COMET-GPU: A GPGPU-ENABLED DETERMINISTIC SOLVER FOR THE CONTINUOUS-ENERGY COARSE MESH TRANSPORT METHOD (COMET)

A Dissertation
Presented to
The Academic Faculty

by

Paul Burke

In Partial Fulfillment
of the Requirements for the Degree
Doctorate in Nuclear Engineering in the
George W. Woodruff School of Mechanical Engineering

Georgia Institute of Technology
May 2021

COPYRIGHT © 2021 BY PAUL BURKE

COMET-GPU: A GPGPU-ENABLED DETERMINISTIC SOLVER FOR THE CONTINUOUS-ENERGY COARSE MESH TRANSPORT METHOD (COMET)

Approved by:

Dr. Farzad Rahnema, Co-Advisor
School of Mechanical Engineering
Georgia Institute of Technology

Dr. Umit Catalyurek, Co-Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Daniel Gill
Advisor Engineer
Naval Nuclear Laboratory, Bettis

Dr. Bojan Petrovic
School of Mechanical Engineering
Georgia Institute of Technology

Dr. Dingkang Zhang
School of Mechanical Engineering
Georgia Institute of Technology

Date Approved: April 22, 2021

To my family, friends, and anyone who helped me along the way. I could not have done this without you all.

ACKNOWLEDGEMENTS

This research was performed under appointment to the Rickover Fellowship Program in Nuclear Engineering sponsored by the Naval Reactors Division of the U.S. Department of Energy.

This research made use of the resources of the High Performance Computing Center at Idaho National Laboratory, which is supported by the Office of Nuclear Energy of the U.S. Department of Energy and the Nuclear Science User Facilities under Contract No. DE-AC07-05ID14517.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

AE	Absolute Error
AHTR	Advanced High-Temperature Reactor
ARI	All-rods-in
BLAS	Basic Linear Algebra Subprograms
CDF	Common Data Format
CE	Continuous Energy
CMFD	Coarse Mesh Finite Difference
COMET	Coarse Mesh Transport
GEMM	Generic Matrix-Matrix
GEMV	Generic Matrix-Vector
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
I^2S	Integral Inherently Safe Reactor
$\boldsymbol{J}$	Global partial current moments vector

k	Core criticality eigenvalue
MAE	Mean Absolute Error
MPI	Message Passing Interface
ms	Milliseconds
RF	Response Function
SIMD	Single-instruction-multiple-data
SM	Streaming Multiprocessor
TRISO	Tri-structural Isotropic
UVM	CUDA Unified Virtual Memory
VHTR	Very-High-Temperature Reactor

SUMMARY

The Continuous-Energy Coarse Mesh Transport (COMET) method is a neutron transport solution method that uses a unique hybrid stochastic-deterministic solution method to obtain high-fidelity whole-core solutions to reactor physics problems with formidable speed. This method involves pre-computing solutions to individual coarse meshes within the global problem, then using a deterministic transport sweep to construct a whole-core solution from these local solutions. In this work, a new implementation of the deterministic transport sweep solver is written which includes the ability to accelerate the calculation using up to 4 Graphics Processing Units (GPUs) on one computational node. The new implementation is written in C++ with GPU-facing logic leveraging the CUDA API.

To demonstrate the new implementation, three whole-core benchmark problems were solved using the previous serial solver and various configurations of the new solver, with the relative performance compared. In this comparison, it was found that the application of one GPU to the problem resulted in between a 100x-150x speedup (depending on the specific problem) relative to the old serial solver. Excellent scaling up to 4 GPUs was observed, which brought the total speedup up to 450x-500x. Although the magnitude of the speedup was found to be problem dependent, it is noted that the overall strategy of the acceleration is not problem dependent.

As an example of a new type of analysis which is enabled by the improved speed of the solver, a sensitivity study was performed on the convergence thresholds used in controlling the inner and outer iteration processes. This study involves repeatedly solving whole-core problems using slightly varying thresholds, including computing a “gold-standard” solution to double-precision. These various runs would be prohibitively expensive if run using the old solver (upwards of weeks) but in this work were completed in around an hour. The results of the sensitivity study show that some examined cases indicate that converging the problem completely to the limits of single-precision numbers can result in pin power errors on the order of 0.1% as compared to a completely converged double-precision result. This behavior was found to be problem dependent.

CHAPTER 1. INTRODUCTION

In the design and operation of a nuclear reactor, accurate modeling of the neutronic characteristics of the reactor is of paramount importance. This involves obtaining a solution to the Boltzmann neutron transport equation. The steady-state neutron transport equation involves 6 independent variables. As a result, the equation cannot be solved analytically in all but the simplest cases. To obtain solutions for real-word problems, numerical solution methods must be used. One such numerical transport method is the continuous-energy (CE) Coarse Mesh Transport (COMET) method developed at the Computational Reactor and Medical Physics Laboratory at Georgia Tech [1]. The COMET method uses a unique hybrid stochastic-deterministic solution method to obtain high-fidelity whole-core solutions to reactor physics problems with formidable speed. In this work, a new implementation of the CE-COMET method will be developed to leverage Graphics Processing Unit (GPU) architectures.

1.1 Motivation

The High-Performance Computing (HPC) landscape for scientific computing has seen a rise in General-Purpose GPU (GPGPU) computing. GPUs differ from regular CPUs in that their architecture is designed to support many more cores (5,120 CUDA cores on the NVIDIA Tesla V100 [2]) than a CPU. These cores are designed to support the execution of the same set of operations on a large number of different data regions. As a result, these devices excel in terms of both speed and efficiency at “data-parallel” problems: problems which perform a similar set of operations repeatedly over a large amount of data. The advancement of this technology has resulted in a shift in recent years toward an increased

GPU to CPU balance ratio. The flagship HPC resources from Oak Ridge National Laboratory demonstrate this trend: Titan (1:1 ratio, 2012, [3]), Summit (6:2 ratio, 2018, [4]), and Frontier (4:1 ratio, scheduled 2021, [5]).

To effectively implement an algorithm on GPU architectures, the maximum possible degree of data parallelism must be exposed in the problem. In this context, the deterministic transport sweep of the CE-COMET method is an excellent fit for GPU parallelization. The transport sweep consists of a nested iteration process in which the inner iterations correspond to a power iteration method used to converge to the dominant eigenvector of the global reactor problem. In the power method, the n^{th} guess at the eigenvector is constructed using data only from the previous guess. With this, the operations used to update the eigenvector can be performed entirely independently, as will be shown in Chapters 2 and 3. This fact places the COMET transport sweep in a class of problems that can be called “embarrassingly parallel”. It is this inherent parallelism that will be exploited in this work to construct a GPU implementation of the method.

Whole-core CE-COMET problems are an excellent candidate for a GPU-accelerated parallel implementation. Calculations for these problems generally involve performing up to several hundred thousand dense matrix multiplications, with matrices up to 720 x 720 or larger in size. These matrix multiplications generally comprise upwards of 99% of the program runtime, with operations taking place on tens-hundreds of gigabytes of data. These operations and memory accesses are highly regular, which are excellent features for a GPU implementation. It is well-demonstrated (as will be discussed in Chapter 2) that, in highly regular operations such as these, one GPU can provide as much as 2+ orders of magnitude speedup as compared to one CPU thread. Such a speedup for a CPU-only parallel implementation would likely require multi-threaded and multi-node execution, which brings extra complication when it comes to communication, problem decomposition, and work distribution. It is thus of great interest to explore shared-memory parallelism and GPU-acceleration to demonstrate the speed that could be achieved on one computational node before moving to a multi-node implementation.

The fact that the block of matrix multiplication operations is so monolithic also implies that more than 1 GPU could be effectively employed in solving the problem. Since GPU-to-GPU data transfer is quite fast, the cost of communication can be kept low. It is thus reasonable to expect that additional GPUs applied to a CE-COMET problem will provide additional speedup, firmly planting the total speedup between 2 and 3 orders of magnitude. A multi-GPU approach also provides an additional benefit in that it enables larger problems to fit entirely into device memory (generally limited to a couple tens of GB). This is an excellent feature that enables continued applicability of the GPU solver as COMET is extended to operate on problems with larger and larger memory footprints.

The promise of more than 2 orders of magnitude speedup and the suitability of COMET for a GPU implementation thus forms the motivation for this work. The implications of such a speedup for the future of the method are extensive. The speed could enable multi-physics, time-dependent, and depletion calculations while maintaining a reasonable total runtime. It will also be demonstrated in this work that this could enable a mixed-precision approach to the numbers involved in calculation. The objectives of this work are designed to demonstrate the applicability of the problem to a GPU-accelerated implementation and show the benefits of the additional speedup.

1.2 Objectives

In this work, a new implementation of the COMET method will be developed that targets calculation on GPU architectures. As will be explained in Chapter 3, part of this undertaking involved the development of new host code as well, converting the serial Fortran solver into a C++ solver. This allows the opportunity to re-order operations and re-arrange memory to allow for an optimized parallel implementation. A CPU-only shared-memory parallel solver will also be developed as a secondary objective.

As a demonstration of the improved performance due to the newly developed capabilities, in Chapter 4 the new solver is applied to a set of whole-core benchmark problems. The performance of the code is demonstrated via comparison against the current serial solver, and scaling studies are performed for 1, 2, and 4 GPU configurations. The GPU-accelerated solver is demonstrated to have 2+ orders of magnitude speedup as compared to the serial solver.

As an example of a type of additional analysis unlocked by the new speed of the GPU-accelerated solver, a sensitivity study was performed on the specific thresholds used in truncating the iteration process. The methodology and results of this study are documented in Chapter 5. This type of analysis would have been technically possible with the serial solver (except for the double-precision runs), but the computation time alone would have spanned weeks, effectively prohibiting such a study. The specific runs for this study were completed in around an hour, including two special runs (per problem) in which gold-standard solutions were converged in double-precision (rather than the single-precision COMET typically uses) to around machine epsilon. The results of the study indicate that the problem-dependent nature of the convergence of COMET solutions can result in varying differences between solutions obtained with a specific set of convergence criteria and the gold-standard solution, including up to 3-4 orders of magnitude difference between the input threshold on relative residual eigenvector norm and the resulting error.

In Chapter 6, a set of conclusions are drawn from the results of this work. Implications of the increased computational speed are discussed, and a recommendation is made for future work both on the solver itself and as studies unlocked by the speed.

CHAPTER 2. BACKGROUND AND THEORY

In this chapter, the requisite background knowledge and theory for the work is described. The first section deals with background information on the neutron transport equation and the COMET solution method (primarily as described in [1]). This first section also includes a review of previous work that examined CPU-only parallelization of the method [6]. The second section gives background knowledge on GPU parallelization and the specific considerations that must be examined when writing GPU-capable programs. This section also includes a review of other neutron transport solvers (stochastic and deterministic) that have leveraged GPU architectures. The third section provides a brief interpretation of this background information that forms the basis for the following work.

2.1 Neutron Transport and the COMET method

The Boltzmann neutron transport equation (eq. 2.1.1) describes the distribution of neutrons in a steady-state system. In this equation, $\vec{r}$ indicates the position in 3-dimensional space, $\hat{\Omega}$ indicates the angle of the neutron flight, E indicates the neutron's energy, $\psi(\vec{r},\hat{\Omega},E)$ indicates the angular neutron flux, σ, σ_s, and σ_f indicate the total, scatter, and fission cross sections, k indicates the criticality eigenvalue, χ indicates the fission spectrum, and ν indicates the neutrons emitted per fission. This "steady-state transport equation" is observed to have 6 independent variables (3 spatial variables, 2 angular variables, and 1 energy variable), which makes direct solution impossible in all but the simplest cases. As such, obtaining solutions to whole-core problems involves the use of numerical methods and approximations.

$$\hat{\Omega}\cdot\nabla\psi\left(\vec{r},\hat{\Omega},E\right)+\sigma(\vec{r},E)\psi\left(\vec{r},\hat{\Omega},E\right)$$
$$=\int_0^\infty dE'\int_{4\pi}d\hat{\Omega}'\,\sigma_s\left(E',\hat{\Omega}'\to E,\hat{\Omega}\right)\psi'\left(\vec{r},\hat{\Omega}',E'\right)$$
$$+\frac{1}{k}\frac{1}{4\pi}\chi(\vec{r},E)\int_0^\infty dE'\,\nu\sigma_f(\vec{r},E')\int_{4\pi}d\hat{\Omega}'\,\psi'\left(\vec{r},\hat{\Omega}',E'\right)\tag{2.1.1}$$

For brevity, we define an operator form of the transport equation (eq. 2.1.2), where **H** is termed the transport operator, **F** the fission operator, and **B** the boundary conditions for the problem.

$$\boldsymbol{H}\psi\left(\vec{r},\hat{\Omega},E\right)=\frac{1}{k}\boldsymbol{F}\psi\left(\vec{r},\hat{\Omega},E\right),\qquad \vec{r}\in V,$$
$$\psi\left(\vec{r},\hat{\Omega},E\right)=\boldsymbol{B}\psi\left(\vec{r},\hat{\Omega},E\right),\qquad \vec{r}\in\partial V\ \&\ \hat{\Omega}\cdot\hat{\mathrm{n}}<0$$
$$\boldsymbol{H}=\hat{\Omega}\cdot\nabla+\sigma(\vec{r},E)-\int_0^\infty dE'\int_{4\pi}d\hat{\Omega}'\,\sigma_s\left(E',\hat{\Omega}'\to E,\hat{\Omega}\right)\tag{2.1.2}$$
$$\boldsymbol{F}=\frac{1}{4\pi}\chi(\vec{r},E)\int_0^\infty dE'\,\nu\sigma_f(\vec{r},E')\int_{4\pi}d\hat{\Omega}'$$

2.1.1 COMET Theory

One such numerical solution method for the neutron transport equation is the continuous-energy COMET method. In the COMET method [1], the spatial domain of the problem V is divided into a number of contiguous coarse meshes V_i. From there, the angular flux that lies on the boundary of each mesh ∂V_i is expanded in terms of an orthogonal basis set as in eq. 2.1.3, where $\psi_{is}^{\pm}$ represents the outgoing or incoming flux on surface s of the

i^{th} coarse mesh, $J_{is}^{\pm,m}$ represents the corresponding surface *flux expansion coefficient*, $W(E)$ is a positive weight function (discussed later in this section), and Γ_m is the m^{th} member of an orthogonal basis function set of *M* members.

$$\psi_{is}^{\pm}(\vec{r},\widehat{\Omega},E) \approx \sum\nolimits_{m=1\ldots M} J_{is}^{\pm,m} W(E)\Gamma_m(\vec{r},\widehat{\Omega},E) \qquad , \qquad \vec{r} \in \partial V_{is} \tag{2.1.3}$$

With the flux on the surface of each mesh expanded in terms of the basis set, we can then express the flux in the interior of each mesh in terms of a *surface-to-volume response function*, as in eq. 2.1.4.

$$\psi_i(\vec{r},\widehat{\Omega},E) = \sum\nolimits_{s,m} J_{is}^{-,m} R_{is}^{m}(\vec{r},\widehat{\Omega},E) \qquad , \qquad \vec{r} \in V_i \tag{2.1.4}$$

These surface-to-volume response functions can be interpreted as the solution to a corresponding local problem in which a surface flux in the shape of the specific response function Γ_m is imposed on surface *s*, as in eq. 2.1.5.

$$\begin{gathered} R_{is}^{m}(\vec{r},\widehat{\Omega},E;k) = \frac{1}{k}\boldsymbol{F}R_{is}^{m}(\vec{r},\widehat{\Omega},E;k) \qquad , \qquad \vec{r} \in V_i \\ R_{is}^{m}(\vec{r},\widehat{\Omega},E;k) = \begin{cases} W(E)\Gamma_m(\vec{r},\widehat{\Omega},E) \qquad , \qquad \vec{r} \in \partial V_{is},\ \widehat{\Omega}\cdot\hat{n}_{is}^{+} < 0 \\ \qquad 0 \qquad , \qquad otherwise,\ \widehat{\Omega}\cdot\hat{n}_{is}^{+} < 0 \end{cases} \end{gathered} \tag{2.1.5}$$

By examining the value of R_{is}^{m} along the surface of the mesh and expanding it in terms of the basis set in the **outgoing** direction, we arrive at the *surface-to-surface response function coefficient* $R_{is's}^{m'm}$, which represents the response on surface *s* in moment *m* due to a unit incoming flux on surface *s*' in moment *m'*. Due to the linearity of the transport and fission operators, the relationship between the outgoing and incoming flux expansion coefficients can be expressed as in eq. 2.1.6.

$$J_{is}^{+,m} = \sum_{s'} \sum_{m'} R_{is's}^{m',m}(k) J_{is'}^{-,m'} \tag{2.1.6}$$

By expressing the partial current expansion coefficients in vector form across the individual moments in the expansion basis set, eq. 2.1.6. can be expressed as a matrix-vector operation as in eq. 2.1.7, where $\boldsymbol{r}_{is's}$ represents an M x M surface-to-surface response function matrix for mesh *i* from surface *s'* to *s*, and $\boldsymbol{j}_{is}^{\pm}$ represents an M x 1 column vector of the outgoing/incoming individual partial current expansion coefficients for surface *s* of mesh *i*.

$$\boldsymbol{j}_{is}^{+} = \sum_{s'} \boldsymbol{r}_{is's}(k) \boldsymbol{j}_{is'}^{-} \tag{2.1.7}$$

These surface-to-surface response function coefficients can be used to construct a solution to the global problem by ensuring continuity between the outgoing flux expansion coefficient for a given mesh and surface and the corresponding incoming flux expansion coefficient on the neighboring mesh/surface. By concatenating the individual flux expansion vectors $\boldsymbol{j}_{is}^{\pm}$ for the problem into a global *partial current expansion coefficients* vector **J**, this continuity condition can be expressed as in eq. 2.1.8, where $\boldsymbol{R}$ represents a block-sparse matrix corresponding to the collection of the above $\boldsymbol{r}_{is's}$ for the global problem, $\boldsymbol{M}$ represents a connectivity matrix that connects the incoming and outgoing partial current expansion coefficients, and λ represents the discontinuity resulting from differences between the evaluated *k* and the true core eigenvalue *k*.

$$\boldsymbol{M}\boldsymbol{R}(k)\boldsymbol{J} = \lambda\boldsymbol{J} \tag{2.1.8}$$

In the case where the response functions for the global problem $\boldsymbol{R}$ are evaluated at the true core eigenvalue, λ is equal to unity and $\boldsymbol{J}$ represents the true inter-mesh partial current expansion coefficients for the problem.

Now, if the orthogonal basis set chosen to expand the surface flux (eq. 2.1.3.) is complete (infinite), then the solution is exact. The approximation in the method is introduced when selecting which basis set to use and at the order at which the set is truncated. For the COMET method, the typical basis set uses Legendre polynomials P_n in the x, y¸ and azimuthal (ϕ) variables and Chebyshev polynomials of the second kind U_n in the (polar) μ variable, as in eq. 2.1.9.

The dependence of the angular flux on the energy variable generally varies too sharply for a similar type of polynomial expansion method to be applicable. Instead, a weight function $W(E)$ is included in the expansion (eq. 2.1.3.) and divisions of unity (or zeroth-order B-splines) across a set of energy groups g are used to satisfy the orthogonality condition. The complete basis set is thus seen in eq. 2.1.9.

$$\Gamma_{ijklg}(x, y, \mu, \phi, E) = P_i(x)P_j(y)U_k(\mu)P_l(\phi)B_{g,0}(E) \tag{2.1.9}$$

For a continuous-energy treatment, the weight function $W(E)$ is typically chosen to be the asymptotic neutron spectrum, computed from a typical single assembly or color set calculation. It is observed that the lowest-order base case corresponding to $W(E) = 1$ describes a multi-group treatment.

To create an expression for the core criticality eigenvalue *k*, it is useful to define two new response functions $RNF_{is}^{m}(k)$ and $RAB_{is}^{m}(k)$, describing the total fission and total absorption in a mesh *i* due to a unit flux imposed on surface *s* with shape *m*. A global particle balance can then be used to create an expression for *k* based on these total fission and absorption response functions and the flux expansion moments $J_{is}^{-,m}$ as in eq. 2.1.10.

$$k = \frac{\sum_{i,s,m} J_{is}^{-,m} RNF_{is}^{m}}{\sum_{i,s,m} J_{is}^{-,m} RAB_{is}^{m} + \sum_{\partial V_{is} \in \partial V} (J_{is}^{+,0} - J_{is}^{-,0})} \tag{2.1.10}$$

It is useful to condense the summation across the *m* index (as in going from eqs. 2.1.6 to 2.1.7 above). By doing so, we express a vector inner product between the surface-wise partial current moments vector $\boldsymbol{j}_{is}^{-}$ and a total absorption or a total fission response vector $\boldsymbol{rab}_{is}$ or $\boldsymbol{rnf}_{is}$ representing the total absorption or fission in the mesh due to an incident flux $\boldsymbol{j}_{is}^{-}$. This condensation results in a new surface-wise particle balance equation, as in eq. 2.1.11.

$$k = \frac{\sum_{i,s} \boldsymbol{rnf}_{is}{}^{T} \boldsymbol{j}_{is}^{-}}{\sum_{i,s} \boldsymbol{rab}_{is}{}^{T} \boldsymbol{j}_{is}^{-} + \sum_{\partial V_{is} \in \partial V} (J_{is}^{+,0} - J_{is}^{-,0})} \tag{2.1.11}$$

2.1.2 COMET Numerical Steps

When solving a reactor problem, COMET proceeds as a two-phase process. In the first phase, Monte Carlo transport is used to generate the requisite response functions by solving the local problems described by eq. 2.1.5. This phase is not the subject of the work described in this document, and thus a description of the process is omitted. In principle, the method used to generate the response functions is not required to even be Monte Carlo transport and could instead be any solver capable of solving the local problems described by eq. 2.1.5. For more information on the response function generation phase, interested readers are directed to [1].

The second phase of the COMET method is a deterministic transport sweep to determine the true core criticality eigenvalue k and partial currents expansion eigenvector $\boldsymbol{j}$ as described in Section 2.1.1. This is a nested iteration process consisting of inner iterations on the eigenvector $\boldsymbol{j}$ and outer iterations on the criticality k. The inner iterations solve the generalized eigenproblem described by eq. 2.1.8. The outer iterations update the guess for the core criticality k using the particle balance described in eq. 2.1.11. With this, the deterministic transport sweep algorithm proceeds (from [1]) as:

1. Guess the initial eigenvalue k_0
2. Perform inner iterations for the specific eigenvalue computed in the previous step.
    a. Normalize the outgoing partial current moments.
    b. Use the external boundary condition or the internal interface condition to update the incoming partial current moments.
    c. Use response function coefficients to compute the outgoing partial current moments.
    d. Repeat (a.) – (c.) until the partial current moments are converged.
3. Use the global particle balance to update the eigenvalue guess k.
4. Repeat (2.) and (3.) until the eigenvalue is converged.

The current implementation of this process is written in Fortran. It stores the response function matrices in Common Data Format (CDF) [7], a self-describing portable binary data format maintained by the Space Physics Data Facility at Goddard Space Flight Center. In practice, the power method is used to solve the inner iteration eigenproblem, with a Chebyshev polynomial filtering acceleration method applied. It leverages Intel MKL [8] to perform the individual matrix multiplies expressed in eq. 2.1.7. The production build is serial, with no parallel computation capabilities (shared- or distributed-memory) for the deterministic transport phase of calculation. The results of the calculations are printed in output text files and parsed with a variety of post-processing tools.

The order in which the code performs the individual small matrix multiplications (called the "sweep order") can have a strong effect on the runtime of the program. For the current serial COMET, the sweep order used in practice groups together all multiplications by coarse mesh fill type and surface, such that consecutive multiplies are likely to use the same underlying response function matrix $\boldsymbol{r}_{is's}$. In this way, it is more likely that the matrix will already reside in cache, and as such can be read and used in multiplication faster than if it needed to be fetched from memory. In principle, however, since the power iteration process only relies on the previous guess for the eigenvector, these multiplications can be performed in any arbitrary order, with any arbitrary degree of concurrency. In this way, the COMET method is ripe for parallelization.

2.1.3 COMET Parallelization Study

A previous Ph.D. dissertation examined the possibility of extending the COMET method to distributed-memory (CPU-only) parallel systems [6]. In this work, a proxy app called COMET-MPI was developed as a fork of the (at the time) production build of the main COMET app. Written in Fortran, it leverages the Message Passing Interface (MPI) to distribute work to many independent processes, either residing on one or many physical CPUs. As a distributed-memory parallel program, each process maintains its own independent memory space and communicates with other processes by explicitly sending and receiving data.

The COMET-MPI study used a geometric decomposition to partition work between the multiple processors, as seen in Figure 1. In such a scheme, the problem is divided such that each processor works on all calculations for a group of meshes that are contiguous in geometric space. In this way, processors need only exchange information about the borders between its own domain and that of the other processors. During each inner iteration, each processor performs the matrix multiplies necessary to update its own domain, then exchanges this new information with its neighbors.

