Algorithmic Choices in WARP - A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs

Ryan M. Bergmann*, Jasmina L. Vujić

Department of Nuclear Engineering, 4155 Etcheverry Hall, University of California - Berkeley, Berkeley, CA 94703-1730

Abstract

In recent supercomputers, general purpose graphics processing units (GPUs) are a significant fraction of the supercomputer's total computational power. GPGPUs have different architectures compared to central processing units (CPUs), and for Monte Carlo neutron transport codes used in nuclear engineering to take advantage of these coprocessor cards, transport algorithms must be changed to execute efficiently on them. WARP is a continuous energy Monte Carlo neutron transport code that has been written to do this. The main thrust of WARP is to adapt previous event-based transport algorithms to the new GPU hardware; the algorithmic choices for all parts of which are presented in this paper. It is found that remapping history data references increases the GPU processing rate when histories start to complete. The main reason for this is that completed data are eliminated from the address space, threads are kept busy, and memory bandwidth is not wasted on checking completed data. Remapping also allows the interaction kernels to be launched concurrently, improving efficiency. The OptiX ray tracing framework and CUDPP library are used for geometry representation and parallel dataset-side operations, ensuring high performance and reliability.

Keywords: Monte Carlo, Neutron Transport, GPU, CUDA, CUDPP, OptiX

*Corresponding author. Tel.: +1 320 333 3089.

Email addresses: ryanbergmann@gmail.com (Ryan M. Bergmann), vujic@nuc.berkeley.edu (Jasmina L. Vujić)

1. Introduction

Graphics processing units, or GPUs, have gradually increased in computational power from the small, job-specific boards of the early 1990s to the programmable powerhouses of today. Compared to more common central processing units, or CPUs, GPUs have a higher aggregate memory bandwidth, much higher rate of floating-point operations per second (FLOPS), and lower energy consumption per FLOP. Because one of the main obstacles in exascale computing is power consumption, many new supercomputing platforms are gaining much of their computational capacity by incorporating GPUs into their compute nodes. Since CPU-optimized parallel algorithms are not directly portable to GPU architectures (or at least not without losing substantial performance), particle transport codes need to be rewritten to execute efficiently on GPUs. Unless this is done, reactor simulations cannot take full advantage of these new supercomputers. In this work, the algorithmic choices used in WARP and their consequences are presented. The ultimate accuracy of WARP, comparisons against other neutron transport codes, and benchmarking are not presented here. This work only concerns algorithmic decisions based on self-comparison. WARP benchmarking, accuracy and speedup comparison with other Monte Carlo codes will be presented in a companion paper.

WARP, which can stand for “Weaving All the Random Particles,” is a three-dimensional (3D) continuous energy Monte Carlo neutron transport code developed at UC Berkeley to efficiently execute on a CPU/GPU platform. WARP accelerates Monte Carlo simulations while preserving the benefits of using the Monte Carlo method, namely, that very few physical and geometrical simplifications are applied. WARP is able to calculate multiplication factors, flux tallies, and fission source distributions for time-independent problems, and can run in both criticality or fixed source modes. WARP can currently transport neutrons in unrestricted arrangements of parallelepipeds, hexagonal prisms, cylinders, and spheres.

The impetus for developing WARP was the research done by Martin and Brown in 1984 [1] for Monte Carlo and by Vujić and Martin in 1991 [2] for collision probability method. In the 1984 paper, a method for mapping the Monte Carlo problem onto SIMD (single instruction multiple data) vector computers is described. SIMD is an execution model some processors use in

36 order to lower the number of instructions needed per amount of computation
37 done, which increases both power and computational efficiency [3]. SIMD
38 requires the same instructions to be carried out over every element in a
39 concurrently-processed data vector. The essential idea in the 1984 paper is
40 to bank the neutrons into vectors based on their required operation. If a
41 neutron is scattering, its data is placed in the scattering buffer. If a neutron
42 needs to do a surface crossing, it is put in the crossing buffer, and so on.
43 Once a buffer becomes full, it is processed in a SIMD fashion by the vector
44 computer. Processing the neutrons makes the buffers contain non-uniform
45 reactions, however, and a “shuffle” operation is done that actually moves
46 data back into contiguous blocks based on the reaction type.

