting took place.

Fig. 5.7 shows excellent predictive ability of Eq. 5.43. A similarly good fit can be observed for other nuclides where the scattering phase shift is sufficiently small and the resonances widths are less than 1/20 the mean resonance spacing. Fortunately, as will be revealed soon, we only must motivate the modeling of the cross section distribution as nearly Cauchy when the resonance spacing to width ratio is large.

Small ϕ values

In the case that the scattering phase shift, ϕ, is large, the resulting distribution of the spin sequence's cross section tends to take on a -3/2 power law distribution. A large phase shift can be typically encountered in the higher angular momentum spin sequences, e.g. ^{238}U's p-wave as we demonstrate here.

If ϕ is relatively small, then the right side of Eq. 5.31 can be approximated as:

$$\frac{(\sigma_t(E) - \sigma_{\text{pot}})\Gamma_t k^2}{4\pi\Gamma_n g_J} = \frac{\cos(2\phi)\coth(\pi\Delta)}{\cos\left(\frac{\pi E}{\langle\Delta E\rangle}\right)^2 + \coth(\pi\Delta)^2 \sin\left(\frac{\pi E}{\langle\Delta E\rangle}\right)^2} \tag{5.44}$$

$$\tag{5.45}$$

By solving for E and differentiating w.r.t. σ_t, the distribution of σ_t can again be found. In this case, we again consider the case where $\Delta \approx 0$, and observe that the probability

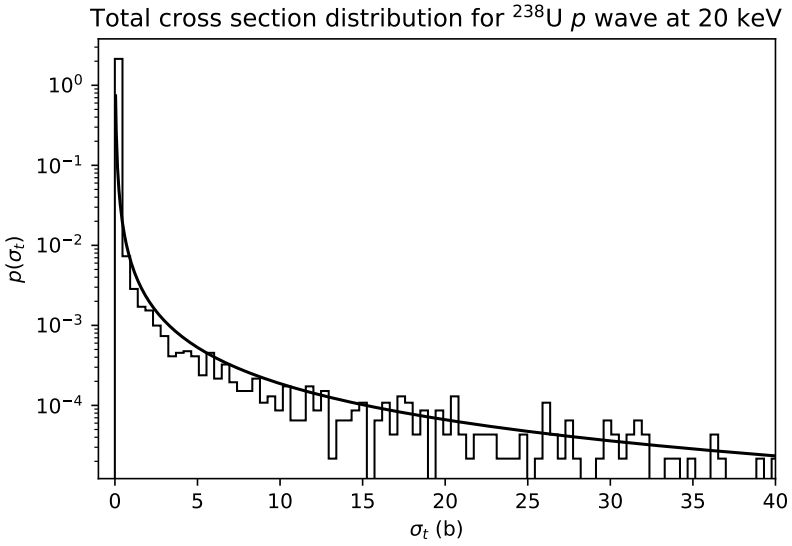


Figure 5.8: Eq. 5.47 qualitatively describes the cross section distribution as a -3/2 power law for widely spaced resonances and a small phase shift, e.g. as commonly encountered for p-wave resonances.

distribution approaches a $\sigma_t^{-3/2}$ power law distribution. Because the normalizing constant of this distribution is undefined if including the left endpoint of its support $\sigma_t = 0$, a reasonable minimum value of σ_t has to be chosen to compensate for the approximations introduced which have brought the distribution to the simple -3/2 power law form.

Using the following minimum cross section to normalize the distribution gives satisfactory results over a wide range of parameters when ϕ is relatively small.

$$\sigma_{min} = \frac{2\langle\Gamma_n\rangle g_J}{\pi\langle\Gamma_t\rangle k^2}\cos(2\phi)\tanh(\pi\Delta) \tag{5.46}$$

The distribution of σ_t for large values of ϕ can then be considered to be:

$$p_{\sigma_t}(\sigma_t) \approx \frac{\sqrt{\sigma_{min}}}{2\sigma_t^{3/2}} \quad . \tag{5.47}$$

Fig. 5.8 shows how the -3/2 power law fits the qualitative behavior of the distribution of the total cross section of the p wave resonances quite well.

In sum, we have shown that in the case of widely spaced resonances, two asymptotic regimes exist corresponding to dominant $\sin(2\phi)$ or dominant $\cos(2\phi)$ terms. Intermediate cases could likely be modeled as a mixture of a -3/2 power law and Cauchy distribution, although we show later that another distribution can model almost any unresolved resonance total cross section with acceptable accuracy.

5.2.2 The intermediate Δ region, e.g. ^{239}Pu

To our knowledge, the only characterization of the behavior of the unresolved resonance cross section distribution was presented in [190]. Hwang observed that in most nuclides of practical interest, the total cross section probability distribution tends to form a unimodal distribution exhibiting two exponential tails. In other words, Hwang observed that semilogarithmic in ordinate plots of the density function of the total cross section for actinides tends to exhibit two straight lines. This behavior can be observed for the intermediate values of Δ in Figs. 5.4 and 5.5. Hwang observed that the hypoexponential distribution matches precisely this behavior, and he showed in [190] that this can be made to adequately fit the unresolved resonance cross section distribution for a few isotopes.

In our experience, however, Hwang's proposed hypoexponential model fails to model a whole library of nuclides. There exist other unimodal distributions which exhibit constant logarithmic derivative behavior on either side of the mode, the Laplace distribution, for example. It was precisely this straight line behavior that inspired Barndorff-Nielsen to develop the hyperbolic distribution [196] to fit the distribution of sizes of wind-blown sand grains in log-log space. We found some moderate success fitting the hyperbolic distribution to unresolved resonance cross section distributions, but another yet-to-be-discussed distribution provides better fits across the full range of Δ.

Rather than undertake the arduous task of provisioning theory showing that the distribution of a randomized sum of single-level Breit-Wigner resonances tends to yield exponential tails, we follow the lead of Hwang and merely relegate this as an empirical observation. The work [197] rigorously demonstrates how to determine the tail probability behavior of randomly weighted sums of dependent random variables. If the SLBW shapes of the form $1/(1+x^2)$ and $x/(1+x^2)$ are treated as dependent random variables (given that the resonance spacing is random) and the factors multiplying them are indepdentent, the framework of [197] can likely be applied to explain Hwang's observations.

5.2.3 The Small Δ region, e.g. ^{235}U

Lastly, Figs. 5.4 and 5.5 show that for small values of Δ, or in other words tightly spaced resonances, the distribution of the total cross section becomes concave on a log-y plot. In fact, the cross section becomes reasonably approximated by a normal distribution.

To explain this fact, we recall that in the central limit theorem, a sum of independent and identically distributed random variables tends towards having a normal distribution. Similarly, a sum of a few independent and identically distributed random variables might be approximated as normally distributed. If we consider the sum over SLBW resonances with

random widths and spacings:

$$\sigma_t(E) - \sigma_{\text{pot}} \propto \sum_{j=-\infty}^{j=\infty} \frac{\hat{\Gamma}_n}{\hat{\Gamma}_t} \frac{\cos(2\phi) + \sin(2\phi)\frac{2(E-E_j)}{\hat{\Gamma}_t}}{1 + \left(\frac{2(E-E_j)}{\hat{\Gamma}_t}\right)^2} \tag{5.48}$$

we can observe that the $\hat{\Gamma}_n/\hat{\Gamma}_t$ values in the summation are all independent from each other. Moreover, they are identically distributed. When the resonances are spaced together closely, the SLBW shape terms on the right under the summation tend to the same value, as $E - E_j$ is much smaller than $\hat{\Gamma}_t$ when the resonances are tightly spaced. Consequently the first few terms in the series nearest the value of E will have similar values of the SLBW shape modulation, since they tend towards unity as $E - E_j$ becomes small. It then follows that the summation consists of a sum over nearly independent and identically distributed terms plus a contribution from far-away resonances.

Future work could more rigorously explore this. It should be possible to rigorously prove that in the limit as $\Delta \to 0$, when properly normalized by some factor, the total cross section should tend towards a normal distribution. We leave this for future work as a more rigorous treatment here adds little to the forthcoming discussion. The key observation is that for small values of Δ, the cross section tends towards a normal distribution.

5.2.4 A Reasonable Form of the Distribution of σ_t For All Regimes

Applicability of the NIG Distribution

A probability distribution defined by a small number of parameters which can exhibit the three properties in our foregoing discussion could satisfactorily model the distribution of σ_t in the unresolved region. Firstly, the distribution must be able to approximate a Cauchy distribution in order to match the behavior of the regime demonstrated in Subsection 5.2.1. Secondly, the distribution should be able to exhibit linear tails in log space to cover the case analyzed in subsection 5.2.2, and if possible, the slope of these tails should be an adjustable parameter. Finally, the distribution should be able to match the shape of a normal distribution as exhibited in Subsection 5.2.3.

The normal inverse Gaussian (NIG) distribution [198] accomplishes exactly these tasks with only four parameters. It is defined by the density function:

$$\mathcal{NIG}(x; \mathfrak{a}, \mathfrak{b}, \mu, \delta) = \frac{\mathfrak{a}\delta K_1\left(\mathfrak{a}\sqrt{\delta^2 + (x-\mu)^2}\right)}{\pi\sqrt{\delta^2 + (x-\mu)^2}} \exp\left(\delta\gamma + \mathfrak{b}(x-\mu)\right) \quad , \tag{5.49}$$

where $\gamma = \sqrt{\mathfrak{a}^2 - \mathfrak{b}^2}$.

In the case of widely spaced resonances, the distribution of σ_t resembles a Cauchy distribution as shown by Fig. 5.7. Near the mode of the distribution, the NIG distribution can approximate a Cauchy distribution. To see this, we recall that for small x, $K_1(x) \sim x^{-1}$. We then have that for $x \approx \mu$ and small δ,

$$\mathcal{NIG}(x; \mathfrak{a}, 0, \mu, \delta) \sim \frac{1}{\pi\delta\left(1 + (\frac{x-\mu}{\delta})^2\right)} \tag{5.50}$$

which is exactly a Cauchy distribution. The $\mathfrak{a}$ parameter is still free and can still model tail effects encountered where the assumptions of Subsection 5.2.1 break down.

Secondly, as discussed in Subsection 5.2.2, a distribution used to model the distribution of σ_t should be able to model tails that are roughly linear on a log plot. It was this behavior that suggested to Hwang [190] that a hypoexponential distribution might suffice. Indeed, the NIG distribution has this property. To see it, we we apply the asymptotic estimate for large x that $K_1(x) \sim \sqrt{\frac{\pi}{2x}}e^{-x}$. It straightforwardly follows that the asymptotic logarithmic derivatives are $\beta - \alpha$ for $x \to \infty$, and $\beta + \alpha$ for $x \to -\infty$. Consequently, it forms straight lines on a log plot in the limit, as originally suggested by Hwang. We note that other distributions in the generalized hyperbolic [196] family of distributions satisfy this property, but the NIG additionally approximates a Cauchy distribution well near its peak.

Finally, Subsection 5.2.3 suggests that a normal distribution might reasonably approximate the cross section distribution for small values of Δ. For this point, we direct the reader to [198] to see that the NIG distribution can arbitrarily closely approximate a Gaussian.

One possible critique of the argument in favor of the NIG distribution's utility here concerns the presence of multiple spin sequences. The previous argument only considered a single spin state sequence of resonances. Noting that the spin sequences are independent of each other (save for zeroed out negative cross sections due to the flaws of SLBW), the overall distribution of σ_t is a convolution of the distributions of the partial total cross sections from each spin sequence. Because the NIG distribution is closed under convolution [199], the resulting sum over several spin states will also be NIG-distributed.

Another potential hole in this reasoning results from the effect of temperature. All the forthcoming reasoning relied on the zero kelvin cross section. The logic may not necessarily hold at nonzero temperature. Consequently, we naively suppose that the NIG distribution might work across a variety of temperatures, which our results demonstrate empirically to be certainly true.

Sampling The Bounded NIG Distribution

The whole family of generalized hyperbolic distributions can be interpreted as infinite mixtures of normal random variables with variances distributed by generalized inverse Gaussian distributions [198]. Specifically, samples z which have $z \sim \mathcal{IG}(\delta(\mathfrak{a}^2 - \mathfrak{b}^2)^{-1/2}, \mathfrak{a}^2 - \mathfrak{b}^2)$ which are used to sample $x \sim \mathcal{N}(\mu + \mathfrak{b}z, z)$ result in $x \sim \mathcal{NIG}(\mathfrak{a}, \mathfrak{b}, \mu, \delta)$. Equivalently,

$$\mathcal{NIG}(x; \mathfrak{a}, \mathfrak{b}, \mu, \delta) = \int_0^\infty \mathcal{IG}(z; \delta/\gamma, \delta^2)\mathcal{N}(x; \mu + \beta z, z)\,\mathrm{d}z \quad . \tag{5.51}$$

This fact can be combined with the efficient sampling procedure for inverse Gaussian distributed variables presented in [200] to efficiently sample a NIG-distributed random variable. Algorithm 13 presents the combined IG sample and variance mixture sampling algorithm. As a rule of thumb, shallow branches can run efficiently on a modern GPU, so we can expect this algorithm to outperform probability tables on our target architecture.

Input : $\mathfrak{a}, \mathfrak{b}, \mu, \delta$
$\gamma \leftarrow \sqrt{\mathfrak{a}^2 - \mathfrak{b}^2}$;
Sample $w_0 \sim \mathcal{N}(0, 1)$;
$w \leftarrow \delta w_0^2/\gamma$;
$c \leftarrow (2\delta\gamma)^{-1}$;
$x \leftarrow \delta/\gamma + c\left(w - \sqrt{w(4\delta^2 + w)}\right)$;
Sample $u \sim \mathrm{Unif}([0, 1])$;
if $u \geq \delta/(\gamma(\delta/\gamma + x))$ **then**
  $z \leftarrow \delta^2/(x\gamma^2)$;
else
  $z \leftarrow x$;
end
return sample from $\mathcal{N}(\mu + \beta z, z)$;

Algorithm 13: Algorithm to sample a variable distributed as $\mathcal{NIG}(\mathfrak{a}, \mathfrak{b}, \mu, \delta)$.

5.3 Results

5.3.1 Generation of a Finely Resolved URR Library

We generated a highly resolved library of σ_t distributions for every energy point of every nuclide exhibiting unresolved resonance behavior in ENDF/B-VIII.0 [161]. Two hundred bin histograms were used to tally the total cross section probability density and conditional expectations of the partial cross section given the total. In contrast, default OpenMC nuclear

data processing options [187] use NJOY to build a twenty bin histogram, with sixty-four independent resonance ladders. While NJOY samples the ladder at ten thousand uniformly sampled energy points, we sampled each ladder at one hundred thousand random points. Our code still employed sixty-four independent ladders for each energy point of each nuclide.

Creating a new program devoted to highly resolved unresolved resonance probability table generation proved necessary. On an AMD Ryzen 7950X CPU, NJOY sampled the full set of probability tables for ^{238}U over six different temperatures in 1593 s. Assuming optimal scaling [2], NJOY would take 15,930 s to build the highly sampled probability tables that our code generated. In contrast, our code took 0.1 s to sample the resonance locations and sort the energy sample grid of each ladder for ^{238}U, with 2.5 s spent reconstructing the cross section on 100,000 points for each of the six temperatures. The six temperatures are 250, 294, 600, 900, 1200, and 2500 kelvin.

5.3.2 Fitting the σ_t Model

Because the moments of the NIG distribution are available in closed form, a method of moments estimator could be used to fit the distributions of σ_t; however, the σ_t samples tend to be bounded above and below, albeit while following a NIG distribution closely over their support. Because the standard NIG distribution has infinite support, the standard expressions for the moments do not apply. Moreover, the moments of a NIG distributed random variable restricted to a contiguous section of the real line do not have analytically tractable expressions, to our knowledge.

Without a method of moments estimator easily available, one might similarly consider a maximum likelihood approach to estimating the parameters of a NIG distribution. Because we sample a total of 6,400,000 realizations of σ_t per temperature and energy point, the resultant nonlinear fitting problem would be inconveniently computationally expensive.

Due to these barriers, we fit the density function of a NIG distribution restricted to the same range as that presented by the first ladder sampled of the σ_t data. The normalizing constant must be numerically calculated to find the correctly scaled value of the density. To fit the densities, we minimize the Hellinger distance,

$$H(P,Q) = \frac{1}{\sqrt{2}} \left(\int \left(\sqrt{P(x)} - \sqrt{Q(x)} \right)^2 dx \right)^{1/2} \quad , \tag{5.52}$$

approximated by numerical integration over our finely divided histograms. This distance

[2] [177] points out that the energy grid sorting algorithm of NJOY is $\mathcal{O}(n^2)$, so the actual number would be higher.

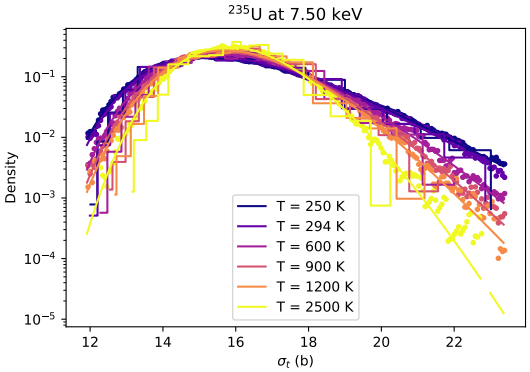
metric has been shown to exhibit desirable properties [201]. The Hellinger distance can be understood as proportional to the root mean square difference between the square roots of two density functions. The use of Hellinger distance over standard least squares minimization between the density function provides a desirable balance between minimizing error in the most probable regions and properly modeling the behavior in the tails of the probability distribution. In contrast, we found minimizing the Kullback-Leibler divergence excessively prioritized matching the values in the tails of the distribution, and naive least-squares minimization between the density function difference excessively prioritized matching the densities around the distribution's mode.

The numerical integration for the NIG distributions normalizing constant was carried out with SciPy [202]. Likewise, the nonlinear optimization problem presented by minimizing the Hellinger distance was solved with the Nelder-Mead method wrapped by SciPy. We found that applying bounds to the NIG parameters improved the reliability of the optimizer substantially, namely that $\alpha > 0$, $\beta > 0$, $\delta^2 > 0$, and $\mu > \min_i \sigma_{t,i}$, where $\sigma_{t,i}$ represents the realizations of the total microscopic cross section. The bound $\beta > 0$ strictly speaking is not required of the NIG distribution; it is instead based on our observation that the σ_t is exclusively skewed to large values and never otherwise.

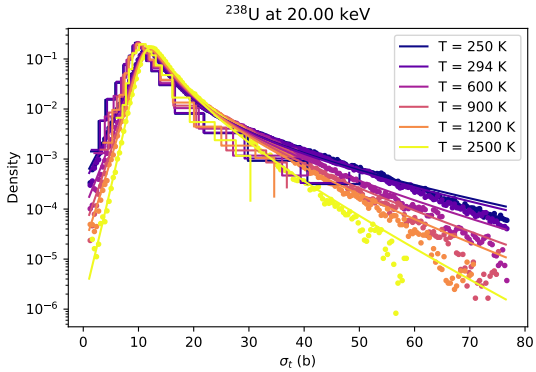
Still, the nonlinear optimizer crashed on many nuclides due to NaN being generated. These were found to result from the multiplication of underflowed zeros and overflowed infinities respectively generated by the K_1 term of Eq. 5.49 and the exponential term. The code must branch to handle this asymptotic case separately. We chose the criterion for the asymptotic branch as any argument to the exponential function over seven hundred.