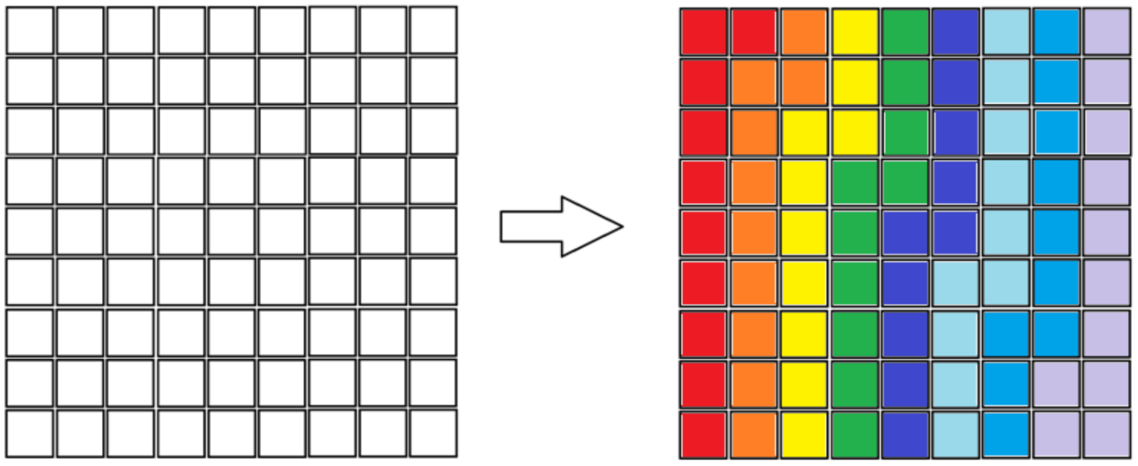


Figure 1: Geometric Domain Composition (Figure from [2])

Testing with the proxy app indicated performance that was promising. The code was demonstrated on up to 125 MPI processors on two PWR problems, one with gadolinium and one with MOX. The results can be seen in Figure 2. It is observed that a maximum speedup of 50x was achieved with 100 processors on the PWR/MOX problem. Efficiency was problem-dependent and even varied depending on the specific run, but hovered between 20% and 60% for the investigated range. It is noted that these runs were performed on a shared cluster, and as such the absolute metrics could be expected to vary from run to run. However, the general trend was that of marginally decreasing efficiency with increased numbers of processors.

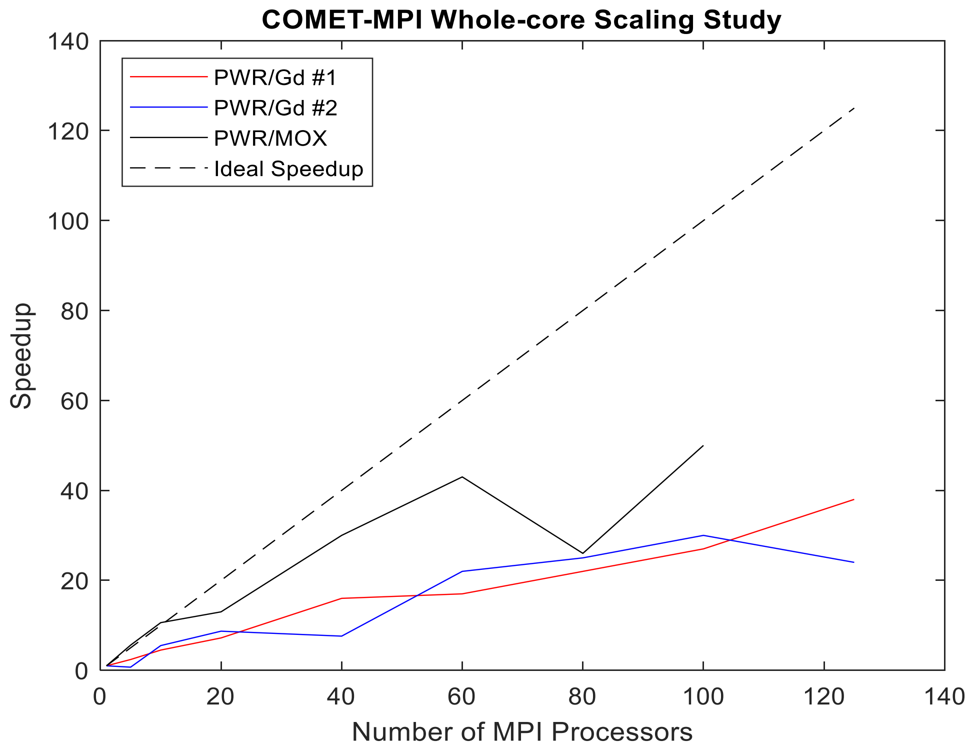


Figure 2: COMET-MPI Scaling Study Results (Reconstructed from data in [2])

The study identified a hybrid approach in which distributed-memory and shared-memory parallelism are combined as an area of possible future work. This was partially in response to a case with higher-order function basis sets in which the distributed-memory-only solver would run out of memory due to the fact that each process needs its own memory space, even when it resides on the same physical processor. The study implemented an initial hybrid implementation with OpenMP + MPI, but the results remained inconclusive, with the code showing either improved or worsened performance.

2.2 General-Purpose GPU Computing

General-purpose GPU (GPGPU) computing is a field in which Graphics Processing Units (GPUs) are applied to non-graphics workloads. This practice has been applied broadly to many different areas of scientific computing, including neutron transport.

2.2.1 GPU Programming Considerations

Due to the differences in design between GPUs and CPUs, writing programs that run on a GPU can involve restructuring large parts of algorithms and implementations. However, depending on the specific workload, this strategy can result in exceptional performance increases due to the strengths of GPUs.

To understand the strengths of GPUs and thus the goals of GPGPU computing, it first of use to establish the differences between a GPU and a CPU. The primary purpose of a CPU is to perform an arbitrary string of operations as quickly as possible. Since the CPU must be general purpose, it needs to be able to quickly handle things like conditional branches that require dedicated circuitry to handle quickly (Figure 3). As a result, CPUs can generally only perform operations on 10s of threads at a time, but each thread can operate independently and branch on its own with minimal performance impact.

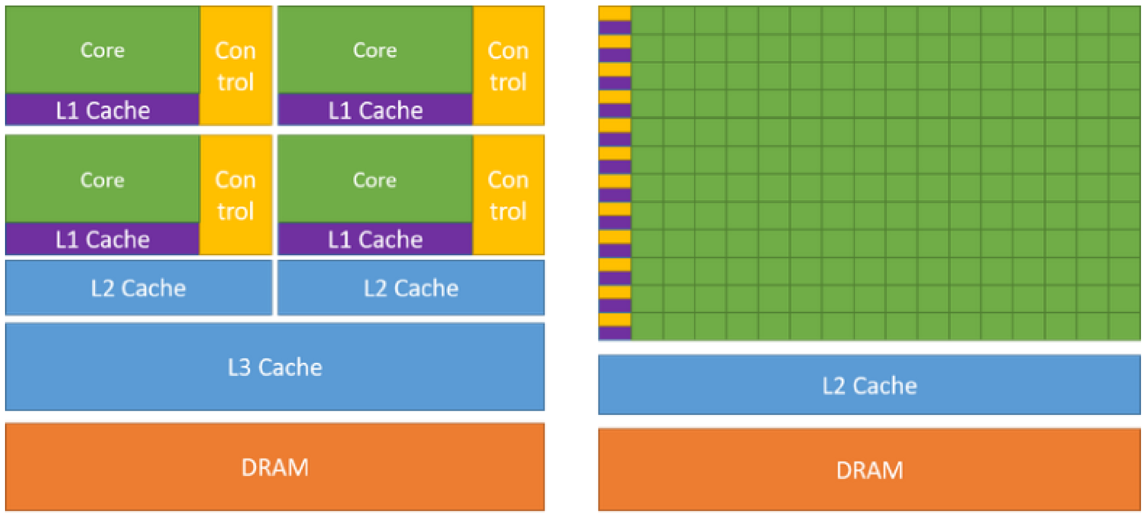


Figure 3: CPU vs. GPU Architectures (Figure from [4])

By contrast, GPUs (often simply called devices in the context of GPGPU computing) excel at single-instruction-multiple-data (SIMD) computation. In such computation, the same set of instructions is performed on multiple different data streams. This type of computation comprises the bulk of graphics workloads, in which points in 2D or 3D space have the same geometric transforms imposed upon them. To accommodate this type of SIMD computation, the silicon at the core of a GPU has a much higher proportion of transistors devoted to arithmetic and logic, rather than control circuits and cache (Figure 3). As a result, any individual GPU thread is comparatively weaker than a CPU thread. However, a GPU can operate many more threads at once than a CPU (1,000s to 10,000s).

As a result of these device design features, there are several key differences between the way programs execute on a GPU vs. on a CPU. The most immediate difference between parallel programs on a CPU and on a GPU is that, on GPUs, threads are broken into groups that must operate in lockstep. For NVIDIA GPUs, this grouping is called a “warp” and contains 32 threads on the Tesla and Ampere architectures [2], [9]. For the AMD CDNA architecture, this grouping is called a “wavefront” and contains 64 “work-items” [10]. For the remainder of this document, unless otherwise specified, the NVIDIA terminology will be used for consistency, since the devices/library used in this work come from NVIDIA.

The warp distinction is significant because all threads in a warp share only one actively executing instruction per cycle. If for some reason (e.g. conditional branching) one thread jumps to a different instruction than the remaining threads in a warp, that thread will sit idle while the others execute until its instruction is reached. For branch-heavy operations, this can be a significant performance penalty. If 16 threads branch to one instruction while the other 16 threads branch to a different instruction, the effective rate at which instructions are being executed is halved, as those threads are forced to sit idle. As such, it is imperative for maximum performance that the number of threads executing the same instruction (to the warp level) is maximized. The ratio between the number of threads in a streaming multiprocessor (SM) that *are* executing to the number that *could be* executing is called **occupancy**. Occupancy is affected by thread divergent execution, but can also be limited by the number of registers that a thread needs relative to the available resources on the SM, etc.

The second difference between effective CPU and GPU programs is in optimal memory access patterns. For CPU programs, memory is often laid out to maximize the use of the cache. As such, it is generally structured such that consecutive **instructions** access consecutive locations in memory. On the first access, the entire page is loaded into cache, so it is likely subsequent instructions will access memory locations that have already been loaded into cache. Since accessing cache is faster than accessing memory, performance is generally improved. An example of this is a matrix-vector multiplication in which the matrix is laid out in **row-major** order, such that a thread reads consecutive locations in memory as it accumulates across a matrix row.

Optimal GPU memory layout requires slightly different logic, due in part to the warp-level threading considerations described above. If a warp is fully executing in lockstep, each of the 32 threads will reach a load instruction at the same time. In general, device memory is only capable of accommodating one request at a time. As such, this request would generally be serialized, and each thread would be served in turn. However, there are some specific access patterns for which these accesses "coalesce" and are condensed into fewer instructions that are served more quickly. The first is a "broadcast" type of read in which threads all read from the same memory location. The second is a read in which consecutive threads read from consecutive locations in memory. If the memory accesses match these patterns (with some paging and alignment concerns, per [11]), these reads will be served at the same time. As such, device memory is generally structured such that consecutive **threads** access consecutive locations in memory. An example of this is a matrix-vector multiplication in which the matrix is laid out in **column-major** order. In such an operation, each thread is responsible for one row, and consecutive threads read consecutive locations in memory from one column at a time.

One more higher-level programming consideration for developing GPU programs derives from the way in which GPUs interact with the system. GPUs are devices that are attached to a host node, typically over the PCI bus. As a result, transferring data between the host and the device is a relatively slow process, especially compared to either the host or device reading from its own memory. As such, it is imperative to transfer as much data as possible to the device before beginning computation, or to hide the transfer time by overlapping transfer and computation via a pipelining scheme.

2.2.2 GPGPU Programming Interfaces

GPU functions are called from a program running on the CPU. To interface with the GPU from such a program, it is required to use one of many APIs, libraries, or interfaces available for most programming languages. These interfaces range in origin from GPU manufacturers to open standards. They also range in granularity of control, with some comprising OpenMP-style compiler pragmas, with others providing explicit control over device memory allocation and movement. A brief overview of a set of the most common GPU interfaces is provided in this section.

2.2.2.1 CUDA

One of the most ubiquitous GPU computing libraries is CUDA, a first-party interface from NVIDIA [11]. CUDA allows the developer to write custom “kernels” that run on the device and provides a C++ API to interact with the devices, allocate and transfer memory, and launch kernels. CUDA programs are compiled using *nvcc*, a custom compiler driver that generates appropriate device code. CUDA defines a programming model that allows for concurrent execution of multiple threads and kernels.

The execution hierarchy in CUDA can be seen in Figure 4. The most atomistic unit of the CUDA programming model is the **thread**. When a kernel launches, each assigned thread runs a copy of the same kernel. Threads are first bundled into a **block**. Currently, blocks can consist of up to 1024 threads. The threads in a block are guaranteed to be assigned to the same compute core and thus can share certain types of on-processor memory/cache. Kernels are launched in a **grid** of such blocks. These blocks are not guaranteed to execute in any particular order or with any guarantee of concurrency or synchronicity. As such, each block needs to represent work that can be processed entirely independently of other blocks in the grid.

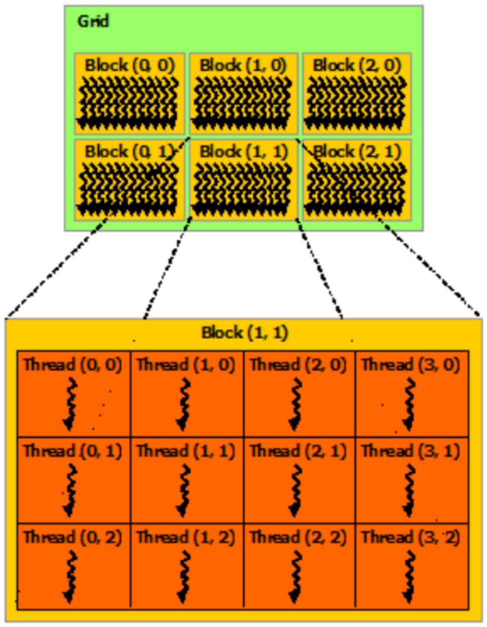


Figure 4: CUDA Execution Hierarchy (Figure from [7])

In the CUDA model, the host and the device maintain separate memory spaces. Memory must be explicitly allocated on the device, and data transfer between the host and the device must be explicitly requested. This generally increases the complexity of CUDA code, as data structures must generally be created on the host, have space allocated for them on the device, then instantiated manually with a memory copy call from host memory to device memory. To manage this complexity, newer CUDA devices also have a “unified” memory space, in which data can be allocated and accessed from both the host and the device at the same address [12]. In the case in which the memory has not been transferred to the device, the page will be fetched from the host memory automatically. However, to maximize performance, unified memory still needs to be pre-fetched or copied directly, which adds some (perhaps lesser) degree of complexity.

The CUDA suite also provides several higher-level libraries and programs to aid in the development of CUDA programs. These include profilers to analyze the performance of CUDA kernels and programs, a linear algebra library (cuBLAS), and others.

2.2.2.2 OpenACC

The OpenACC standard is a higher-level approach to enabling GPGPU computation [13]. It takes the form of a set of compiler directives to directly expose parallelism in the code and allow the compiler to build parallel versions for a variety of different device backends. The primary goal of OpenACC is to prioritize portability across different compilers and device architectures. A limitation of this approach is that it often results in code that is marginally less efficient than a full hand-tuned implementation, but portability is vastly improved as opposed to e.g., a CUDA implementation, as the same code could be compiled and run with a different compiler/architecture stack.

Programs that leverage OpenACC are marked up with pragmas that direct the compiler to data and operations that can be parallelized. The semantics of data allocation, transfer, specific calculation parameters, and results transfer are (unless necessary) the responsibility of the compiler, rather than the developer. In this way, the details and best practices associated with different device architectures need not be handled by the developer, and the compiler can generate correct code with good performance.

2.2.2.3 Kokkos

Kokkos is a C++ library developed at Sandia National Laboratory [14]. It provides a programming model (and layer of abstraction) above vendor-specific GPU backends that aims to provide portability to different architectures while maintaining a degree of fine-grained control over device memory and operations. The goal of this approach is to allow developers to write one set of source code that can be effectively deployed across parallel architectures, including multi-core and CPU-GPU architectures.

Kokkos accomplishes this portability by allowing the developer to write algorithms that use templated Kokkos-specific data types and parallel execution patterns (including parallel_for, reduce, etc.). Beneath this layer of abstraction is one of several different "Execution Spaces" and "Memory Spaces" available to the developer. These spaces serve to translate the intermediate Kokkos representation to the specific representation that makes the most sense for the desired architecture. In this way, optimizations such as the row- vs. column- major matrix example described above are abstracted away from the user. Along with the library, the Kokkos ecosystem also includes extra features including BLAS functionality, profiling tools, and debugging tools.

2.2.2.4 <u>OpenMP Accelerator Support</u>

The OpenMP standard is a standard that is most often used to enable shared-memory multi-threading on a CPU via a set of compiler directives. Beginning with the OpenMP 4.0 standard in 2013, the standard also provides a set of pragmas that enable the offloading of computation to an attached device [15]. As another pragma-based interface, the pros and cons of working with OpenMP accelerator support are similar to those of OpenACC. The primary benefit of such an interface is an increase in portability, as the compiler can translate these instructions for any arbitrary target architecture. The cost of this portability is not being able to control the more fine-grained aspects of memory movement and execution.

2.2.2.5 <u>HIP</u>

The Heterogeneous-compute Interface for Portability (HIP) is an API that aims to provide developers a level of control over execution that is akin to CUDA but does not limit the user to NVIDIA GPUs [16]. The HIP API includes many functions that represent a direct counterpart to a CUDA function (for example, hipMalloc/hipMemcpy vs. cudaMalloc/cudaMemcpy). The similarities are so strong that the HIP ecosystem offers a set of tools called hipify which are able to scan CUDA source code and either offer a direct HIP translation or identify the calls which are not translatable.

2.2.3 GPGPU As Applied to Neutron Transport

GPGPU computing techniques have been applied to a wide range of scientific computing workloads, and neutron transport is no exception. There have been examples of deterministic and stochastic methods that have been applied to GPU architectures with varying degrees of success. This section provides a brief overview of a few of the most prominent examples, including case studies of both stochastic and deterministic methods.

2.2.3.1 WARP

The WARP (which can stand for “Weaving All the Random Particles”) code is a neutron transport solver developed from the ground up for computation on GPU architectures [17]. The algorithm used for WARP is based off research performed in 1984 by Brown and Martin [18] which established a unique “event-based” Monte Carlo algorithm for SIMD vector computers.

In a traditional Monte Carlo transport algorithm (referred to for comparison as “history-based” algorithm), threads proceed by calculating the entirety of a particle’s lifetime, then proceeding to a new particle. This type of algorithm is not well-suited for computation on a SIMD vector processor (or, in modern cases, GPUs) because, by the very definition of the Monte Carlo method, different particle lifetimes will play out differently. As such, consecutive particle histories can (and indeed are expected to) vary in terms of number of collisions, number of surfaces met, etc. As such, a new “event-based” algorithm was proposed in which particles are grouped into vectors based on the operation which is required by the current point in their history. For example, all particles currently undergoing a surface crossing operation are placed in a surface crossing buffer, and so on for other operations including scattering, etc. This exposes a much more SIMD calculation pattern, as most events of the same type require similar operations, even if being performed on different histories.

WARP was programmed in CUDA C++ and leveraged the NVIDIA first-party OptiX library to handle geometric specification and ray-tracing operations. Since the code was designed from the ground up for GPU computation, the work does not provide a direct performance comparison to an equivalent CPU execution, since no such direct comparison exists. A looser comparison can be found in a companion paper [19], wherein WARP was benchmarked against the mature CPU-only Monte Carlo transport codes Serpent and MCNP. In this study, it was found that one GPU using the warp code achieved performance equivalent to 0.84x-7.61x the performance of a modern CPU node.

2.2.3.2 Shift

Shift is a massively parallel Monte Carlo radiation transport package from Oak Ridge National Laboratory [20]. A 2019 study using both a history-based and an event-based approach to extend the code to include operation on GPUs [21]. The GPU calculation logic for this study was written in CUDA C++. The study found that the event-based algorithm in which kernels only process one type of event at a time (either transport or collision) vastly outperforms the history-based algorithm in which threads process an entire particle history. The study found that this was due to increased occupancy (as described in Section 2.2.1 above) resulting from decreased divergent execution, which is likely to occur in a history-based approach as, e.g., some particles die off before others.

The results produced by the study were very promising. For the event-based approach (with an additional fuel-partitioning optimization), it was determined that an NVIDIA V100 GPU provided the performance of about 150 individual CPU cores for the examined benchmark problems.

2.2.3.3 <u>OpenMOC</u>

OpenMOC is an open-source method of characteristics neutral particle transport code developed at the Massachusetts Institute of Technology [22]. The code uses the method of characteristics to solve 2D problems, with solvers for multi-core CPUs as well as GPUs. The GPU solver is written in CUDA C++. The initial documentation surrounding the release of the code indicated an acceleration of 50x with respect to a single-threaded CPU solver when using a Tesla C2070 GPU.

2.2.3.4 <u>nTRACER</u>

The nTRACER code package is a direct whole-core neutron transport solver developed by Seoul National University [23]. nTRACER employs a hybrid method in which 2D problems are solved to high detail using the method of characteristics, and the solutions of these 2D solves are fed into a 3D coarse mesh finite difference (CMFD) solver. In a study which examined expanding the nTRACER neutronic solver to GPU architectures, it was determined that the GPU-accelerated version of the code would see a speedup of between 8.3x to 11x relative to a CPU-only version [24].

2.3 Interpretation

In this section, a description of the COMET method was provided, and results from a previous CPU-only COMET parallelization study were presented. A set of basic considerations for GPU programming were presented, along with an overview of a selection of interfaces and libraries available to write GPU programs. Finally, overviews for a set of GPU acceleration studies for other neutron transport codes (both stochastic and deterministic) were presented.

It is clear from this review that there is a strong case to be made for GPU acceleration of the COMET method. The matrix multiplies required to perform one inner iteration can be performed entirely independently, placing the method in a class of problems which can be called “embarrassingly parallel”. This parallelism was exploited in the COMET-MPI project in a distributed-memory manner, and shared-memory parallelism was suggested for further study.

Furthermore, the specific considerations necessary for efficient GPU programming can be handled by the method. The response function libraries can fit entirely in device memory, which restricts communication of large amounts of data over the PCI bus to primarily the beginning of each outer iteration. The bulk of the runtime for COMET is associated with the small dense matrix multiplies (eq. 2.1.7), operations which can be speedily performed with the floating-point compute units on a GPU. The matrices can be laid out in column-major order on the device, ensuring that memory accesses will coalesce to a high degree, and thus memory performance (which usually bottlenecks matrix multiplies as low arithmetic intensity operations) can be made high.

CHAPTER 3. COMET-CPP DEVELOPMENT AND METHODOLOGY

In this section, the development of a GPU-accelerated version of the COMET deterministic solver (including multi-GPU support) will be described. Although it would be technically possible to modify the existing COMET Fortran codebase to add GPU support, attempting to do so was not likely to result in an efficient implementation due to the order in which calculations are currently performed. Due to this ordering, memory layout concerns, and the lack of relative maturity for GPU interfaces in Fortran, the decision was made to develop an all-new implementation of the transport sweep portion of the COMET method in C++. The benefit of the ability to re-order calculations, restructure the way in which memory is laid out, and add a degree of modularity/separation between the philosophical representation of the problem and the raw matrices and vectors involved in calculation was determined to outweigh the cost of developing this all-new implementation. For the remainder of this document, the new implementation will be called "COMET-cpp" or the "new implementation", and the original Fortran solver will be called the "serial solver" or the "old implementation".

It was decided that the GPU calculation capability for the new solver would be implemented in CUDA. There are several reasons for which CUDA was selected as the interface for use in this study:

- As described in Section 2.2.3, CUDA has already been used to accelerate several different neutron transport methods.

- The machines on which the development of COMET-cpp was to take place all contained NVIDIA GPUs, which prevented the need for a more portable interface. The solver was developed to allow for clean connection points between the calculation front-end and back-end such that, if more portability is desired, a new calculation back-end could be written in a more portable interface. One possibility is porting the CUDA directly using the hipify toolchain discussed in Section 2.2.2.5.
- The CUDA ecosystem itself includes the possibility for many different implementations for the GPU operations, including the use of the first-party linear algebra library cuBLAS and the use of CUDA unified memory. These were both investigated before the final full implementation was written.
- As described in Section 2.2.2.1, the CUDA interface allows for very fine-grained control of memory allocation, movement, and GPU operation. Since the solver was already being rewritten to pay careful attention to memory layout, the opportunity to write this logic from the ground up was already present.

The first section of this chapter describes the new implementation of the CPU-only solver, including design elements to allow for switching on the calculation back-end. A CPU-only BLAS back-end is also described. The second section of the chapter describes the design of the GPU-accelerated algorithm, including details about memory movement and calculation kernel structure. This section also includes details about early attempts to leverage CUDA unified memory and the cuBLAS library, before settling on the final implementation that uses explicit memory management and custom calculation kernels.

The third section of this chapter describes a set of newly developed post-processing and visualization tools to interpret the text-based output from the new solver and plot the results.

3.1 Host COMET-cpp Solver Development

As described in the introduction, it was decided that a new host solver should be created as the development target for GPU calculation capabilities. To maintain a reasonable scope for the project while maximizing the utility of the solver (and allowing for the possibility for continued development), a set of goals for the solver were set forth, with some items explicitly excluded from scope:

- The solver should have total backward compatibility with the current libraries and input files read by the serial solver. This clearly facilitates the continued development and use of the solver, but it also aids in restricting scope, as testing can be performed with the library of benchmark problems that have already been solved with COMET.
- The solver should have the capability to perform regular forward transport calculations, with or without Chebyshev polynomial acceleration. The other functions of the deterministic solver (e.g. adjoint calculations, uncertainty calculations) rely on similar calculations, so regular forward calculation with Chebyshev acceleration is sufficient to demonstrate the techniques that would be involved in implementation of the other features.
- Because COMET post-processing tools are currently written on a per-problem basis, backwards compatibility with the output file format was decided not to

be a goal of the new solver. Instead, a new format for output files was created, and a unified post-processing suite of tools was developed (per Section 3.3).

- The new solver should have a modern build system generator and should be configurable at compile-time and run-time to include support for CPU-only (with configurable BLAS backend) or GPU-accelerated calculation on supported systems.