47 This new approach was named “event-based” Monte Carlo, since the
48 neutron events are tracked and processed as a group [1]. This was a very
49 different way of performing a Monte Carlo simulation at the time. Almost all
50 computers were strictly serial, and SIMD lanes were only available in super-
51 computers. Therefore, the pervasive method was the “task-based” method
52 in which neutrons are tracked for their entire lifetime in series. Since GPUs
53 are massively parallel and rely on SIMD, an event-based algorithm seems to
54 be the appropriate approach to GPU-accelerated neutron transport.

55 WARP uses an event-based algorithm, but with some important differ-
56 ences. Vectorizing the Monte Carlo algorithm should allow for efficient GPU
57 execution, but a paper by Zhang [4] also shows that by remapping data refer-
58 ences, the thread divergence in GPU warps can be minimized. Moving data
59 is expensive, so WARP uses a remapping vector of pointer/index pairs to di-
60 rect GPU threads to the data they need to access. The remapping vector is
61 sorted by reaction type after every transport iteration using a high-efficiency
62 parallel radix sort, which serves to keep the reaction types as contiguous as
63 possible and removes completed histories from the transport cycle. Sorting
64 reduces the amount of divergence in GPU “thread blocks,” keeps the SIMD
65 units as full as possible, and eliminates using memory bandwidth to check
66 if a neutron in the batch has been terminated or not. Using a remapping
67 vector means the data access pattern is irregular, but this is mitigated by
68 using large batch sizes where the GPU can effectively eliminate the high cost
69 of irregular global memory access.

70 WARP sets itself apart from any previous endeavors in its breath of scope
71 and its novel adaption of the event-based transport algorithm. Previous
72 codes have also either used synthetic, simplified, and/or incomplete nuclear
73 data [5, 6, 7]. WARP loads standard data files and accurately simulate each

74 reaction type specified in the data. WARP also uses a flexible, scalable,
 75 and optimized geometry representation where previous studies have used
 76 simplified and restricted geometry models [5]. Previous works have exam-
 77 ined event-parallel algorithms, but have not parallelized them effectively and
 78 therefore did not see the benefits of adopting such an algorithm on a GPU
 79 [5]. WARP uses highly-parallelized algorithms and slightly modify the origi-
 80 nal vision of the event-based algorithm to better suit execution on the GPU.
 81 The previous event-based algorithms tried to implement a “shuffle” opera-
 82 tion where neutron data was actually sorted into reaction-contiguous blocks
 83 [5], or used small, synthetic nuclear data and were not able to capture the
 84 effects of loading large nuclear datasets [6]. Other studies have been done
 85 that offload specific parts of the neutron transport routine to the GPU, such
 86 as that done on RMC at Tsinghua University [7]. They were able to obtain a
 87 113x speedup over a CPU version without a flux tally and 36x speedup with
 88 a flux tally [8]. Again, one group cross sections were used.

89 WARP also changes the unionized energy grid data format to reduce the
 90 number of data loads needed to scan cross sections. Instead of storing a
 91 matrix of pointers indexed by reaction type and energy, WARP stores three
 92 matrices. The first contains cross section values, the second contains pointers
 93 to angular distributions, and a third contains pointers to energy distributions.
 94 This linked list type of layout increases memory usage, but lowers the number
 95 of data loads that are needed to determine a reaction by eliminating a pointer
 96 load to find a cross section value.

97 Optimized, high-performance GPU code libraries are also used by WARP
 98 wherever possible. The CUDA performance primitives (CUDPP) library
 99 is used to perform the parallel reductions, sorts and sums, the CURAND
 100 library is used to seed the linear congruential random number generators,
 101 and the OptiX ray tracing framework is used for geometry representation
 102 [9, 10, 11]. OptiX is a highly-optimized library developed by NVIDIA that
 103 automatically builds hierarchical acceleration structures around user-input
 104 geometry so only surfaces along a ray line need to be queried in ray tracing
 105 [11]. WARP also performs material and cell number queries with OptiX by
 106 using a point-in-polygon like algorithm. The reaction sampling routines have
 107 been coded in CUDA according to the appropriate ENDF laws.