Some fraction of negative σ_t observations may be encountered when using the SLBW model to generate unresolved resonance region models. In practice, negative cross sections on a probability table are modeled instead as some small value; NJOY arbitrarily uses a microbarn. In order to maintain the same behavior as NJOY, we fit to the dataset including negative values but return a microbarn total cross section when negative cross sections are sampled. More physically realistic methods for sampling the cross section distribution such as R-matrix limited [182] may reveal new behaviors in the small cross section region that require a new form of modeling, and our method could incorporate that by using it as part of a mixture distribution.

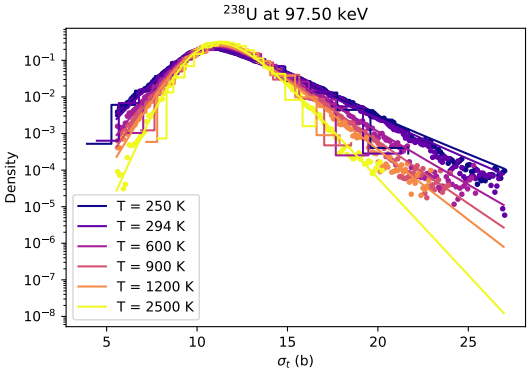
A select few nuclides of importance to reactor simulation are presented in Fig. 5.9 and 5.10. The plots show some representative results of obvious importance for ^{235}U, ^{238}U, ^{239}Pu, and ^{240}Pu. ^{107}Ag has been included due to its use in control rods and natural abundance around 52%. ^{137}Cs has been presented due to its prominence in radiological source terms. We include some results for ^{55}Mn due to it constituting all natural manganese; we have



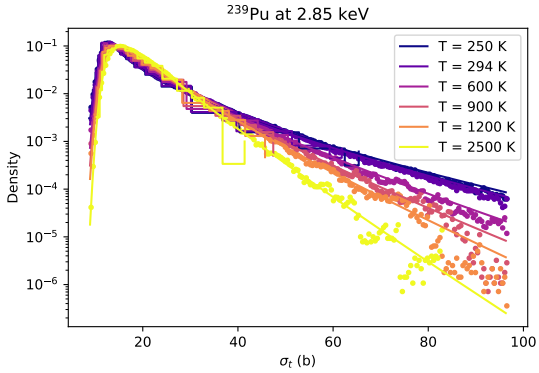
(a) ^{235}U σ_t distribution at 7.5 keV.



(b) ^{238}U σ_t distribution at 20.0 keV.



(c) ^{238}U σ_t distribution at 97.5 keV.

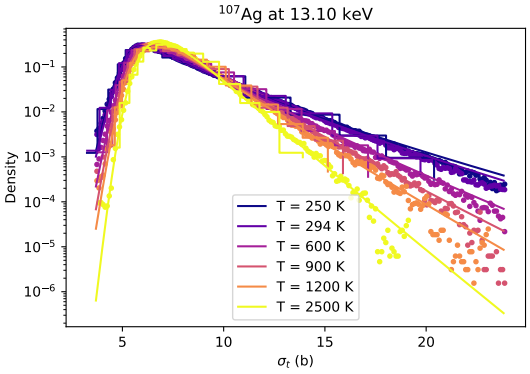


(d) ^{239}Pu σ_t distribution at 2.85 keV.

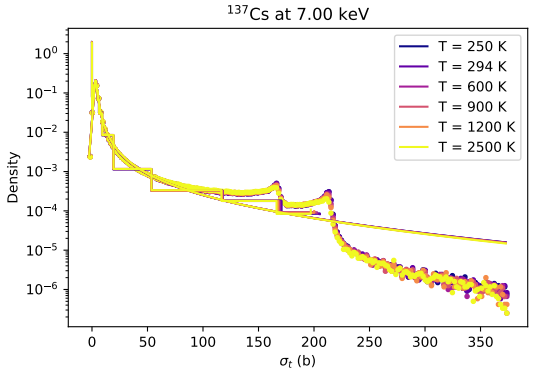
Figure 5.9: Comparison of pointwise σ_t probability density estimates from our `urrtools.py` simulation (points), fits to NIG distributions (smooth lines), and NJOY PURR results (stairsteps).

observed that in both our code and in NJOY the SLBW approach yields a large fraction of negative cross sections. Notably, in Fig. 5.10c we have not zeroed out the negative values in order to show the fit of our model to the raw data.

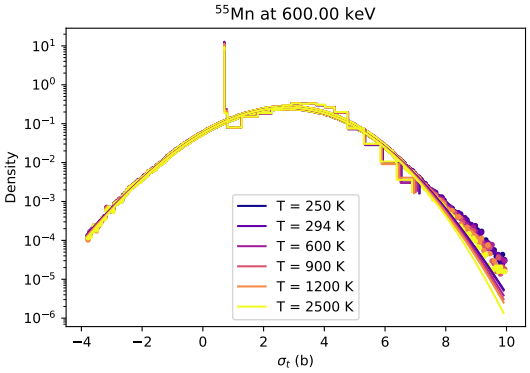
Figure 5.11 shows one case where a poor match against NJOY was discovered. Our initial hypothesis for this discrepancy pertained to the biased resonance width sampling, but a patched version of NJOY proved that to be false. Given the large number of numerical approximations present in NJOY, we suspect some difference in our modeling choices, for example the number of "outer" resonances not in the sampling window, likely plays a role here. Further research seeking to deploy this method in a practical setting will have to resolve this discrepancy.



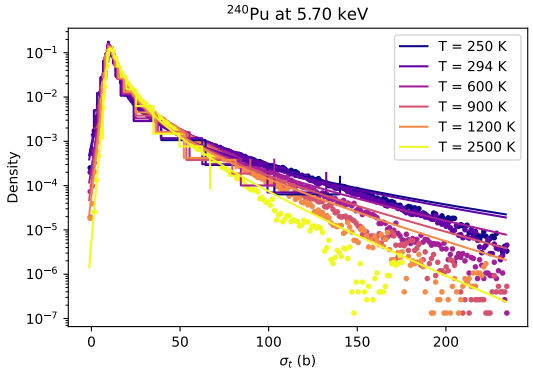
(a) ^{107}Ag σ_t distribution at 13.1 keV.



(b) ^{137}Cs σ_t distribution at 7.0 keV.



(c) ^{55}Mn σ_t distribution at 600 keV.



(d) ^{240}Pu σ_t distribution at 5.70 keV.

Figure 5.10: Same as Fig. 5.9 but for some other nuclides.

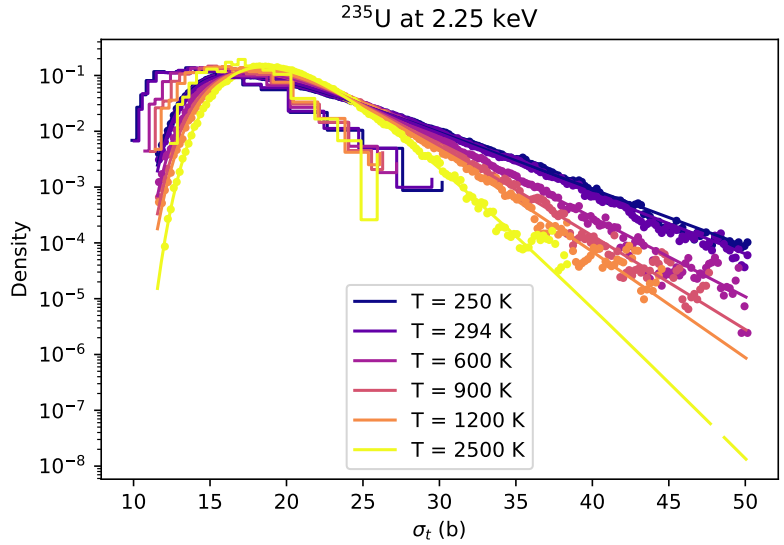


Figure 5.11: ^{235}U σ_t distribution at 2.25 keV matches NJOY poorly.

5.3.3 Fitting the Partial Cross Section Model

As discussed in Subsection 5.1.3, the conditional expectations of the partial cross section components suffice to obtain properly self-shielded fluxes in a Monte Carlo simulation. Consequently, we must obtain a curve fit that outputs the expected partial cross section of interest given a value of σ_t. In order to yield a clear benefit over the traditional probability tables approach, the model should require a fairly small amount of data yet approximate the cross section curve accurately.

Ideally, a physically motivated expression for $\mathbb{E}[\sigma_x|\sigma_t]$ could be obtained. Our preliminary investigation, under some crude assumptions, suggested that the absorption and fission partial cross sections might be approximated accurately using a rational polynomial expression. Regardless, rational polynomial functions have universal approximation properties to analytic and more generally Lipschitz-continuous functions [203], so further justification of this approach is unnecessary.

On top of this, Hwang's final paper [194] also shows that the expression for $\mathbb{E}[\sigma_x|\sigma_t]$ should take the form of a ratio of square roots, which may be closely approximated by rational polynomials $\sqrt{\sigma_t}$ when $\sqrt{\sigma_t}$ is not near its upper or lower limiting values. This stands as even more reason to believe that low-order rational approximations can match the desired behavior closely.

NJOY or `urrtools.py` both output a tabulation of the expectation value of each partial cross section contributing to the total within in each histogram bin. The task at hand then, is simple curve-fitting to these outputs. The traditional probability table approach would save these as points to be interpolated. The benefit of a curve fitting approach is twofold: firstly the noise from the simulation used to generate the data is suppressed by using a low number of degrees of freedom, and secondly the function becomes more applicable for GPU applications which, at peak performance, employ a higher floating point to memory access operation ratio. Moreover, the memory access pattern is predictable, unlike the probability table approach. Preventing memory access divergence is essential to utilize GPUs efficiently.

Rational polynomial fitting is a nontrivial research problem itself, and algorithms specialized for it typically must be used as discussed in [204]. The recently presented adaptive Antoulas-Anderson (AAA) algorithm [205] is, in our opinion, the most straightforward and powerful means of rational polynomial fitting at present. The strength of this algorithm originates from its representation of rational functions in rational barycentric form,

$$r(z) = \frac{n(z)}{d(z)} = \frac{\sum_{j=1}^{m} \frac{w_j f_j}{z - z_j}}{\sum_{j=1}^{m} \frac{w_j}{z - z_j}} \quad , \tag{5.53}$$

where $n(z)$ and $d(z)$ are polynomials, and z_j are evaluation points to evaluate function values f_j at. The coefficients of the polynomials $n(z)$ and $d(z)$ need not be computed. The work [206] discusses the numerical advantages that the barycentric rational form possess. For a certain choice of w_j, the classical Lagrange interpolant is obtained. For the choice $w_j = \{1, -1, 1, -1, \ldots\}$, a smooth rational interpolant is obtained [207].

Our initial attempts at fitting to the conditional expectations of the partial cross sections using general rational polynomials of the direct rational form $n(z)/d(z)$ were clear failures. The coefficients in the numerator and denominator can have enormous differences in their influence on the fit quality: for example high order monomial coefficients clearly vary the fit more than the lower order parts. Using a Chebyshev basis for each polynomial might have remediated this issue, but a nonlinear fitting approach using the barycentric rational form offers superior numerical characteristics and interpretability.

An initial guess to a nonlinear fitting problem like this can accelerate computations appreciably, especially considering the problem of curve fitting at every energy point, at each temperature, for each nuclide. Barycentric rational forms furnish a straightforward approach to this. The number of summation terms m is decided ahead of time in building the library; $m = 5$ appears to be more than enough for the problem at hand. Next, five points z_j have to be chosen. Because the data at hand comes from conditional expectations within each bin of a histogram of the total cross section, the points z_j can be distributed evenly in σ_t probability.

To find the points, the cumulative distribution function approximated by summing histogram bins can be found, calling this function $C(\sigma_t)$. Then, the z_j are $z_j = C^{-1}(j(\max \hat{\sigma}_t - \min \hat{\sigma}_t)/(m-1) + \min \hat{\sigma}_t)$. The interpolation points thus cluster around the most likely values of σ_t. The starting guess values for the partial cross section, f_j, are the conditional expectations of partial cross sections from each histogram bin. The weights are set to Berrut's set, $\{1, -1, 1, \ldots\}$.

At this point, another layer of complexity can be added. The weights and interpolation points can be shared among all the temperatures data must be tabulated at, while the interpolation values f_j can vary with respect to temperature. We find fits by solving:

$$\min_{\vec{sigma}_t, \vec{w}, \vec{\sigma}_x(T_t)} \sum_{t=1}^{n_t} \sum_{i}^{n_\sigma} \tilde{P}(\sigma_{t,i})^2 \left(\mathbb{E}\left[\sigma_x(T_t, \sigma_{t,i}) | \sigma_{t,i}\right] - b(\vec{\sigma}_t, \vec{w}, \vec{\sigma}_x(T_t), \sigma_{t,i}) \right) \tag{5.54}$$

where $\sigma_x(T_t, \sigma_{t,i})$ is the random partial cross section on the histogram bin i. $b(\vec{\sigma}_t, \vec{w}, \vec{\sigma}_x(T_t), x)$ is the barycentric rational approximation formed by using interpolation points $\vec{\sigma}_t$, weights $\vec{w}$, and interpolation values $\sigma_x(T_t)$, evaluated at the point x. The outer sum in Eq. 5.54 is over temperature points T_t, and the inner sum is over $\sigma_{t,i}$, the midpoints of each histogram bin.

Notably, the sum is weighted by the square of the probability density function estimated on the histogram. Squaring is not *per se* necessary, but we found the resulting fits to be most aesthetically appealing if using this choice. The black-box Nelder-Mead optimizer from `scipy` provided reasonable results in a reasonable amount of compute time. The use of the barycentric rational form appears to transform the problem to a space of coordinates which is reasonably well-conditioned and lacking in local optima, as our experience using black-box solvers to solve Eq. 5.54 has been generally positive. The same can not be said of directly using a rational polynomial form.

After obtaining a fit, the parameters of this model offer excellent interpretability. The interpolation points are where the conditional expectation curve is evaluated, and the interpolation values again are easy to interpret. On the the other hand, interpreting the weights is a bit more opaque. At least, the majority of fit parameters have a clear physical meaning.

Because σ_t is defined as the sum of a capture, elastic scattering, and fission component, we only need to store the fits to the capture and fission components. In sampling a value of σ_t and its associated partials, the elastic scattering component can be found by subtracting the capture and fission parts from the total.

Figures 5.12-5.14 show some examples of key nuclides' conditional expectations of partial cross sections that were fit by this procedure.

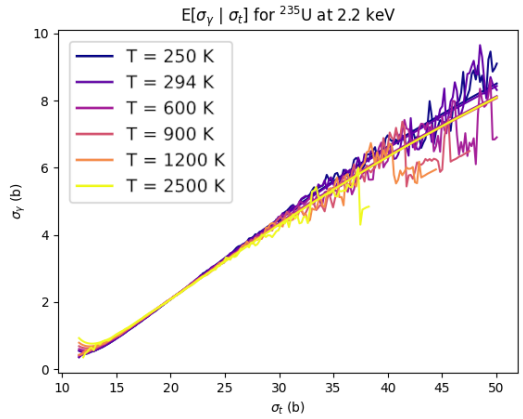
We call the modeling approximation of NIG-distributed σ_t values and $\mathbb{E}[\sigma_x|\sigma_t]$ approximated by barycentric rational forms AURA (Analytic Unresolved Resonance Algorithm).

5.4 Discussion

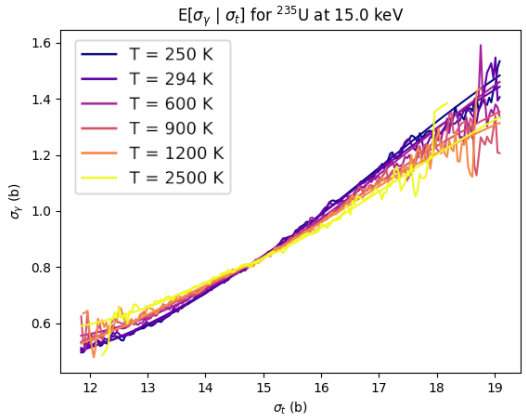
We have illustrated the applicability of fitting NIG distributions to the URR cross section distribution over a variety of temperatures in Fig. 5.9 and 5.10. The NIG distribution has been shown to fit the distribution of the cross section over a range of energies and temperatures for a variety of nuclides. Fig. 5.9a shows the new method performing well for ^{235}U at the lower energy end of the URR range.

Figs. 5.9b-5.9c demonstrate the applicability of our fitting technique to ^{238}U over two energies. The ratio of resonance width to spacing tends to increase as energy increases, and these plots demonstrate this when placed in the context of Subsecs. 5.2.1-5.2.3. The distribution at 20 keV has more clearly pronounced exponentially varying tails, while the distribution at 97.5 keV exhibits more parabolic behavior over a wider range of σ_t.

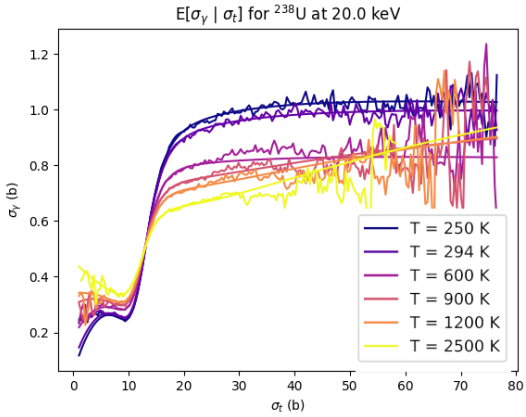
In the case of ^{55}Mn presented in Fig. 5.10c, we can observe that the single level Breit-Wigner model clearly falls short to a greater extent than for most other isotopes in this case. This particularly unphysical behavior deserves study in greater detail using the methods



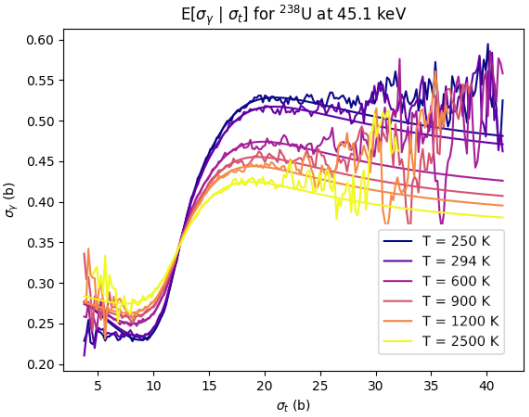
(a) ^{235}U conditional expectation of σ_γ given σ_t function at 2.2 keV.



(b) ^{235}U conditional expectation of σ_γ given σ_t function at 15 keV.