The result of this refactor of the serial code was a new implementation of the deterministic sweep calculation which was more suitable for parallel computation, both CPU-only as described in Section 3.1.2 and GPU-accelerated as in Section 3.2.

3.1.1 Solver Front-end

The purpose of the COMET-cpp solver front end is to construct an interpretation of the reactor core represented by the text-based COMET input file for the run and interface with the CDF database files that contain the underlying $\boldsymbol{r}_{is's}$ matrices used in calculation. The front-end first constructs a series of objects that contain information about the problem meshes (in-core location, material composition, etc.). From there, a set of information is assembled into packages that get passed to the back-end:

- A mapping is made for each individual mesh i, surface s, and moment m entry into one global eigenvector $\boldsymbol{J}$.
- A list of small dense matrix operations is constructed (eq. 2.1.7) that, in total, describe the surface-to-surface current transfer for the global problem. These operation objects containing the following information:

    - Offset into the global eigenvector of the "from" surface, giving the location of the first element of $\boldsymbol{j}_{is'}^{-}$,
    - Offset into the global eigenvector of the "to" surface, giving the location of the first element of $\boldsymbol{j}_{is}^{+}$,
    - An object that describes the surface-to-surface response function matrix $\boldsymbol{r}_{is's}$ to use in the multiplication.
- A list of boundary condition operations is constructed to enforce periodic or reflective boundary conditions for the global problem, if necessary.
- A list of total absorption operations is constructed that, in total, comprise the total absorption for the problem (corresponding to the $\boldsymbol{rab}$ term in eq. 2.1.11). These operation objects contain the following information:
    - Offset into the global eigenvector of the "from" surface, giving the location of the first element of $\boldsymbol{j}_{is}^{-}$,
    - An object that describes the total absorption response vector $\boldsymbol{rab}_{is}$.
- A list of total fission operations is constructed that, in total, encompass the total fission for the problem (corresponding to the $\boldsymbol{rnf}$ term in eq. 2.1.11). These operation objects contain much the same information as the absorption operations above, except the objects describe the total fission response vector $\boldsymbol{rnf}_{is}$ instead of the absorption vector.
- A list of the ordinate combinations i and s for which the surface lies on the boundary of the global problem (that is, $\partial V_{is} \in \partial V$). This is necessary to construct the leakage term in eq. 2.1.11.

The information passed to the back-end is designed to completely describe the problem and the calculations necessary for the inner and outer iteration process while avoiding information that may vary between different calculation back ends. A graphical depiction of the exchange of information between the front-end and any particular back-end can be seen in Figure 5 below. In this figure, the logic contained in the blue front-end box is constant regardless of the particular back-end used, and the logic contained in the green back-end box can vary depending on the target architecture, library, etc.

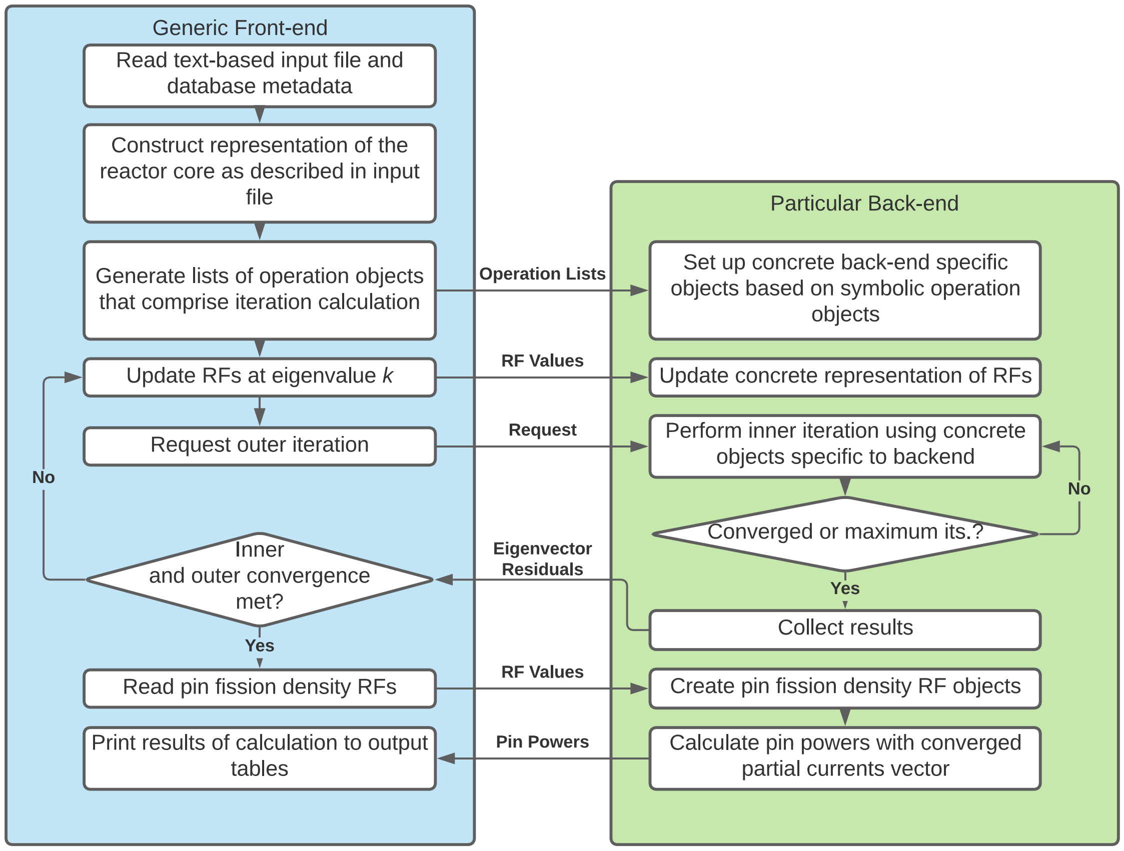

Generic Front-end
Read text-based input file and database metadata
Construct representation of the reactor core as described in input file
Generate lists of operation objects that comprise iteration calculation
Update RFs at eigenvalue k
Request outer iteration
No
Inner and outer convergence met?
Yes
Read pin fission density RFs
Print results of calculation to output tables
Operation Lists
RF Values
Request
Eigenvector Residuals
RF Values
Pin Powers
Particular Back-end
Set up concrete back-end specific objects based on symbolic operation objects
Update concrete representation of RFs
Perform inner iteration using concrete objects specific to backend
No
Converged or maximum its.?
Yes
Collect results
Create pin fission density RF objects
Calculate pin powers with converged partial currents vector

Figure 5: Separation and Exchange Between Front- and Back-ends

The COMET-cpp front-end has two external dependencies. The gflags library [25] is used for command-line argument parsing. The CDF C library [7] is used to interface with the CDF response function database files. Rather than use an external library to parse the text-based input files used by COMET, a new custom parser was developed.

3.1.2 CPU-only Generic BLAS Back-end

A CPU-only back-end for COMET-cpp was also developed in this work. This back-end was developed for several reasons:

- If COMET-cpp were to continue as the main development branch of the project, a CPU-only solver would be required to enable calculation on systems for which GPUs are not present.
- By expressing the problem in the symbolic objects described in Section 3.1.1 above, an opportunity for shared-memory parallelism is immediately available. It is thus of interest to gauge the performance of a shared-memory parallel implementation of the method (as recommended by the COMET-MPI study described in Section 2.1.3 above).
- Having a CPU-only solver that matches the program flow of the GPU solver is a useful tool for debugging and correctness checks.

Although it is not the main subject of this study, the details of the CPU-only shared-memory parallel back-end are useful for understanding the techniques that will be used in the GPU back-end logic. As such, to provide a basis for full illustration of the similarities and differences between the two, the CPU-only solver is documented fully in this section.

The first notable distinguishing feature of a particular back-end is the object that it uses to represent a particular $\boldsymbol{r}_{is's}$ matrix in memory. For the CPU-only backend, this object is relatively straight-forward. It uses a C++ standard library vector of single-precision (usually, although a double-precision version is explored in Chapter 5) floating-point values. The matrix is stored in row-major order for the cache-optimization rationale described in Section 2.2.1 above. The pointer to this matrix which is ultimately fed into the Basic Linear Algebra Subprograms (BLAS) function calls is obtained by calling the data() function on this vector.

As described in Sections 2.1.2 and 2.1.3, a key feature of the COMET method which makes it an excellent candidate for parallel implementations is that the individual small matrix multiplications that comprise an inner iteration can be performed in any arbitrary order, called the “sweep order”. In the current serial COMET implementation, this sweep order groups together all operations that use the same underlying response function matrix (that is, operations on meshes that have the same material composition, from the same surface number to the same surface number) [1], [6]. This is done to optimize performance by maximizing re-use of matrices between contiguous operations, thereby maximizing cache use.

In Section 3.1.1 and Figure 5 above, it is described how the individual operations are communicated from the front-end to the back-end in a 1-dimensional list of operation objects. In this context, any particular sweep order can be thought of as one permutation of this master list of operations. To recreate the sweep order explicitly enforced by the nature of the order of the serial solver operations, the COMET-cpp CPU-only backend begins setup by sorting the master list of operations according to the following variables in order of importance:

- Material number
- "From" surface number
- "To" surface number
- "From" offset into global eigenvector
- "To" offset into global eigenvector

This sorting operation results in a list ordering that maximizes cache use in the same manner as the serial solver, grouping together operations that use the same underlying matrix.

It is noted from eq. 2.1.7 that each outgoing partial current expansion coefficient for surface s of mesh i is a summation over the results of several matrix multiplies over the incoming surfaces s' for the mesh. This means that any individual $\boldsymbol{j}_{is}^{+}$ for the problem is obtained by summing the results of *S* matrix multiplies, where *S* is the number of surfaces per coarse mesh for the specific problem geometry (4 for 2-D Cartesian, 6 for 3-D Cartesian, 8 for 3-D hexagonal). On the surface, this poses a problem for the independence of the operations, as the current due to one surface must be added to the current due to the other incoming surfaces. COMET-cpp gets around this problem by setting up a set of *S* unique "intermediate results" buffers to which the results of the multiplies can be written independently. Once all the multiplies have been performed, these buffers can be summed and written to a final combined results buffer.

A graphical depiction of this process for a 2-dimensional Cartesian problem with 2 meshes can be seen below. A depiction of the problem geometry and the resulting global partial currents vector $\boldsymbol{J}$ can be seen in Figure 6. It is noted that, due to the continuity condition, there is no explicit entry in the vector for the entries $\boldsymbol{j}_{0,1}^{+}$ and $\boldsymbol{j}_{1,0}^{+}$. The partial currents moments that correspond to these surfaces are already represented by $\boldsymbol{j}_{1,0}^{-}$ and $\boldsymbol{j}_{0,1}^{-}$ respectively, and as such there is no need for them. In general, *outgoing* partial current moments are only explicitly given their own entry in the global vector if they lie on the global boundary of the problem, and as such are not represented by any incoming partial current moments.

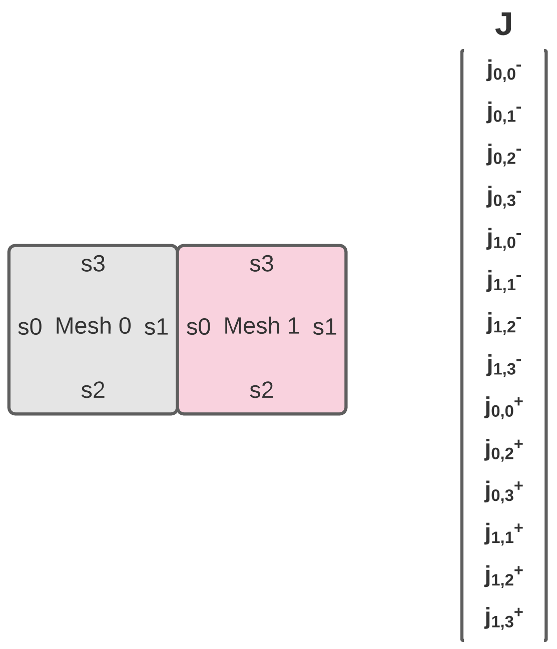


Figure 6: Example Problem Geometry and Vector

The individual small dense matrix multiplies for the problem can be seen in Figure 7. The full set of small matrix multiplies for the problem is of size 32 (4 incoming surfaces per mesh * 4 outgoing surfaces per mesh * 2 meshes), 10 of which are depicted in the figure. It is observed that, by having the results of each multiplication written to a different intermediate buffer based on the inward-facing surface number, no two operations write to the same location in memory. As such, there is no need for the operations to be performed with any particular order, synchronicity, or grouping. Thus, the need for atomic operations is not present, and the ordering of these operations can be changed arbitrarily.

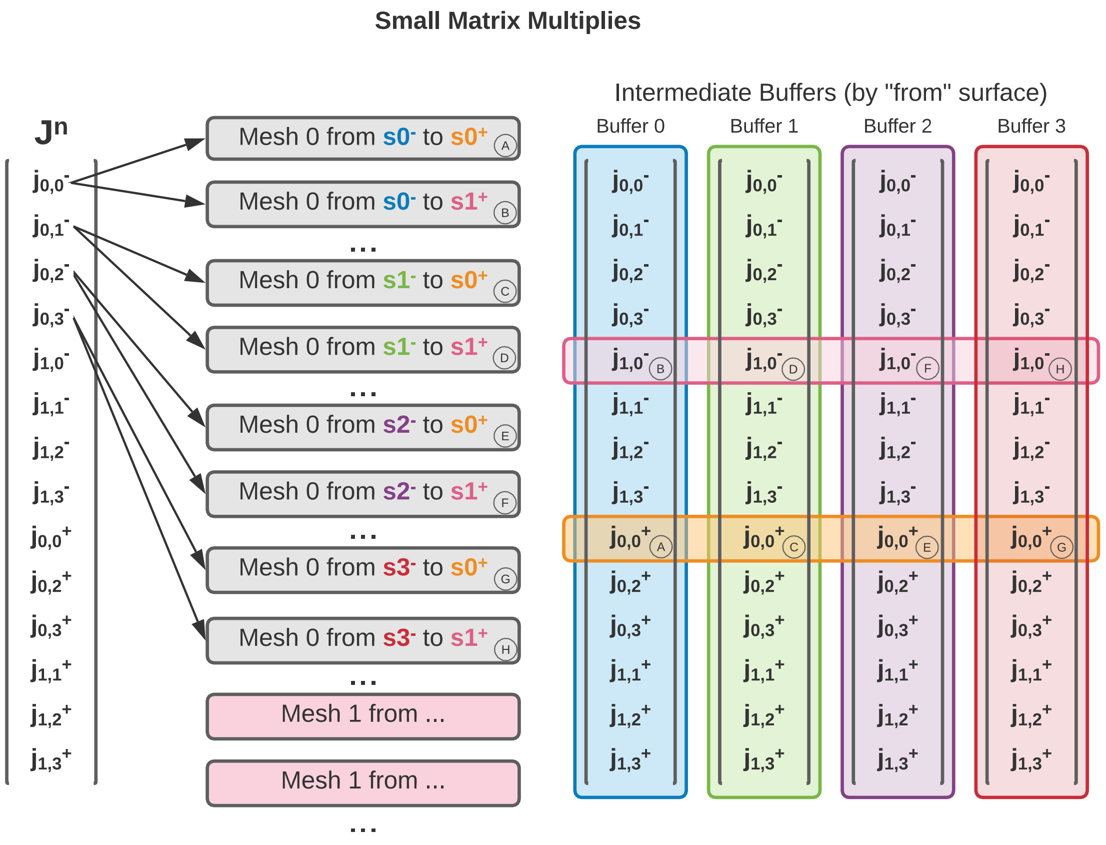


Figure 7: Small Dense Matrix Multiplications for Inner Iteration

To obtain the full results of these multiplications, the intermediate buffers must be reduced, as in Figure 8. In the CPU-only backend, the results of this reduction are stored in the primary buffer as the most up-to-date guess for the eigenvector.

As with the old serial solver, the Intel Math Kernel Library (MKL) [8] is leveraged to perform the matrix multiplies. However, to aid in the portability of the code to arbitrary systems, compile-time switches were also implemented to allow for building of the code with an open-source alternative, the OpenBLAS [26] library.

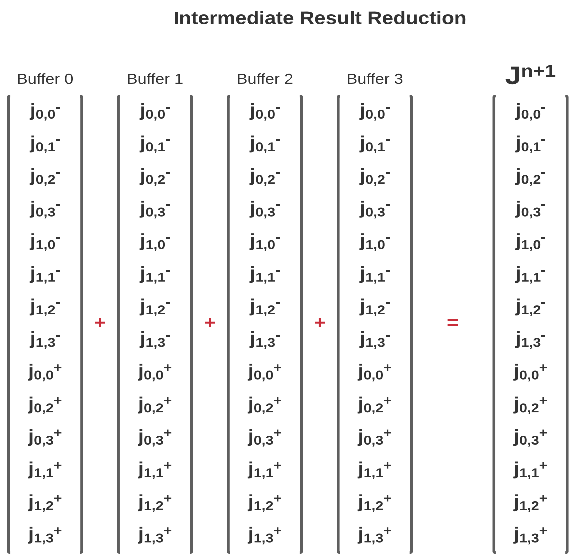


Figure 8: Intermediate Results Reduction

To exploit the parallelism exposed by the above-described untangling of front- and back-ends, a simple OpenMP loop over the (sorted) list of matrix multiplies is used, giving each thread a contiguous chunk of the global list. Because CPU parallelism was not the primary focus of this work, no further effort was made to optimize this strategy. Regardless, the examination of program execution seen in Chapter 4 clearly demonstrates the strength of the overall approach used by COMET-cpp to separate these operations.

3.2 GPU Back-end Development

The primary focus of this work is the development of a GPU-accelerated version of the solver. With the host front-end described in Section 3.1.1 and Figure 5, this problem is reduced to the development of a GPU-accelerated back-end.

The first notable feature of the GPU backend is the object used to store the $\boldsymbol{r}_{is's}$ matrices used in computation. The initial implementation of this object was one that stored the matrix in CUDA unified virtual memory (UVM) [12]. This is a natural first test for a number of reasons, not least of which being that the same memory address can be directly written to on the host then pre-fetched on the device. The relative performance of this initial implementation was demonstrated in a brief test on a machine with a Ryzen 3800X and an NVIDIA GTX 1080 with 8GB of video memory. This test demonstrates the throughput of (an older build of) the algorithm on the VHTR benchmark problem [27], [28]. Note, this test was not performed with the most efficient version of the compute kernels (for compatibility reasons), and as such the absolute metrics should NOT be taken as indicative of overall performance. This test is only relevant as one measure of the relative performance of the UVM and explicit memory treatments achieved by COMET-cpp. The results of the test can be seen in Table 1. It is observed that the use of UVM to store the underlying matrices (with careful attention paid to prefetching) results in a modest performance penalty as compared to an explicit treatment. However, in the pursuit of maximum performance (as well as explicit control over the movement of memory between the host and one or more devices) an explicit treatment was subsequently developed.

Table 1: UVM Test Results

Memory Treatment	Throughput (GB/s)	Relative performance
Explicit Management	789	-
UVM	667	84.5%

The final implementation of the matrix storage object uses explicitly allocated and managed CUDA memory. Each buffer used by the object stores the matrix in column-major order for the access-coalescing reason described in Section 2.2.1 above. The overall storage object consists of a host copy of the buffer that lives in OS-pinned memory, and one or more device mirrors of the host buffer. OS-pinned memory is used to maximize the performance of data transfers during operation, per [11]. The object only creates a mirror on any particular device when one is requested by the calculation space. This ensures that there are no redundant copies on any device that does not need one (if its particular chunk of operations does not involve that matrix). These device mirrors are updated whenever the underlying RF matrix is updated (e.g., due to a change in the guess for *k*). Such updates never occur during an outer iteration, meaning that the data is ONLY read (never written to) during the inner iteration process. This means that there is no communication between the host and the device related to these matrices during calculation, aligning with the communication-minimization goal described in Section 2.2.1. The object typically stores the numbers in each buffer in single-precision, but the program can be compiled for double-precision calculation.

As described above, a key feature of the COMET-cpp front-end is that it provides to the back-end a list of matrix operations that can be considered truly independent, in that they can be performed in any arbitrary order with any degree of concurrency (with the intermediate results buffer strategy described in Section 3.1.2). The COMET-cpp GPU back-end takes this feature one step further and divides the operations between up to 4 GPUs in the system. To implement such a division efficiently, it is clear that some degree of distributed-memory parallel principles must be introduced, and inter-GPU communication must occur.

The COMET-MPI study [6] discussed several different methods to separate a COMET problem for distributed-memory systems, and ultimately settled on a geometric decomposition. In such a decomposition, the only inter-processor communication that occurs is the communication of the currents that lie on the border between two geometric regions. Since it had been proven in a previous study (in addition to being generally applied for many parallel physics problems), such a decomposition was considered as a starting point for a multi-GPU problem division. This was ultimately decided against for several reasons:

- GPU-to-GPU communication is generally quite fast when links are present, often being on par with or faster than host-to-GPU communication. This is much less relatively expensive than communication between multiple processors. As such, minimizing communication between GPUs is not as much of a concern as minimizing communication between processors in an MPI program.
- Geometric decomposition introduces a degree of problem-dependence on the work division. It is preferable to investigate methods that are not problem-dependent, although in practice the problem-dependence of this strategy on only up to 4 devices may be tolerable.
- The material-major sweep ordering has a strong effect on runtime. Changing it to a geometry-based sweep ordering (as was done in the COMET-MPI study [6]) is worth avoiding, if possible, to maximize cache use, which still has a strong runtime effect even on a GPU.

- Depending on the layout of the problem, it is possible that the same matrix may be required to reside on and be operated with on more devices than is really required. Grouping the operations by matrix type and then splitting the list evenly increases the likelihood that matrices will only need to reside on 1 device, allowing problems with higher numbers of unique coarse meshes to fit on the total device memory. Such memory concerns were identified in the COMET-MPI study [6] as an issue with geometric decomposition as the number of unique meshes increases.

For these reasons, a more basic approach was selected in which each device keeps a copy of the entire global eigenvector ***J***, and the vector is reduced across all devices after each iteration. As a result, the communication between GPUs during each inner iteration consists of copies of the entire vector, rather than some smaller subset being exchanged. In whole-core problems, this vector is no more than a couple hundred megabytes in size. This relatively small size of the vector and fast inter-device communication means that communication is not a dominant contributor to the runtime of the program. The matrix multiplications still contribute the vast majority of the runtime, as demonstrated in Section 4.3. The reduction of the eigenvector involves 1 round of pairwise communication if 2 GPUs are used or 2 rounds of pairwise communication if 4 GPUs are used, the pattern of which matches the hypercubic pattern typically used as the device interconnect topology. This communication can be seen in Figure 9.

The remaining algorithmic choice in the division of work between multiple devices is the permutation of the operations list (as described in Section 3.1.2) and the distribution of this list between the multiple devices. In this work, it was selected to maintain the material-major ordering used in both the serial solver and the CPU-only COMET-cpp back-end. The logic behind this decision is that, although the primary concern for memory layout in GPU programs is the access coalescing described in Section 2.2.1, the L1 and L2 caches present on each SM still provide fast access to memory that was recently used. As such, if a cache-aware batched matrix multiplication algorithm is used (as described in this section below), it is generally beneficial to group together operations that use the same matrix.

The distribution of this list between the devices used in this study is an even distribution of contiguous chunks. The reasoning behind this is two-fold. First, it maintains the cache optimization described above. Second, the as-needed allocation of device mirrors of the matrix storage objects means that it is likely that a representation of a specific matrix will only need to live on one device at a time. Although some matrices may be used more or less often than others in a problem, this will loosely scatter the matrices between the devices, enabling larger problems to fit into on-device memory. As problems grow in size, it is possible that this may need to be more strictly enforced in an alternate scheme, giving each device an even number of matrices, rather than operations.

An example of the division of work and communication pattern for a 4 GPU example problem is seen in Figure 9. The top of the figure depicts the list of operations passed to the GPU back-end by the front-end, after the application of the material-major sorting operation. This list is split evenly between the 4 devices.