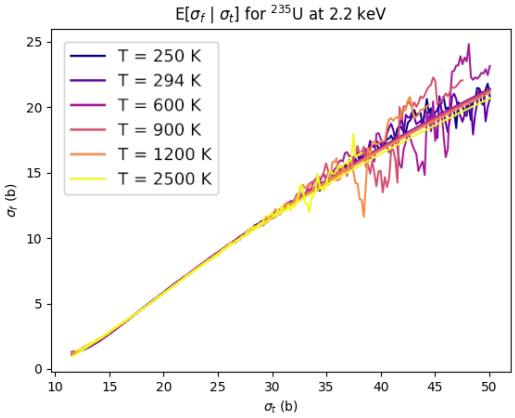


(c) ^{238}U conditional expectation of σ_γ given σ_t function at 20 keV.

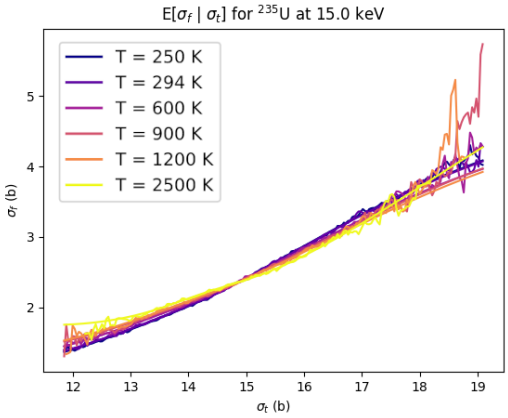


(d) ^{238}U conditional expectation of σ_γ given σ_t function at 45.1 keV.

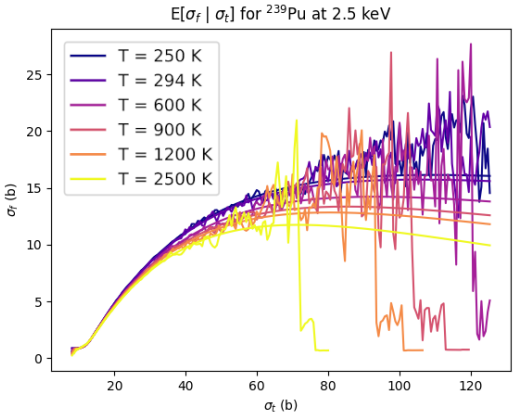
Figure 5.12: The $\mathbb{E}[\sigma_\gamma|\sigma_t]$ curves for ^{235}U and ^{238}U at two energies. The jagged lines are estimates from the `urrtools.py` simulation, and the smooth lines are fits obtained by solving Eq. 5.54.



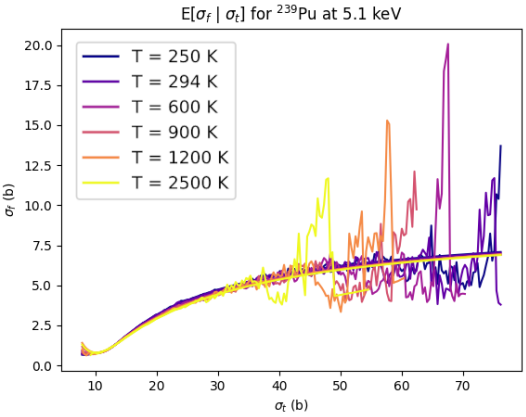
(a) ^{235}U conditional expectation of σ_f given σ_t function at 2.2 keV.



(b) ^{235}U conditional expectation of σ_f given σ_t function at 15 keV.

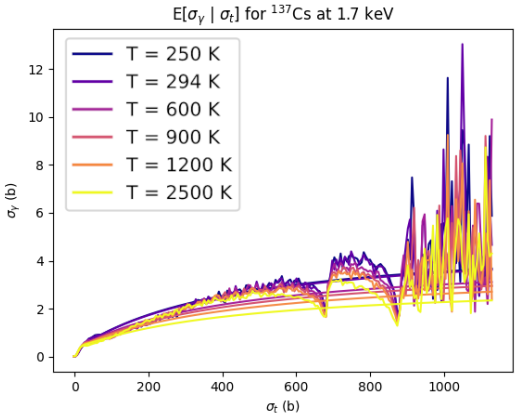


(c) ^{239}Pu conditional expectation of σ_f given σ_t function at 2.5 keV.

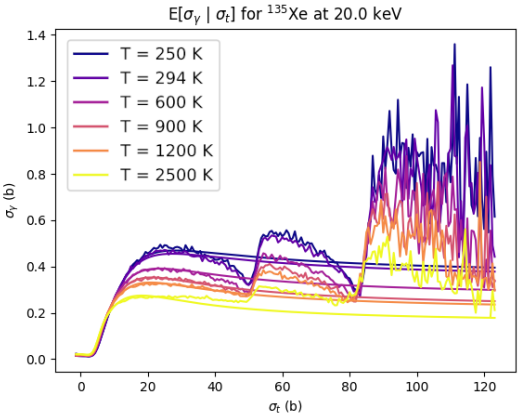


(d) ^{239}Pu conditional expectation of σ_f given σ_t function at 5.1 keV.

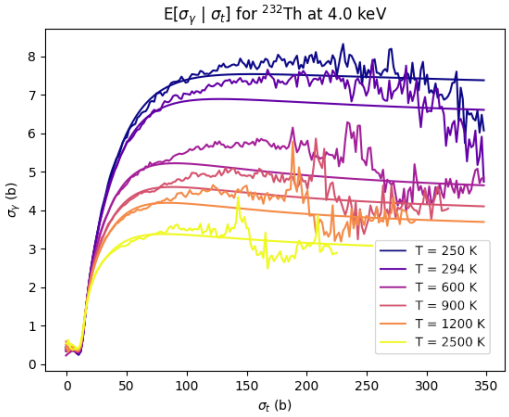
Figure 5.13: The $\mathbb{E}[\sigma_f|\sigma_t]$ curves for some key nuclides. Again, the jagged lines are estimates from the `urrtools.py` simulation, and the smooth lines are fits obtained by solving Eq. 5.54.



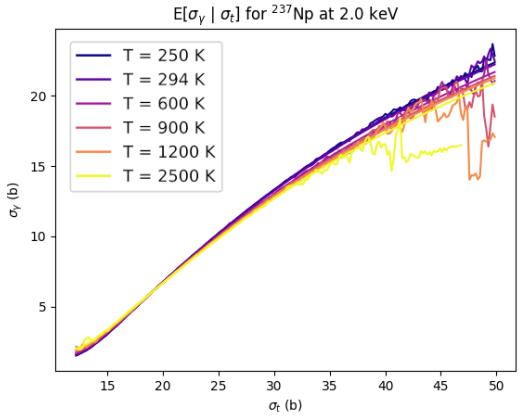
(a) ^{137}Cs conditional expectation of σ_γ given σ_t function at 1.7 keV.



(b) ^{135}Xe conditional expectation of σ_γ given σ_t function at 20 keV.



(c) ^{232}Th conditional expectation of σ_γ given σ_t function at 4 keV.



(d) ^{237}Np conditional expectation of σ_γ given σ_t function at 2 keV.

Figure 5.14: The $\mathbb{E}[\sigma_\gamma|\sigma_t]$ curves for some other key nuclides. Again, the jagged lines are estimates from the `urrtools.py` simulation, and the smooth lines are fits obtained by solving Eq. 5.54.

developed by [182], where otherwise somewhat small differences between the SLBW and R-matrix-limited approaches had been shown for key actinides.

The visible steep dropoffs of the cross-section in Figure 5.10b can be explained using Hwang's model [194]. Hwang showed that when considering an isolated Breit-Wigner resonance, a jump in the probability distribution before a steep dropoff can be observed. Figure 5.10b shows exactly this same effect but with the superposition of two independent spin sequences each with their own maximum cross-section values. It appears to be the case that neglecting this jump effect produces practically usable cross-section distributions for use in reactor analysis.

The logarithmic derivatives of the cross section distribution tend to vary linearly in temperature above a certain temperature. This would allow straightforward extrapolation to high temperatures. We have not thus found a satisfactory intuitive mathematical argument as to why this might be the case. One possibility stems from the fact that for sufficiently high temperature, it can be shown that a SLBW resonance peak tends to decrease in a manner proportional to $1/T$. Making a change of variables in the probability distribution for σ_t at zero kelvin to the high temperature case results in coefficients multiplying σ_t which are proportional to temperature. Unfortunately, the highest values of the cross section seem to originate not just from peaks in the cross section, but also exceptional cases where a few resonances stack together to produce a large cross section. Future work could more rigorously analyze this in order to potentially determine reasonable models for the temperature dependence of the distribution parameters used to model the total cross section.

In [182] the capability to generate probability tables based on rigorous R-matrix theory rather than summed SLBW resonances indicated small changes in the distribution around the smallest σ_t values. However, the binning of the grid was not fine enough to evince the actual structure of the probability distribution. We suspect that a distinct probability distribution shape around $\sigma_t \approx 0$ will arise when using finely gridded probability tables generated by realistic resonance models rather than SLBW. Because the major behavior in the large σ_t appears to be unchanged, the NIG distribution approach can likely still be used—it just may need to be a mixture distribution with something describing the small σ_t regime. Whether this has any practical impact in downstream nuclear computations is an open question, and it appears to not influence criticality benchmarks in any meaningful way. We therefore pose the problem of contriving a neutron transport problem sensitive to small σ_t behavior–perhaps deep penetration shielding through a material described by URR cross-sections.

The optimization problem Eq. 5.54 was solved with a black-box method from `scipy`, but its structure would allow specialized optimization approaches. One that takes advantage of

the barycentric rational form would be a coordinate-descent type approach. The f_j values can be found as a solution to a linear least squares problem; the w_j can be found via an linearization then SVD approach similar to that employed as an intermediate step in the AAA algorithm, and the z_j could be optimized with some other specialized method, although the choice of that is not clear to us at present. One thing is for sure: two of the three subvectors can be optimized in closed-form with an approach like this. The optimization solver would likely converge faster and perhaps yield an even better fit on convergence.

5.5 Criticality Benchmark Results

To gauge the accuracy of our new method relative to the conventional approach we solved a few standard criticality benchmarks. Many criticality benchmarks do not exhibit a difference a few pcm due to URR effects, so we turned Mosteller [176] to find a few problems to evaluate our new mtehod's accuracy. To do so, we run each case with the standard approach to URR, an approach only using averaged values from the pointwise grid, and lastly our new method normalized and multiplying the pointwise grid. Our new method exhibits clear discrepancies when not normalized and multiplying the pointwise grid as the current generation of ENDF data does not intend for URR parameters to be used in this manner. Therefore this mode of operation was not investigated here and only stands as a potentially high-performance calculation mode to be explored for future use.

All runs used ENDF/B-VIII.0 data. With our new technique, the distribution fits and barycentric fits were generated using ENDF/B-VIII.0 unresolved resonance parameters but were not processed via the usual NJOY route. Instead our aforementioned GPU-accelerated ladder sampling code employing the same algorithm as NJOY's PURR was used. To normalize the distributions and the conditional expectation curves, we precompute the means of each parameter ahead of time at each energy and temperature point, then use those to normalize any values pulled from those grids after multiplying by the pointwise cross-section. This is exactly the same technique prescribed by modern nuclear data evaluations to use with tabular probability distributions rather than our fitted ones.

We referred to [176] to find a few standard criticality benchmark problems exhibiting an appreciable difference from unresolved resonance effects. Thermal spectrum problems tend to be negligibly affected by URR effects, but some thermal benchmarks exhibit discrepancies of a few pcm. The Big Ten benchmark and Zebra-8H are the two commonly solved problems particularly influenced by URR effects. According to Mosteller's work the most ZPR (Zero Power Reactor) configurations also exhibit sensitivity to URR effects so we provide solutions to the ZPR-3/53 and ZPR-6/10 cases. Lastly we also solve two thermal spectrum problems:

PNL-10 and SHEBA-2 which are respectively a tank of water-reflected aqueously dissolved plutonium nitrate and uranium dioxide dissolved in hydrofluoric acid. These are expected to present only subtle differences, but ensure that the new method does not interfere with results where probability tables otherwise do not make a difference.

Table 5.3 shows the associated eigenvalues and uncertainties for each benchmark problem. The k eigenvalue computed without probability tables, with probability tables, and with AURA are all presented. In all cases 20 inactive cycles with a total of 3000 batches were employed. In all cases but ZEBRA-8H, 100,000 particles per generation were simulated. In the case of ZEBRA-8H, only 10,000 neutrons per cycle were employed because this presented an acceptable level of uncertainty relative to the impact that URR modeling has on this problem. The original input files were designed for use with ENDF/B-VII.1 which includes natural carbon as a separate nuclide, which ENDF/B-VIII.0 employed here does not include. Because of this, we replaced all natural carbon with ^{12}C, which may cause slight disagreement of these results against others including ^{13}C. The code to reproduce these results can be found at https://github.com/gridley/openmc/tree/analytic_urr.

Table 5.3: Comparison of k eigenvalues for different URR modeling methods.

Benchmark	**No URR Modeling**	**Probability Tables**	**AURA**
Big Ten	1.00133 ± 0.00004	1.00442 ± 0.00004	1.00414 ± 0.00004
ZEBRA-8H	1.01433 ± 0.00011	1.02330 ± 0.00011	1.02329 ± 0.00011
SHEBA-II	1.01206 ± 0.00006	1.01190 ± 0.00006	1.01197 ± 0.00006
PNL-10	0.99650 ± 0.00006	0.99659 ± 0.00006	0.99654 ± 0.00006
ZPR-3/53	0.97933 ± 0.00005	0.97953 ± 0.00005	0.97945 ± 0.00005
ZPR-6/10	1.00393 ± 0.00005	1.00377 ± 0.00005	1.00377 ± 0.00005

Table 5.4: Differences in k eigenvalues between modeling methods.

Benchmark	**AURA - Prob. Tables**	**Prob. Tables - No URR**
Big Ten	-0.00028	$+0.00309$
ZEBRA-8H	-0.00001	$+0.00897$
SHEBA-II	$+0.00007$	-0.00016
PNL-10	-0.00005	$+0.00009$
ZPR-3/53	-0.00008	$+0.00020$
ZPR-6/10	0.00000	-0.00016

5.6 GPU Performance

5.6.1 Implementation

We implemented the NIG distribution sampling and barycentric rational polynomial evaluation in our CUDA-based version of OpenMC described in Chapter 3. There were no novel changes required to evaluate the barycentric rational polynomials in a GPU-friendly way. On the other hand, to implement Algorithm 13 for NIG distribution sampling to run efficiently on GPU, we sought to obviate any accesses to global memory. Therefore, techniques like ziggurat sampling are specifically avoided. Noticing that two samples from a standard normal random distribution are required in Algorithm 13, the Box-Muller transform stands out as the ideal technique under these constraints. The CUDA code which samples from the NIG distribution that results is:

```
__device__ double sample_nig(double alpha, double beta, double
    mu, double delta2, uint64_t* seed) {

  double u1 = prn(seed);
  double u2 = prn(seed);
  double R = std::sqrt(-2.0 * std::log(u1));
  double phi = 2.0 * M_PI * u2;

  // Two independent normal variates are obtained:
  double nv1 = R * std::cos(phi);
  double nv2 = R * std::sin(phi);

  // Sample the inverse Gaussian distribution, z
  double mu_ig = std::sqrt(delta2 / (alpha*alpha-beta*beta));
  double w = mu_ig * nv1 * nv1;
  double c = 0.5 * mu_ig / delta2;
  double z = mu_ig + c * (w - std::sqrt(w*(4*delta2+w)));
  if (prn(seed) >= mu_ig / (mu_ig + z)) {
    z = mu_ig * mu_ig / z;
  }

  // Sample the NIG distribution, which is a mixture over IGs
  return std::sqrt(z) * nv2 + beta * z + mu;
```

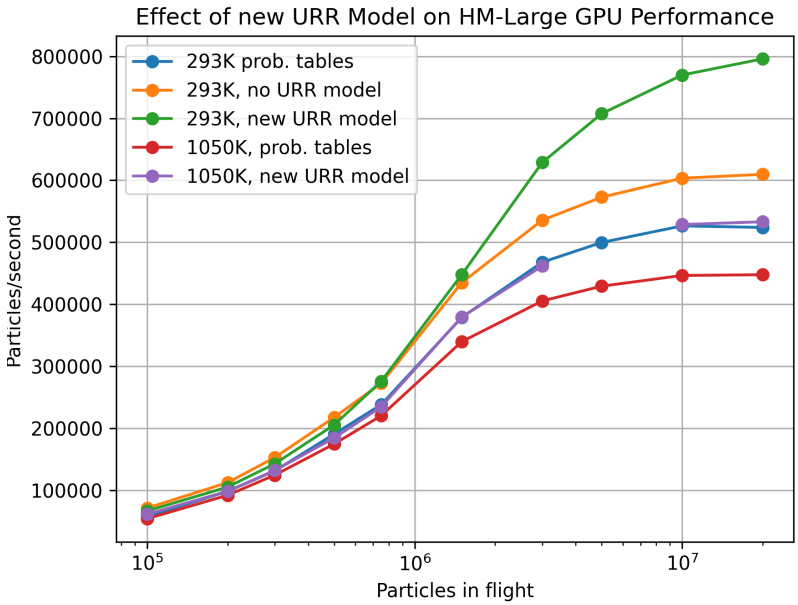


Figure 5.15: Performance comparison of two URR modeling approaches on the H100 GPU with Tramm's Hoogenboom-Martin large benchmark at two temperatures.

```
}
```

5.6.2 Hoogenboom-Martin Model

We simulated the Hoogenboom-Martin large benchmark proposed by Tramm [116] on an NVIDIA H100 GPU to test the new method. Figure **??** compares the particle processing rate of the conventional probability table and our newly proposed method. Because temperature interpolation introduces additional overhead as pointed out in Chapter 3 (in contrast to CPU-based Monte Carlo with stochastic temperature interpolation), we also present results with the reactor fuel temperature set to 1050K. The new URR modeling technique in this case not normalized against values on a pointwise grid so some error is to be expected. The model using standard probability tables calculated $k = 1.0007 \pm 6$pcm, while the new URR model calculated $k = 1.0057 \pm 6$pcm, both obtained on the room temperature model. Future iterations of this modeling technique will do away with the ENDF assumptions that URR probability distributions should be normalized against values on a pointwise grid by directly normalizing the distributions' parameters. The per-event performance breakdown is not presented for each case because the vast majority of the difference is in a single event–cross-section lookup.

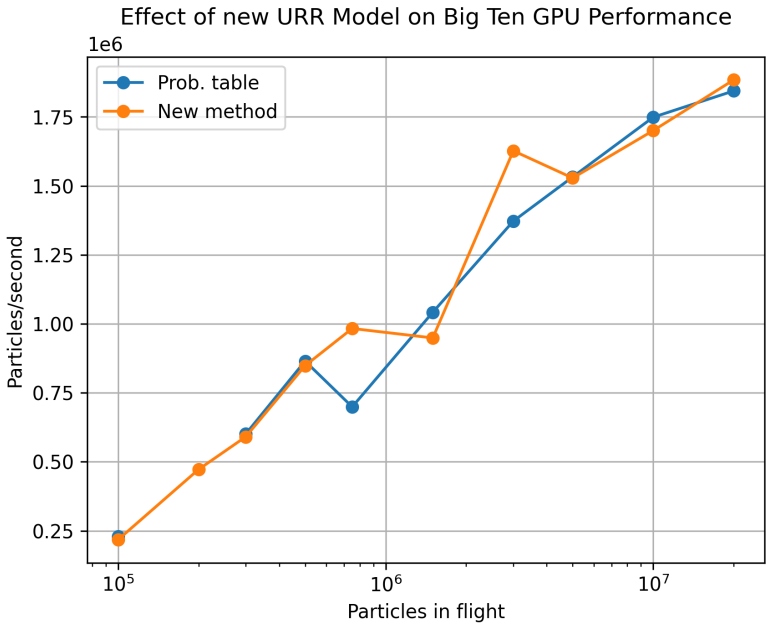


Figure 5.16: Performance comparison of two URR modeling approaches on the H100 GPU applied to the Big Ten criticality benchmark.

5.6.3 Big Ten Model

Figure 5.16 shows the calculation rate performance of the new method on the Big Ten criticality benchmark as a function of the number of particles in flight. The calculation runs exhibited some noise in the tracking rate; within this noise no substantial performance difference can be discerned. This can be attributed to the small number of nuclides in the problem. In this case, the scattered accesses to particle data are overpowering the coalesced accesses to cross-sections. Moreover, the GPU was never quite saturated in this run. Figure 5.17 shows the relative time taken by each event when using the new method, which looks similar to when the old method is used. The surface crossing event takes a relatively enormous amount of time at lower particle counts because the entire GPU stalls when appending to the neighbor lists arrays. This gets amortized in later generations, but is not properly captured by this performance testing.

5.7 Discussion

The results presented in Tables 5.3 and 5.4 suggest that the newly presented method, named AURA, successfully models the unresolved resonance effect. Some discrepancies can be observed in the cases with uranium present, which can be explained by the mismatch between

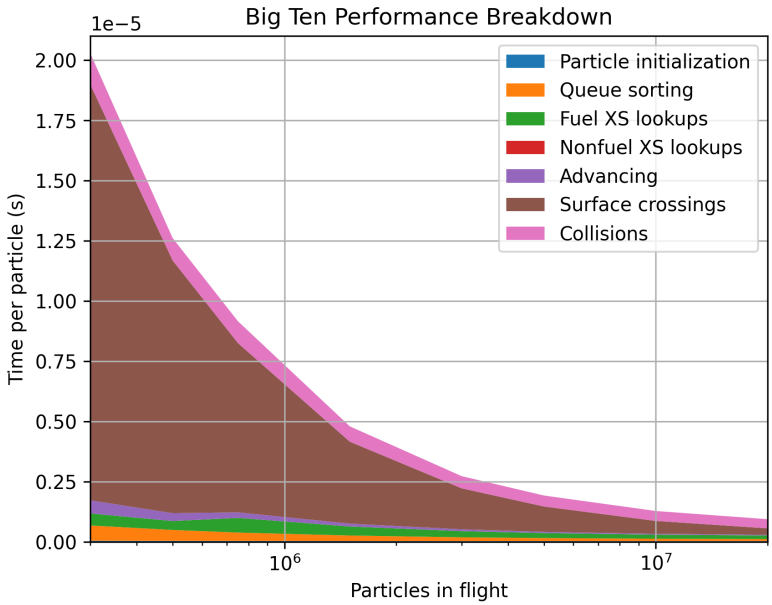


Figure 5.17: Performance breakdown of the new URR model on the H100 GPU applied to the Big Ten criticality benchmark.

NJOY and `urrtools.py` visible in Figure 5.11. For ^{235}U in particular, the agreement between our code and NJOY is poor, while simultaneously the other results match very well. This is driven home by the perfect match on the reactivity value of the ZPR-6/10 problem which was fueled entirely by plutonium. The impact of this disagreement is most pronounced in the Big Ten problem, dominated by uranium.

Nonetheless, the mismatches are representative of problems with our underlying resonance sampling code and not the AURA method itself. Its utility in GPU computing has been conclusively proven on the Hoogenboom-Martin large benchmark as shown by Figure 5.15. The performance gains were not visible on the Big Ten benchmark when executed on GPU–this comes as a result of having two orders of magnitude less nuclides to do cross-section lookup operations on. GPU Monte Carlo neutron transport is most efficiently accelerated when the bulk of the computational work is confined to particularly expensive cross-section lookup calculations.

In an attempt to eliminate the discrepancy encountered on the Big Ten benchmark here, we wrote a patched version of NJOY that samples unbiased χ^2 random variables. While a discernible effect on the shape of the probability tables themselves was observed, the reactivity differences related to this change appear to be less than a few pcm. As [172] points out, NJOY can be a difficult code to get agreement with due to its numerous approximations and opaque, un-documented Fortran programming.

The visible differences of the probability tables when sampling from unbiased χ^2 distributions pertain to the inclusion of extreme values. Because the χ^2 distributions for the partial resonance widths come from a finite set of twenty values in NJOY, large values that push the ratio Γ_n/Γ_t to its limit are never encountered. Extreme values in the cross-section occur in NJOY and `urrtools.py` when multiple nearby resonances with large widths are stacked together. This situation that NJOY misses may be unphysical, though, because of the level repulsion effect [178] which tends to make many resonances being nearby less likely relative to the NJOY asymptotic approximation where each spacing is governed by an independent distribution.

One of the main reasons for our development of a separate resonance sampling code to fit our new URR model to was our perception that NJOY failed to sample values in the tails of the distribution sufficiently: crucial for properly fitting the α and β parameters. Instead, we found that these tail values were simply missing due to the failure of PURR to sample large values from χ^2 distributions. In future work, the best approach in making comparisons to criticality benchmarks and evaluating the merit of the AURA method will be to fit directly to NJOY results. Nuclear data is tuned to match experiment, and NJOY has stood as a keystone of that process for decades. We can practically acknowledge that nuclear data may be tuned to match experiments that are analyzed using potentially flawed tools.

Although our code disagrees by -28 pcm on the Big Ten benchmark relative to probability tables, the true k value for the benchmark is 1.00490 ± 0.00080. Given the uncertainty on the benchmark, it may be the case that our code used to generate resonance samples avoids now-unnecessary numerical approximations (such as to the $w(z)$ function) and therefore is in fact closer to physical reality. After all, if the k measurement were a not-unlikely $+1\sigma$ sample over the real value of k, the actual k value would be 1.00410. This is 4 pcm below new method's result and 32 pcm below the prediction made using standard probability tables.

A possible source of systematic discrepancy in all current URR modeling pertains to the incorporation of cross-sections not modeled in sampling resonances. At present, the suggested appraoch is to generate probability tables in the form of multiplying factors on the pointwise cross-section. These factors multiply the capture, fission, and elastic scattering cross-sections. When the capture multiplying factors are considered in NJOY's PURR module or our own code, only the (n, γ) is included. In actual calculations, the "capture" contribution to the cross-section is considered to be the absorption component minus the fisison component. Therefore, by multiplying this quantity, the charged particle cross-sections are inadvertently also being multiplied by variations from resonant structure. We found that this can create differences around 10 pcm in the Big Ten benchmark, but more in-depth research on this topic is required.

The results presented by Figure 5.16 show little performance benefit from the new approach on GPUs. In fact, attaining a 1.5 million particle/second tracking rate on this problem is not particularly impressive. While GPUs in event mode can indeed deliver potent speedups on problems dominated by cross-section lookup, problems with a small number of nuclides are dominated by the totally uncoalesced, scattered memory accesses into the particle array, as Figure 5.17 shows where the surface crossing kernel dominates the computation time. To give a feel for how unimpressive that is on the H100 GPU, our Apple M3 Max attains a tracking rate around 800,000 particles per second. In cases like this, a history-based kernel would likely outperform on GPU.

Temperature correlations effects, discussed in depth in Walsh's thesis [181], have not been explicitly discussed so far. Suppose that a sample of the cross-section has been taken at some fixed energy and temperature. If the temperature is adjusted by an infinitesimal amount, we know from the continuity of Doppler broadening that the cross-section should also change by an infinitesimal amount. Consequently, this rules the physicality of resampling the cross-section at each temperature invalid. At present, our approach is to use the same random number stream to compute the cross-section at each temperature. By doing so, the continuity of cross-section changes with temperature is captured, and the correct marginal distribution of σ_t is preserved at each temperature. To our knowledge, no mainstream studies have measured the in-depth impact of URR model temperature dependence, so we have not discussed this aspect in this work.

The NIG distribution given in Eq. 5.49 has a closed-form moment generating function [196]. Recall that the moment generating function of a random variable X is $\mathbb{E}[e^{tX}]$. Consequently, the attenuation of a large population of neutron through a sample of thickness t can be calculated with the moment-generating function of the NIG distribution:

$$\mathbb{E}[e^{-\sigma_t t} = e^{-\mu t + \delta\left(\gamma - \sqrt{\mathfrak{a}^2 - (\mathfrak{b}-t)^2}\right)} \quad . \tag{5.55}$$

The average cross-section in the above case is $\mathbb{E}[\sigma_t] = \mu + \delta\beta/\gamma$. Consequently, by measuring the attenuation through a large number of samples of varying thicknesses and fitting the four-parameter curve from Eq. 5.55, the parameters of the URR model can be extracted from differential experiments without any recourse to intermediate probability tables or resonance width information. We propose that future work expanding on our results here interpret differential experiments through this lens.

Chapter 6

Conclusion

In this thesis, we introduced the challenges associated with writing a performant, maintainable Monte Carlo neutron transport simulation code. The first chapter introduced the key principles. The second chapter provided a contribution in the direction of algorithm analysis for Monte Carlo methods on GPU. The third chapter introduced programming techniques, optimizations, and novel developments associated with our CUDA-based version of OpenMC. Next in chapter four we introduced a new algorithm for handling the resonance upscatter effect, vital to calculating correct temperature feedbacks in fission reactors, without the addition of any extra data–this is a first in several ways. Lastly, we identified the performance impact of unresolved resonance region modeling in Monte Carlo reactor neutronics simulations, critically analyzed the requirements of unresolved resonance region modeling from first principles, and presented the first practically usable analytic approach to unresolved resonance region modeling and demonstrated its efficacy in GPU computing.

One drawback not examined in many papers so far in the GPU neutronics domain is that reactor problems have received the vast majority of the attention due to the nature of these problems focusing most of the computational effort on the cross-section lookup operation. A variety of performant GPU Monte Carlo applications have been developed for simulating reactors with hundreds of nuclides in the fuel. On the other hand, a relatively small amount of attention has been directed towards Monte Carlo neutronics needs for nuclear security, shielding, and other applications. In those cases, again, paradigm shifts in the Monte Carlo algorithm may be desirable for optimal operation. At the moment, we are unaware of any performant GPU-based continuous energy, full-physics Monte Carlo radiation transport solver that has focused on shielding with variance reduction or coupled neutron-gamma problems. Plenty of future work could be dedicated to addressing the computational intricacies of these other problems.

To obtain performant code on non-reactor problems exhibiting a relatively smaller num-

ber of nuclides, substantial performance improvements could be made by finding methods that do not require pulling and pushing a large amount of particle data to and from global memory. It is this un-coalesced pushing and pulling operation that explains the hybrid history-event method presented in PRAGMA's performance optimization paper [29]. In that method, event queues are used, but a few events in a potentially divergent execution pattern are still carried out after each event. Despite the thread divergence, performance gains are still realized due to the high expense of pushing particle data back into global memory and instead keeping it in registers. If a method to avoid the need for queue-based event methods could be devised (for example by dramatically reducing the amount of data needed to describe a particle), again, substantial performance benefits could likely be realized. A particularly promising direction would be to implement a scheme similar to dynamic re-coalescence available in recent ray-tracing programs for Nvidia GPUs in the method called shader execution re-ordering [208].

If we were to embark again on writing a GPU-based Monte Carlo code scratch, we would not write it with any intention of sharing large amounts of code with the corresponding CPU codebase. Considering that both the optimal data structure layouts and algorithms differ so much, this practically forces the programmer to write separate codebases. For example, we ended up completely rewriting all the cross-section lookup code in a separate CUDA kernel. On the other hand, all the physics code was shared between the CPU and GPU versions–this may not be the best approach to performance.

Chapter 3 made clear the high code complexity of collision physics modeling. Ideally, a single purely numerical collision formula could be devised. We are unaware of any research that has sought to unify for example the double-differential cross-section of elastic scattering and the Kalbach-Mann systematics under the same formula which can nonetheless approximate either case. One admittedly computationally expensive approach to universal approximation like this would be normalizing flows as explored and proven to work in [209]. However, the computational performance seen there was lackluster. A numerical method inspired by the physical constraints might be able to address this. The fruit of this research would be drastically simplified, non-divergent collision processing code that might allow the benefits of our novel resonance upscatter method to be realized. Otherwise, the extreme branching in the collision kernel tends to damp out any performance gains made to any single branch.

Plenty more advances in data format can be made to improve performance of GPU Monte Carlo applications. In Chapter 3, we showed the advantage of the windowed multipole formalism over the pointwise method when interpolating between two temperatures. The unionized grid method [69] has delivered impressive performance gains in CPU applications.

Such a technique to optimize the accesses to windowed multipole data in principle could be devised, and may present a tangible performance benefit for GPU computing at the cost of a little bit of extra memory.

Bibliography

[1] R. Eckhardt, "Stan Ulam, John von Neumann, and the Monte Carlo Method," *Los Alamos Science*, vol. Special Issue, 1987. [Online]. Available: https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-88-9068 (visited on 01/14/2021).

[2] R. J. J. Stamm'Ler and M. J. Abbate, *Methods of Steady-State Reactor Physics in Nuclear Design*. London ; New York: Academic Pr, Jul. 1983, 506 pp., ISBN: 978-0-12-663320-7.

[3] A. Hébert, *Applied reactor physics*, Second Edition, P. internationales Polytechnique, Ed. Presses internationales Polytechnique, Jan. 1, 2016, 406 pp., ISBN: 978-2-553-01698-1.

[4] B. L. Sjenitzer, "The Dynamic Monte Carlo Method for Transient Analysis of Nuclear Reactors," 2013. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3A6a4bfa4c-2d1f-4648-9698-6eb7ec7e2d11 (visited on 01/13/2021).

[5] A. F. Henry, "The Application of Reactor Kinetics to the Analysis of Experiments," *Nuclear Science and Engineering*, vol. 3, no. 1, pp. 52–70, Jan. 1, 1958, ISSN: 0029-5639. DOI: 10.13182/NSE58-1. [Online]. Available: https://doi.org/10.13182/NSE58-1 (visited on 01/13/2021).

[6] Q. Shen, Y. Wang, D. Jabaay, B. Kochunas, and T. Downar, "Transient analysis of C5G7-TD benchmark with MPACT," *Annals of Nuclear Energy*, vol. 125, pp. 107–120, Mar. 1, 2019, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2018.10.049. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306454918305772 (visited on 01/13/2021).

[7] D. Ferraro, M. García, V. Valtavirta, U. Imke, R. Tuominen, J. Leppänen, and V. Sanchez-Espinoza, "Serpent/SUBCHANFLOW pin-by-pin coupled transient calculations for the SPERT-IIIE hot full power tests," *Annals of Nuclear Energy*, vol. 142, p. 107 387, Jul. 1, 2020, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2020.107387. [On-

line]. Available: http://www.sciencedirect.com/science/article/pii/S0306454920300852 (visited on 01/13/2021).

[8] S. C. Shaner, "Development of high fidelity methods for 3D Monte Carlo transient analysis of nuclear reactors," Thesis, Massachusetts Institute of Technology, 2018. [Online]. Available: https://dspace.mit.edu/handle/1721.1/119034 (visited on 01/13/2021).

[9] ORNL. "Summit," Oak Ridge Leadership Computing Facility. (2018), [Online]. Available: https://www.olcf.ornl.gov/summit/ (visited on 01/13/2021).

[10] F. B. Brown, "Recent advances and future prospects for Monte Carlo," Los Alamos National Lab. (LANL), Los Alamos, NM (United States), LA-UR-10-05634; LA-UR-10-5634, Jan. 1, 2010. [Online]. Available: https://www.osti.gov/biblio/1028796 (visited on 01/13/2021).

[11] J. Nickolls and W. J. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar. 2010, ISSN: 1937-4143. DOI: 10.1109/MM.2010.41.

[12] T. N. Theis and H. P. Wong, "The End of Moore's Law: A New Beginning for Information Technology," *Computing in Science Engineering*, vol. 19, no. 2, pp. 41–50, Mar. 2017, ISSN: 1558-366X. DOI: 10.1109/MCSE.2017.29.

[13] Top500. "November 2020 | TOP500." (Nov. 2020), [Online]. Available: https://www.top500.org/lists/top500/2020/11/ (visited on 01/14/2021).

[14] J. Spanier and E. M. Gelbard, *Monte Carlo Principles and Neutron Transport Problems*. Mineola, N.Y: Dover Publications, Feb. 4, 2008, 256 pp., ISBN: 978-0-486-46293-6.

[15] L. C. Evans, *Partial Differential Equations: Second Edition*, 2nd edition. Providence, R.I: American Mathematical Society, Mar. 3, 2010, 749 pp., ISBN: 978-0-8218-4974-3.

[16] K. M. Case and P. F. Zweifel, *Linear Transport Theory*. Addison-Wesley Publishing Company, 1967, 360 pp., Google-Books-ID: uQtRAAAAMAAJ.

[17] R. Haberman, *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems*, 5th edition. Place of publication not identified: Pearson, Aug. 24, 2012, ISBN: 978-0-321-79706-3.

[18] E. Troubetzkoy, H. Steinberg, and M. Kalos, "Monte Carlo Radiation Penetration Calculations on a Parallel Computer," *Trans. Am. Nucl. Soc*, vol. 17, no. 260,

[19] W. Martin and F. Brown, "Status of Vectorized Monte Carlo for Particle Transport Analysis," *International Journal of High Performance Computing Applications*, vol. 1, Jun. 1, 1987. DOI: 10.1177/109434208700100203.

[20] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972, ISSN: 1557-9956. DOI: 10.1109/TC.1972.5009071.

[21] C. Calvin and D. Nowak, "High Performance Computing in Nuclear Engineering," in *Handbook of Nuclear Engineering*, D. G. Cacuci, Ed., Boston, MA: Springer US, 2010, pp. 1449–1517, ISBN: 978-0-387-98149-9. DOI: 10.1007/978-0-387-98149-9_12. [Online]. Available: https://doi.org/10.1007/978-0-387-98149-9_12 (visited on 01/14/2021).

[22] N. Matloff, *Programming on Parallel Machines*.

[23] Nvidia. "CUDA Toolkit Documentation v10.2.89." (), [Online]. Available: https://docs.nvidia.com/cuda/ (visited on 02/13/2020).

[24] "Inside volta: The world's most advanced data center GPU," NVIDIA Technical Blog. (May 10, 2017), [Online]. Available: https://developer.nvidia.com/blog/inside-volta/ (visited on 07/09/2024).

[25] V. Volkov, "Better Performance at Lower Occupancy," in *GPU Technology Conference*, Silicon Valley, 2010.

[26] F. B. Brown and W. R. Martin, "Monte Carlo methods for radiation transport analysis on vector computers," *Progress in Nuclear Energy*, vol. 14, no. 3, pp. 269–299, Jan. 1, 1984, ISSN: 0149-1970. DOI: 10.1016/0149-1970(84)90024-6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0149197084900246 (visited on 01/14/2021).