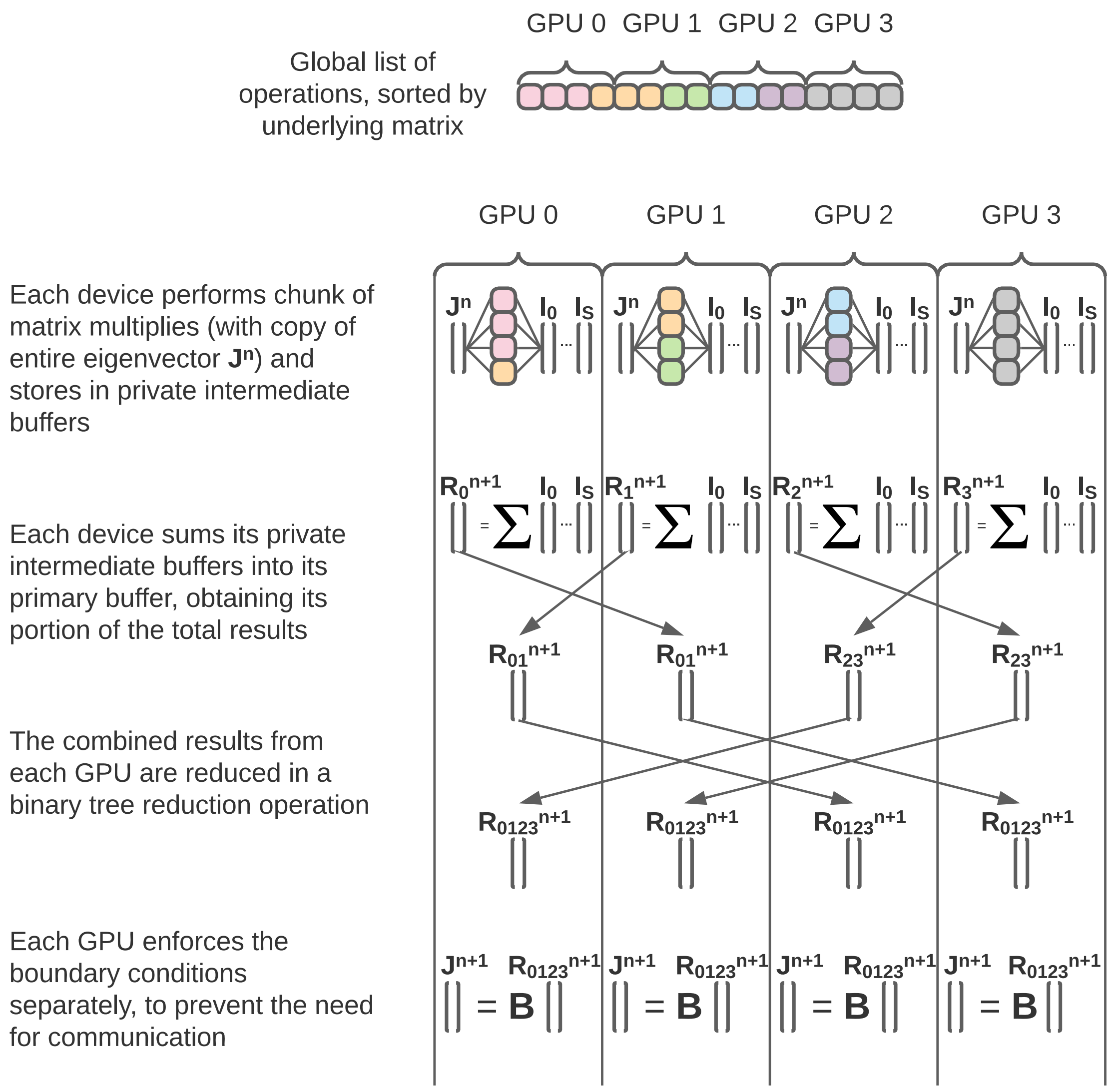

GPU 0 GPU 1 GPU 2 GPU 3
Global list of operations, sorted by underlying matrix
GPU 0
GPU 1
GPU 2
GPU 3
Each device performs chunk of matrix multiplies (with copy of entire eigenvector J^n) and stores in private intermediate buffers
J^n
I_0
I_S
Each device sums its private intermediate buffers into its primary buffer, obtaining its portion of the total results
R_0^{n+1}
R_1^{n+1}
R_2^{n+1}
R_3^{n+1}
R_01^{n+1}
R_23^{n+1}
The combined results from each GPU are reduced in a binary tree reduction operation
R_0123^{n+1}
Each GPU enforces the boundary conditions separately, to prevent the need for communication
J^{n+1}
= B

Figure 9: Multi-GPU Calculation Scheme

The first step of an inner iteration is the execution of the per-device list of small dense matrix multiplies. These operations are performed in parallel by a custom CUDA kernel, as will be described below. This process is by-and-large the same as it is for the CPU-only back-end, and as such this step for each device bears a resemblance to Figure 7 above. In the same way, parallelization of these multiplies is fully unlocked by the use of S intermediate buffers to which results can be written, where S is the number of surfaces per coarse mesh in the problem. In this way, there are no conflicting writes, and the multiplies can be performed with any degree of parallelization or synchronicity. These intermediate results buffers are then summed to get the portion of the total results buffer due to the operations performed on this specific device.

To obtain the full results vector, these partial representations are then reduced via a binary tree reduction operation, seen in the next two steps of Figure 9. This step entails a device-to-device copy of the partial results buffer, followed by a kernel to add the fetched buffer to the device primary buffer. This is a synchronous operation (between the devices) to ensure that the full results are fetched at each step. As described above, due to the fact that these transfers are generally performed over device-to-device interconnects (such as NVIDIA's NVLink [29]), these transfers are relatively fast.

After this reduction operation, each device has a copy of the total results vector which is obtained as if it performed all the operations itself. To obtain the true next guess for the eigenvector $\boldsymbol{J^{n+1}}$, each device enforces the boundary conditions separately, via the list of boundary condition operations passed to the back-end by the front-end (as described in Section 3.1.1). This represents an inefficiency in that the same work is being performed multiple times (redundantly). However, these boundary condition update operations typically represent only a small amount of work. As such, this redundant execution is worth it to avoid the need to communicate the results of this work to each device, if it were only performed on one device.

The matrix multiplications shown in Figure 7 and Figure 9 comprise what could loosely be considered a "batched-GEMV" operation, where GEMV is the typical terminology used by BLAS programs to denote a GEneric Matrix-Vector operation (multiplication). The concept of batched BLAS calls has gained popularity in recent years due in part to the proliferation of machine-learning workloads in which batched-GEMM (GEneric Matrix-Matrix) calls play a large part [30]. Since a GEMV operation can be expressed as a GEMM operation, an initial implementation of this step leveraged the built-in batched-GEMM available in the cuBLAS library [31]. To demonstrate the relative performance of this call as compared to a naïve custom implementation, a brief test was performed on a machine with a Ryzen 3800X and an NVIDIA GTX 1080 with 8GB of video memory. The results can be seen in Table 2 below. Note, the kernel used in comparison is a very simple initial implementation, and NOT the final cache-aware kernel developed later in this work. As such the results should not be taken in absolute terms. This is only provided as a rough comparison.

Table 2: cuBLAS Comparison Results

Calculation Executor	Throughput (GB/s)	Relative performance
Custom batched-GEMV kernel	479	-
cuBLAS batched-GEMM	243	50.7%

It can be seen from this test that the batched-GEMM provided by cuBLAS is not a good fit for the type of calculations involved in a typical whole-core COMET-cpp run. This may not be altogether too surprising, considering the typical use case of batched-GEMM calls. Typically, these calls are used to perform matrix multiplications A x B in which both A and B are typically both of dimensions around 8 x 8 (or slightly smaller/larger). The heuristic optimizations likely present in the cuBLAS library for these ranges likely do not apply to COMET operations, which involve the multiplications of up to a 720 x 720 matrix (or larger) and a 720 x 1 vector. As such, it was determined that a custom kernel should be developed and tuned for the operations that occur during a COMET run.

A typical matrix used in a COMET run (720 x 720) yields a total storage requirement for one matrix of around 2 MB (in single-precision, or 4 MB in double-precision). This is too large to fit into L1 cache in a typical modern card (128 KB on a V100 [2] and 192 KB on an A100 [9]). As such, it is desirable to construct a kernel that attempts to work on smaller chunks of the matrix that can be loaded into and kept in L1 cache. This approach is described below.

Before describing the cache-aware kernel developed in this work, it is worth noting that one clear alternative approach would be to explicitly manage the fast-access memory available to the SM via the use of CUDA *shared memory*. Shared memory can be thought of as an explicitly managed cache, and indeed, on the V100 and A100 architectures they are in fact substitute goods that share the same physical memory on the unit [2], [9]. It has been demonstrated, however, that this “unified cache” provides cache-only performance that increasingly rivals an explicit shared memory treatment, up to around 93% in relative performance [2]. Explicitly leveraging shared memory often involves high amounts of additional logic which, as applied to COMET, would likely require a degree of problem dependence (e.g., depending on the size of the problem basis set). Additionally, there is likely a trade-off between shrinking chunks to fit entirely in cache and the increased stashing of intermediate results as described below. For these reasons, an explicit shared memory treatment was not pursued in this work. It is identified as a possible area for future work, subject to the caveats described above. Instead, the kernel used in this work was designed to group operations to most effectively leverage the L1 cache.

As described in Section 2.2.2.1, the CUDA execution hierarchy describes the way that the many threads concurrently executing on a GPU interact with each other:

- A **thread** is the most granular level of execution. It has a separate register file from all other threads.
- 32 threads execute in a group called a **warp**. These threads execute in lock-step. If threads branch differently, some threads will sit idle until their execution path rejoins with the others. Certain memory access patterns occur very quickly in a process called coalescing.
- A group of warps execute in a thread **block**. This block executes on one SM and shares a level of cache and a separate shared memory bank.
- A group of blocks executes in a **grid**. The blocks are distributed across the available resources on the device and execute with no guarantee of concurrency or synchronicity. Ordering of operations can only be ensured by launching two separate kernels with two separate grids.

We begin constructing a layout for the custom batched-GEMV kernel in the context of this execution hierarchy from the bottom up. Since results accumulate across a row in matrix multiplication, it is natural to have threads be responsible for 1 row and proceed across the columns in the matrix. By executing such that consecutive threads are responsible for consecutive rows in the matrix, we can leverage the column-major memory layout and ensure that threads access consecutive memory locations, and as such those accesses coalesce. Additionally, threads will only branch differently at the bottom of the matrix, when there are not enough remaining rows to saturate the number of dispatched threads. Since the idle threads are not waiting to perform useful work (and instead have finished their work), this does not represent an inefficiency.

As mentioned above, the matrices used in calculation do not fit entirely into L1 cache. As such, one algorithmic optimization is to operate on smaller chunks of the matrix that are likely to fit in cache. Since matrices are laid out in column-major order, this corresponds to some number of whole columns. When the last column in the chunk is reached, the intermediate results are stashed, and the threads stride down to the next group of rows in the matrix. Since this memory is likely to already reside in cache, these subsequent accesses are likely to be fast.

After completing calculation with the matrix chunk, instead of moving on to the next chunk in the operation, the threads then proceed to the next operation in the list. Due to the material-major operation ordering described in Section 3.1.2, this subsequent operation is likely to use the same underlying matrix as the previous operation. As such, the threads will likely be operating on the same chunk of data as the previous operation. In this case, the matrix chunk will already reside in cache, and accesses will be fast. If the operation does not use the same matrix, a new chunk will be loaded in, and the previous chunk will be evicted from cache. Since the threads have already done all they can with the previous chunk, this eviction does not represent an eviction of useful data. The threads proceed in this manner through all operations in the list. After the end of the operation list is reached, the threads wrap back around to the start of the list, proceeding to the next chunk of the first operation. Rather than start with a 0-initialized accumulator value, threads load the previously stashed intermediate results from prior chunks and proceed.

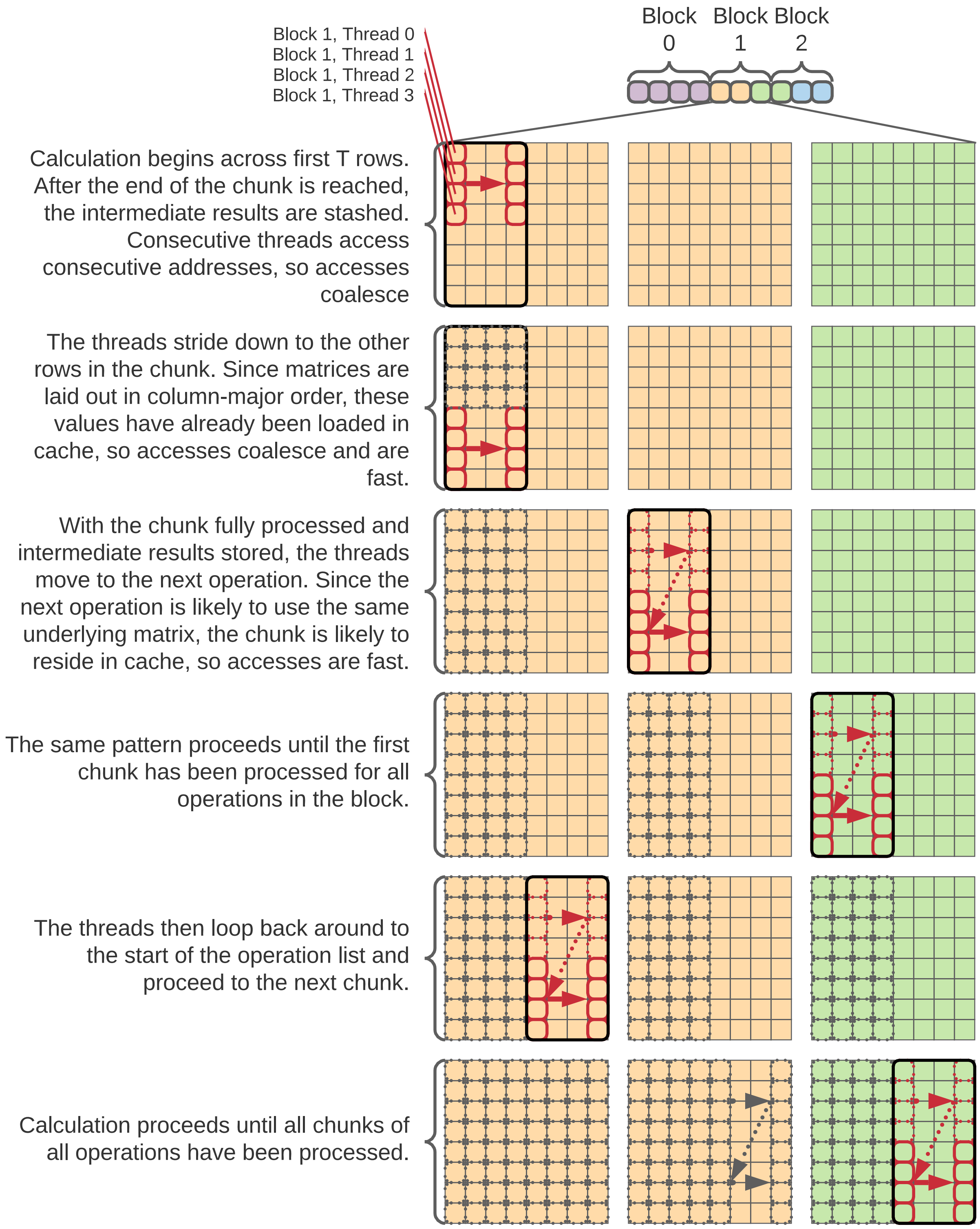
The pseudo-code representing the final custom cache-aware batched-GEMV kernel used in this work can be seen in Figure 10. A graphical depiction of this ordering of operations can be seen in Figure 11. The resulting algorithm includes three primary tuning "knobs" which can be changed for specific devices and problems:

- The number of threads per block
- The number of blocks in the grid
- The number of columns per chunk

Values for these parameters were fuzzed and set to values that result in generally good performance for the whole-core benchmark problems tested in this study.

```
Batched-GEMV Kernel:
for chunk in chunks:
  for op in operations:
    for row in rows:
      if chunk_index == 0:
        accumulator = 0
      else:
        accumulator = y_list[op][row]
      for column in cols_in_chunk:
        accumulator += x_list[op][col] * matrix[op][col][row]
      y_list[op][row] = accumulator
```

Figure 10: Batched-GEMV Pseudo-code



Calculation begins across first T rows. After the end of the chunk is reached, the intermediate results are stashed. Consecutive threads access consecutive addresses, so accesses coalesce

The threads stride down to the other rows in the chunk. Since matrices are laid out in column-major order, these values have already been loaded in cache, so accesses coalesce and are fast.

With the chunk fully processed and intermediate results stored, the threads move to the next operation. Since the next operation is likely to use the same underlying matrix, the chunk is likely to reside in cache, so accesses are fast.

The same pattern proceeds until the first chunk has been processed for all operations in the block.

The threads then loop back around to the start of the operation list and proceed to the next chunk.

Calculation proceeds until all chunks of all operations have been processed.

Figure 11: Batched-GEMV Kernel Visualization

3.3 Post-processing and Visualization Tools

In this project, a new suite of post-processing and visualization tools was developed. There are several reasons this suite was targeted as a development goal for this project:

- A tool to visualize the output of the program is instrumental in the process of debugging, ensuring output correctness, etc.
- Since it is an important goal of this work to match the solutions produced by the serial solver to a high precision (within convergence criteria), a “differ” that compares two cores (e.g., one produced by the serial solver and one produced by the new solver) is necessary to demonstrate agreement.
- As the solver is being entirely re-written, it is a natural fit to overhaul the format of the output printed by the solver and develop a new suite of tools to take advantage of the increased level of information printed in these files.

To improve the usability of the solver, a set of new output print tables was developed with the goal of being comprehensive, fully detailed, and self-describing. All floating-point numbers in the output files are printed to the full precision allowed by the variable data type (single- or double- precision). The output tables that can be printed by the new solver are described in Table 3.

Table 3: Output Tables

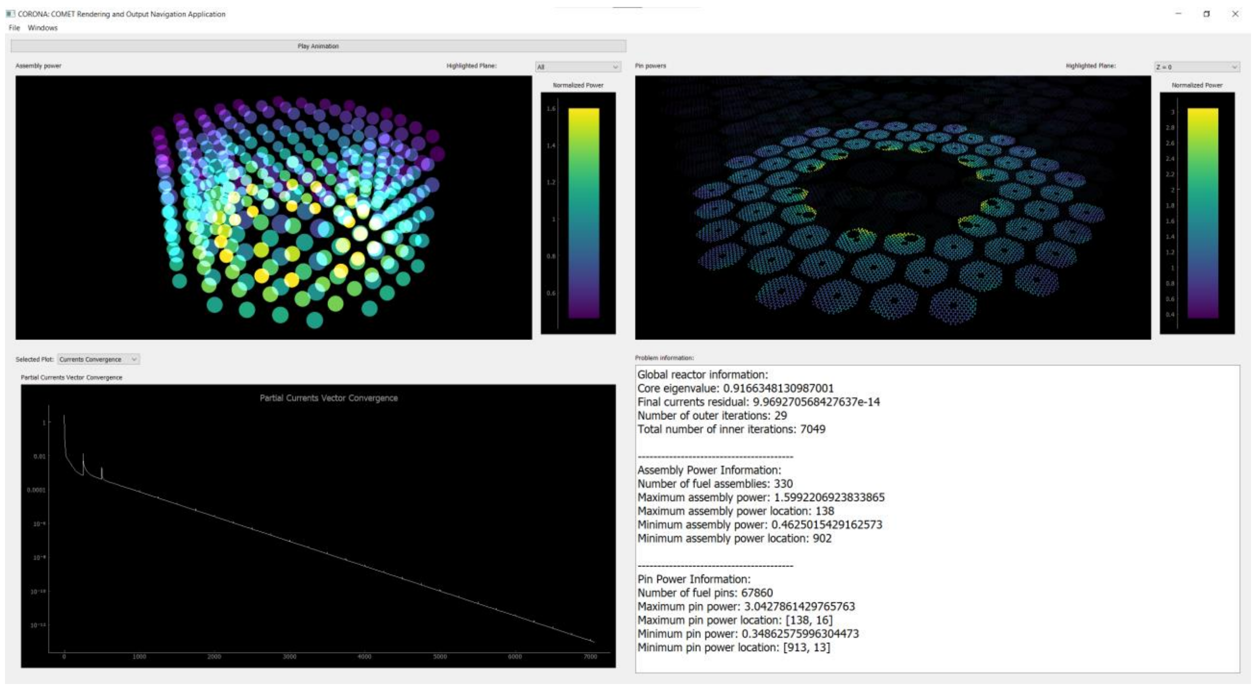
Table Number	*Table Description*	*Data contained in File*
100	"Contains a representation of the problem as understood by COMET"	• Geometry metadata • RF database metadata • Core layout by material • Core layout by mesh index • Individual mesh coordinates and spectral identities by surface
101	"Contains the assembly and pin powers calculated by COMET per coarse mesh in the problem"	• For all fuel-containing meshes in the problem: ○ Index ○ Assembly power ○ Number of pins ○ Pin powers
102	"Contains the final converged inward-facing partial currents by mesh for the problem"	• Number of surfaces per mesh • Number of 0th-order partial current expansion moments in the basis set • For all meshes in the problem: ○ All 0th-order partial current moments for

Table Number	*Table Description*	*Data contained in File*
		each surface in the mesh
103	"Contains information about the core eigenvalue and its convergence throughout the run"	• Final converged eigenvalue • Total fission, absorption, and leakage for the problem • Total number of outer iterations • Criticality eigenvalue by outer iteration number
104	"Contains information about the convergence behavior of the partial currents during the run"	• Total number of outer iterations • Final residual norm • For each outer iteration: ○ Number of inner iterations ○ Calculated residual norms in the outer iteration

To facilitate post-processing of this data, a suite of Python tools was developed. The first in this set of tools is a module that parses these text-based output files and produces a Python dictionary with the information from any output files requested to be printed. This custom module uses only Python built-in functions (such that it has no dependencies). The dictionary describing the reactor core is structured to closely mirror the internal representation of the reactor core in the COMET-cpp solver to facilitate activities like debugging. This module also includes the ability to read and parse the text-based output files generated by the old serial solver.

The second tool is a comprehensive analysis and visualization tool called the COmet Rendering and Output Navigation Application (CORONA). This tool uses the COMET parsing tool described above to read the output files produced by the solver and provides a number of useful metrics and result visualizations to help interpret the results of the program. As a result of the high number of individual data points that are visualized (up to a couple million pin powers, for example), a GPU-accelerated plotting library is required. After testing of a couple libraries, the library was selected to be PyQtGraph [32]. PyQtGraph was selected for its OpenGL acceleration capabilities which provide excellent performance on the large number of points required for the visualization tool and the ease of incorporating the visualization panes in a larger PyQt application. A PyQt user interface was developed around these visualizations to facilitate navigation.

The application has two main modes: single-core visualization mode and a two-core comparison mode. A screenshot of the single-core visualization mode can be seen in Figure 12.



Global reactor information:
Core eigenvalue: 0.9166348130987001
Final currents residual: 9.969270568427637e-14
Number of outer iterations: 29
Total number of inner iterations: 7049

Assembly Power Information:
Number of fuel assemblies: 330
Maximum assembly power: 1.5992206923833865
Maximum assembly power location: 138
Minimum assembly power: 0.4625015429162573
Minimum assembly power location: 902

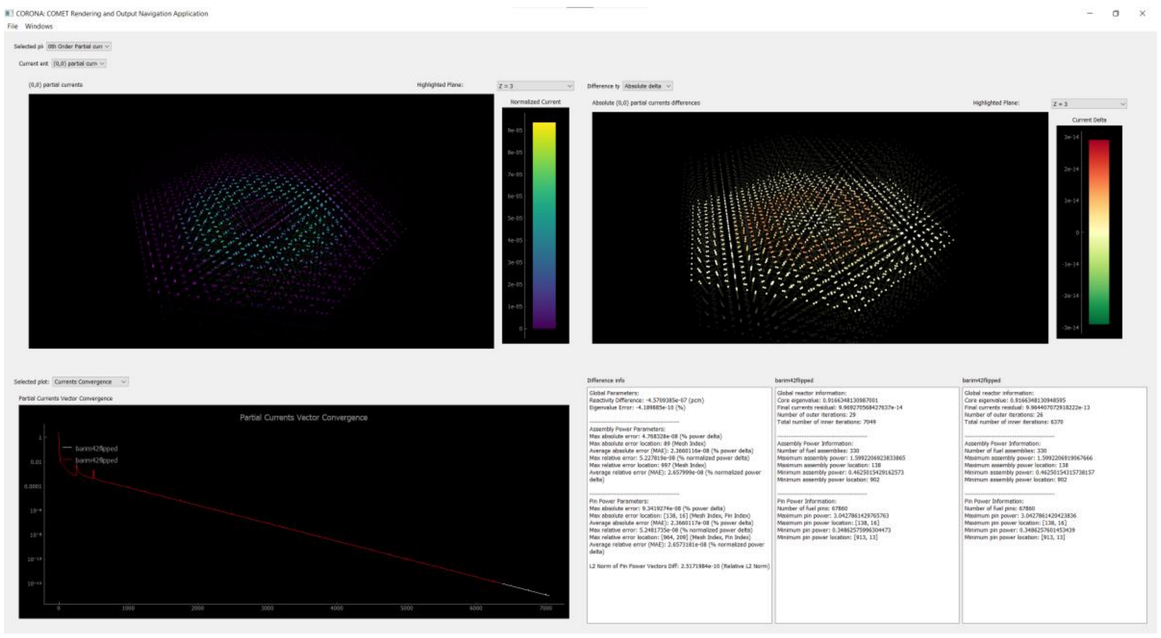
Pin Power Information:
Number of fuel pins: 67860
Maximum pin power: 3.0427861429765763
Maximum pin power location: [138, 16]
Minimum pin power: 0.34862575996304473
Minimum pin power location: [913, 13]

Figure 12: Single-core Visualization Mode Screenshot

This single-core visualization mode contains the following four information panes:

- A 3-D assembly power visualization in which fuel-containing meshes are plotted in 3-D space and colored according to the power being produced. The viewport can be moved, rotated, zoomed, and panned in real-time. This pane includes the ability to select one plane of the core to highlight for clarity while the rest of the code fades out to only 10% opacity.
- A 3-D pin power visualization in which a pin-mapping file is used to map individual pin powers into 3-D space, coloring them according to the power produced. The same movement and highlighting capabilities are present in this pane as are present in the assembly power pane. The assembly and pin power panes can be animated in sync, highlighting every Z-plane in the core in sequence, spaced 1 second apart.