[27] P. K. Romano and A. R. Siegel, "Limits on the efficiency of event-based algorithms for Monte Carlo neutron transport," *Nuclear Engineering and Technology*, Special Issue on International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering 2017 (M&C 2017), vol. 49, no. 6, pp. 1165–1171, Sep. 1, 2017, ISSN: 1738-5733. DOI: 10.1016/j.net.2017.06.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1738573317302966 (visited on 05/21/2020).

[28] S. P. Hamilton and T. M. Evans, "Continuous-energy Monte Carlo neutron transport on GPUs in the Shift code," *Annals of Nuclear Energy*, vol. 128, pp. 236–247, Jun. 2019, ISSN: 03064549. DOI: 10.1016/j.anucene.2019.01.012.

[29] N. Choi, K. M. Kim, and H. G. Joo, "Optimization of neutron tracking algorithms for GPU-based continuous energy Monte Carlo calculation," *Annals of Nuclear Energy*, vol. 162, p. 108 508, Nov. 2021, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2021.108508.

[30] C. J. Werner, J. S. Bull, C. J. Solomon, *et al.*, "MCNP Version 6.2 Release Notes," Los Alamos National Lab. (LANL), Los Alamos, NM (United States), LA-UR-18-20808, Feb. 5, 2018. DOI: 10.2172/1419730. [Online]. Available: https://www.osti.gov/biblio/1419730-mcnp-version-release-notes (visited on 01/20/2021).

[31] N. Choi, K. Kim, and H. Joo, "Initial Development of PRAGMA - A GPU-Based Continuous Energy Monte Carlo Code for Practical Applications," Oct. 25, 2019.

[32] D. Göddeke, *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*. Berlin: Logos Verlag, Feb. 2011, ISBN: 978-3-8325-2768-6.

[33] G. Giudicelli, A. Lindsay, L. Harbour, *et al.*, "3.0 - MOOSE: Enabling massively parallel multiphysics simulations," *SoftwareX*, vol. 26, p. 101 690, May 1, 2024, ISSN: 2352-7110. DOI: 10.1016/j.softx.2024.101690. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S235271102400061X (visited on 07/05/2024).

[34] L. Murray, "GPU Acceleration of Runge-Kutta Integrators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 1, pp. 94–101, Jan. 2012, ISSN: 1558-2183. DOI: 10.1109/TPDS.2011.61.

[35] Yuguang Li, Xu Dai, Feng Zhao, and Hong Shang, "GPU-based acceleration for Monte Carlo ray-tracing of complex 3D scene," *2012 IEEE International Geoscience and Remote Sensing Symposium, Geoscience and Remote Sensing Symposium (IGARSS), 2012 IEEE International*, pp. 4240–4243, Jul. 2012, ISSN: 978-1-4673-1158-8. DOI: 10.1109/IGARSS.2012.6351732.

[36] B. Plazolles, D. El Baz, M. Spel, V. Rivola, and P. Gegout, "SIMD Monte-Carlo Numerical Simulations Accelerated on GPU and Xeon Phi," *International Journal of Parallel Programming*, vol. 46, no. 3, pp. 584–606, Jun. 2018, ISSN: 08857458. DOI: 10.1007/s10766-017-0509-y. (visited on 12/13/2019).

[37] W. Boyd, S. Shaner, L. Li, B. Forget, and K. Smith, "The OpenMOC method of characteristics neutral particle transport code," *Annals of Nuclear Energy*, vol. 68, pp. 43–52, Jun. 1, 2014, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2013.12.012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306454913006634 (visited on 07/05/2024).

[38] H. G. Lee, S. U. Jae, N. Choi, and J. Kang, "Progress of GPU Acceleration Module in nTRACER for Cycle Depletion," in *Transactions of the Korean Nuclear Society Virtual Spring Meeting*, Jul. 2020.

[39] N. Choi, J. Kang, H. G. Lee, and H. G. Joo, "Practical acceleration of direct whole-core calculation employing graphics processing units," *Progress in Nuclear Energy*, vol. 133, p. 103 631, Mar. 1, 2021, ISSN: 0149-1970. DOI: 10.1016/j.pnucene.2021.103631. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0149197021000032 (visited on 07/05/2024).

[40] S. Jeon, H. Hong, N. Choi, and H. G. Joo, "Methods and performance of a GPU-based pinwise two-step nodal code VANGARD," *Progress in Nuclear Energy*, vol. 156, p. 104 528, Feb. 2023, ISSN: 0149-1970. DOI: 10.1016/j.pnucene.2022.104528. (visited on 09/05/2023).

[41] K. M. Kim, L. Han Gyu, and J. Han Gyu, "GPU-based Method of Characteristics with CMFD Acceleration in Unstructured Mesh Geometry," in *Transactions of the Korean Nuclear Society*, Jeju, South Korea, May 2023.

[42] T. Stitt, K. Belcher, A. Campos, T. Kolev, P. Mocz, R. N. Rieben, A. Skinner, V. Tomov, A. Vargas, and K. Weiss, "Performance portable graphics processing unit acceleration of a high-order finite element multiphysics application," *Journal of Fluids Engineering*, vol. 146, no. 41102, Feb. 9, 2024, ISSN: 0098-2202. DOI: 10.1115/1.4064493. [Online]. Available: https://doi.org/10.1115/1.4064493 (visited on 07/05/2024).

[43] J. Andrej, N. Atallah, J.-P. Bäcker, *et al.*, *High-performance finite elements with MFEM*, Feb. 24, 2024. DOI: 10.48550/arXiv.2402.15940. arXiv: 2402.15940[cs,math]. [Online]. Available: http://arxiv.org/abs/2402.15940 (visited on 07/05/2024).

[44] J. R. Tramm and A. R. Siegel, "Immortal rays: Rethinking random ray neutron transport on GPU architectures," *Parallel Computing*, vol. 108, p. 102 832, Dec. 1, 2021, ISSN: 0167-8191. DOI: 10.1016/j.parco.2021.102832. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819121000806 (visited on 07/05/2024).

[45] J. Willert, C. Kelley, D. Knoll, Han Dong, M. Ravishankar, P. Sathre, M. Sullivan, and W. Taitano, "Hybrid Deterministic/Monte Carlo Neutronics Using GPU Accelerators," *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering & Science, Distributed Computing and Applications to Business, Engineering & Science (DCABES), 2012 11th International Symposium on, International Symposium on Distributed Computing and Applications to Busi-*

ness, Engineering and Science, pp. 43–47, Oct. 2012, ISSN: 978-1-4673-2630-8. DOI: 10.1109/DCABES.2012.37.

[46] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, APPLICATION ACCELERATORS IN HPC, vol. 38, no. 8, pp. 391–407, Aug. 2012, ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.10.002. (visited on 10/09/2020).

[47] M. J. Harvey and G. De Fabritiis, "Swan: A tool for porting CUDA programs to OpenCL," *Computer Physics Communications*, vol. 182, pp. 1093–1099, Apr. 2011, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2010.12.052. (visited on 02/04/2020).

[48] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, vol. 74, no. 12, pp. 3202–3216, Dec. 2014, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003. (visited on 07/12/2021).

[49] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. W. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019.

[50] T. W. D. Medina, A. Modave, and V. Tech, "The OCCA abstract threading model," p. 27,

[51] D. S. Medina, A. St-Cyr, and T. Warburton, "OCCA: A unified approach to multi-threading languages," *arXiv:1403.0968 [cs]*, Mar. 2014. arXiv: 1403.0968 [cs]. (visited on 07/12/2021).

[52] R. M. Bergmann and J. L. Vujić, "Algorithmic choices in WARP – A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs," *Annals of Nuclear Energy*, vol. 77, pp. 176–193, Mar. 1, 2015, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2014.10.039. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306454914005787 (visited on 01/13/2021).

[53] G. Wilson, D. A. Aruliah, C. T. Brown, *et al.*, "Best Practices for Scientific Computing," *PLOS Biology*, vol. 12, no. 1, e1001745, Jan. 7, 2014, ISSN: 1545-7885. DOI: 10.1371/journal.pbio.1001745. [Online]. Available: https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745 (visited on 07/10/2021).

[54] J. Tramm, P. Romano, J. Doerfert, A. Lund, P. Shriwise, A. Siegel, G. Ridley, and A. Pastrello, "Toward Portable GPU Acceleration of the OpenMC Monte Carlo Particle Transport Code," May 2022. DOI: 10.13182/PHYSOR22-37847.

[55] J. Tickner, "Monte Carlo simulation of X-ray and gamma-ray photon transport on a graphics-processing unit," *Computer Physics Communications*, vol. 181, no. 11, pp. 1821–1832, Nov. 1, 2010, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2010.07.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465510002249 (visited on 01/12/2021).

[56] J. E. Sweezy, "A Monte Carlo Volumetric-Ray-Casting Estimator for Global Fluence Tallies on GPUs," *Journal of Computational Physics*, vol. 372, pp. 426–445, Nov. 2018, ISSN: 00219991. DOI: 10.1016/j.jcp.2018.06.032. arXiv: 1706.07476. (visited on 02/04/2020).

[57] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. [Online]. Available: https://www.mcs.anl.gov/papers/P5064-0114.pdf.

[58] F. Brown, "Vectorized Monte Carlo.," Ph.D. dissertation, University of Michigan, 1981. (visited on 01/13/2021).

[59] Y. Morimoto, H. Maruyama, K. Ishii, and M. Aoyama, "Neutronic Analysis Code for Fuel Assembly Using a Vectorized Monte Carlo Method," *Nuclear Science and Engineering*, vol. 103, no. 4, pp. 351–358, Dec. 1989, ISSN: 0029-5639. DOI: 10.13182/NSE89-A23688. (visited on 12/28/2020).

[60] A. G. Nelson, "MONTE CARLO METHODS FOR NEUTRON TRANSPORT ON GRAPHICS PROCESSING UNITS USING CUDA," Dec. 21, 2009. [Online]. Available: https://etda.libraries.psu.edu/catalog/10202 (visited on 01/13/2021).

[61] T. Scudiero, "Monte Carlo Neutron Transport: Simulat-ing Nuclear Reactions One Neutron at a Time," in *Proc. GPU Technology Conf. (GTC)*, San Jose, CA, Mar. 2013.

[62] R. M. Bergmann, "Performance and accuracy of criticality calculations performed using WARP – A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs," 2017, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2017.01.027.

[63] K. Rowland, R. Bergmann, R. Slaybaugh, and J. Vujic, "Delta-tracking in the GPU-accelerated WARP Monte Carlo Neutron Transport Code," in *International Conference on Mathematics&Computational Methods Applied to Nuclear Science&Engineering*, Jeju, South Korea, Apr. 2017.

[64] T. Liu, X. Du, W. Ji, G. Xu, and F. Brown, "A comparative study of history-based versus vectorized Monte Carlo methods in the GPU/CUDA environment for a simple neutron eigenvalue problem," Jun. 6, 2014, p. 04 206, ISBN: 978-2-7598-1269-1. DOI: 10.1051/snamc/201404206.

[65] T. Liu, N. Wolfe, C. D. Carothers, W. Ji, and X. G. Xu, "Optimizing the Monte Carlo Neutron Cross-Section Construction Code XSBench for MIC and GPU Platforms," *Nuclear Science and Engineering*, vol. 185, no. 1, pp. 232–242, Jan. 2017, ISSN: 0029-5639. DOI: 10.13182/NSE16-33. (visited on 06/01/2021).

[66] S. P. Hamilton, S. R. Slattery, and T. M. Evans, "Multigroup Monte Carlo on GPUs: Comparison of history- and event-based algorithms," *Annals of Nuclear Energy*, vol. 113, pp. 506–518, Mar. 2018, ISSN: 03064549. DOI: 10.1016/j.anucene.2017.11.032.

[67] R. C. Bleile, P. S. Brantley, S. A. Dawson, M. J. O'Brien, and H. Childs, "Investigation of portable event-based monte carlo transport using the NVIDIA thrust library," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), LLNL-CONF-681383, Jan. 25, 2016. [Online]. Available: https://www.osti.gov/biblio/1259778 (visited on 07/07/2024).

[68] Y. Wang, "Optimization of Monte Carlo Neutron Transport Simulations with Emerging Architectures," Ph.D. dissertation, Dec. 2017.

[69] J. Leppänen, "Two practical methods for unionized energy grid construction in continuous-energy monte carlo neutron transport calculation," *Annals of Nuclear Energy*, vol. 36, no. 7, pp. 878–885, Jul. 1, 2009, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2009.03.019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306454909001108 (visited on 07/07/2024).

[70] F. Brown, "New hash-based energy lookup algorithm for monte carlo codes," in *American Nuclear Society, 2014 Winter Meeting*, LA-UR-14-24530, Anaheim, California, United States, Nov. 9–13, 2014.

[71] C. Josey, P. Ducru, B. Forget, and K. Smith, "Windowed multipole for cross section Doppler broadening," *Journal of Computational Physics*, vol. 307, pp. 715–727, Feb. 2016, ISSN: 0021-9991. DOI: 10.1016/j.jcp.2015.08.013.

[72] N. Choi and H. G. Joo, "Domain decomposition for GPU-Based continuous energy Monte Carlo power reactor calculation," *Nuclear Engineering and Technology*, vol. 52, no. 11, pp. 2667–2677, Nov. 2020, ISSN: 1738-5733. DOI: 10.1016/j.net.2020.04.024. (visited on 09/05/2023).

[73] S. P. Hamilton, T. M. Evans, K. E. Royston, and E. D. Biondo, "Domain decomposition in the GPU-accelerated Shift Monte Carlo code," *Annals of Nuclear Energy*, vol. 166, p. 108 687, Feb. 2022, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2021.108687. (visited on 12/01/2022).

[74] D. Ozog, A. D. Malony, and A. R. Siegel, "A Performance Analysis of SIMD Algorithms for Monte Carlo Simulations of Nuclear Reactor Cores," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 733–742. DOI: 10.1109/IPDPS.2015.105.

[75] K. M. Kim, J. Im, N. Choi, H. G. Lee, and H. G. Joo, "High-Performance and High-Fidelity GPU-Based Monte Carlo Solutions to the BEAVRS Benchmark," *Nuclear Science and Engineering*, vol. 197, no. 8, pp. 1823–1844, Aug. 2023, ISSN: 0029-5639. DOI: 10.1080/00295639.2022.2148812. (visited on 09/05/2023).

[76] J. Im, M. J. Jeong, N. Choi, K. M. Kim, H. K. Cho, and H. G. Joo, "Multiphysics Analysis System for Heat Pipe–Cooled Micro Reactors Employing PRAGMA-OpenFOAM-ANLHTP," *Nuclear Science and Engineering*, vol. 197, no. 8, pp. 1743–1757, Aug. 2023, ISSN: 0029-5639. DOI: 10.1080/00295639.2022.2143209. (visited on 09/05/2023).

[77] B. Molnar, G. Tolnai, and D. Legrady, "A GPU-based direct Monte Carlo simulation of time dependence in nuclear reactors," *Annals of Nuclear Energy*, vol. 132, pp. 46–63, Oct. 2019, ISSN: 03064549. DOI: 10.1016/j.anucene.2019.03.024.

[78] R. Rahaman, D. Medina, A. Lund, J. Tramm, T. Warburton, and A. Siegel, "Portability and Performance of Nuclear Reactor Simulations on Many-Core Architectures," in *Proceedings of the 3rd International Conference on Exascale Applications and Software*, ser. EASC '15, Edinburgh, UK: University of Edinburgh, Apr. 2015, pp. 42–47, ISBN: 978-0-9926615-1-9. (visited on 02/04/2020).

[79] K. Bossler, "Methods for Computing Monte Carlo Tallies on the GPU.," Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), Tech. Rep. SAND2018-3123C, Mar. 2018. (visited on 05/01/2023).

[80] F. Shriver, S. Lee, S. P. Hamilton, J. S. Vetter, and J. Watson, "Enhancing monte carlo proxy applications on GPUs," Oak Ridge National Lab. (ORNL), Oak Ridge, TN (United States), Nov. 1, 2019. [Online]. Available: https://www.osti.gov/biblio/1606716 (visited on 07/06/2024).

[81] J. Salmon and S. McIntosh-Smith, "Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC," presented at the 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Nov. 2019, pp. 19–29. DOI: 10.1109/PMBS49563.2019.00008.

[82] R. C. Bleile, P. S. Brantley, H. H. Childs, D. Richards, S. Dawson, M. S. McKinley, M. O'Brien, and H. Childs, "Thin-Threads: An Approach for History-Based Monte Carlo on GPUs," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), LLNL-TR-781441, Jul. 15, 2019. DOI: 10.2172/1542743. [Online]. Available: https://www.osti.gov/biblio/1542743-thin-threads-approach-history-based-monte-carlo-gpus (visited on 03/16/2020).

[83] M. S. McKinley, R. Bleile, P. S. Brantley, S. Dawson, M. O'Brien, M. Pozulp, and D. Richards, "Status of LLNL Monte Carlo Transport Codes on Sierra GPUs," p. 8,

[84] M. M. Pozulp, R. C. Bleile, P. S. Brantley, S. A. Dawson, M. S. McKinley, M. J. O'Brien, A. P. Robinson, and M. Yang, "Progress porting LLNL monte carlo transport codes to nvidia GPUs," Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), LLNL-PROC-848684, May 1, 2023. [Online]. Available: https://www.osti.gov/biblio/1995429 (visited on 07/07/2024).

[85] R. Friedman. "OPENMP 5.0 IS a MAJOR LEAP FORWARD," OpenMP. (Nov. 8, 2018), [Online]. Available: https://www.openmp.org/press-release/openmp-5-0-is-a-major-leap-forward/ (visited on 07/08/2024).

[86] J. E. Hoogenboom, W. R. Martin, and B. Petrovic, "The monte carlo performance benchmark test - aims, specifications and first results," in *M&C 2011: International conference on mathematics and computational methods applied to nuclear science and engineering*, Brazil, 2011.

[87] J. P. Morgan, A. Mote, S. L. Pasmann, G. Ridley, T. S. Palmer, and K. E. Niemeyer, "The monte carlo computational summit–october 25 & 26, 2023–notre dame, indiana, usa," *Journal of Computational and Theoretical Transport*, pp. 1–22, 2024.

[88] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[89] S. Chakraborty and R. D. Gupta, "Exponentiated Geometric Distribution: Another Generalization of Geometric Distribution," *Communications in Statistics - Theory and Methods*, vol. 44, no. 6, pp. 1143–1157, Mar. 19, 2015, ISSN: 0361-0926. DOI: 10.1080/03610926.2012.763090. [Online]. Available: https://doi.org/10.1080/03610926.2012.763090 (visited on 02/22/2020).

[90] L. Devroye, "Chapter 4 Nonuniform Random Variate Generation," in *Handbooks in Operations Research and Management Science*, ser. Simulation, S. G. Henderson and B. L. Nelson, Eds., vol. 13, Elsevier, Jan. 1, 2006, pp. 83–121. DOI: 10.1016/S0927-0507(06)13004-2. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0927050706130042 (visited on 02/13/2020).

[91] G. Marsaglia and W. W. Tsang, "A simple method for generating gamma variables," *ACM Transactions on Mathematical Software (TOMS)*, vol. 26, no. 3, pp. 363–372, Sep. 1, 2000, ISSN: 0098-3500. DOI: 10.1145/358407.358414. [Online]. Available: https://doi.org/10.1145/358407.358414 (visited on 02/13/2020).

[92] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith, "OpenMC: A state-of-the-art Monte Carlo code for research and development," *Annals of Nuclear Energy*, Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013, SNA + MC 2013. Pluri- and Trans-Disciplinarity, Towards New Modeling and Numerical Simulation Paradigms, vol. 82, pp. 90–97, Aug. 1, 2015, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2014.07.048. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S030645491400379X (visited on 03/10/2020).

[93] W. Rothenstein, "Neutron scattering kernels in pronounced resonances for stochastic Doppler effect calculations," *Annals of Nuclear Energy*, A Special Issue in Honour of M. M. R. Williams, vol. 23, no. 4, pp. 441–458, Mar. 1, 1996, ISSN: 0306-4549. DOI: 10.1016/0306-4549(95)00109-3. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0306454995001093 (visited on 03/10/2020).

[94] A. Terenin, S. Dong, and D. Draper, "GPU-accelerated Gibbs sampling: A case study of the Horseshoe Probit model," *Statistics and Computing*, vol. 29, no. 2, pp. 301–310, Mar. 1, 2019, ISSN: 1573-1375. DOI: 10.1007/s11222-018-9809-3. [Online]. Available: https://doi.org/10.1007/s11222-018-9809-3 (visited on 02/13/2020).

[95] A. Ahmadi-Javid and A. Moeini, "An economical acceptance–rejection algorithm for uniform random variate generation over constrained simplexes," *Statistics and Computing*, vol. 26, no. 3, pp. 703–713, May 1, 2016, ISSN: 1573-1375. DOI: 10.1007/s11222-

015-9553-x. [Online]. Available: https://doi.org/10.1007/s11222-015-9553-x (visited on 03/10/2020).

[96] G. Marsaglia and W. W. Tsang, "The Ziggurat Method for Generating Random Variables," *Journal of Statistical Software*, vol. 5, no. 1, pp. 1–7, 1 Oct. 2, 2000, ISSN: 1548-7660. DOI: 10.18637/jss.v005.i08. [Online]. Available: https://www.jstatsoft.org/index.php/jss/article/view/v005i08 (visited on 04/11/2020).

[97] S. Basu and A. DasGupta, "The Mean, Median, and Mode of Unimodal Distributions:A Characterization," *Theory of Probability & Its Applications*, vol. 41, no. 2, pp. 210–223, Jan. 1, 1997, ISSN: 0040-585X. DOI: 10.1137/S0040585X97975447. [Online]. Available: https://epubs.siam.org/doi/10.1137/S0040585X97975447 (visited on 02/22/2020).

[98] M. Abramowitz and I. A. Stegun, Eds., *Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables*, 0009-Revised edition. New York, NY: Dover Publications, Jun. 1965, ISBN: 978-0-486-61272-0.

[99] P. Romano and J. Walsh, "An improved target velocity sampling algorithm for free gas elastic scattering," *Annals of Nuclear Energy*, vol. 114, pp. 318–324, Apr. 2018.

[100] T. Kunz, A. Thomaz, and H. Christensen, "Hierarchical rejection sampling for informed kinodynamic planning in high-dimensional spaces," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 89–96. DOI: 10.1109/ICRA.2016.7487120.

[101] R. D. Peng, *6.3 Rejection Sampling | Advanced Statistical Computing*. [Online]. Available: https://github.com/rdpeng/advstatcomp (visited on 09/16/2020).

[102] "Intel® Advanced Vector Extensions 512 Overview." (), [Online]. Available: intel.com/avx512 (visited on 09/09/2020).

[103] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proc. High Performance Graphics*, Aug. 1, 2009. [Online]. Available: https://research.nvidia.com/publication/understanding-efficiency-ray-traversal-gpus (visited on 09/09/2020).

[104] A. Pfeffer, "Sampling with Memoization," in *AAAI*, 2007.

[105] G. Ridley and B. Forget, "Design and Optimization of GPU Capabilities in OpenMC," in *ANS Winter Conference*, Washington D.C., Dec. 2021.

[106] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz—Open Source Graph Drawing Tools," in *Graph Drawing*, P. Mutzel, M. Jünger, and S. Leipert, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2002, pp. 483–484, ISBN: 978-3-540-45848-7. DOI: 10.1007/3-540-45848-4_57.

[107] "CUDA Toolkit Documentation v10.2.89." (), [Online]. Available: https://docs.nvidia.com/cuda/ (visited on 02/13/2020).

[108] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st edition. Addison-Wesley Professional, Jul. 19, 2010, 320 pp., ISBN: 978-0-13-138768-3.

[109] A. Trkov and D. A. Brown, "ENDF-6 formats manual: Data formats and procedures for the evaluated nuclear data files," Brookhaven National Lab. (BNL), Upton, NY (United States), BNL-203218-2018-INRE, Jan. 31, 2018. DOI: 10.2172/1425114. [Online]. Available: https://www.osti.gov/biblio/1425114 (visited on 09/11/2023).

[110] M. Park. "Variant: Discriminated union with value semantics." Programming Language C++ Library Evolution Group, Document #: P0080. (Jul. 2015), [Online]. Available: https://open-std.org/Jtc1/sc22/wg21/docs/papers/2015/p0080r0.pdf.

[111] D. Vandevoorde, N. M. Josuttis, and D. Gregor, *C++ Templates: The Complete Guide*, 2 edition. Boston: Addison-Wesley Professional, Sep. 2017, ISBN: 978-0-321-71412-1.

[112] N. M. Josuttis, *C++17 - The Complete Guide: First Edition*. NicoJosuttis, Sep. 6, 2019, 454 pp., ISBN: 978-3-96730-017-8.

[113] A. Alexandrescu and D. Lafferty, *Modern C++ Design: Generic Programming and Design Patterns Applied*, 1st edition. Boston, MA: Addison-Wesley Professional, Feb. 13, 2001, 360 pp., ISBN: 978-0-201-70431-0.

[114] P. E. Burke, K. E. Remley, and D. P. Griesheimer, "GPU ACCELERATION OF DOPPLER BROADENING FOR NEUTRON TRANSPORT CALCULATIONS," *EPJ Web of Conferences*, vol. 247, p. 04 017, 2021, Publisher: EDP Sciences, ISSN: 2100-014X. DOI: 10.1051/epjconf/202124704017. [Online]. Available: https://www.epj-conferences.org/articles/epjconf/abs/2021/01/epjconf_physor2020_04017/epjconf_physor2020_04017.html (visited on 07/10/2024).

[115] J. E. Hoogenboom, "THE MONTE CARLO PERFORMANCE BENCHMARK TEST - AIMS, SPECIFICATIONS AND FIRST RESULTS," presented at the International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2011), Rio de Janeiro, RJ, Brazil: American Nuclear Society,

May 8, 2011, ISBN: ISBN 978-85-63688-00-2. [Online]. Available: https://inis.iaea.org/collection/NCLCollectionStore/_Public/48/022/48022330.pdf?r=1 (visited on 07/21/2024).

[116] J. Tramm. "Jtramm/openmc_offloading_benchmarks." (), [Online]. Available: https://github.com/jtramm/openmc_offloading_benchmarks (visited on 07/21/2024).

[117] "CUDA pro tip: Optimized filtering with warp-aggregated atomics," NVIDIA Technical Blog. (Oct. 2, 2014), [Online]. Available: https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/ (visited on 07/17/2024).

[118] S. Harper, P. Romano, B. Forget, and K. Smith, "Efficient dynamic threadsafe neighbor lists for monte carlo ray tracing," presented at the M&C 2019 Topical Meeting, Portland, OR, Aug. 25, 2019.

[119] M. E. O'Neill, "Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation," Harvey Mudd College, Claremont, CA, Tech. Rep. HMC-CS-2014-0905, Sep. 2014.

[120] E. Biondo, "Implementation of the windowed multipole method in shift," OAK RIDGE NATIONAL LABORATORY, Oak Ridge, TN, United States, ORNL/TM-2021/2056, Jul. 15, 2021. [Online]. Available: https://info.ornl.gov/sites/publications/Files/Pub159896.pdf.

[121] S. Johnson, *Faddeeva Package*, http://ab-initio.mit.edu/wiki/index.php/Faddeeva_Package.

[122] H. Henryson II, B. J. Toppel, and C. G. Stenberg, "Mc^{2}2: A code to calculate fast neutron spectra and multigroup cross sections," Argonne National Laboratory, Tech. Rep. Argonne-8144, 1976.

[123] B. Forget, J. Yu, and G. Ridley, "Performance Improvements of the Windowed Multipole Formalism using a Rational Fraction Approximation of the Faddeeva Function," in *International Conference on Physics of Reactors 2022*, Pittsburg, PA, May 2022.

[124] J. Humlicek, "An efficient method for evaluation of the complex probability function: The voigt function and its derivatives," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 21, pp. 309–313, 1979.

[125] J. Humlicek, "Optimized computation of the voigt and complex probability functions," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 27, pp. 437–444, 1982.

[126] A. Hui, B. Armstrong, and A. Wray, "Rapid computation of the voigt and complex error functions," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 19, pp. 509–516, 1978.

[127] J. Tramm, P. Romano, P. Shriwise, A. Lund, J. Doerfert, P. Steinbrecher, A. Siegel, and G. Ridley, *Performance portable monte carlo particle transport on intel, NVIDIA, and AMD GPUs*, Mar. 18, 2024. DOI: 10.48550/arXiv.2403.12345. arXiv: 2403.12345[cs]. [Online]. Available: http://arxiv.org/abs/2403.12345 (visited on 07/31/2024).

[128] E. M. Gelbard, "Epithermal Scattering in VIM," Argonne National Laboratory, Argonne, IL, United States, Tech. Rep. FRA-TM-123, Dec. 1979.

[129] W. Rothenstein and R. Dagan, "Two-body kinetics treatment for neutron scattering from a heavy Maxwellian gas," *Annals of Nuclear Energy*, vol. 22, no. 11, pp. 723–730, Nov. 1995, ISSN: 0306-4549. DOI: 10.1016/0306-4549(95)00002-A.

[130] M. Ouisloumen and R. Sanchez, "A Model for Neutron Scattering Off Heavy Isotopes That Accounts for Thermal Agitation Effects," *Nuclear Science and Engineering*, vol. 107, no. 3, pp. 189–200, Mar. 1991, ISSN: 0029-5639. DOI: 10.13182/NSE89-186.

[131] D. Lee, K. Smith, and J. Rhodes, "The impact of 238U resonance elastic scattering approximations on thermal reactor Doppler reactivity," *Annals of Nuclear Energy*, PHYSOR 2008, vol. 36, no. 3, pp. 274–280, Apr. 2009, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2008.11.026.

[132] T. MORI and Y. NAGAYA, "Comparison of Resonance Elastic Scattering Models Newly Implemented in MVP Continuous-Energy Monte Carlo Code," *Journal of Nuclear Science and Technology*, vol. 46, no. 8, pp. 793–798, Aug. 2009, ISSN: 0022-3131. DOI: 10.1080/18811248.2007.9711587.

[133] N. Choi and H. G. Joo, "Relative Speed Tabulation Method for Efficient Treatment of Resonance Scattering in GPU-Based Monte Carlo Neutron Transport Calculation," *Nuclear Science and Engineering*, vol. 195, no. 9, pp. 954–964, Sep. 2021, ISSN: 0029-5639. DOI: 10.1080/00295639.2021.1887701.

[134] B. Becker, R. Dagan, and G. Lohnert, "Proof and implementation of the stochastic formula for ideal gas, energy dependent scattering kernel," *Annals of Nuclear Energy*, vol. 36, no. 4, pp. 470–474, May 2009, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2008.12.001.

[135] R. Dagan, "On the use of S(α,β) tables for nuclides with well pronounced resonances," *Annals of Nuclear Energy*, vol. 32, no. 4, pp. 367–377, Mar. 2005, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2004.11.003.

[136] A. Zoia, E. Brun, C. Jouanne, and F. Malvagi, "Doppler broadening of neutron elastic scattering kernel in Tripoli-4®," *Annals of Nuclear Energy*, vol. 54, pp. 218–226, Apr. 2013, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2012.11.023.

[137] T. H. Trumbull and T. E. Fieno, "Effects of applying the doppler broadened rejection correction method for LEU and MOX pin cell depletion calculations," *Annals of Nuclear Energy*, vol. 62, pp. 184–194, Dec. 2013, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2013.06.013.

[138] S. Hart, G. Maldonado, S. Goluoglu, and B. Rearden, "Implementation of the doppler broadening rejection correction in keno," *Transactions of the American Nuclear Society*, vol. 108, pp. 423–425, Jan. 2013.

[139] T. Viitanen and J. Leppänen, "Explicit Treatment of Thermal Motion in Continuous-Energy Monte Carlo Tracking Routines," *Nuclear Science and Engineering*, vol. 171, no. 2, pp. 165–173, Jun. 2012, ISSN: 0029-5639. DOI: 10.13182/NSE11-36.

[140] J. A. Walsh, B. Forget, and K. S. Smith, "Accelerated sampling of the free gas resonance elastic scattering kernel," *Annals of Nuclear Energy*, vol. 69, pp. 116–124, Jul. 2014, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2014.01.017.

[141] J. Liang, P. Ducru, and B. Forget, "Target Velocity Sampling for Resonance Elastic Scattering Using Windowed Multipole Cross Section Data," vol. 119, Nov. 2018, pp. 1163–1166.

[142] E. Biondo, V. Sobes, A. Holcomb, S. Hamilton, and T. Evans, "Algorithm for Free Gas Elastic Scattering without Rejection Sampling," in *The International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*, Raleigh, NC, Oct. 2021. DOI: 10.13182/M\&C21-33659.

[143] S. Liu, X. Peng, C. Josey, J. Liang, B. Forget, K. Smith, and K. Wang, "Generation of the windowed multipole resonance data using Vector Fitting technique," *Annals of Nuclear Energy*, vol. 112, pp. 30–41, Feb. 2018, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2017.09.042.

[144] P. Ducru, A. Alhajri, I. Meyer, B. Forget, V. Sobes, C. Josey, and J. Liang, "Windowed multipole representation of R-matrix cross sections," *Physical Review C*, vol. 103, no. 6, p. 064 610, Jun. 2021. DOI: 10.1103/PhysRevC.103.064610.

[145] R. N. Hwang, "A Rigorous Pole Representation of Multilevel Cross Sections and Its Practical Applications," *Nuclear Science and Engineering*, vol. 96, no. 3, pp. 192–209, Jul. 1987, ISSN: 0029-5639. DOI: 10.13182/NSE87-A16381.

[146] J. Humlíček, "An efficient method for evaluation of the complex probability function: The Voigt function and its derivatives," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 21, no. 4, pp. 309–313, Apr. 1979, ISSN: 0022-4073. DOI: 10.1016/0022-4073(79)90062-1.

[147] A. Deaño and N. M. Temme, "Analytical and numerical aspects of a generalization of the complementary error function," *Applied Mathematics and Computation*, vol. 216, no. 12, pp. 3680–3693, Aug. 2010, ISSN: 0096-3003. DOI: 10.1016/j.amc.2010.05.025.

[148] R. Aïd, L. Campi, and N. Langrené, "A Structural Risk-Neutral Model for Pricing and Hedging Power Derivatives," *Mathematical Finance*, vol. 23, no. 3, pp. 387–438, Feb. 2012. DOI: 10.1111/j.1467-9965.2011.00507.x.

[149] C. M. Bender and S. A. Orszag, *Advanced Mathematical Methods for Scientists and Engineers I: Asymptotic Methods and Perturbation Theory*, Softcover reprint of hardcover 1st ed. 1999 edition. New York Heidelberg: Springer, Dec. 2010, ISBN: 978-1-4419-3187-0.

[150] A. ; M. F. G.; Harry Bateman Erdelyi, *Tables of Integral Transforms Volume 1*, 1st Edition. McGraw-Hill Book Company, Jan. 1954.

[151] F. G. Lether, "Constrained near-minimax rational approximations to Dawson's integral," *Applied Mathematics and Computation*, vol. 88, no. 2, pp. 267–274, Dec. 1997, ISSN: 0096-3003. DOI: 10.1016/S0096-3003(96)00330-X.

[152] F. G. Lether and P. R. Wenston, "Elementary approximations for Dawson's integral," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 46, no. 4, pp. 343–345, Oct. 1991, ISSN: 0022-4073. DOI: 10.1016/0022-4073(91)90099-C.

[153] W. Gautschi, "Recursive Computation of Certain Integrals," *Journal of the ACM*, vol. 8, no. 1, pp. 21–40, Jan. 1961, ISSN: 0004-5411, 1557-735X. DOI: 10.1145/321052.321054.

[154] D. G. Hummer, "Exapansions of Dawson's Function in a Series of Chebyshev Polynomials," *Mathematics of Computation*, vol. 18, no. 86, pp. 317–319, 1964, ISSN: 0025-5718. DOI: 10.2307/2003311.

[155] W. J. Cody, K. A. Paciorek, and H. C. Thacher, "Chebyshev Approximations for Dawson's Integral," *Mathematics of Computation*, vol. 24, no. 109, pp. 171–178, 1970, ISSN: 0025-5718. DOI: 10.2307/2004886.

[156] *Cephes*, https://netlib.org/cephes/.

[157] F. Schreier, "The Voigt and complex error function: Humlíček's rational approximation generalized," *Monthly Notices of the Royal Astronomical Society*, vol. 479, no. 3, pp. 3068–3075, Sep. 2018, ISSN: 0035-8711. DOI: 10.1093/mnras/sty1680.

[158] S. M. Abrarov and B. M. Quine, "Efficient algorithmic implementation of the Voigt/-complex error function based on exponential series approximation," *Applied Mathematics and Computation*, vol. 218, no. 5, pp. 1894–1902, Nov. 2011, ISSN: 0096-3003. DOI: 10.1016/j.amc.2011.06.072.

[159] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd edition. Cambridge, UK ; New York: Cambridge University Press, Sep. 10, 2007, 1256 pp., ISBN: 978-0-521-88068-8.

[160] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, Mar. 2020, ISSN: 1548-7105. DOI: 10.1038/s41592-019-0686-2.

[161] D. A. Brown, M. B. Chadwick, R. Capote, *et al.*, "ENDF/B-VIII.0: The 8th Major Release of the Nuclear Reaction Data Library with CIELO-project Cross Sections, New Standards and Thermal Scattering Data," *Nuclear Data Sheets*, Special Issue on Nuclear Reaction Data, vol. 148, pp. 1–142, Feb. 2018, ISSN: 0090-3752. DOI: 10.1016/j.nds.2018.02.001. (visited on 07/29/2023).

[162] L. B. Levitt, "The Probability Table Method for Treating Unresolved Neutron Resonances in Monte Carlo Calculations," *Nuclear Science and Engineering*, vol. 49, no. 4, pp. 450–457, Dec. 1972, ISSN: 0029-5639. DOI: 10.13182/NSE72-3. (visited on 12/22/2021).

[163] N. A. Gibson, K. Smith, and B. Forget, "Simple benchmark for evaluating self-shielding models," in *Proceedings of the Mathematics and Computations, Supercomputing in Nuclear Applications and Monte Carlo International Conference, 2015.*, Nashville, TN: American Nuclear Society, Apr. 2015, ISBN: 9781510808041. (visited on 07/31/2023).