- A stackable pane with 3 options:
    - A plot of the convergence of the core criticality eigenvalue against outer iteration number.
    - A plot of the convergence of the partial currents vector against cumulative inner iteration number (displayed in Figure 13).
    - A 3-D visualization of the inward-facing surface-wise 0^{th}-order current expansion moments. This pane includes a drop down to select the desired energy bin for visualization. The points are colored based on the magnitude of the expansion coefficient. This pane contains the same highlighting capability as the assembly and pin power panes but cannot be animated.
- A text pane containing the following information about the core:
    - Final eigenvalue
    - Final currents residual
    - Number of inner and outer iterations
    - Number of fuel assemblies
    - Maximum assembly power (and the location of this maximum power)
    - Minimum assembly power (and its location)
    - Number of fuel pins
    - Maximum pin power (and its location)
    - Minimum pin power (and its location)


CORONA: COMET Rendering and Output Navigation Application
File Windows
Selected plot: 0th Order Partial curr
Current ent: (0,0) partial curr
(0,0) partial currents
Highlighted Plane: Z = 3
Normalized Current
Difference ty: Absolute delta
Absolute (0,0) partial currents differences
Current Delta
Selected plot: Currents Convergence
Partial Currents Vector Convergence
barim42flipped
barim42flipped

```
Difference info
Global Parameters:
Reactivity Difference: -4.5709385e-07 (pcm)
Eigenvalue Error: -4.189885e-10 (%)

-------------------------------------
Assembly Power Parameters:
Max absolute error: 4.768328e-08 (% power delta)
Max absolute error location: 89 (Mesh Index)
Average absolute error (MAE): 2.3660116e-08 (% power delta)
Max relative error: 5.227819e-08 (% normalized power delta)
Max relative error location: 997 (Mesh Index)
Average relative error (MAE): 2.657999e-08 (% normalized power
delta)

-------------------------------------
Pin Power Parameters:
Max absolute error: 9.3419274e-08 (% power delta)
Max absolute error location: [138, 16] (Mesh Index, Pin Index)
Average absolute error (MAE): 2.3660117e-08 (% power delta)
Max relative error: 5.2481735e-08 (% normalized power delta)
Max relative error location: [964, 209] (Mesh Index, Pin Index)
Average relative error (MAE): 2.6573181e-08 (% normalized power
delta)

L2 Norm of Pin Power Vectors Diff: 2.5171984e-10 (Relative L2 Norm)
```

```
barim42flipped
Global reactor information:
Core eigenvalue: 0.9166348130987001
Final currents residual: 9.969270568427637e-14
Number of outer iterations: 29
Total number of inner iterations: 7049

-------------------------------------
Assembly Power Information:
Number of fuel assemblies: 330
Maximum assembly power: 1.5992206923833865
Maximum assembly power location: 138
Minimum assembly power: 0.4625015429162573
Minimum assembly power location: 902

-------------------------------------
Pin Power Information:
Number of fuel pins: 67860
Maximum pin power: 3.0427861429765763
Maximum pin power location: [138, 16]
Minimum pin power: 0.3486257599630473
Minimum pin power location: [913, 13]
```

```
barim42flipped
Global reactor information:
Core eigenvalue: 0.9166348130948595
Final currents residual: 9.984407072918222e-13
Number of outer iterations: 26
Total number of inner iterations: 6370

-------------------------------------
Assembly Power Information:
Number of fuel assemblies: 330
Maximum assembly power: 1.5992206919867666
Maximum assembly power location: 138
Minimum assembly power: 0.4625015431573815 7
Minimum assembly power location: 902

-------------------------------------
Pin Power Information:
Number of fuel pins: 67860
Maximum pin power: 3.0427861420423836
Maximum pin power location: [138, 16]
Minimum pin power: 0.3486257601453439
Minimum pin power location: [913, 13]
```

Figure 13: Two-Core Difference Visualization Mode Screenshot

A screenshot of the two-core “differ” mode of the visualization application can be seen in Figure 13. This mode reads in the results of a reference COMET run and a perturbed COMET run with some differing parameters and compares the results. In this mode, all the information from the single-core visualization mode is present as well as:

- The absolute and relative differences between the assembly powers, pin powers, and 0th-order partial current expansion moments, all plotted in 3-D space.
- The following text-based information about the difference between the cores:
    - Absolute and relative reactivity difference between the cores
    - Maximum and mean error between the assembly powers (absolute and relative)
    - Maximum and mean error between the pin powers (absolute and relative)

- Relative L2 norm of the difference between the global vector of pin powers (consisting of all the pin powers in the problem concatenated into one vector)

This difference tool is the basis of the convergence threshold sensitivity study documented in Chapter 5 of this document. Additionally, in the development of the solver, this mode was used to verify that the solutions produced by COMET-cpp and the serial solver agree to within convergence criteria.

There are several other applications where such a mode could be useful, including a similar sensitivity study performed on the order of flux expansion used in the problem, and a quick assessment of the difference between two cores that differ in a small way (e.g., two assemblies swapped in a loading pattern, or control rods inserted to differing depths).

CHAPTER 4. BENCHMARK TIMING STUDY

To assess the performance of the GPU-accelerated solver, a timing study was performed on a set of benchmark problems. Three different whole-core problems were selected for demonstration, described in Section 4.1. A methodology based on inner iteration throughput was selected so as not to unfairly penalize the serial solver for extra operations not performed in COMET-cpp, or COMET-cpp optimizations not related to the iteration procedure. This methodology is described in Section 4.2. The results of the study for the three benchmark problems are presented in Section 4.3.

Since COMET-cpp and the serial solver were verified (with the "differ" visualization tool discussed in Section 3.3) to produce the same solution to within convergence criteria, the accuracy of the COMET-cpp solutions relative to MCNP are not discussed explicitly. Instead, the description of each benchmark problem in Section 4.1 contains a brief description of the accuracy of the serial solver results.

4.1 Benchmark Problems

To demonstrate the capabilities of the new solver with respect to a wide variety of applications, a set of benchmark problems were selected which varied in geometry, size, and reactor type (e.g., coolant and moderator material). Three problems were selected from the range of benchmarks already solved with the COMET method: the Very-High-Temperature Reactor (VHTR), the Integral Inherently Safe Light Water Reactor (I^2S-LWR, or just I^2S), and the Advanced High-Temperature Reactor (AHTR). In this section, each of these benchmark problems will be described, as well as the accuracy of the COMET solutions to the problem relative to MCNP.

4.1.1 Very-High-Temperature Reactor

The VHTR is a Generation IV reactor concept that features a graphite-moderated, helium-cooled thermal core [27]. The reactor poses a challenge to neutronic modelling tools due to its complex geometry, high degree of heterogeneity due to the uranium oxycarbide tri-structural isotropic (TRISO) fuel spheres embedded in the fuel pins, and its large active core region. To aid in the development and licensing of this reactor concept, a set of full-heterogeneity benchmark problems were created to test the validity of neutronics codes on this concept [27].

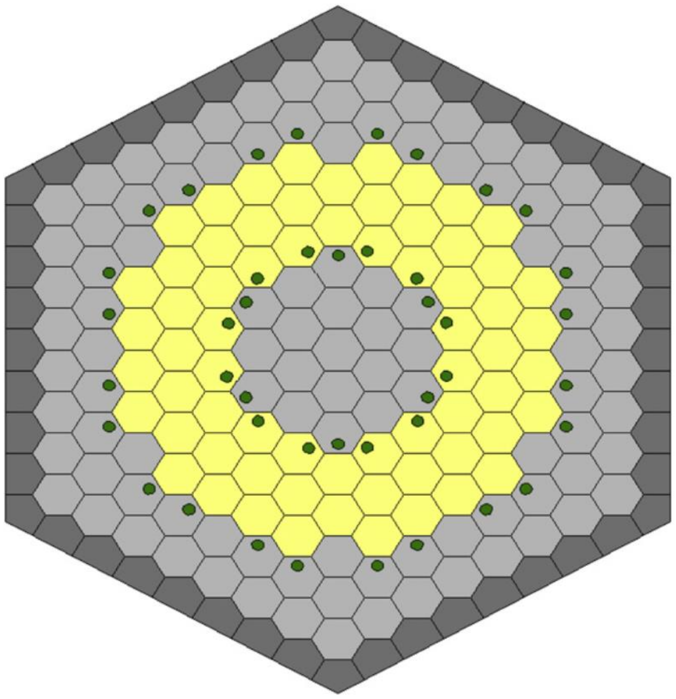


Figure 14: VHTR Layer Depiction (Figure from [29])

The VHTR benchmark core has 14 axial layers, each comprising 7 full rings of blocks around a central column with an additional fractional reflector layer at the periphery, as seen in Figure 14. In this figure, the gray blocks represent reflector blocks and the yellow blocks represent fuel assemblies, each with or without a channel for a control element.

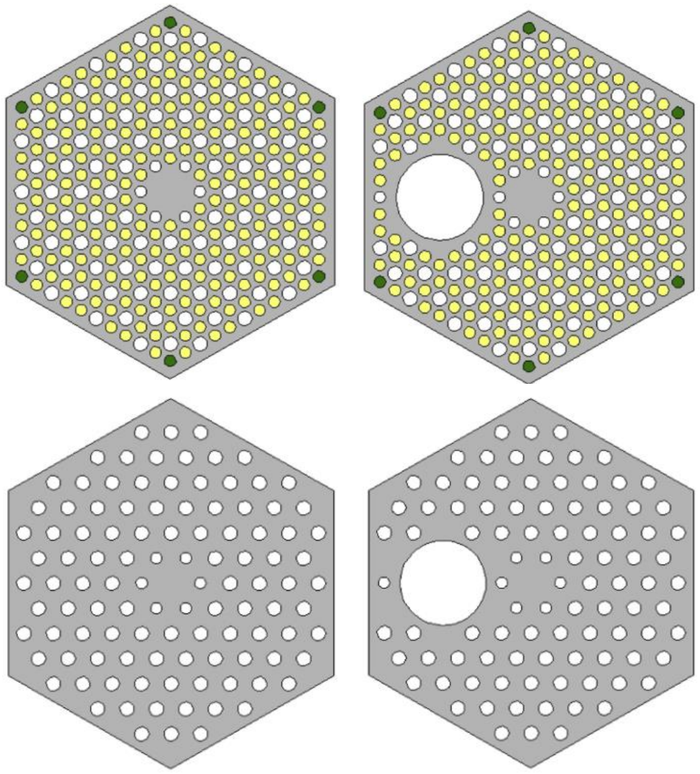


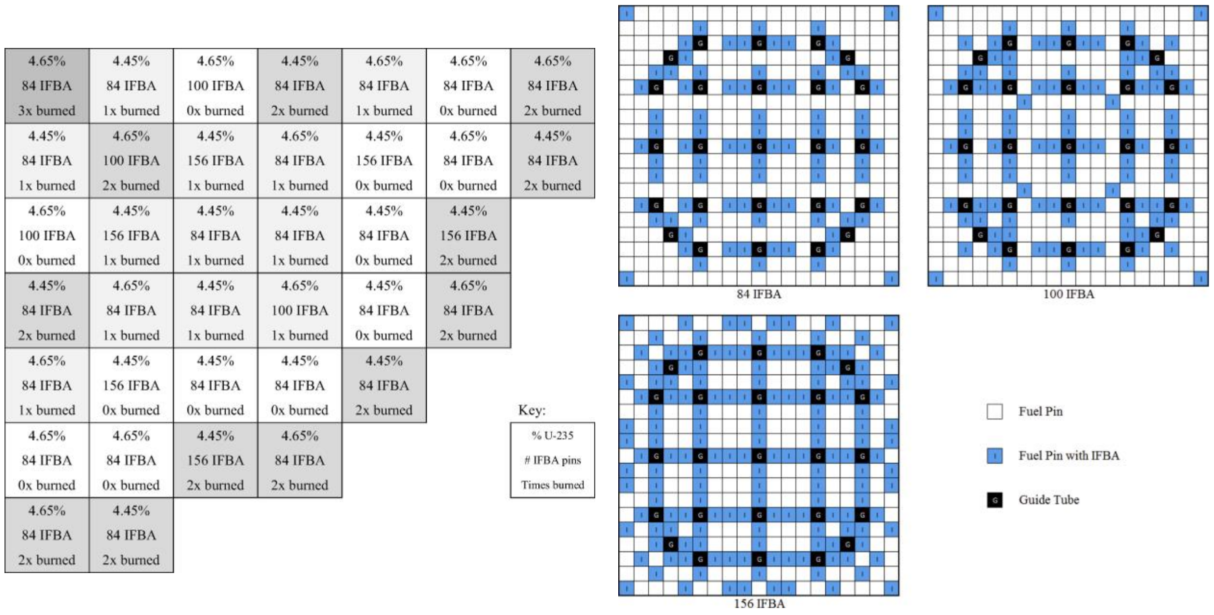
Figure 15: VHTR Component Blocks (from [29]): Fuel Block (top left), Fuel Block with Control Channel (top right), Reflector Block (bottom left), Reflector Block with Control Channel (bottom right)

A zoomed in view of the fuel and reflector blocks can be seen in Figure 15. In this figure, gray represents the underlying graphite structure, yellow represents a fuel pin region, green represents a burnable absorber region, and white represents a channel for the helium core coolant. It is noted that the high degree of heterogeneity results from the fact that the yellow regions of Figure 15 comprise thousands of uranium oxycarbide TRISO fuel particles, modeled discretely.

COMET solutions to this prismatic VHTR benchmark problem have been obtained and documented [28]. A previous study using the serial solver found that the solutions to the uncontrolled and controlled benchmark problems agreed with the MCNP solutions to 0.36% and 0.53% in average pin fission density difference, respectively. The calculated core criticality eigenvalue was found to agree to 66 ± 6 and 69 ± 6 pcm, respectively. In this work, the all-rods-in (ARI) benchmark problem is used.

4.1.2 *Integral Inherently Safe Light Water Reactor*

The Integral Inherently Safe Light Water Reactor (I^{2}S) is a pressurized water reactor concept designed to leverage several inherent passive safety mechanisms (including the use of uranium silicide fuel) in a gigawatt-scale reactor [33]. The core consists of 121 assemblies loaded according to an equilibrium loading plan depicted in Figure 16. In the multi-group COMET model of the I^{2}S core, there are 21 axial layers [34].



4.65% 84 IFBA 3x burned	4.45% 84 IFBA 1x burned	4.65% 100 IFBA 0x burned	4.45% 84 IFBA 2x burned	4.65% 84 IFBA 1x burned	4.65% 84 IFBA 0x burned	4.65% 84 IFBA 2x burned
4.45% 84 IFBA 1x burned	4.65% 100 IFBA 2x burned	4.45% 156 IFBA 1x burned	4.65% 84 IFBA 1x burned	4.45% 156 IFBA 0x burned	4.65% 84 IFBA 0x burned	4.45% 84 IFBA 2x burned
4.65% 100 IFBA 0x burned	4.45% 156 IFBA 1x burned	4.45% 84 IFBA 1x burned	4.45% 84 IFBA 1x burned	4.45% 84 IFBA 0x burned	4.45% 156 IFBA 2x burned	
4.45% 84 IFBA 2x burned	4.65% 84 IFBA 1x burned	4.45% 84 IFBA 1x burned	4.65% 100 IFBA 1x burned	4.45% 84 IFBA 0x burned	4.65% 84 IFBA 2x burned	
4.65% 84 IFBA 1x burned	4.45% 156 IFBA 0x burned	4.45% 84 IFBA 0x burned	4.45% 84 IFBA 0x burned	4.45% 84 IFBA 2x burned		
4.65% 84 IFBA 0x burned	4.65% 84 IFBA 0x burned	4.45% 156 IFBA 2x burned	4.65% 84 IFBA 2x burned			
4.65% 84 IFBA 2x burned	4.45% 84 IFBA 2x burned					

Key:

% U-235 # IFBA pins Times burned

Figure 16: I^{2}S Loading Plan (from [28])

The multi-group COMET solutions to the I^2S benchmark problem are notable because they were the basis for a study on Monte Carlo solution convergence [34]. In this study, both the radially integrated and pin-wise fission densities obtained by COMET were compared against several MCNP runs and demonstrated to agree within ~0.2% in axial fission density against a 50-run MCNP average and 0.4% in pin fission density average against a representative MCNP run.

4.1.3 *Advanced High-Temperature Reactor*

The Advanced High-Temperature Reactor (AHTR) is a fluoride-salt-cooled high-temperature reactor (FHR) concept that features plank-type TRISO-containing fuel with a graphite moderator [35]. This reactor poses similar difficulties to neutronics modelling tools as were discussed for the VHTR, most notably a very high degree of heterogeneity resulting from the TRISO fuel particles embedded in the fuel planks.

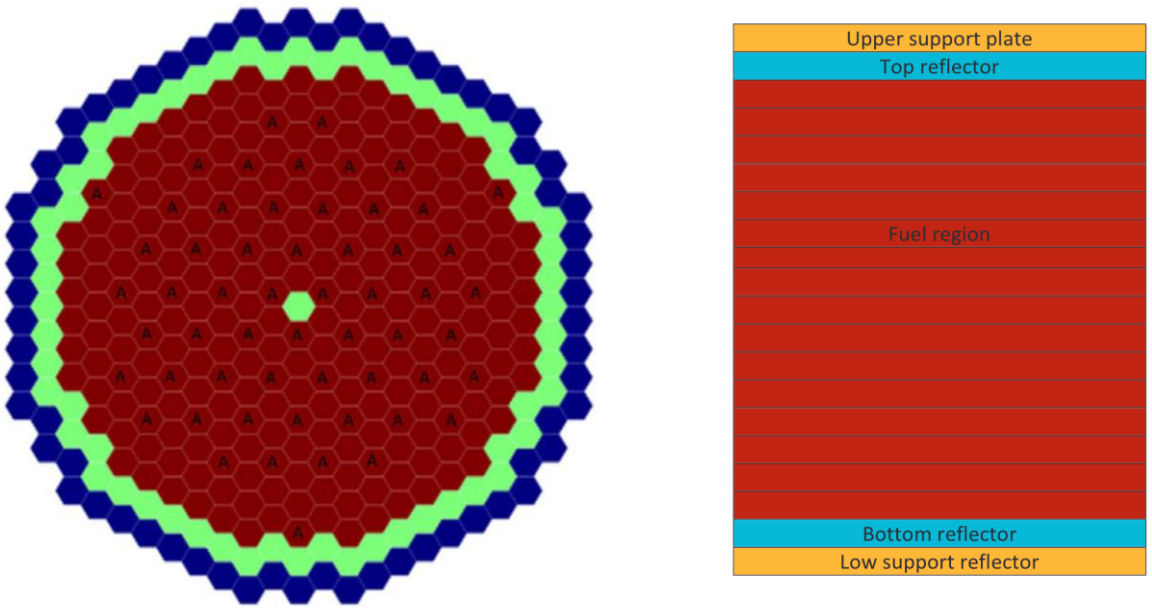


Figure 17: AHTR Problem Layout (from [36]): Radial (left), Axial (right)

The AHTR benchmark problem consists of 20 axial layers, with each layer consisting of 10 complete rings of meshes around a central mesh, as seen in Figure 17. In the radial layout, blue represents a permanent reflector block, green represents a replaceable reflector block, and brown represents a fuel block.

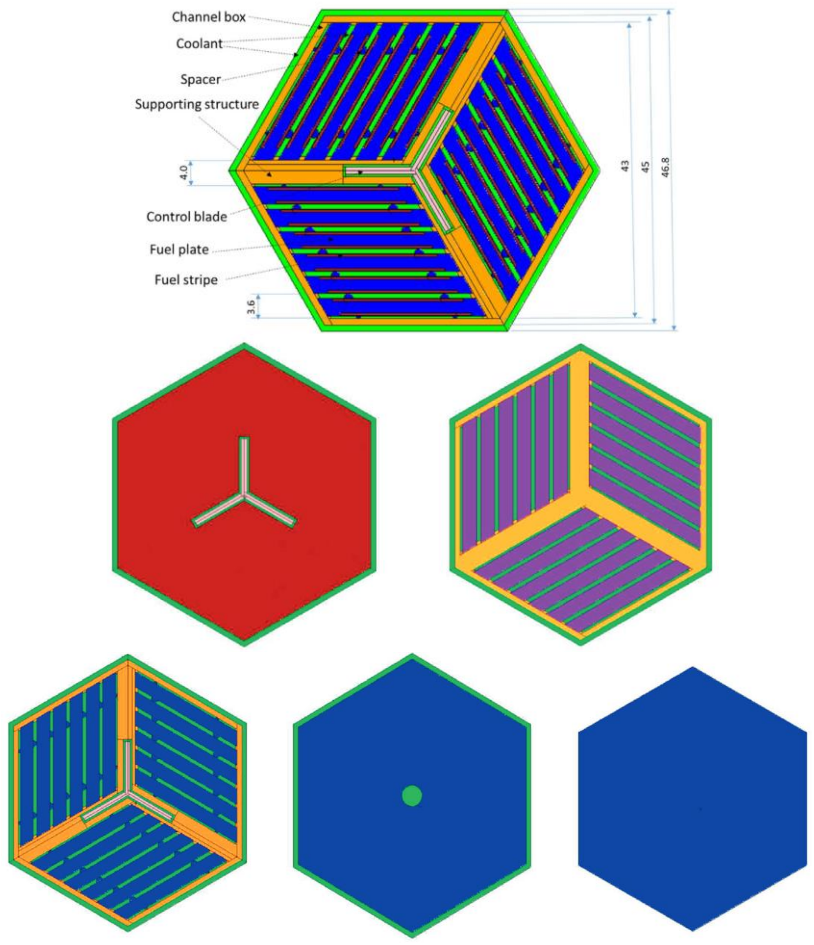


Figure 18: AHTR Mesh Composition (from [36]): Fuel (top), Upper Support (middle left), Lower Support (middle right), Top/Bottom Reflector (bottom left), Replaceable Reflector (bottom middle), Permanent Reflector (bottom right)

The composition of these blocks can be seen in Figure 18. It is noted that the region labelled "Fuel stripe" in this figure contains many TRISO particles pressed into the graphite matrix. It is this design feature that introduces the high degree of heterogeneity.

COMET solutions to this benchmark problem have been obtained and documented [36]. Average stripe-wise fission densities were found to agree with MCNP to under 0.5% on average, with a discrepancy in core criticality eigenvalue of between 17-88 pcm for the benchmarks calculated. For the remainder of this analysis, the all-blades-out benchmark problem is used.

4.2 Timing Comparison Methodology

In devising a fair methodology for the purpose of comparing the relative performance of the GPU-accelerated solver to the serial solver, there are a couple confounding factors that need to be considered. First, since the accelerated solver was being re-written from the ground up, care was taken to add additional optimizations in other non-iteration procedures in the code, including threading and an in-memory cache of the library when interfacing with the RF database CDF files. Since the focus of this work is the suitability of the *iteration method* for GPU-acceleration, this additional performance ought not be taken into account. A second difference between the codes is that the current GPU-accelerated solver does not include low-order acceleration capabilities. Since low-order acceleration involves solving the same problem but with a reduced basis set, it was decided that this would not be explicitly implemented for this initial study. As such, low-order acceleration must be excluded from the comparison. A third difference is that there are subtle differences in the ways in which the two solvers determine whether or not a problem has converged, and as such they may perform slightly different numbers of iterations (inner and outer) before terminating. For these reasons, it was determined that total runtime alone is an insufficient metric for comparison.

The metric settled on for comparison used in this study is inner iterations per second ($^{inner\ its.}/_{s}$) performed on a problem with no acceleration. This metric corresponds to the inverse of the time it takes the solver to perform one $\boldsymbol{MR}(k)\boldsymbol{J}$ multiplication (as in eq. 2.1.8) and enforce any relevant boundary conditions. This metric provides the fairest comparison for this analysis, as it avoids the extraneous operations which may differ between the two solvers. In interpreting the results of this study, however, it must be noted that the speedup interpreted from the comparison of this metric would not be applicable to the runtime of the entire program. This is due to the presence of non-parallelizable work in accordance with Amdahl's law including startup, RF database interfacing, output printing, and others. The presence of other optimizations in the solver means that the speedup experienced over the entire runtime of the program might be correspondingly more or less than what is reported when comparing these metrics.

To put this inner iteration throughput data into context, it is useful to also define a related speedup metric. This metric is useful in providing a direct comparison of the relative performance of two different configurations of the solver. For some configuration A with inner iteration throughput R_A and configuration B with throughput R_B, the speedup of B relative to A is defined as in eq. 4.2.1, with higher speedup values representing improved performance.

$$S_{B,A} = \frac{R_B}{R_A} \tag{4.2.1}$$

The speedup metric is useful because it allows comparison between two configurations, even when there are architectural differences between them. When the main difference between the two configurations is a mere increase in compute units (such as number of threads on a CPU, number of CPUs, or number of GPUs), we can use this speedup metric to define an efficiency metric. This metric compares a configuration with 1 compute unit against a configuration with n_{CU} compute units and gives an idea of how well the problem scales across this range. The definition of this efficiency metric can be seen in eq. 4.2.2, where T_n represents the execution time with n_{CU} compute units.

$$\varepsilon = \frac{1}{n_{CU}}\frac{T_1}{T_n} = \frac{1}{n_{CU}}\frac{R_n}{R_1} = \frac{S_{n,1}}{n_{CU}} \qquad (4.2.2)$$

This metric generally varies between 0 and 1, with 1 called the “ideal” or “linear” speedup case, the case in which there is no increased cost associated with the use of the additional compute units. In some cases, the observed efficiency may exceed 1, in a case called “super-linear” speedup. This behavior can result from several factors and can be harder to track down. Often, this results from the increased amount of cache available to the system since additional compute units usually bring their own cache.

Two other metrics will be presented in this study: the achieved memory bandwidth in reading the RF matrices (GB/s) and the achieved single-precision floating-point operations per second (SP-FLOPs/s, or just FLOPs/s). These metrics are more representative of the true calculations going on in hardware, and as such provide a way of normalizing performance across different problems. These values can be derived from the inner iteration throughput R defined above by multiplying by the memory read per iteration or the number of required operations per iteration, respectively. Since the matrix multiplies dominate the iteration work, other operations are neglected, resulting in a lower bound for the true achieved bandwidth and FLOPs/s. The required memory M and operations O per iteration can be seen in eqs. 4.2.3 and 4.2.4 respectively, where n_{mult} is the number of individual small dense matrix multiplications performed per iteration, m is the size of the basis set used in the calculation, and s is the size of the floating-point representation used to store the matrices and vector (4 for single-precision, 8 for double-precision). The factor of 2 in eq. 4.2.4 comes from the fact that each matrix element requires a fused multiply-add (FMA), which involves 2 floating-point operations.

$$M = n_{mult} * (m^2 + 2m) * s \tag{4.2.3}$$

$$O = 2 * n_{mult} * m^2 \tag{4.2.4}$$

In this study, the three benchmark problems described in Section 4.1 are solved for six individual configurations of the solver: one configuration for the serial solver and five configurations for COMET-cpp. In each case, an average raw inner iteration throughput is presented, as above. In cases where the notion of a corresponding 1 compute unit case applies, a speedup metric is also presented corresponding to a comparison against this 1 compute unit case. In these cases, an efficiency metric is also presented. The six cases examined in this study (along with the metrics to be presented) are:

a) Serial solver
    - Raw iteration throughput

b) COMET-cpp, CPU-only, 1 thread
    - Raw iteration throughput
    - Speedup relative to a)

c) COMET-cpp, CPU-only, all available threads

- Raw iteration throughput
- Speedup relative to a)
- Speedup and efficiency relative to b)

d) COMET-cpp, GPU-accelerated, 1 device

- Raw iteration throughput
- Speedup relative to a)

e) COMET-cpp, GPU-accelerated, 2 devices

- Raw iteration throughput
- Speedup relative to a)
- Speedup and efficiency relative to d)

f) COMET-cpp, GPU-accelerated, 4 devices

- Raw iteration throughput
- Speedup relative to a)
- Speedup and efficiency relative to d)

4.3 Results

The results in this section were generated on one of the GPU nodes of the Sawtooth cluster at Idaho National Laboratory. These nodes contain dual Xeon Platinum 8268 processors (24 cores each) and four NVIDIA V100 GPUs, each with 32GB of on-device memory. The serial solver was compiled with the Intel Fortran compiler version 19.1. The COMET-cpp executable was compiled with the Intel C++ compiler version 19.1 and the NVIDIA CUDA compiler version 10.2.

The inner iteration throughput metrics presented in this section were collected via 10 repeated trials. Only the resulting average and standard deviation are presented in this section. Since the variance for this metric is found to be very low, the uncertainty is not carried through to the derived metrics. The full results (including the result of each individual trial) can be seen in Appendix A.

The performance of the serial solver for each problem is seen in Table 4. This performance forms the baseline for comparison against COMET-cpp.

Table 4: Serial Solver Performance

	VHTR	I^2S	AHTR
Iteration throughput (inner its./s)	0.10979 ± 0.00002	0.08031 ± 0.00012	0.02342 ± 0.00014
Matrix bandwidth (GB/s)	22.13	22.28	23.56
Achieved computation speed (GFLOPs/s)	11.07	11.14	11.78

The performance of the COMET-cpp solver using the CPU-only backend is seen in Table 5.

Table 5: COMET-cpp CPU-only Performance

	Metric	VHTR	I^2S	AHTR
1 thread	Iteration throughput (inner its./s)	0.11620 ± 0.00068	0.08333 ± 0.00037	0.02325 ± 0.00016
	Speedup relative to serial solver	1.06	1.04	0.993
	Matrix bandwidth (GB/s)	23.42	23.12	23.39
	Achieved computation speed (GFLOPs/s)	11.71	11.56	11.70
48 threads	Iteration throughput (inner its./s)	2.636 ± 0.013	1.901 ± 0.019	0.5398 ± 0.0043
	Speedup relative to serial solver	**24.01**	**23.67**	**23.05**

	Metric	VHTR	I^{2}S	AHTR
	Speedup relative to 1-thread config.	22.68 **(47% eff.)**	22.81 **(48% eff.)**	23.21 **(48% eff.)**
	Matrix bandwidth (GB/s)	531.36	527.51	543.04
	Achieved computation speed (GFLOPs/s)	265.68	263.75	271.52

These results clearly demonstrate the immediate speedup resulting from shared-memory parallelism available from the restructuring of the algorithm in Chapter 3 and Section 3.1.2. The CPU-only matrix multiplication function represents one of the simplest batched-GEMV implementations (simply OpenMP parallel-for looping over individual GEMV BLAS calls – effectively a 1 line addition), but this alone is sufficient to achieve 23x speedup relative to the serial solver.

The parallel efficiency between the three problems for the all-available-threads case remained consistent at around 47-48%. Since matrix multiplication is generally a memory-bound operation, this is reasonable, as thread execution is likely being limited by memory operations. To investigate this further, a brief thread-wise scaling study was performed on the VHTR benchmark. The results of the study can be seen in Figure 19 below.

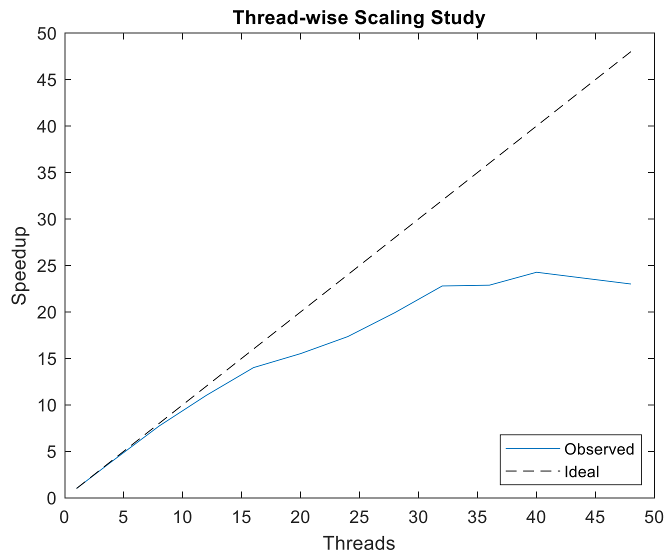


Figure 19: CPU-only Thread-wise Scaling Study

It can be seen from these results that scaling generally starts out quite near ideal (99%, 98% and 96% efficiency for 2, 4, and 8 threads respectively) but departs farther as tens of threads are used. This indicates that small numbers of threads can be effectively fed from memory, but as more and more threads are introduced, they begin to compete for bandwidth.

The performance of the COMET-cpp solver using the GPU-accelerated backend can be seen in Table 6. A plot of the overall achieved speedups can be seen in Figure 20.

Table 6: COMET-cpp GPU-Accelerated Performance

	Metric	**VHTR**	**I^{2}S**	**AHTR**
1 GPU	Iteration throughput (inner its./s)	15.710 ± 0.045	9.177 ± 0.022	2.2836 ± 0.0042
	Speedup relative to serial solver	**143.09**	**114.28**	**97.52**
	Matrix bandwidth (GB/s)	3,167	2,546	2,297
	Achieved computation speed (GFLOPs/s)	1,583	1,273	1,140
2 GPUs	Iteration throughput (inner its./s)	32.006 ± 0.056	20.924 ± 0.066	5.1520 ± 0.0068
	Speedup relative to serial solver	**291.53**	**260.55**	**220.02**
	Speedup relative to 1-GPU config.	2.04 **(102% eff.)**	2.28 **(114% eff.)**	2.26 **(113% eff.)**
	Matrix bandwidth (GB/s)	6,452 (3,226 per GPU)	5,806 (2,903 per GPU)	5,183 (2,591 per GPU)
	Achieved computation speed (GFLOPs/s)	3,226 (1,613 per GPU)	2,903 (1,451 per GPU)	2,591 (1,296 per GPU)
4 GPUs	Iteration throughput (inner its./s)	55.20 ± 0.11	36.30 ± 0.14	11.379 ± 0.021
	Speedup relative to serial solver	**502.75**	**452.05**	**485.93**
	Speedup relative to 1-GPU config.	3.51 **(88% eff.)**	3.96 **(99% eff.)**	4.98 **(125% eff.)**
	Matrix bandwidth (GB/s)	11,127 (2,782 per GPU)	10,073 (2,518 per GPU)	11,446 (2,862 per GPU)
	Achieved computation speed (GFLOPs/s)	5,563 (1,391 per GPU)	5,037 (1,259 per GPU)	5,723 (1,431 per GPU)

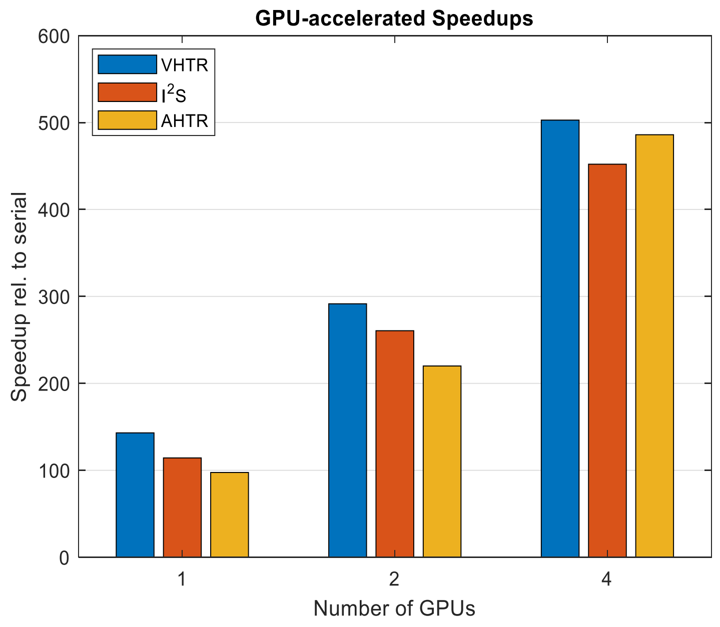


Figure 20: GPU Acceleration Relative to Serial Solver

The acceleration shown in Table 6 and Figure 20 demonstrates that the use of 1 GPU brings acceleration between 100x and 150x relative to a single CPU thread, and excellent scaling up to 2 and 4 GPUs (often super-linear, as is explored below) is observed. This result is well in line with results observed for other neutron transport programs (as in Section 2.2.3), and clearly demonstrates the data-parallel nature of the COMET method as expressed in Chapter 3. The maximum speedup achieved was 502x for the VHTR problem running with 4 GPUs, a substantial reduction in computation time.

One phenomenon apparent in Table 6 is that super-linear speedup is observed for the 2 GPU configuration of the VHTR and I^2S problems and the 2 GPU and 4 GPU configurations of the AHTR problem. Super-linear speedup is not impossible, but it is exceedingly rare. This is typically the result of an entire problem space fitting in the increased levels of cache resulting from multiple machines, a condition which is assuredly not met in the large problems being solved in this study. As such, additional profiling was performed to characterize the source of this behavior more carefully. This profiling was performed on the AHTR benchmark since it demonstrates super-linear speedup across both the 2 and 4 GPU cases.

A screenshot of an NVIDIA Visual Profiler session documenting a run of the AHTR problem on 1 GPU can be seen in Figure 21 below. The large teal block represents the kernel that performs the matrix multiplies. This dominates the runtime, accounting for upwards of 97% of GPU computation. In this run, it was found that an average matrix multiplication kernel took about 427 milliseconds (ms). The intermediate buffer summing took 2.5 ms, and other operations took 2.5 ms, for a total of around 432 ms per inner iteration.

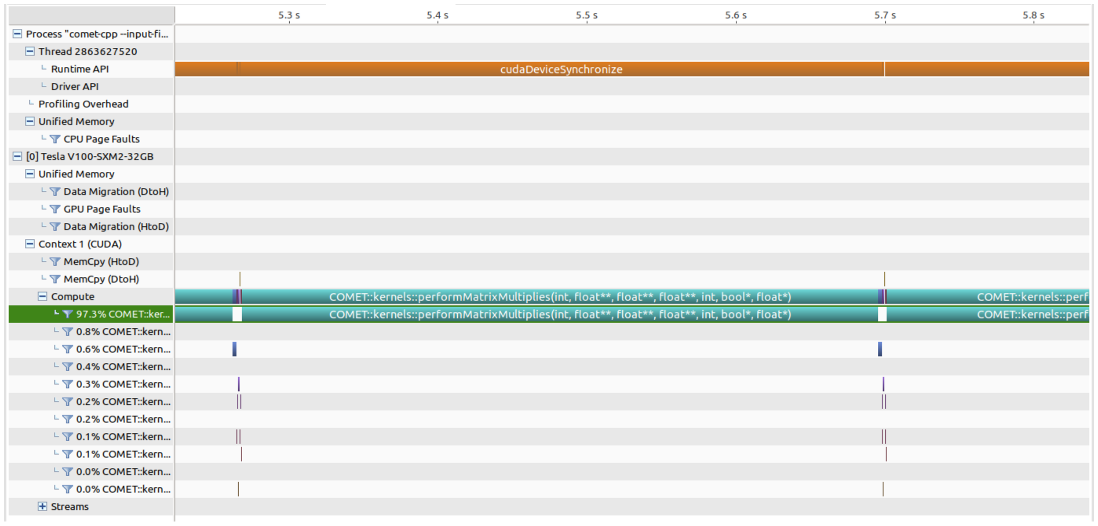
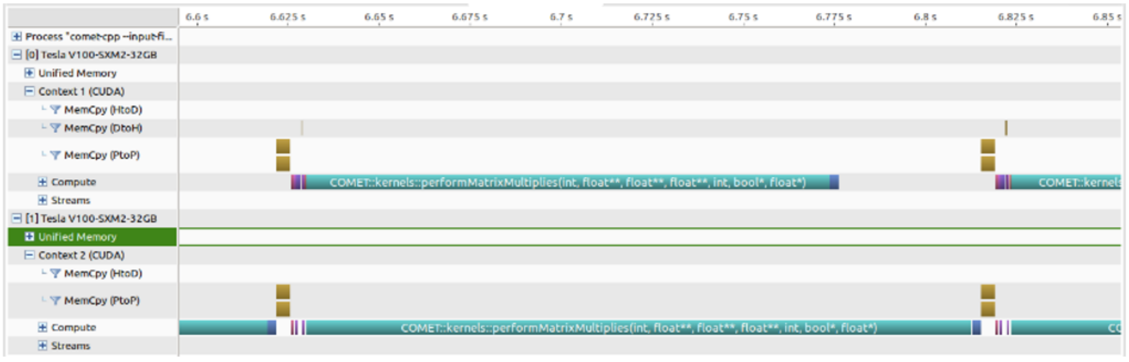


Figure 21: AHTR 1 GPU Profiling

Runs for the 2 GPU and 4 GPU configurations were examined with a similar methodology. The execution time associated with various GPU operations for these configurations (and the 1 GPU configuration) can be seen in Table 7. Screenshots of the profiler for these runs can be seen in Figure 22.

Table 7: AHTR Inner Iteration Execution times

	1 GPU	2 GPUs		4 GPUs			
		GPU 0	GPU 1	GPU 0	GPU 1	GPU 2	GPU 3
Matrix Multiplication	427 ms	144 ms	183 ms	68 ms	66 ms	67 ms	71 ms
Intermediate Summing	2.5 ms	2.5 ms	2.5 ms	2.5 ms	2.5 ms	2.5 ms	2.5 ms
Waiting	-	39 ms	-	3 ms	5 ms	4 ms	-
Communication	-	5.6 ms		10.3 ms			
Other Operations	2.5 ms	2.6 ms		2.7 ms			
Total:	432 ms	193.7 ms		86.5 ms			
Matrix Multiplication Total:	**427 ms**	**327 ms**		**272 ms**			



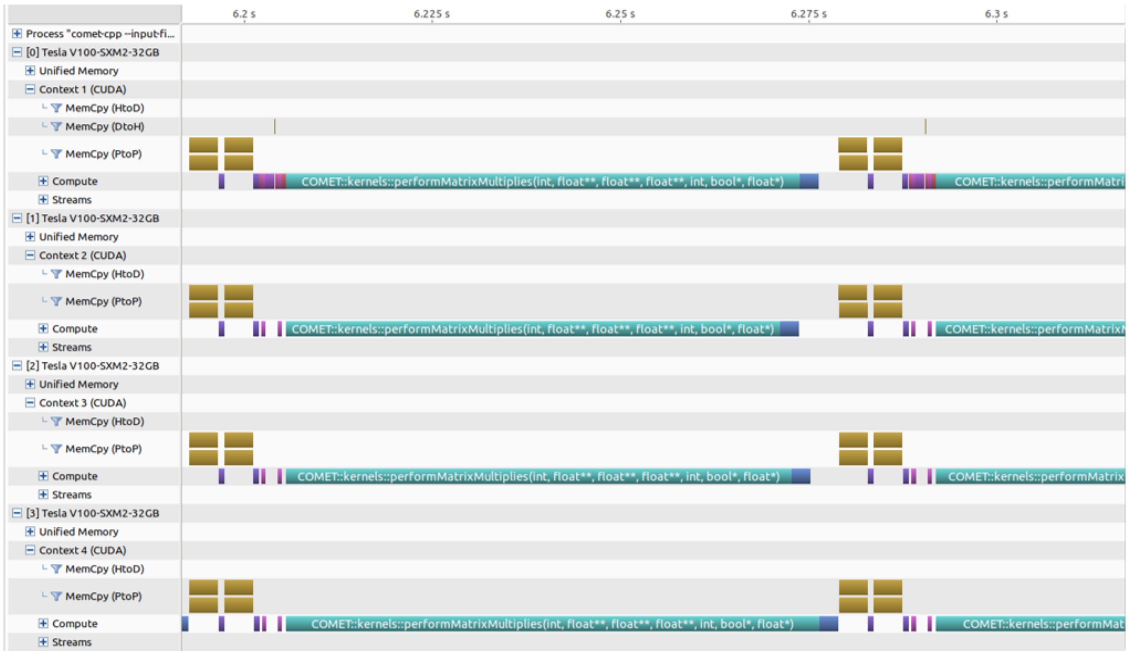


Figure 22: AHTR Multi-GPU Profiling: 2 Devices (top), 4 GPU (bottom)

It is seen from Table 7 that, although communication accounts for 2.9% and 11.9% of the total runtime per inner iteration of the 2 and 4 GPU cases, respectively, the costs are far outweighed by a drastic reduction in the total GPU time devoted to matrix multiplication, which reduces by 100 ms and 150 ms. Such a substantial reduction is notable, and not explainable by the cache effect alone. To investigate this effect, a custom one-off version of the solver was developed that breaks the list of matrix multiply operations into 4 chunks, but executes them all on 1 device. Effectively, this solver breaks up the monolithic 1 GPU kernel seen in Figure 21 into the 4 kernels seen in the bottom of Figure 22, but still dispatches them all to 1 device. Profiling data for this run can be seen in Figure 23. The execution time of the 4 separate kernels seen in Figure 23 can be seen in Table 8.

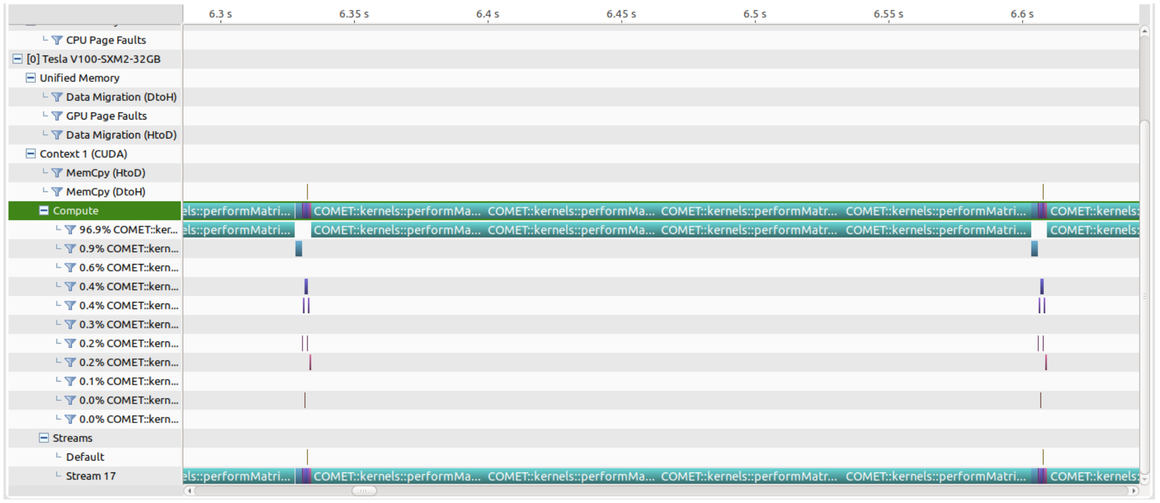


Figure 23: Small Chunk Solver Profiling

Table 8: Small Chunk Kernel Execution Times

	Batch 1	**Batch 2**	**Batch 3**	**Batch 4**	**Total**
Execution Time	66 ms	65 ms	68 ms	71 ms	270 ms

It is observed that the execution times for the 4 batches in Table 8 match up quite well with the execution times seen for the individual batches executing on 4 different GPUs in Table 7. This is an indication that the primary difference between the 1 GPU and 4 GPU matrix multiplication execution times in Table 7 is not due to a cache effect, but rather due to the specific parameters used in launching the kernels (such as the number of thread blocks in the grid relative to the number of operations in the list). As described in Chapter 3, these parameters can have a strong runtime effect and are generally optimized by fuzzing over a range of values. The results seen from the custom small-chunk solver indicate that these parameters are a poor fit for the AHTR problem on only 1 GPU. As such, the super-linear speedup observed in the general timing study is merely a relic of an inefficiency in the 1 GPU case due to the selected execution parameters being sub-optimal for these problems. To demonstrate this, Table 9 demonstrates a re-calculation of the scaling seen in Table 6 using the custom smaller-chunk solver as the new 1 GPU case. It is observed that the smaller-chunk solver has an inner iteration throughput improvement of 60% relative to the base result in Table 6.

Table 9: AHTR Scaling, Small-Chunk Solver

	1 GPU	**2 GPU**	**4 GPU**
Inner iteration Throughput	**3.584**	5.1520	11.379
Scaling relative to 1 GPU case	-	1.43x	3.17x
Parallel Efficiency	-	**71.5%**	**79.3%**

These results are more in line with what might be expected from the multi-GPU scaling. Indeed, even the 2 GPU efficiency being lower than the 4 GPU efficiency can be explained as higher workload imbalance, indicated by the large time GPU 0 waits per iteration seen in Table 7 and Figure 22 above. As such, the super-linear speedup observed in the overall results is best explained as inefficiency in the 1 GPU case.

These results are also reasonably taken to be an indication that there is further room to optimize these execution parameters. Due to the very large parameter space over which optimization would need to be done (including problem-specific characteristics, architecture characteristics, etc.) no such problem-dependent tuning of the parameters was included in the final solver. This means that super-linear speedup will be observed for some problems. A more thorough parameter optimization study is identified as an area for future work.

One final aspect of these benchmark timing results worth examining is where the raw metrics of achieved memory bandwidth and operations per second sit relative to the maximum achievable on the device. The achieved values of these metrics as seen in Table 6 are summarized and compared against architecture-specific maximum values (obtained from [2] and the NVIDIA Visual Profiler) in Table 10.

Table 10: Achieved GPU Performance Metrics

		VHTR	I^2S	AHTR
Memory Bandwidth	Achieved	2,800-3,200 GB/s	2,500-2,900 GB/s	2,300-2,900 GB/s
	Theoretical Device	900 GB/s (global memory)		
Operations per Second	Achieved	1,400-1,600 GFLOPs/s	1,200-1,500 GFLOPs/s	1,100-1,400 GFLOPs/s
	Theoretical Device	15,700 GFLOPs/s (single-precision)		

It is seen from these results that, as expected, the overall speed of the execution is limited by memory bandwidth. This is expected, as matrix multiplication has a low arithmetic intensity (FLOPs per byte of memory) and is thus generally memory bound. It is also seen from these results that, to some degree, the unified cache on the device is being effectively utilized, and the limit imposed by simply reading from global memory is well exceeded. This is an indication that the cache-aware batched-GEMV kernel developed in Chapter 3 does provide additional computational speedup relative to a naïve kernel. These raw performance metrics are seen to vary by problem, but overall indicate strong performance relative to device capabilities.

CHAPTER 5. CONVERGENCE THRESHOLD SENSITIVITY STUDY

As an example of a type of analysis unlocked by the additional computational speed of the GPU-accelerated solver, a sensitivity study was performed on the specific thresholds used as convergence criteria in controlling the inner and outer iteration processes. The purpose of this analysis is to attempt to quantify the type of statements that can be made about the error of the solution relative to the residual resulting from one step of the iteration process and how this behavior changes with respect to problem type. Although this study would have been technically possible with the serial solver (not-withstanding the lack of double-precision support), the runs would have been prohibitively expensive in terms of computation time, with the gold-standard runs alone taking up to 1-2 days (estimated) to complete. The methodology of this study is discussed in Section 5.1, with the results and interpretation discussed in 5.2.

5.1 Methodology

As described in Section 2.1.2, the COMET method consists of a nested iteration process. The inner iterations consist of a power iteration method used to converge to the dominant eigenvector of the global problem, and the outer iterations serve to update the guess for the global criticality eigenvalue *k*. Thus, two different criteria are constructed with which convergence is checked: one for the inner iteration process and one for the outer iteration process.

A natural choice for the parameter used in the inner iteration convergence criterion is the relative norm of the residual vector for the global partial current moments eigenvector. This choice is suggested by Saad [37], among others. One useful feature of this specific convergence metric is that it lends itself easily to specific thresholds that represent the limits of single-precision or double-precision calculation of around machine epsilon. These example conditions are expressed in eq. 5.1.1.

$$\frac{\left|\frac{\boldsymbol{MR}(k)\boldsymbol{J^n}}{\lambda^n} - \boldsymbol{J^n}\right|}{\left|\frac{\boldsymbol{MR}(k)\boldsymbol{J^n}}{\lambda^n}\right|} < \epsilon_J \, , \epsilon_J = \begin{cases} 10^{-7}, & single - precision \\ 10^{-14}, & double - precision \end{cases} \qquad (5.1.1)$$

The limit expressed in eq. 5.1.1 represents close to the maximum convergence that can be achieved for a given degree of precision for the numerical representation of the vectors and matrices used in calculation. A reasonable question is the degree to which this level of convergence impacts the accuracy of the solution obtained with the method. Rephrased with slightly more rigor, the guiding motivation for this study is what types of statements one can make about the *numerical error* of the solution given some information about the *residual* resulting from an iteration step.