[164] R. Macfarlane, D. W. Muir, R. M. Boicourt, I. I. I. Kahler, and J. L. Conlin, "The NJOY Nuclear Data Processing System, Version 2016," Los Alamos National Lab. (LANL), Los Alamos, NM (United States), Tech. Rep. LA-UR-17-20093, Jan. 2017. DOI: 10.2172/1338791. (visited on 05/17/2023).

[165] J.-C. Sublet and P. Ribon, "A Probability Table Based Cross Section Processing System: CALENDF - 2001," *Journal of Nuclear Science and Technology*, vol. 39, no. sup2, pp. 856–859, Aug. 2002, ISSN: 0022-3131. DOI: 10.1080/00223131.2002.10875233. (visited on 04/27/2023).

[166] J. Walsh, B. Forget, K. Smith, B. Kiedrowski, and F. Brown, "Direct, On-the-fly Calculation of Unresolved Resonance Region Cross Sections in Monte Carlo Simulations," Apr. 2015.

[167] A. Jimenez carrascosa, E. Fridman, N. Garcia herranz, F. Alvarez valarde, P. Romojaro, and F. Bostelmann, "About the Impact of the Unresolved Resonance Region in Monte Carlo Simulations of Sodium Fast Reactors," Oak Ridge National Lab. (ORNL), Oak Ridge, TN (United States), Tech. Rep., May 2019. (visited on 04/26/2023).

[168] C. E. Porter and R. G. Thomas, "Fluctuations of Nuclear Reaction Widths," *Physical Review*, vol. 104, no. 2, pp. 483–491, Oct. 1956. DOI: 10.1103/PhysRev.104.483. (visited on 04/27/2023).

[169] A. Foderaro, *The Elements of Neutron Interaction Theory*. Cambridge, Mass.: The MIT Press, Mar. 2003, ISBN: 978-0-262-56160-0.

[170] A. Trkov and D. A. Brown, "ENDF-6 Formats Manual: Data Formats and Procedures for the Evaluated Nuclear Data Files," Brookhaven National Lab. (BNL), Upton, NY (United States), Tech. Rep. BNL-203218-2018-INRE, Jan. 2018. DOI: 10.2172/1425114. (visited on 09/11/2023).

[171] L. Dresner, *Resonance Absorption in Nuclear Reactors: International Series of Monographs on Nuclear Energy, Vol. 4*. Pergamon, Nov. 1960, ISBN: 978-1-4832-0929-6.

[172] C. Jeannesson, "Development of a methodology to exploit nuclear data in the unresolved resonance range and the impact on criticality safety and reactor applications," Ph.D. dissertation, Université Paris-Saclay, Dec. 2020. (visited on 07/31/2023).

[173] D. E. Cullen, "A Short History of ENDF/B Unresolved Resonance Parameters," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. LLNL-TR-461199, Oct. 2010. DOI: 10.2172/1124805. (visited on 07/31/2023).

[174] D. E. Cullen, *PREPRO 2019*, https://www-nds.iaea.org/publications/nds/iaea-nds-0229/. (visited on 07/31/2023).

[175] S. Liu, Y. Yuan, J. Yu, D. She, and K. Wang, "On-the-fly treatment of temperature dependent cross sections in the unresolved resonance region in RMC code," *Annals of Nuclear Energy*, vol. 111, pp. 234–241, Jan. 2018, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2017.09.010. (visited on 09/15/2023).

[176] R. D. Mosteller and R. C. Little, "Impact of MCNP Unresolved Resonance Probability-Table Treatment on Uranium and Plutonium Benchmarks," Los Alamos National Lab. (LANL), Los Alamos, NM (United States), Tech. Rep. LA-UR-99-2300, Sep. 1999. (visited on 09/15/2023).

[177] J. Yu, H. Huo, W. Li, G. Yu, and K. Wang, "Validation of a New PURC Module for Calculating Probability Table in Unresolved Resonance Region in RXSP Code," in *2014 22nd International Conference on Nuclear Engineering*, American Society of Mechanical Engineers Digital Collection, Nov. 2014. DOI: 10.1115/ICONE22-30307. (visited on 09/15/2023).

[178] M. E. Dunn and L. C. Leal, "Calculating Probability Tables for the Unresolved-Resonance Region Using Monte Carlo Methods," *Nuclear Science and Engineering*, vol. 148, no. 1, pp. 30–42, Sep. 2004, ISSN: 0029-5639. DOI: 10.13182/NSE04-A2438. (visited on 12/23/2021).

[179] F. Frohner, *Evaluation and Analysis of Nuclear Resonance Data* (Evaluation et Analyse Des Donnees Relatives Aux Resonances Nucleaires). Nuclear Energy Agency of the OECD (NEA): Organisation for Economic Co-Operation and Development, 2000, ISBN: 978-92-64-18272-1.

[180] F. J. Dyson, "Statistical Theory of the Energy Levels of Complex Systems. I," *Journal of Mathematical Physics*, vol. 3, no. 1, pp. 140–156, Dec. 2004, ISSN: 0022-2488. DOI: 10.1063/1.1703773. (visited on 04/27/2023).

[181] J. A. (A. Walsh, "On-the-fly nuclear data processing methods for Monte Carlo simulations of intermediate and fast spectrum systems," Thesis, Massachusetts Institute of Technology, 2016. (visited on 07/31/2023).

[182] A. Holcomb, L. Leal, F. Rahnema, and D. Wiarda, "A New Method for Generating Probability Tables in the Unresolved Resonance Region," *Nuclear Science and Engineering*, vol. 186, no. 2, pp. 147–155, May 2017, ISSN: 0029-5639. DOI: 10.1080/00295639.2016.1273632. (visited on 07/31/2023).

[183] X. Wu, P. Liu, and Z. Ge, "Implementation of Probability Table Generation Using Ladder Method in Ruler," *EPJ Web of Conferences*, vol. 239, p. 10 005, 2020, ISSN: 2100-014X. DOI: 10.1051/epjconf/202023910005. (visited on 04/27/2023).

[184] K. Hu, X. Ma, X. Ma, Y. Huang, C. Zhang, and Y. Chen, "Development and verification of a new nuclear data processing code AXSP," *Frontiers in Energy Research*, vol. 10, 2022, ISSN: 2296-598X. (visited on 07/31/2023).

[185] D. A. Brown, "A tale of two tools: Mcres.py, a stochastic resonance generator, and grokres.py, a resonance quality assurance tool," Brookhaven National Lab. (BNL), Upton, NY (United States), Tech. Rep. BNL-209313-2018-INRE, Oct. 2018. DOI: 10.2172/1478482. (visited on 04/27/2023).

[186] D. E. Cullen and A. Trkov, *URR-PACK: Calculating Self-Shielding in the Unresolved Resonance Energy Range*, https://www-nds.iaea.org/publications/indc/indc-nds-0711/. (visited on 07/31/2023).

[187] P. Romano and S. Harper, "Nuclear data processing capabilities in OpenMC," *EPJ Web of Conferences*, vol. 146, p. 06 011, Jan. 2017. DOI: 10.1051/epjconf/201714606011.

[188] P. Ribon and J. Maillard, "Probability Tables and Gauss Quadrature: Application To Neutron Cross-Sections in the Unresolved Energy Range," Saratoga Springs, NY, Sep. 1986. [Online]. Available: https://inis.iaea.org/collection/NCLCollectionStore/_Public/18/030/18030192.pdf (visited on 04/01/2024).

[189] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012, ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001.

[190] R. N. Hwang, "An overview of current resonance theory for fast-reactor applications," *Annals of Nuclear Energy*, vol. 9, no. 1, pp. 31–44, Jan. 1982, ISSN: 0306-4549. DOI: 10.1016/0306-4549(82)90058-5. (visited on 04/27/2023).

[191] D. Brown and T. Kawano, "An analytic approach to probability tables for the unresolved resonance region," *EPJ Web of Conferences*, vol. 146, p. 12 008, 2017, ISSN: 2100-014X. DOI: 10.1051/epjconf/201714612008. (visited on 07/31/2023).

[192] T. Jordanov and A. Lukyanov, "Analytical Model of Neutron Resonance Cross Sections in the Unresolved Energy Region," *Bulgarian Journal of Physics*, vol. 21, no. 1, pp. 8–17, Sep. 1993.

[193] S. Kumar, B. Dietz, T. Guhr, and A. Richter, "Distribution of Off-Diagonal Cross Sections in Quantum Chaotic Scattering: Exact Results and Data Comparison," *Physical Review Letters*, vol. 119, no. 24, p. 244 102, Dec. 2017, Publisher: American Physical Society. DOI: 10.1103/PhysRevLett.119.244102. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.119.244102 (visited on 04/20/2024).

[194] R. N. Hwang, "Exploration of an analytical model for treating the unresolved resonances," *Nuclear Science and Engineering*, vol. 167, no. 1, pp. 1–39, Jan. 1, 2011, Publisher: Taylor & Francis _eprint: https://doi.org/10.13182/NSE10-004, ISSN: 0029-5639. DOI: 10.13182/NSE10-004. [Online]. Available: https://doi.org/10.13182/NSE10-004 (visited on 07/31/2024).

[195] M. Aigner and G. M. Ziegler, "Cotangent and the Herglotz trick," in *Proofs from THE BOOK*, M. Aigner and G. M. Ziegler, Eds., Berlin, Heidelberg: Springer, 2010, pp. 149–154, ISBN: 978-3-642-00856-6. DOI: 10.1007/978-3-642-00856-6_23. (visited on 09/13/2023).

[196] O. Barndorff-Nielsen, "Hyperbolic distributions and distributions on hyperbolae," *Scandinavian Journal of Statistics*, vol. 5, no. 3, pp. 151–157, 1978, Publisher: [Board of the Foundation of the Scandinavian Journal of Statistics, Wiley], ISSN: 0303-6898. [Online]. Available: https://www.jstor.org/stable/4615705 (visited on 11/17/2023).

[197] D. Cheng, "Randomly weighted sums of dependent random variables with dominated variation," *Journal of Mathematical Analysis and Applications*, vol. 420, no. 2, pp. 1617–1633, Dec. 15, 2014, ISSN: 0022-247X. DOI: 10.1016/j.jmaa.2014.06.048. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022247X14005952 (visited on 11/17/2023).

[198] O. E. Barndorff-Nielsen, "Normal Inverse Gaussian Distributions and Stochastic Volatility Modelling," *Scandinavian Journal of Statistics*, vol. 24, no. 1, pp. 1–13, 1997, ISSN: 0303-6898. JSTOR: 4616433. (visited on 05/15/2023).

[199] K. Podgórski and J. Wallin, "Convolution-invariant subclasses of generalized hyperbolic distributions," *Communications in Statistics - Theory and Methods*, vol. 45, no. 1, pp. 98–103, Jan. 2, 2016, Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/0361092 ISSN: 0361-0926. DOI: 10.1080/03610926.2013.821489. [Online]. Available: https://doi.org/10.1080/03610926.2013.821489 (visited on 11/20/2023).

[200] J. R. Michael, W. R. Schucany, and R. W. Haas, "Generating Random Variates Using Transformations with Multiple Roots," *The American Statistician*, vol. 30, no. 2, pp. 88–90, May 1976, ISSN: 0003-1305. DOI: 10.1080/00031305.1976.10479147. (visited on 05/12/2023).

[201] R. Beran, "Minimum hellinger distance estimates for parametric models," *The Annals of Statistics*, vol. 5, no. 3, pp. 445–463, May 1977, Publisher: Institute of Mathematical Statistics, ISSN: 0090-5364, 2168-8966. DOI: 10.1214/aos/1176343842. [Online]. Available: https://projecteuclid.org/journals/annals-of-statistics/volume-5/issue-

3/Minimum-Hellinger-Distance-Estimates-for-Parametric-Models/10.1214/aos/1176343842.full (visited on 07/02/2024).

[202] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[203] V. Peiris, N. Sharon, N. Sukhorukova, and J. Ugon, "Generalised rational approximation and its application to improve deep learning classifiers," *Applied Mathematics and Computation*, vol. 389, p. 125 560, Jan. 15, 2021, ISSN: 0096-3003. DOI: 10.1016/j.amc.2020.125560. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0096300320305166 (visited on 07/02/2024).

[204] J. M. Hokanson, *Multivariate rational approximation using a stabilized sanathanan-koerner iteration*, Sep. 22, 2020. DOI: 10.48550/arXiv.2009.10803. arXiv: 2009.10803[cs,math]. [Online]. Available: http://arxiv.org/abs/2009.10803 (visited on 07/02/2024).

[205] Y. Nakatsukasa, O. Sète, and L. N. Trefethen, "The AAA algorithm for rational approximation," *SIAM Journal on Scientific Computing*, vol. 40, no. 3, A1494–A1522, Jan. 2018, ISSN: 1064-8275, 1095-7197. DOI: 10.1137/16M1106122. arXiv: 1612.00337[math]. [Online]. Available: http://arxiv.org/abs/1612.00337 (visited on 07/02/2024).

[206] J.-P. Berrut and L. N. Trefethen, "Barycentric lagrange interpolation," *SIAM Review*, vol. 46, no. 3, pp. 501–517, Jan. 2004, Publisher: Society for Industrial and Applied Mathematics, ISSN: 0036-1445. DOI: 10.1137/S0036144502417715. [Online]. Available: https://epubs.siam.org/doi/10.1137/S0036144502417715 (visited on 07/02/2024).

[207] M. S. Floater and K. Hormann, "Barycentric rational interpolation with no poles and high rates of approximation," *Numerische Mathematik*, vol. 107, no. 2, pp. 315–331, Aug. 1, 2007, ISSN: 0945-3245. DOI: 10.1007/s00211-007-0093-y. [Online]. Available: https://doi.org/10.1007/s00211-007-0093-y (visited on 07/02/2024).

[208] "Improve shader performance and in-game frame rates with shader execution reordering," NVIDIA Technical Blog. (Oct. 13, 2022), [Online]. Available: https://developer.nvidia.com/blog/improve-shader-performance-and-in-game-frame-rates-with-shader-execution-reordering/ (visited on 08/03/2024).

[209] B. Forget and A. Alhajri, "Normalizing flows for thermal scattering sampling," *Annals of Nuclear Energy*, vol. 170, p. 108 974, Jun. 2022, ISSN: 0306-4549. DOI: 10.1016/j.anucene.2022.108974. (visited on 07/10/2023).

Appendix A

GPU Rejection Sampling Experiment

```
// This calculates the distribution of the number of iterations
   required for a warp
// of threads to complete a rejection sampling iteration.
#define WARPSIZE 32
#include <iostream>
#include <vector>

constexpr uint64_t master_seed {1};
constexpr uint64_t prn_mult   {28061969105067807 09LL};   //
   multiplication
constexpr uint64_t prn_add    {1};                        //
   additive factor, c
constexpr uint64_t prn_mod    {0x8000000000000000};       // 2^63
constexpr uint64_t prn_mask   {0x7fffffffffffffff};       // 2^63 -
    1
constexpr double   prn_norm   {1.0 / prn_mod};            // 2^-63

// A linear congruential random number generator
__device__ __inline__ double prn(uint64_t* seed)
{
  *seed = (prn_mult * (*seed) + prn_add) & prn_mask;
  return (*seed) * prn_norm;
}

// Each block takes on its own rejection probability
template<unsigned Warpsize>
```

```
__global__ void calculate_mean_iterations_required(double*
    rejection_probabilities,
                                                   double*
                                                      mean_iterations_required
                                                      ,
                                                   double*
                                                      mean_squared_iterations_required
                                                      ,
                                                   uint64_t* random_seeds)
                                                      {
  int tid = threadIdx.x;
  int n_outer_loops = 1000;
  extern __shared__ int max_warp_iterations[];

  double mean_result = 0;
  double mean_square_result = 0;
  double accept_prob = 1.0-rejection_probabilities[blockIdx.x];

  for (int outer=0; outer<n_outer_loops; ++outer) {
    int num_iterations = 0;
    bool sample_not_obtained = true;
    while (sample_not_obtained) {
      double xi = prn(random_seeds+threadIdx.x+blockDim.x*blockIdx
          .x);

      // Surrogate rejection sampling:
      if (xi < accept_prob) sample_not_obtained = false;

      num_iterations++;
    }

    max_warp_iterations[tid] = num_iterations;

    // Now need to take the max num_iterations across the warp,
       because all threads would
    // have waited on that last one. Parallel reduce using max as
       the binary operator.