A secondary area of interest is the convergence of the core criticality eigenvalue. It is natural to define a similar convergence metric for this parameter representing the relative change resulting from one outer iteration. Equivalent thresholds can be given to match the specific values from eq. 5.1.1, corresponding to effectively the limits of single- or double-precision numbers. Such a threshold can be seen in eq. 5.1.2.

$$\frac{|k^{n+1} - k^{n}|}{|k^{n+1}|} < \epsilon_k , \epsilon_k = \begin{cases} 10^{-7}, & single - precision \\ 10^{-14}, & double - precision \end{cases} \tag{5.1.2}$$

In this sensitivity study, the values of ϵ_J and ϵ_k were set to values logarithmically spaced between 1.77E-04 and 3.16E-07. As a gold-standard run for comparison (used as the true value for the "error" calculations), a double-precision version of the solver was developed. In this solver, all floating-point numbers (including the RF matrices, vectors, and accumulators for the total fission, absorption, and leakage) are represented in double-precision. The results in this double-precision solver were converged with ϵ_J and ϵ_k = 1.0E-13. It is estimated that these gold-standard runs alone would take around 1-2 days each with the serial solver, if double-precision capability were to be implemented.

In the solver, these thresholds can be implemented separately, and iteration ceases only when both convergence criteria have been satisfied. There is a moderate additional complexity to this, however, since the convergence behavior of the outer iterations depends strongly on the behavior of the inner iterations. As an example, take a situation in which the inner iterations have already converged. The guess for *k* is updated, and the RF matrices are updated for the new guess. After a small number of inner iterations, the relative norm of the residual vector is likely to match the level it was before the outer iteration update, and thus the inner iteration process will cease. As such, the difference between k^{n+1} and k^{n} will only be representative of the result of a very small number (less than 5, often) of inner iterations. As such, a hugely integral quantity such as the core eigenvalue is not likely to change that much, and as such it will be more likely that the convergence criterion is satisfied.

In this sensitivity study, the inner iteration process is truncated after 250 iterations per outer iteration (to prevent such vector over-converging for an under-converged eigenvalue), and ϵ_J and ϵ_k were only varied in tandem (that is, set to the same input value). This was selected as a balance between the over- and under- converge conditions. A more detailed analysis of this phenomenon (including the possibility of setting a minimum number of inner iterations, in addition to a maximum) is recommended as future work.

The results of these analyses were interpreted and analyzed with the new COMET post-processing tools developed along with the new solver as described in Section 3.3. The pin powers and eigenvalues for each run were read into a Python object. The results were then examined on both a pin-to-pin and assembly-to-assembly basis, tabulating the **relative** pin/assembly power error. These errors are combined into a set of aggregate quantities: the mean absolute error (MAE) in the pin/assembly power and the max absolute error (max AE) in pin/assembly power. Note the subtlety in this context of the word absolute. In this analysis, the word absolute refers only to the fact that it has had the absolute value operation applied. At an individual level, these errors are **relative** errors, meaning that the difference between the test value and the gold-standard value have been normalized by the gold standard value. The definitions of these parameters can be seen in eqs. 5.1.3, 5.1.4, and 5.1.5, where *N* represents the total number of pins/assemblies in the core.

$$e_{i,relative} = \frac{p_{i,test} - p_{i,GS}}{p_{i,GS}} \tag{5.1.3}$$

$$E_{MAE} = \frac{\sum_{i=1}^{N} \left| e_{i,relative} \right|}{P} \tag{5.1.4}$$

$$E_{Max\ AE} = \max\left(\left|e_{i,relative}\right|\ for\ i\ in\ N\right) \tag{5.1.5}$$

The analyses in this section were performed on the same Sawtooth cluster at Idaho National Laboratory used in Chapter 4. For maximum speed, the 4 GPU configuration was used. The convergence criteria were controlled using the currents-epsilon and eigenvalue-epsilon command line arguments for the solver, and the batch of tests was automated using a Bash script. The gold-standard runs were done with a version of the solver built with the COMET_DOUBLE_PRECISION CMake variable set to on, which converts most floating-point variables in the solver to double-precision variables.

5.2 Results

A quantitative assessment of the results is given in Section 5.2.1. A per-problem discussion of the differences between cores converged to varying tolerances is given in Section 5.2.2.

5.2.1 Overall Quantitative Results

A plot of the error in core criticality eigenvalue relative to the input threshold used for ϵ_J and ϵ_k can be seen in Figure 24. For the reasons mentioned above relating to the tangling of the inner and outer iterations, these results should be taken with a grain of salt, as the specific values are heavily dependent on parameters invisible to this analysis, such as the maximum/minimum number of inner iterations per outer iteration. The main results that should be taken away from this specific portion of the analysis is that the convergence behavior is problem-dependent, and that for the specific case of 250 maximum inner iterations per outer iteration, the error in criticality eigenvalue could be expected to vary anywhere from slightly less than the input convergence threshold to up to an order of magnitude higher than the convergence threshold.

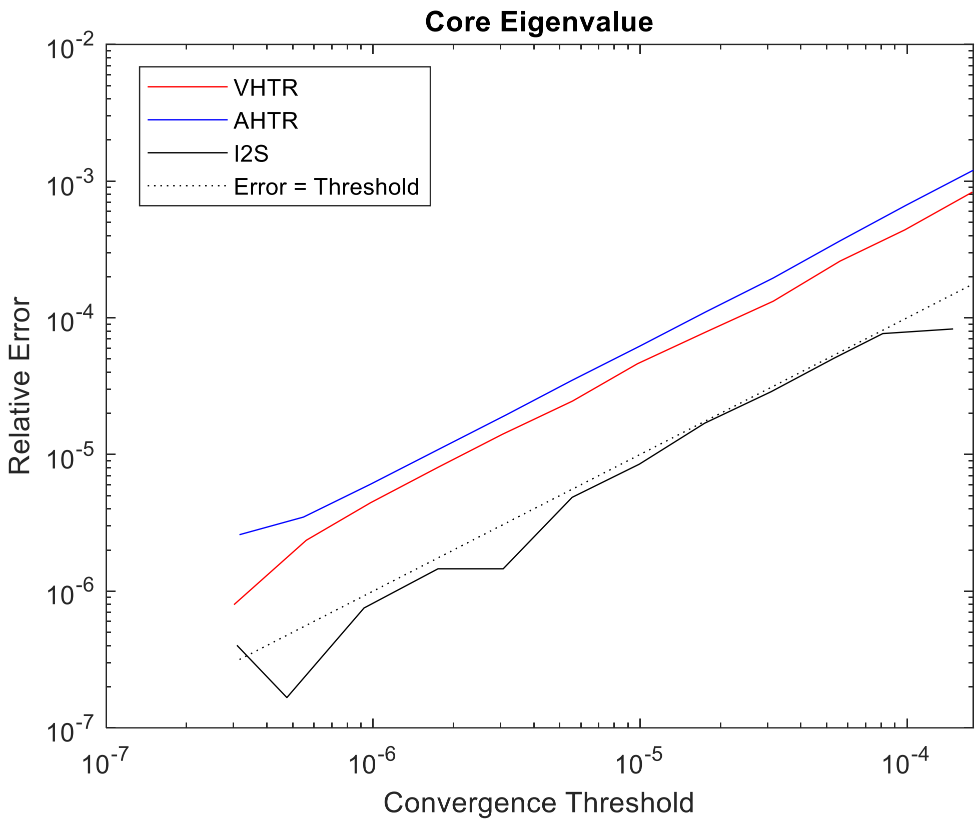


Figure 24: Criticality Eigenvalue Sensitivity

A plot of the errors (both maximum AE and mean AE) in assembly and pin power relative to the input threshold (ϵ_J and ϵ_k) can be seen in Figure 25. It can be seen from these plots that the difference between the input threshold (again, compared against the relative norm of the *residual* vector of partial current moments) and the resulting powers can be up to 3 orders of magnitude in mean error, and up to 4 orders of magnitude in maximum error. This substantial difference is due to the fact that the ratio of the second-largest and largest eigenvalues (sometimes called the dominance ratio) corresponding to the global problem represented in eq. 2.1.8. can be very high. The power iteration method converges according to this ratio [37], so convergence on the global eigenvector can be very slow. The net result of this is that, although any one individual step may be relatively small, so many of these steps are being taken that the overall error is far greater.

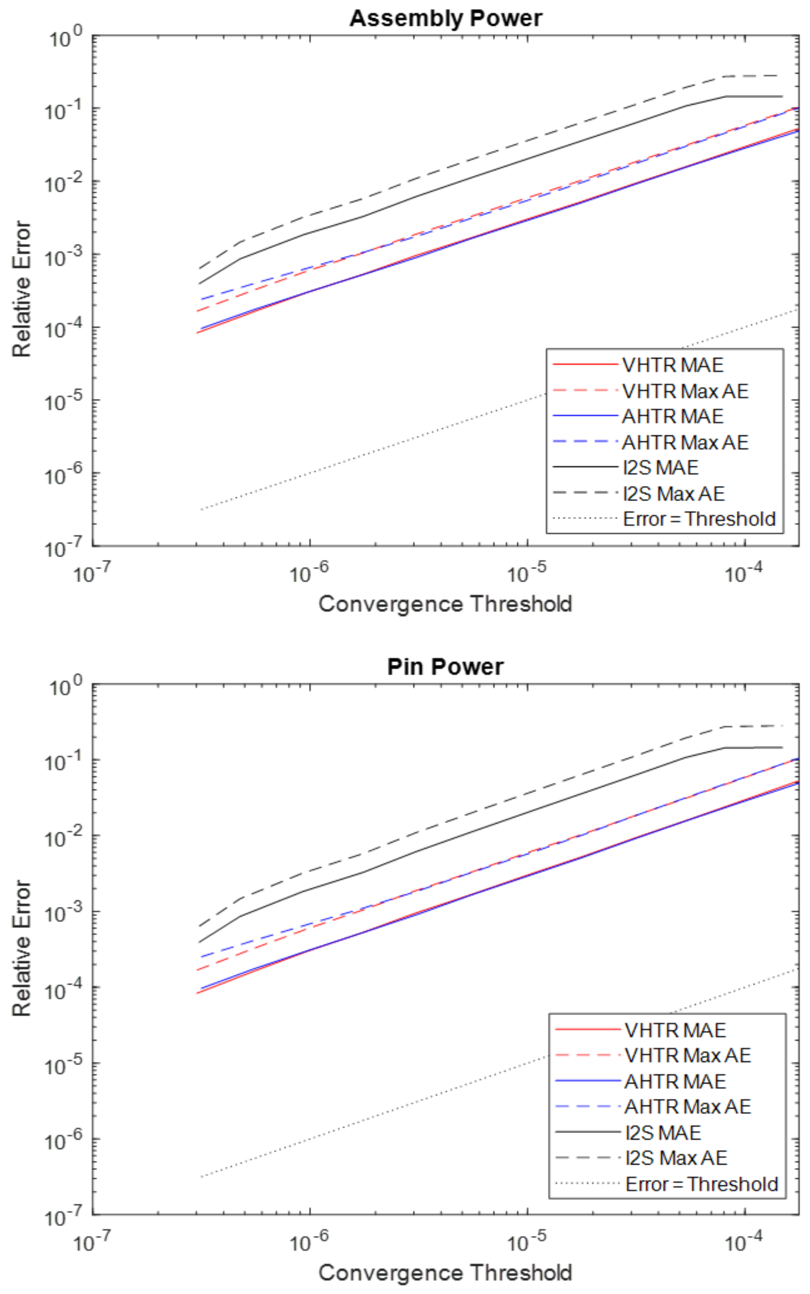


Figure 25: Assembly (top) and Pin (bottom) Power Sensitivities

These results bring a couple of interesting implications. The first is that the residual cannot be used on its own as an analogue for the numerical error, at least in terms of magnitude. Especially when it comes to maximum error in pin or assembly powers, a difference of 3-4 orders of magnitude is not negligible and must be taken explicitly into account when accounting the accuracy of the obtained solutions. The second implication derives from the first, as applied to the limits of single-precision numbers. In keeping calculated parameters in single-precision, for problems like the I^2S, accuracy in parameters like maximum pin power error can be limited to at most around 0.1%. Although the worth of that last 0.1% could be debated (and likely depends on the context in which the calculation and results are to be used), it is important to understand this limitation for versions of the solver built with only single-precision support, especially if stronger claims about the numerical error relative to a specific level of convergence are desired.

5.2.2 Per-Problem Convergence Assessment

It is also of interest to take a qualitative inventory of the differences between two cores that represent the same input problem converged to different tolerances. In this analysis, a visualization of such differences can be obtained quickly using the “differ” mode of the COMET Rendering and Output Navigation Application. This mode allows for a quick visualization of the difference between two cores in assembly power, pin powers, and 0th-order currents, in addition to the quantitative differences discussed above. The VHTR is discussed in Section 5.2.2.1. The I^2S is discussed in Section 5.2.2.2. The AHTR is discussed in Section 5.2.2.3.

5.2.2.1 VHTR

To visualize the type of error caused by under-convergence in the VHTR, figures depicting the absolute and relative pin power differences are presented for three different pairs of cores. These individual plots can be seen in Figure 26. A plot of the residual norm with respect to the cumulative inner iteration number for the problem can be seen in Figure 27.

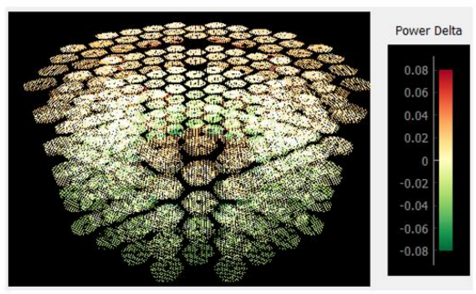
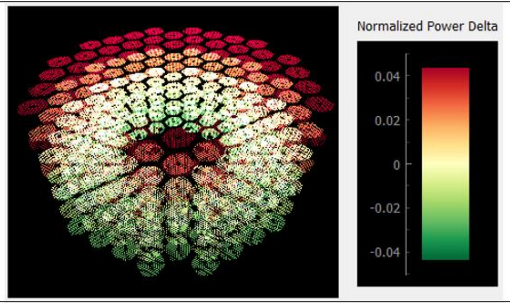
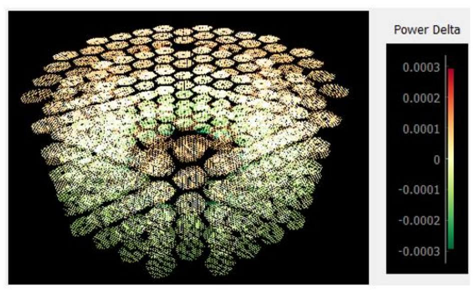
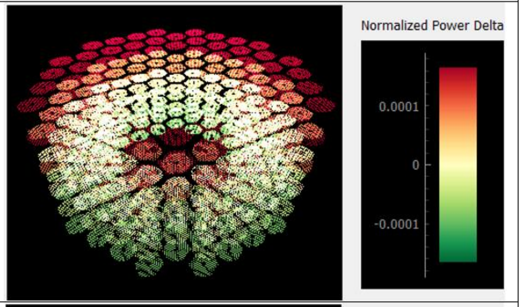
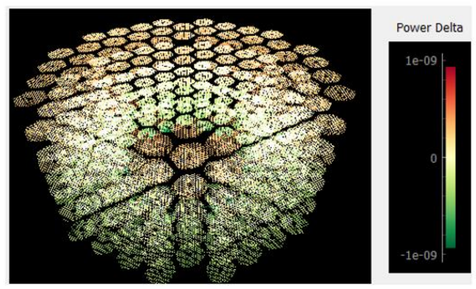
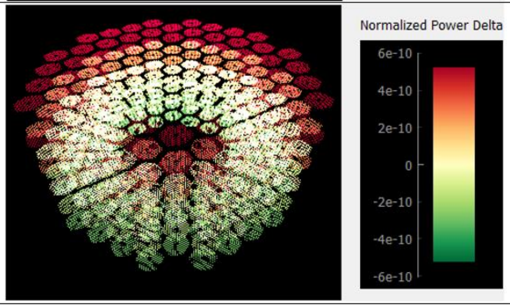
	Absolute Delta	**Relative Delta**
1.00E-04 vs. 1.78E-04	 Power Delta 0.08 0.06 0.04 0.02 0 -0.02 -0.04 -0.06 -0.08	 Normalized Power Delta 0.04 0.02 0 -0.02 -0.04
3.16E-07 vs. 5.62E-07	 Power Delta 0.0003 0.0002 0.0001 0 -0.0001 -0.0002 -0.0003	
1.00E-13 vs. 1.00E-12 (double-precision)		 Normalized Power Delta 6e-10 4e-10 2e-10 0 -2e-10 -4e-10 -6e-10

Figure 26: VHTR Pin Power Differences

The error pattern for the VHTR is observed to have a very even shape across the entire convergence range (after iteration ~500) with the main difference simply being decreasing magnitude of the error. The pattern can be described as general undershoot towards the bottom of the problem and general overshoot towards the top of the problem, with the level of error being quite consistent within any one z-plane. It is noted that this problem is reflected on the bottom, and as such this physically represents an undershoot in the middle region of the core and overshoot at the axial periphery.

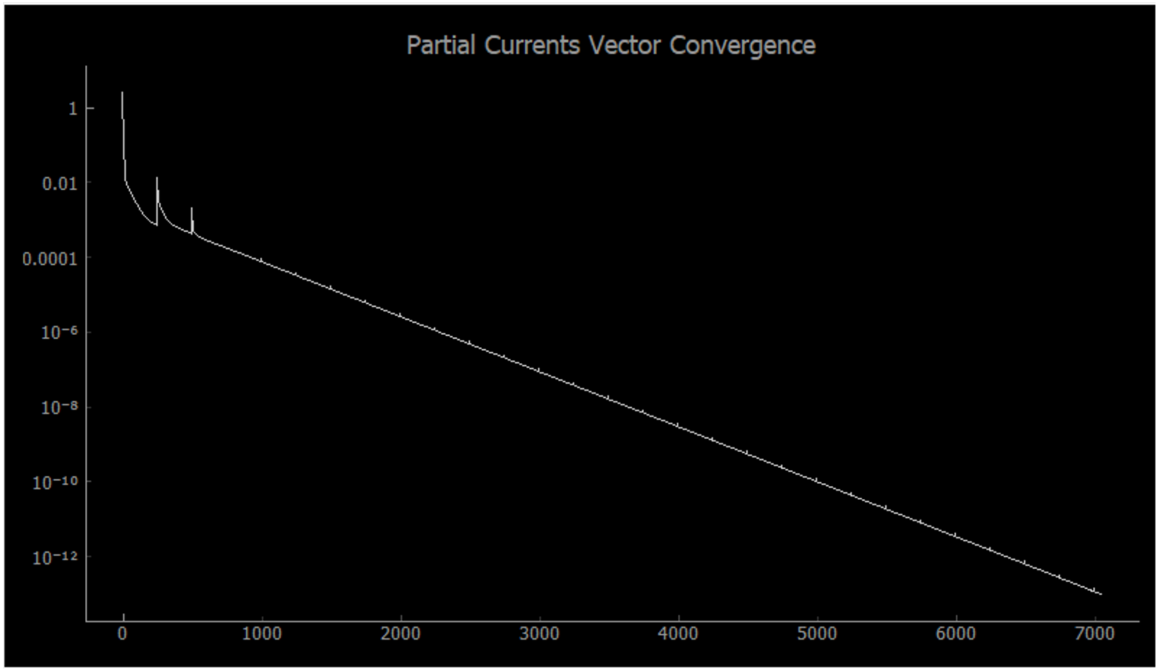


Figure 27: VHTR Partial Currents Vector Convergence

5.2.2.2 I^2S

To demonstrate this behavior for the I^2S, the same set of convergence thresholds are pictured in Figure 28. Due to the high number of pins and axial layers in the problem, the pin power plot is too dense to be effectively translated to a 2-D still image, and as such the plots in Figure 28 are of the assembly power errors instead. The pin power plots demonstrate a similar trend, so this substitute is still representative of the trend.

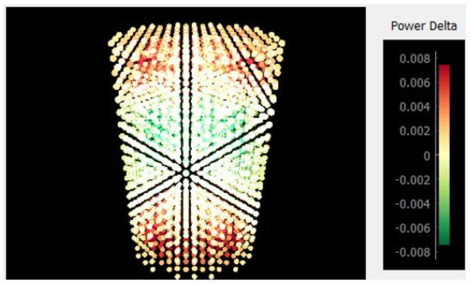
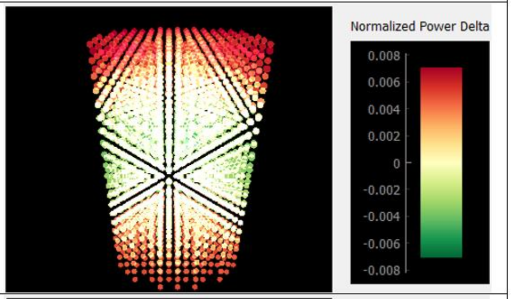
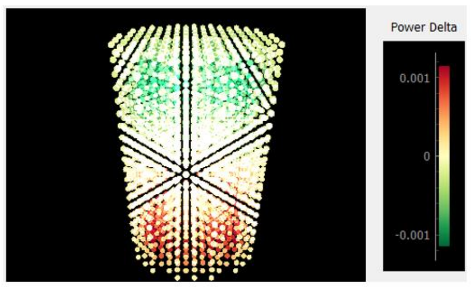
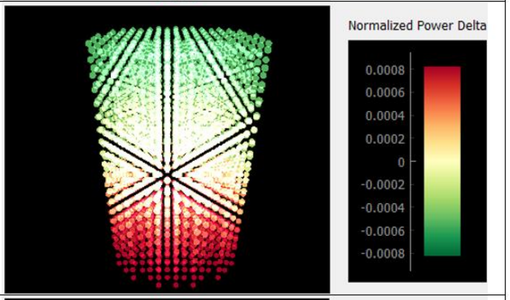
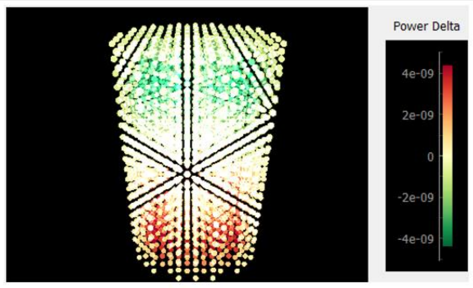
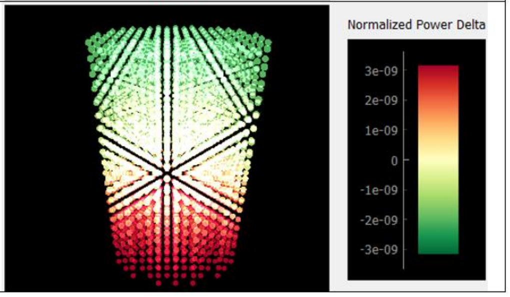
	Absolute Delta	Relative Delta
1.00E-04 vs. 1.78E-04		
3.16E-07 vs. 5.62E-07		
1.00E-13 vs. 1.00E-12 (double-precision)		

Figure 28: I^{2}S Assembly Power Differences

A plot of the residual vector norm with respect to cumulative inner iteration number is seen in Figure 29. This plot combines with the patterns in Figure 28 to give us a clearer picture of the convergence process. The first row of Figure 28 (indicating the comparison between thresholds of 1.00E-04 and 1.78E-04) depicts a noticeably different error pattern than the remaining rows. Examining Figure 24 reveals that, due in part to the Chebyshev acceleration used in this problem, a convergence threshold this loose results in a "converged" problem that has not yet actually converged out the components of the vector guess due to the eigenpairs of more than the second-largest magnitude. As such, a very different error pattern is observed, likely corresponding to the contribution of these lesser-magnitude eigenvectors. After the final region is reached, however, a regular pattern is reached (indicating the difference between the dominant and second-most dominant eigenvectors) as seen in the bottom two rows of the figure. This pattern is similar (although inverted) to the pattern seen above for the VHTR. This is notable since the I^{2}S problem is not reflected as the VHTR problem is.

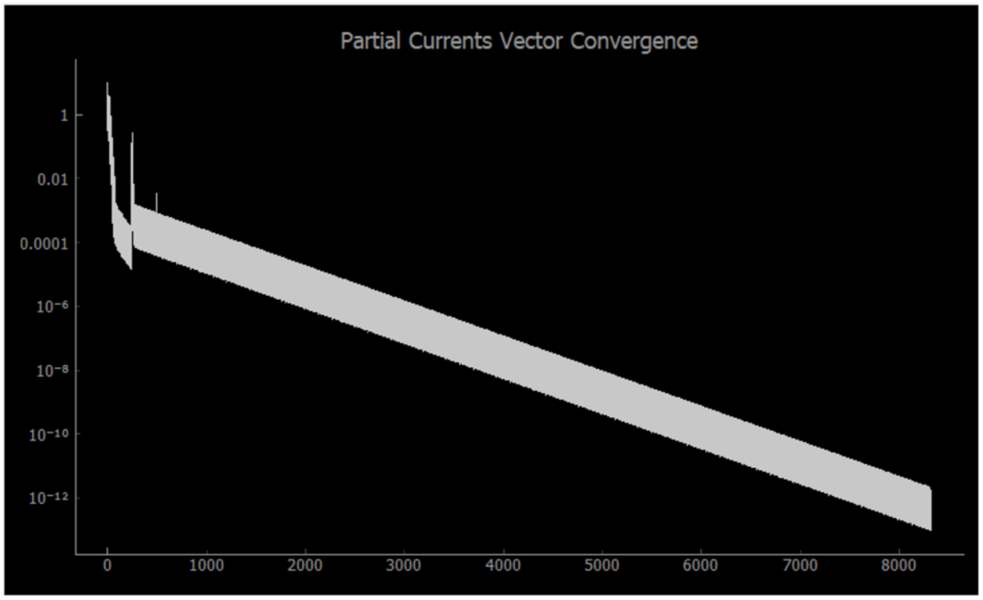


Figure 29: I^{2}S Partial Currents Vector Convergence

5.2.2.3 <u>AHTR</u>

The behavior of the convergence of the partial currents vector for the AHTR problem has a bit more nuance. To begin the discussion of this behavior, it is of use to first examine the plot of the residual vector norm with respect to cumulative inner iteration number, seen in Figure 30. In contrast to the previous two problems, this plot demonstrates a clear intermediate region between iterations ~500 and ~4200. In this region, it is likely that the component of the vector guess due to the third-largest eigenpair is still being converged out. This component is not fully reduced until around iteration ~4200, at which point the component due to the second-largest eigenpair begins to dominate convergence. This claim can be bolstered by examining the differences between the solutions in the two different regions.

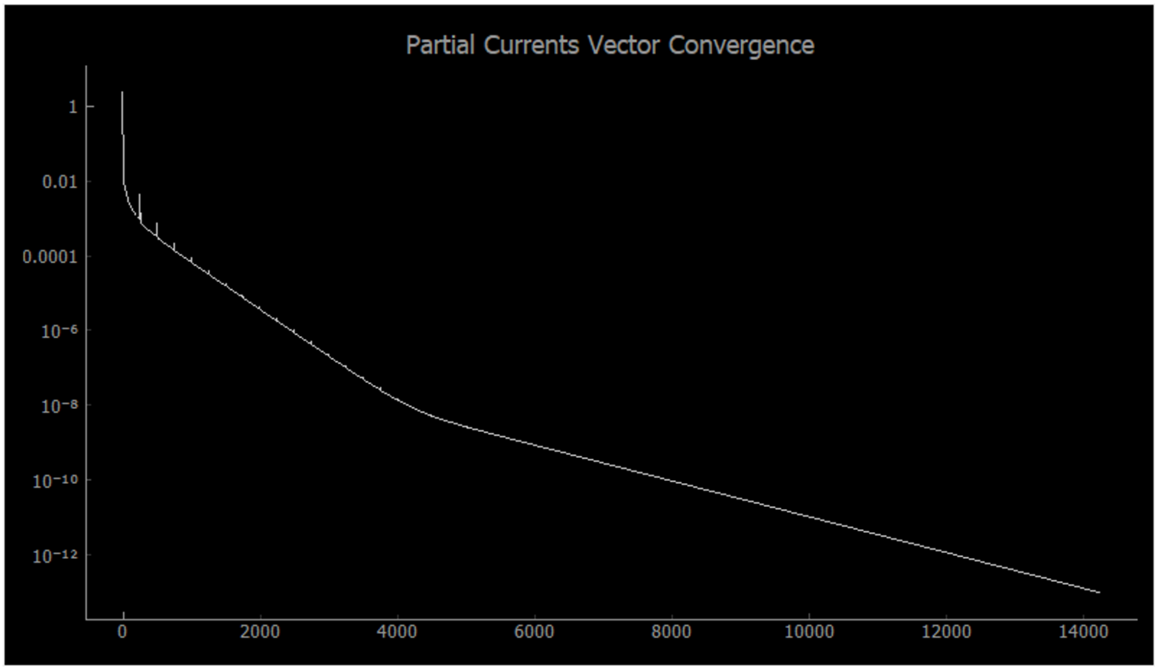


Figure 30: AHTR Partial Currents Vector Convergence

A plot of the relative assembly power differences (with the plane corresponding to Z = 9 highlighted) between solutions converged to 3.16E-07 and 5.62E-07 can be seen in Figure 31. This error pattern is quite interesting relative to those observed in the VHTR and I^{2}S analyses above. Rather than being flat within one z-plane, it is relatively constant *between* z-planes, mostly varying *within* a z-plane. Within these planes, the general pattern is under-prediction in the center of the plane and over-prediction towards the periphery of the plane. By contrast, a plot of the relative assembly power differences with the same plane highlighted for solutions converged to 1.0E-13 and 1.0E-12 can be seen in Figure 32.

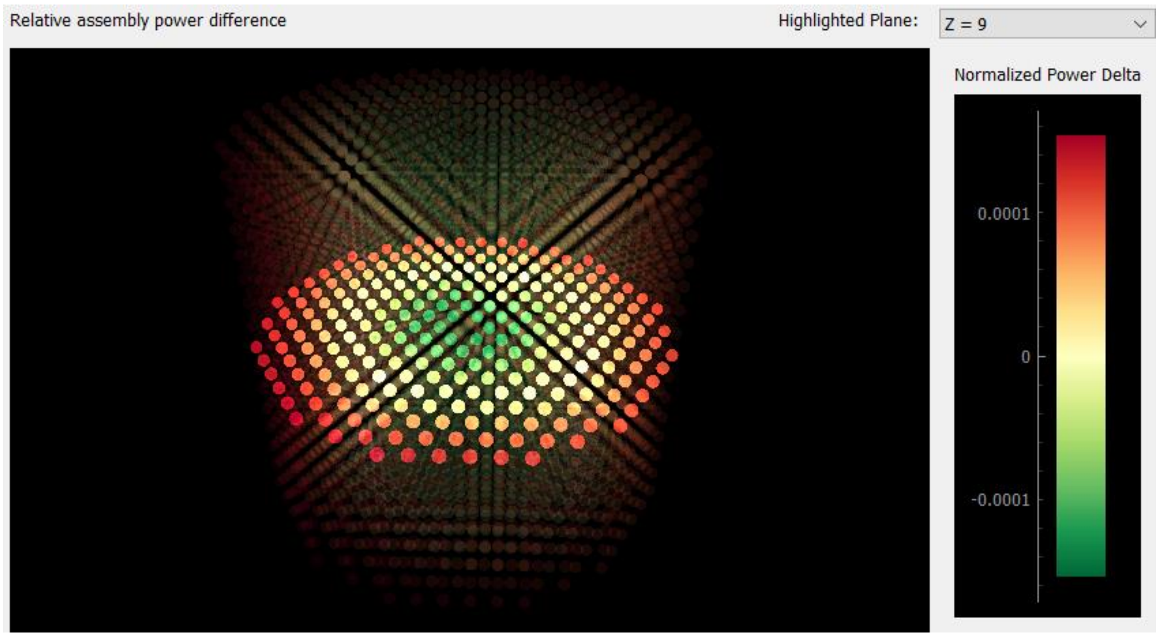


Figure 31: AHTR Assembly Power Relative Differences, 3.16E-07 vs. 5.62E-07

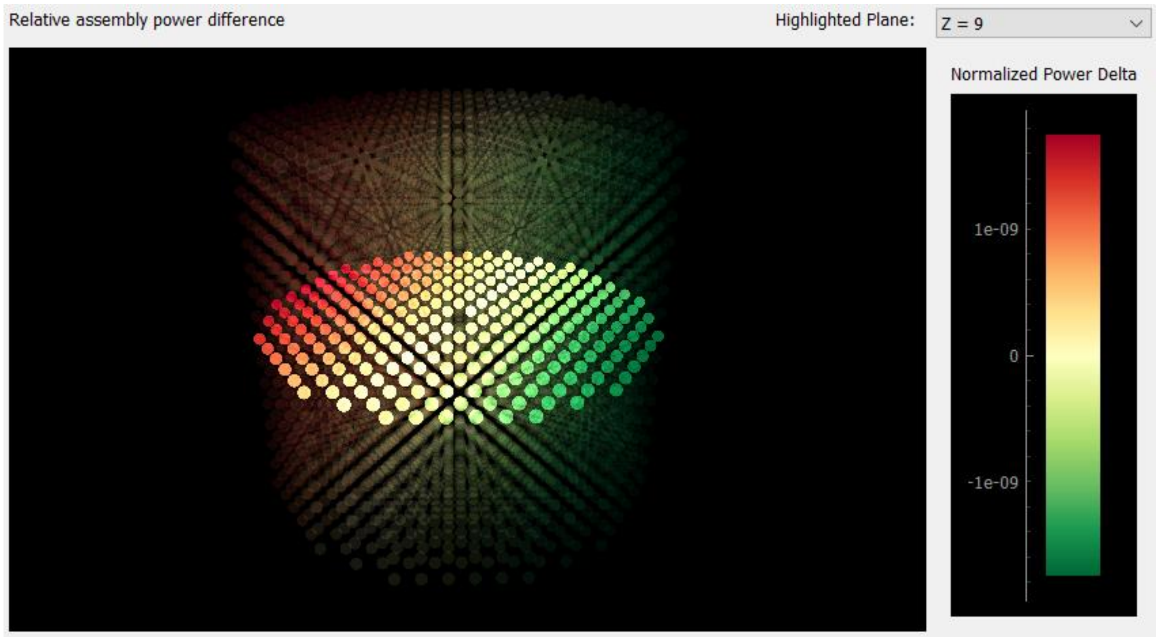


Figure 32: AHTR Assembly Power Relative Differences, 1.0E-13 vs. 1.0E-12

The error pattern seen in Figure 32 is clearly different from that seen in Figure 31, indicating that this is an entirely different regime of convergence, bolstering the claim above that this represents a reduction in largely different components of the vector guess. Like the VHTR and I^{2}S primary convergence patterns, this pattern is monotonic across one axis, starting with over-prediction and ending with under-prediction. However, unlike those problems, the dimension of variance is a radial dimension, rather than the axial dimension. The pattern is, in fact, relatively consistent across multiple z-planes, and only strongly varying within the planes.

This phenomenon is quite notable because the transition in convergence region occurs **after** the vector has converged past the limit of single-precision computing which occurs around 10^{-7} in relative residual norm. This means that single-precision versions of the solver are inherently reasonably limited to the region of the problem in which there are contributions to the guess at the primary eigenvector not only in the direction of the second-most dominant eigenvector, but also the third-most dominant eigenvector. As such, attempts to estimate the ratio of $\left|{}^{\lambda_2}/_{\lambda_1}\right|$ by the convergence behavior would result not in the true value of this ratio, and would in fact likely be closer to the ratio $\left|{}^{\lambda_3}/_{\lambda_1}\right|$.

Now, it must be re-iterated that, per Figure 24 and Figure 25, in this region of convergence, the error in the criticality eigenvalue and the powers themselves have already reached values of less than a fraction of a percent. As such, the use of single-precision numbers does not pose a challenge to the overall correctness of the solution. Rather, there is more to the overall convergence behavior that is not captured.

CHAPTER 6. SUMMARY, CONCLUSIONS, AND FUTURE WORK

In this work, a new solver for the deterministic transport sweep phase of the continuous-energy COMET method was developed. The new solver leverages a slight re-ordering of operations to expose massive inherent parallelism in the method and exploits this parallelism for both CPU-only and heterogenous CPU-GPU architectures. The new solver is demonstrated on a set of 3 whole-core benchmark problems. Relative to the serial solver, the new solver is demonstrated to have a 100-150x speedup for the 1 GPU case, and 450-500x speedup for the 4 GPU case. As an example of the types of analysis enabled by the improved computational speed, a sensitivity study is performed on the specific thresholds used to control convergence of the inner and outer iteration processes. In this study, it is found that there can be up to 3-4 orders of magnitude of difference between the input threshold used to compare the relative residual norm of the eigenvector guess and the "error" of the solutions relative to a fully converged, double-precision solution. Additionally, for the largest problem, there are behaviors observed which indicate that holding the vector and matrices in single-precision may result in an incomplete picture of the convergence of the problem.

The implications of this work on the future of the continuous-energy COMET method are promising. With a 500x speedup, whole-core benchmark problems like those examined in Chapters 4 and 5 can be solved in under 1 or 2 minutes depending on the size of the problem and the specific criteria used for convergence control. The application of low-order acceleration could be expected to further reduce this runtime by 2-3x [1]. As features continue to be added to the COMET method, including time-dependent calculation, burnup/depletion analyses, and multi-physics capabilities, the computational speed provided by the GPU acceleration will be invaluable in keeping the overall computational cost of the calculations to a reasonable level. The problem-independent nature of the speedup and the fact that there is only minimal CPU-GPU communication (with no large transfers of data) within one outer iteration are particularly promising features of the method and implementation in this respect.

There are several areas of future work identified in this analysis. The first area of future work recommended by this study is a thorough parameter optimization, taking into account problem-specific and architecture-specific parameters. The super-linear speedup observed in Chapter 4 was attributed via profiling and the development of a differently-tuned solver to inefficiency in the 1 GPU case resulting from the input parameters on execution. With an adaptive tuning of these parameters, the performance of the solver in cases like this could be improved.

Additionally, as described in Chapter 3, the choice was made in this study to pursue a matrix multiplication kernel that leverages the unified cache implicitly, rather than explicitly leveraging CUDA shared memory. This study and the parameter-optimization study could each bring separate improvements in execution speed, with improvements of up to 50% or more resulting in some cases, as was shown with the smaller-chunk solver in Chapter 4.

Another area of future work recommended in this analysis stems from the convergence criteria threshold study in Chapter 5. A study that examines the interplay between the separate thresholds ϵ_J and ϵ_k used in assessing convergence of the partial currents eigenvector and core criticality eigenvalue, respectively, would be beneficial in more accurately characterizing the behavior of the core eigenvalue convergence. The intricacies of this behavior are briefly discussed in Chapter 5, but the net result is that examining the error of k with respect to the threshold ϵ_k is in general insufficient, as differences in the convergence of the partial currents eigenvector can result in situations in which this criterion is readily satisfied. This presents a confounding factor in assessing the convergence of the problem not taken into account in this work.

One additional area of future work resulting from the sensitivity study in Chapter 5 is an examination of double- or mixed-precision calculation. One of the example problems showed interesting convergence behavior that occurred after the limit of single-precision numbers. Although it is firmly demonstrated that this does not pose an issue to the correctness of the solutions, it does indicate that storing some of the variables (such as the accumulators used in matrix multiplication) in a higher precision could unlock other behavior masked by floating-point round-off.

The final area of future work recommended by this study is to leverage the newly developed separation between the front-end and back-end. With this separation, the workload required to port the solver to new architectures or implement new solution methods is significantly reduced. As an example, it could be interesting to attempt to use other eigenproblem solution methods via a linear algebra package such as Trilinos/Anasazi [38].

With the results of this work, the continuous-energy COMET method is set to take advantage of GPU architectures as they continue to become more prevalent in the high-performance computing landscape. The implementation developed in this work uses the tremendous data-parallel computation power of these devices to deliver transport-quality solutions to whole-core problems with great speed. With this capability, the COMET method is well set up to continue into the future with additions for time-dependent, multi-physics, and other extended calculation capabilities.

APPENDIX A: RAW BENCHMARK TIMING DATA

Table 11: Raw VHTR Benchmark Timing Data

Config:	Serial	CPU-only 1 thread	CPU-only 48 threads	1 GPU	2 GPUs	4 GPUs
Num CUs:	1	1	48	1	2	4
Run 1 throughput	0.10973	0.11699	2.61935	15.80923	31.95143	55.06714
Run 2 throughput	0.1098	0.11698	2.63678	15.76372	32.11376	55.37819
Run 3 throughput	0.1098	0.117	2.62725	15.72372	32.05961	55.26159
Run 4 throughput	0.109795	0.11703	2.63594	15.70919	32.04183	55.33927
Run 5 throughput	0.1098	0.11559	2.61751	15.68204	32.0248	55.24175
Run 6 throughput	0.10977	0.11568	2.65341	15.69351	31.99846	55.20074
Run 7 throughput	0.109774	0.11567	2.63498	15.69367	32.0124	55.19388
Run 8 throughput	0.109793	0.11568	2.63943	15.66691	31.9398	55.12877
Run 9 throughput	0.1098	0.11572	2.65668	15.6674	31.94633	55.11327
Run 10 throughput	0.1098	0.11568	2.63726	15.68642	31.97473	55.02926
Average:	0.109786	0.116202	2.635859	15.709581	32.006315	55.195386
Std. dev:	0.000023	0.000688	0.012655	0.045298	0.055896	0.113485
Std. dev as %:	0.02%	0.59%	0.48%	0.29%	0.17%	0.21%

Table 12: Raw I^{2}S Benchmark Timing Data

Config:	Serial	CPU-only 1 thread	CPU-only 48 threads	1 GPU	2 GPUs	4 GPUs
Num CUs:	1	1	48	1	2	4
Run 1 throughput	0.080463	0.083773	1.87894	9.19283	20.87047	36.06554
Run 2 throughput	0.080013	0.083772	1.90353	9.177073	20.87325	36.22335
Run 3 throughput	0.080191	0.08353	1.9042	9.18145	20.97271	36.404011
Run 4 throughput	0.080362	0.083918	1.85916	9.19324	20.97791	36.4007
Run 5 throughput	0.080367	0.082982	1.91536	9.18459	20.98336	36.40626
Run 6 throughput	0.080339	0.08303	1.92602	9.14465	20.83576	36.122861
Run 7 throughput	0.080327	0.083046	1.91023	9.18517	20.96023	36.42828
Run 8 throughput	0.080351	0.083075	1.89821	9.19331	20.97359	36.39222
Run 9 throughput	0.080349	0.083101	1.90655	9.18895	20.97588	36.41024
Run 10 throughput	0.080307	0.083098	1.90897	9.12979	20.81807	36.17539
Average:	0.08030690	0.08333250	1.90111700	9.177105	20.9241	36.3028
Std. dev:	0.00012291	0.00037123	0.01907075	0.021965	0.06642	0.14025
Std. dev as %:	0.15%	0.45%	1.00%	0.24%	0.32%	0.39%

Table 13: Raw AHTR Benchmark Timing Data

Config:	Serial	CPU-only 1 thread	CPU-only 48 threads	1 GPU	2 GPUs	4 GPUs
Num CUs:	1	1	48	1	2	4
Run 1 throughput	0.0237954	0.0236718	0.54523	2.29326	5.16584	11.41408
Run 2 throughput	0.0234011	0.0232015	0.54614	2.28739	5.1584	11.40838
Run 3 throughput	0.0233917	0.0231473	0.54001	2.2858	5.15411	11.3886
Run 4 throughput	0.0233849	0.023155	0.53867	2.2813	5.15359	11.38785
Run 5 throughput	0.0233862	0.0232266	0.53817	2.28331	5.15387	11.38383
Run 6 throughput	0.023396	0.0232	0.542285	2.28029	5.14581	11.36889
Run 7 throughput	0.0233497	0.023301	0.53393	2.28115	5.14926	11.36095
Run 8 throughput	0.0233322	0.023171	0.53288	2.27903	5.14293	11.3612
Run 9 throughput	0.023365	0.023145	0.53983	2.28183	5.14571	11.35778
Run 10 throughput	0.0233584	0.023316	0.54114	2.28238	5.15006	11.35435
Average:	0.0234160	0.02325352	0.5398285	2.283574	5.151958	11.378591
Std. dev:	0.0001351	0.00015881	0.0042683	0.004221	0.006785	0.021314
Std. dev as %:	0.58%	0.68%	0.79%	0.18%	0.13%	0.19%

REFERENCES

[1] F. Rahnema and D. Zhang, "Continuous energy coarse mesh transport (COMET) method," *Annals of Nuclear Energy,* vol. 115, pp. 601-610, 2018.

[2] NVIDIA Corporation, *NVIDIA Tesla V100 GPU Architecture*, 2017.

[3] K. E. Jones, "Farewell, Titan," Oak Ridge National Laboratory, Oak Ridge, TN, 2019.

[4] S. Vazhkudai, B. de Supinski, A. Bland, A. Geist, J. Sexton, J. Kahle, C. Zimmer, S. Atchley, S. Oral, D. Maxwell, V. Vergara Larrea, A. Bertsch, R. Goldstone, W. Joubert and C. Chambreau, "The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems," in *Scientific Computing 2018*, Dallas. TX, 2018.

[5] Oak Ridge National Laboratory, "U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL," Oak Ridge National Laboratory, Oak Ridge, TN, 2019.

[6] K. Remley, "Development of Methods for High Performance Computing Applications of the Deterministic Stage of COMET Calculations," Georgia Tech Library, Atlanta, GA, 2016.

[7] Space Physics Data Facility: NASA Goddard Space Flight Center, *Common Data Format (CDF)*, Greenbelt, MD, 2021.

[8] G. Fedorov, K. Nguyen, P. Harrison and A. Singh, "Intel(R) Math Kernel Library Release Notes and New Features," Intel Corporation, 16 July 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-math-kernel-library-release-notes-and-new-features.html.

[9] NVIDIA Corporation, *NVIDIA A100 Tensor Core GPU Architecture*, 2020.

[10] Advanced Micro Devices, Inc. , *AMC CDNA Architecture*, 2020.

[11] NVIDIA Corporation, *CUDA C++ Programming Guide v11.2.2*, 2021.

[12] N. Sakharnykh, "Maximizing Unified Memory Performance in CUDA," NVIDIA Corporation, 19 November 2017. [Online].

[13] OpenACC.org, "OpenACC Programming and Best Practices Guide," openacc-standard.org, 2015.

[14] H. C. Edwards and C. Trott, "Kokkos, Manycore Device Performance Portability for C++ HPC Applications," in *GPU Technology Conference 2015*, San Jose, CA, 2015.

[15] K. Li, "OpenMP Accelerator Support for GPUs," OpenMP, 2017.

[16] GPUOpen Professional Compute, "It's HIP to be Open," Advanced Micro Devices, 2016.

[17] R. M. Bergmann and J. L. Vujic, "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, 2014.

[18] 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, 1984.

[19] R. M. Bergmann, K. L. Rowland, N. Radnovic, R. N. Slaybaugh and J. L. Vujic, "Performance and accuracy of criticality calculations performed using WARP – A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs," *Annals of Nuclear Energy*, vol. 103, pp. 334-349, 2017.

[20] T. M. Pandya, S. R. Johnson, T. M. Evans, G. G. Davidson, S. P. Hamilton and A. T. Godfrey, "Implementation, capabilities, and benchmarking of Shift, a massively

parallel Monte Carlo radiation transport code," *Journal of Computational Physics,* vol. 308, pp. 239-272, 2016.

[21] S. Hamilton and T. Evans, "Continuous-energy Monte Carlo neutron transport on GPUs in the Shift code," *Annals of Nuclear Energy,* vol. 128, pp. 236-247, 2019.

[22] 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, 2014.

[23] Y. S. Jung, C. B. Shim, C. H. Lim and H. G. Joo, "Practical numerical reactor employing direct whole core neutron transport and subchannel thermal/hydraulic solvers," *Annals of Nuclear Energy,* vol. 62, pp. 357-374, 2013.

[24] N. Choi, J. Kang and H. Joo, "Preliminary Performance Assessment of GPU Acceleration Module in nTRACER," in *Transactions of the Korean Nuclear Society Autumn Meeting 2018*, Yeosu, Korea, 2018.

[25] C. Silverstein and A. Schuh, "How To Use gflags (formerly Google Commandline Flags)," 14 January 2021. [Online]. Available: https://gflags.github.io/gflags/. [Accessed 20 March 2021].

[26] Z. Xianyi and M. Kroeker, "OpenBLAS: An optimized BLAS library," 13 December 2020. [Online]. Available: http://www.openblas.net. [Accessed 20 March 2021].

[27] K. Connolly, F. Rahnema and P. Tsvetkov, "Prismatic VHTR neutronic benchmark problems," *Nuclear Engineering and Design,* vol. 285, pp. 207-240, 2015.

[28] D. Zhang and F. Rahnema, "COMET neutronics solutions to the prismatic VHTR benchmark problem," *Nuclear Engineering and Design,* vol. 358, 2020.

[29] S. Gupta, "What Is NVLink? And How Will It Make the World's Fastest Computers Possible?," 14 November 2014. [Online]. Available: https://blogs.nvidia.com/blog/2014/11/14/what-is-nvlink/. [Accessed 23 March 2021].

[30] Z. Zhao, "Introducing Batch GEMM Operations," 14 September 2017. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/introducing-batch-gemm-operations.html. [Accessed 23 March 2021].

[31] NVIDIA Corporation, "CUDA Toolkit Documentation," NVIDIA Corporation, 4 August 2020. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html.

[32] L. Campagnola, "PyQtGraph: Scientific Graphics and GUI Library for Python," University of North Carolina at Chapel Hill, 2021. [Online]. Available: http://www.pyqtgraph.org/. [Accessed 2021 28 March].

[33] R. Hon, G. Kooreman, F. Rahnema and B. Petrovic, "Stylized whole-core benchmark of the Integral Inherently Safe Light Water Reactor (I2S-LWR) concept," *Annals of Nuclear Energy,* vol. 100, pp. 12-22, 2017.

[34] D. Zhang and F. Rahnema, "Monte Carlo solution convergence analysis via COMET in a stylized integral inherently safe LWR benchmark problem," *Annals of Nuclear Energy,* vol. 100, pp. 3-11, 2017.

[35] D. Zhang and F. Rahnema, "A stylized 3D Advanced High Temperature Reactor (AHTR) benchmark problem," *Annals of Nuclear Energy,* vol. 120, pp. 178-185, 2018.

[36] D. Zhang and F. Rahnema, "Continuous-energy COMET solution to the stylized AHTR benchmark problem," *Annals of Nuclear Energy,* vol. 121, pp. 284-294, 2018.

[37] Y. Saad, Numerical Methods for Large Eigenvalue Problems, 2nd ed., Minneapolis, Minnesota: Society for Industrial and Applied Mathematics, 2011.

[38] C. Baker, U. Hetmaniuk, R. Lehoucq and H. Thornquist, "Anasazi software for the numerical solution of large-scale eigenvalue problems," *ACM Transactions on Mathematical Software*, vol. 36, no. 3, 2009.