```

```
52      if (Warpsize > 64) { // this statement should evaluate at
            compile time
53        __syncthreads();
54        if (tid < 64) max_warp_iterations[tid] = max(
              max_warp_iterations[tid],
55             max_warp_iterations[tid+64]);
56      }
57      if (Warpsize > 32) { // this statement should evaluate at
            compile time
58        __syncthreads();
59        if (tid < 32) max_warp_iterations[tid] = max(
              max_warp_iterations[tid],
60             max_warp_iterations[tid+32]);
61      }
62      __syncthreads();
63      if (tid < 16) max_warp_iterations[tid] = max(
           max_warp_iterations[tid],
64            max_warp_iterations[tid+16]);
65      __syncthreads();
66      if (tid < 8) max_warp_iterations[tid] = max(
           max_warp_iterations[tid],
67            max_warp_iterations[tid+8]);
68      __syncthreads();
69      if (tid < 4) max_warp_iterations[tid] = max(
           max_warp_iterations[tid],
70            max_warp_iterations[tid+4]);
71      __syncthreads();
72      if (tid < 2) max_warp_iterations[tid] = max(
           max_warp_iterations[tid],
73            max_warp_iterations[tid+2]);
74      __syncthreads();
75      if (tid == 0) {
76        max_warp_iterations[tid] = max(max_warp_iterations[tid],
77            max_warp_iterations[tid+1]);
78        mean_result = (outer * mean_result + max_warp_iterations[0])
              / (outer+1);
79        mean_square_result = (outer * mean_square_result +
```

```
80            max_warp_iterations[0]*max_warp_iterations[0]) / (outer
                 +1);
81      }
82    }
83
84    if (tid==0) {
85      mean_iterations_required[blockIdx.x] = mean_result;
86      mean_squared_iterations_required[blockIdx.x] =
           mean_square_result;
87    }
88  }
89
90  int main(int argc, char* argv[]) {
91
92    // Command line arguments are min rejection probability, max
       rejection probability
93    if (argc != 4) {
94      std::cout << "Incorrect argument count. First input min
           rejection probability, "
95        "max, then the number of linearly spaced probabilities to
             check." << std::endl;
96    }
97    double min_r = std::stod(argv[1]);
98    double max_r = std::stod(argv[2]);
99    int n_probs = std::stoi(argv[3]);
100   if (min_r < 0 or min_r > 1) std::cerr << "MIN_R INVALID" << std
       ::endl;
101   if (max_r < 0 or max_r > 1) std::cerr << "MAX_R INVALID" << std
       ::endl;
102   if (min_r >= max_r) std::cerr << "BADLY ORDERED PROBABILITIES"
       << std::endl;
103
104   // Create the rejection probabilities to be used and move them
       to device
105   double* dev_rej_probs;
106   size_t nbytes = sizeof(double) * n_probs;
107   std::vector<double> rejection_probabilities(n_probs);
108   double dp = (max_r-min_r)/(n_probs-1);
```

```
109   for (int i=0; i< n_probs; ++i)
110     rejection_probabilities[i] = min_r + dp * i;
111   cudaMalloc(&dev_rej_probs, nbytes);
112   cudaMemcpy(dev_rej_probs, rejection_probabilities.data(),
113       nbytes, cudaMemcpyHostToDevice);
114
115   // Allocate memory for results on device
116   double* mean_device;
117   double* mean_sq_device;
118   cudaMalloc(&mean_device, nbytes);
119   cudaMalloc(&mean_sq_device, nbytes);
120
121   // Allocate memory for random number seeds on device
122   uint64_t* device_rngs;
123   cudaMalloc(&device_rngs, sizeof(uint64_t) * n_probs * WARPSIZE);
124   std::vector<uint64_t> rngs_host(n_probs * WARPSIZE);
125   for (uint64_t i=0; i<n_probs*WARPSIZE; ++i)
126     rngs_host[i] = master_seed + i;
127   cudaMemcpy(device_rngs, rngs_host.data(),
128       sizeof(uint64_t) * n_probs * WARPSIZE,
          cudaMemcpyHostToDevice);
129
130   // Run the main kernel, one block per rejection probability
131   calculate_mean_iterations_required<WARPSIZE><<<n_probs,
132       WARPSIZE, sizeof(int)*WARPSIZE>>>(
133       dev_rej_probs, mean_device, mean_sq_device, device_rngs);
134
135   // Bring results back to host
136   std::vector<double> mean_iterations(n_probs);
137   std::vector<double> mean_squared_iterations(n_probs);
138   cudaMemcpy(mean_iterations.data(), mean_device, nbytes,
      cudaMemcpyDeviceToHost);
139   cudaMemcpy(mean_squared_iterations.data(), mean_sq_device,
      nbytes, cudaMemcpyDeviceToHost);
140
141   // And finally, print out the results
142   for (int i=0; i<n_probs; ++i)
143     std::cout << rejection_probabilities[i] << " "
```

```
144                 << mean_iterations[i] << " " <<
                       mean_squared_iterations[i] << std::endl;
145
146   cudaFree(dev_rej_probs);
147   cudaFree(mean_device);
148   cudaFree(mean_sq_device);
149   cudaFree(device_rngs);
150 }
```

Appendix B

C++ Rational Approximation to $w(z)$

```
std::complex<double> faddeeva(std::complex<double> z)
{
  z += std::complex<double>(1.31183j);
  const auto zz = z * z;
  constexpr std::array<std::complex<double>, 16> aa = {41445.0374210222,
    -136631.072925829j, -191726.143960199, 268628.568621291j, 173247.907201704,
    -179862.56759178j, -63310.0020563537, 56893.7798630723j, 11256.4939105413,
    -9362.62673144278j, -1018.67334277366, 810.629101627698j, 44.5707404545965,
    -34.5401929182016j, -0.740120821385939, 0.564189583547714j};
  constexpr std::array<std::complex<double>, 16> bb = {7918.06640624997, 0.0,
    -126689.0625, 0.0, 295607.8125, 0.0, -236486.25, 0.0, 84459.375, 0.0,
    -15015.0, 0.0, 1365.0, 0.0, -60.0, 0.0};
  return ((((((((((((((aa[15] * z + aa[14]) * z + aa[13]) * z + aa[12]) * z +
                           aa[11]) * z + aa[10]) * z + aa[9]) * z + aa[8]) * z
                   + aa[7]) * z + aa[6]) * z + aa[5]) * z + aa[4]) * z +
            aa[3]) * z + aa[2]) * z + aa[1]) * z + aa[0]) / (((((((zz +
                     bb[14]) * zz + bb[12]) * zz + bb[10]) * zz + bb[8]) * zz
              + bb[6]) * zz + bb[4]) * zz + bb[2]) * zz + bb[0]);
}
```

Appendix C

Derivation of Eq. 4.20

The forthcoming discussion has not been made mathematically rigorous for sake of brevity and the context of a nuclear engineering journal. We begin by defining the auxiliary complex function F(z):

$$F(z) = e^{z^2} w(z, x) \tag{C.1}$$

The complex line integral theorem can then be applied when $\Im[z] > 0$:

$$F(z) - F(0) = \int_0^z \frac{\mathrm{d}F}{\mathrm{d}z}|_{z=z'} \, \mathrm{d}z' \tag{C.2}$$

Computing $\frac{\mathrm{d}F}{\mathrm{d}z}$ and inserting then reveals:

$$e^{z^2} w(z, x) - w(0, x) = \int_0^z \left(2z' e^{z'^2} w(z', x) - e^{z'^2} \frac{i}{\pi} \int_{-\infty}^{x} \frac{e^{-t^2} \, \mathrm{d}t}{(z' - t)^2} \right) \mathrm{d}z' \tag{C.3}$$

where the linearity of integration has been employed, and the interchange of differentiation and integration has also been used. The innermost integrals can now be computed exactly, carrying the z' through to the integral defining the incomplete Faddeeva function. This results in:

$$e^{z^2} w(z, x) - w(0, x) = \frac{i}{\pi} \left(\int_0^z \frac{e^{z'^2} e^{-x^2}}{x - z'} \, \mathrm{d}z' - \sqrt{\pi}(1 + \mathrm{erf}(x)) \int_0^z e^{z'^2} \, \mathrm{d}z' \right) \tag{C.4}$$

Recalling that the Faddeeva function can be defined as

$$w(z) = e^{-z^2} \left(1 + \frac{2i}{\sqrt{\pi}} \int_0^z e^{t^2} \, \mathrm{d}t \right) \quad , \tag{C.5}$$

we can identify $w(z)$ as the trailing term of Eq. C.4. The term $w(0, x)$ must be interpreted

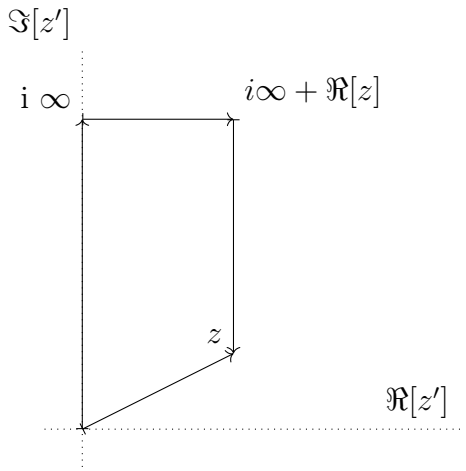


Figure C.1: Modified contour used to cancel exponential integral in Eq. C.6. The contribution from the top line is zero.

in a principal value sense, which results in a contribution in the form of a Heaviside function. The following expression then results:

$$w(z,x) = e^{-z^2}\left(-\frac{1}{2}(\mathrm{Ei}(-x^2) + \frac{i}{\pi}e^{-x^2}\int_0^z \frac{e^{z'^2}}{x-z'}\,\mathrm{d}z'\right) + \frac{1}{2}(1+\mathrm{erf}(x))(w(z) - e^{-z^2}) + h(x)e^{-z^2} \quad . \tag{C.6}$$

This result could perhaps be used for numerical calculations of $w(z,x)$. However, it suffers the shortcoming that the exponential integral term goes to infinity for $x = 0$, which is cancelled out by the integral term. However, $w(z,x)$ is well-defined at $x = 0$, and the addition of branching logic to numerical routines to handle this case would be cumbersome. The expression can be made more amenable to numerical approximation with some further simplification.

The integral term goes from 0 to z, and the integrand encloses no poles of the following path whenever $x \neq 0$. As such, a change of integration path is employed: the contour of Fig. C.1 results in a convenient cancellation of terms. The top leg of the contour is zero from the e^{t^2} term, resulting in:

$$\int_0^z \frac{e^{t^2}}{x-t}\,\mathrm{d}t = \int_0^{i\infty} \frac{e^{t^2}}{x-t}\,\mathrm{d}t - \int_z^{i\infty} \frac{e^{t^2}}{x-t}\,\mathrm{d}t \tag{C.7}$$

A little bit of algebra shows that:

$$\int_0^{i\infty} \frac{e^{t^2}}{x-t}\,\mathrm{d}t = \frac{1}{2}e^{x^2}\left(-i\pi(\mathrm{erf}(x) - \mathrm{sign}(x)) + \mathrm{Ei}(-x^2)\right) \tag{C.8}$$

The sign function term ends up cancelling out the Heaviside term upon substitution of Eq. C.7 back to Eq. C.6. The second term in Eq. C.7 easily can be transformed via the change of variables $t' = -it$ to the integral in Eq. 4.20, thus yielding Eq. 4.20.

Appendix D

Derivation of Eq. 4.50

The goal is to find the smallest n such that

$$\frac{x^n E_1(x)}{n! E_{n+1}(x)} < 1 \tag{D.1}$$

The variation of the left hand side function of Eq. D.1 as n increases, for a constant value of x, has been shown to be first increasing from one, reaching a maximum, and monotonically decreasing from that point, never becoming negative [153]. Firstly, the exponential integrals are replaced with the equivalent upper incomplete gamma function:

$$E_n(x) = x^{n-1}\Gamma(1-n, x) \tag{D.2}$$

so the equation becomes:

$$\Gamma(0, x) = n!\Gamma(-n, x) \tag{D.3}$$

Using the asymptotic formula for the upper incomplete gamma function $\Gamma(s, x) \to x^{s-1}e^{-x}$ gives this approximation to solve:

$$x^n \approx n! \tag{D.4}$$

Which can be solved approximately by first inserting Stirling's formula:

$$x^n \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{D.5}$$

Taking the nth root results in

$$ex \approx (2\pi n)^{\frac{1}{2n}} n \tag{D.6}$$

the second term on the right can be approximated by expanding the exponential, for large n:

$$ex \approx (1 + \frac{1}{2n} \log(2\pi n))n \tag{D.7}$$

This can be solved exactly in terms of the Lambert W function:

$$n \approx \frac{1}{2} W \left(\frac{e^{2ex}}{\pi} \right) \tag{D.8}$$

The asymptotic property of the Lambert W function that $W(z) \approx \log z - \log \log z$ is then used to obtain Eq. 4.50.

Appendix E

Asymptotic Approximation for $R(m, z)$ of Eq. 4.32

An efficient approximation can be found by grouping the integrand as:

$$R(m,z) = \int_0^\infty f^{(0)}(t,m)\sin(2zt)\,\mathrm{d}t = -\frac{1}{2z}\int_0^\infty f^{(0)}(t,m)\frac{\partial}{\partial t}\left[\cos(2zt)\right]\mathrm{d}t \tag{E.1}$$

where

$$f^{(0)}(t,m) = \frac{e^{-t^2}}{t^2+m^2} \quad .$$

Integrating the rightmost expression expression in Eq. E.1 by parts repeatedly results in a divergent series approximation of the form:

$$R(m,z) = \frac{f^{(0)}(0,m)}{2z} + \frac{f^{(2)}(0,m)}{8z^3} + \frac{f^{(4)}(0,m)}{32z^5} + \frac{f^{(6)}(0,m)}{128z^7} + \cdots \tag{E.2}$$

where $f^{(n)}(t,m)$ denotes the nth derivative of $f^{(0)}(m,t)$ with respect to t. Some of the subsequent values evaluated about $t = 0$ are:

$$f^{(2)}(0,m) = -\frac{2(1+m^2)}{m^4} \tag{E.3}$$

$$f^{(4)}(0,m) = \frac{12(2+2m^2+m^4)}{m^6} \tag{E.4}$$

$$f^{(6)}(0,m) = -\frac{120(6+6m^2+3m^4+m^6)}{m^8} \tag{E.5}$$

$$f^{(8)}(0,m) = \frac{1680(24+24m^2+12m^4+4m^6+m^8)}{m^{10}} \tag{E.6}$$

While seemingly progressing without a clear pattern, after a considerable amount of staring at these expressions, a simple recursive formula can be obtained to compute these values: prime for computer implementation. Consider the sequences $a_n, c_n \in \mathbb{R}$ defined by

$$a_0 = m^{-2}; \quad c_0 = 2 \tag{E.7}$$

and

$$a_{n+1} = \frac{2n(2n-1)a_n + c_n}{m^2}; \quad c_{n+1} = (4n+2)c_n \quad . \tag{E.8}$$

Using this, one can show that $f^{(2n)}(0, m) = a_n$. This allows for easy evaluation of the asymptotic series of Eq. E.2. Numerical experimentation has shown this to be an excellent approximation with a maximum error around 10^{-6} when $|z| = 5$ and $|m| = 1$, retaining only five terms. The error rapidly falls from there as $|z| \to \infty$ and $|m| \to \infty$.

Appendix F

Asymptotic Approximation for $J(z, x)$ of Eq. 4.51

Compared to the asymptotic approximation for the $R(m, z)$ integral, a clean expression for simple computer code is not available to our knowledge. Obtaining an asymptotic expression thus relies on access to a computer algebra system. Finding this starts by applying a simple change of variables to Eq. 4.51 to find:

$$J(z, x) = \int_0^{\Im[z]} \frac{e^{-t^2} e^{2i\Re[z]t}\, \mathrm{d}t}{i(\Re[z] - x) - t} \tag{F.1}$$

Where it becomes clear that numerical difficulty from expanding the exponential term originates from the $e^{2i\Re[z]t}$ modulation. This is the term to isolate to obtain the correct asymptotic behavior as the integrand becomes increasingly oscillatory. The standard repeated integration by parts procedure can then be applied. This is pure tedium, so we simply report the C++ code which evaluates five terms below.

```
const std::complex<double> pp1 = 6.0 + m2*(6.0 + 3.0*m2) +
  zr*(m*(-3.0 - 3.0*m2) +
  zr*(m2*(2.0 + 2.0*m2) +
  zr*(-2.0*m2*m + 4.0*m2*m2*zr))) +
  zi*(zr*(3.0*ii + m2* (3.0*ii - 6.0*ii*m2) +
  zr*(m*(-4.0*ii - 4.0*ii*m2) +
  zr*(m2*(6.0*ii - 4.0*ii*m2) - 16.0*ii*m2*m*zr))) +
  zi*(6.0 + m2*(6.0
  - 12*m2) + zr*(m*(-3.0 - 18.0*m2) +
  zr*(-2 - 4.0*m2*m2 + zr*(m*(6.0
  - 16.0*m2) - 24.0*m2*zr))) + zi*(40.0*ii*m2*m + zr*(3.0*ii
```

```
  + m2*(18.0*ii + 4.0*ii*m2) + zr*(m*(-4.0*ii + 16.0*ii*m2) +
  zr*(-2.0*ii + 24.0*ii*m2 + 16.0*ii*m*zr))) +
  zi*(3.0 + m2*(48.0 +
  4.0*m2) + zi*(m*(-24.0*ii - 16.0*ii*m2) +
  zi*(-4.0 - 24.0*m2 +
  zi*(16.0*ii*m + 4.0*zi + 4.0*ii*zr) +
  (-16.0*m - 4.0*zr)*zr) +
  zr*(-24.0*ii*m2 + (-16.0*ii*m - 4.0*ii*zr)*zr)) +
  zr*(m*(6.0 +
  16.0*m2) +
  zr*(-2.0 + 24.0*m2 + zr*(16.0*m + 4.0*zr)))))));
const double pp2 = (-6.0 + m2*(-6.0 - 3.0*m2) +
  zr*(m*(3.0 + 3.0*m2) + zr*(m2*(-2.0 -
  2.0*m2) + zr*(2.0*m2*m - 4.0*m2*m2*zr))))/(m2*m2*m);

// Note:std::exp(zi*(zi - 2.0*ii*zr)) = exp(-z^2) / exp(-zr^2)
result =
  (pp2 + pp1 / (cache.emz2 / cache.emrz2 * std::pow(m - ii * zi,
     5))) /
  (8. * std::pow(zr, 5));
```

Appendix G

Root Finding Bootstrap Function

```
double rootfinding_bootstrap_guess(double xi,
         double apprx_0_cdf, // approx CDF at x=0
         double dcdx, // approx PDF at x=0
         double jump, // probability jump at resonance
         std::complex<double> z) {

  // Note: can approximate length as e^{-z^2} * 3 Im[z]
  double yjumplo = 0.5 * (std::erf(z.real() -
    1.5 * z.imag()) + 1.0);
  double yjumphi = 0.5 * (std::erf(z.real() +
    1.5 * z.imag()) + 1.0);

  if (xi <= apprx_0_cdf) {
    if (yjumphi > 0.5 && yjumplo < 0.5) {
      jump *= (0.5 - yjumplo) / (yjumphi - yjumplo);
      yjumphi = 0.5;
    } else if (yjumplo > 0.5) {
      yjumplo = 0.0;
      yjumphi = 0.0;
      jump = 0.0;
    }
    if (jump > apprx_0_cdf) jump = apprx_0_cdf;
    const double d = yjumphi - yjumplo;
    const double sout = (apprx_0_cdf - jump) / (0.5 - d);
    const double sinv = jump > 0.0 ? d / jump : 0.0;
    if (xi >= sout * yjumplo + jump) {
```

```
      const auto r = sout * yjumplo + jump;
      const auto a = (r - dcdx * (yjumphi - 0.5) - apprx_0_cdf)
        / std::pow(yjumphi - 0.5, 2);
      return 0.5 * (-dcdx + std::sqrt(std::pow(dcdx, 2) -
        4.0 * (apprx_0_cdf - xi) * a)) / a + 0.5;
    } else if (xi > sout * yjumplo) {
      return (xi - sout * yjumplo) * sinv + yjumplo;
    } else {
      if (sout > 0.0)
        return xi / sout;
      else return 0.5 * yjumplo;
    }
  } else { // xi > apprx_0_cdf
    if (yjumplo < 0.5 && yjumphi > 0.5) {
      jump *= (yjumphi - 0.5) / (yjumphi - yjumplo);
      yjumplo = 0.5;
    } else if (yjumphi < 0.5) {
      yjumplo = 1.0;
      yjumphi = 1.0;
      jump = 0.0;
    }

    // Clip innapropriately large jumps
    if (jump > 1.0 - apprx_0_cdf) jump = 1.0 - apprx_0_cdf;
    const auto d = yjumphi - yjumplo;
    const auto sout = (1.0 - jump - apprx_0_cdf) / (0.5 - d);
    const auto sinv = jump > 0.0 ? d / jump : 0.0;
    const auto thresh1 = sout * (yjumplo - 0.5) + jump +
       apprx_0_cdf;
    const auto thresh2 = sout * (yjumplo - 0.5) + apprx_0_cdf;

    if (xi >= thresh1)
      return (xi - thresh1) / sout + yjumphi;
    else if (xi > sout * (yjumplo - 0.5) + apprx_0_cdf)
      return (xi - thresh2) * sinv + yjumplo;
    else {
      const auto a = (thresh2 - dcdx * (yjumplo - 0.5) -
        apprx_0_cdf)/std::pow(yjumplo - 0.5, 2);
```

```
64        return 0.5 * (-dcdx + std::sqrt(std::pow(dcdx, 2) -
65        4.0 * (apprx_0_cdf - xi) * a)) / a + 0.5;
66      }
67    }
68
69    UNREACHABLE();
70  }
```