108 WARP is designed to read ACE-formatted data [12] and perform all re-
 109 action types as prescribed by the data. ACE is an acronym for “A Compact
 110 ENDF,” and is the format which MCNP and Serpent both read [13, 12]. ACE
 111 data is loaded with the PyNE package [14] and passed to the main C++ code

112 via the Python C application programming interface (API). WARP uses a
 113 Serpent-like unionized energy grid to regularize data access (Serpent is a
 114 Finnish Monte Carlo reactor physics code). A unionized energy grid is one
 115 where the individual energy grids of every isotope used in a simulation have
 116 been combined into a single energy vector. The cross section vectors are
 117 then interpolated on this larger unionized energy grid [13]. The host-side
 118 code in WARP is written in C/C++. Single precision floating point num-
 119 bers are used throughout in order to realize the full computational capacity
 120 of the GPU and to allow simulations to be carried out on more affordable
 121 and higher clocked GeForce cards [15, 16]. Using single precision numbers
 122 may lead to inaccuracies when there are very dilute isotopes or very rare
 123 reactions reactions present, as roundoff error may make their contributions
 124 zero. Buffer overflow and roundoff error in the tallies may also be a problem
 125 with single precision, but this can be mitigated by accumulating the tallies
 126 frequently in a double precision vector. Roundoff error may also be problem-
 127 atic in calculating the multiplication factor, but this can also be mitigated by
 128 accumulating the integer secondary neutron yield values into a large host-side
 129 variable. If double precision data is found to be needed, WARP can easily
 130 be changed, and doing so may be an interesting experiment in the future.

131 Section 2 discusses how OptiX is used to perform surface detection and
 132 query the current material a neutron is traveling through. The testing done
 133 to determine the optimal OptiX configuration is also discussed. Section 3
 134 deals with the way the nuclear data is loaded, reformatted to use a unionized
 135 energy grid, and the details about data layout and access patterns. Section 4
 136 details each individual CUDA kernel (i.e. GPU program) used by WARP to
 137 process the neutron histories. A Section 5 presents the results of the OptiX
 138 testing and performance improvements attributed to reference remapping,
 139 and conclusions based on the results are made in Section 6.

140 **2. Ray Tracing and Geometry Representation with OptiX**

141 A preliminary study was conducted to investigate how OptiX performs
 142 when ray tracing is done from randomized points and to find the optimal
 143 OptiX configuration for WARP. In rendering, rays are initialized in a uniform
 144 array, but OptiX also allows for arbitrary starting points and directions to be
 145 used. This flexibility comes from being able to write custom ray-generation
 146 programs in OptiX. OptiX processes rays concurrently, so the ray tracing in
 147 WARP is done in a batched way, i.e. the traces are done for all neutrons

148 in a single step as shown in Figure 1. Initialization, surface detection, and
 149 particle movement calculations are performed across all rays simultaneously
 150 instead of a single ray start-to-finish and then moving on to the next ray.

Figure 1: Ray tracing and surface detection done in a batched, data-parallel way.

151 There are two types of scaling in OptiX that need to be characterized in
 152 order to ensure optimal performance – scaling with regard to the number of
 153 concurrent rays traced and scaling with regard to the number of geometrical
 154 objects in the scene. Determining how OptiX scales with the number of
 155 concurrent rays is important in knowing how large neutron batches need
 156 to be in WARP for the geometry routines to execute efficiently. Nuclear
 157 reactor simulations can contain thousands of material zones, and knowing
 158 how OptiX scales with the number of geometrical objects is important for
 159 choosing a configuration that will allow WARP to perform neutron transport
 160 well in complex geometries. The scaling studies and the algorithms used in
 161 them are outlined here and the results of the studies are shown in Section 5.

162 The first scaling study focuses on scaling with regard to the number
 163 of concurrent rays. Three different geometries were used in the test – an
 164 assembly-like hexagonal array of cylinders, an assembly of the same size but
 165 with two interleaved arrays, and a much larger version of the assembly-like
 166 geometry. The assembly-like configuration has 631 instances of a single prim-
 167 itive type, the interleaved array has 381 instances of three different primitives
 168 types, and the large array consists of 1801 instances of a single primitive type.
 169 These cases were chosen to determine if there are any large differences in the
 170 ray scaling when the number and types of objects are changed. Each array

171 resides in a large hexagonal prism, which is in turn nested in a large cube
 172 that defines the outer limit of the scene.

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

182 2.1. Point-in-Polygon / Material Query

Figure 2: The point-in-polygon-like algorithm for determining the entering cell/material number by using ray tracing

183 An important job for the geometry routine in a Monte Carlo neutron
 184 transport code is to determine the cell and material IDs of a particle based
 185 only on its coordinates. In Woodcock tracking, surface intersections are not
 186 calculated and the material query is the only place where geometric infor-
 187 mation enters into the simulation. In WARP and other ray-tracing codes,
 188 material information is only updated when a sampled interaction distance is
 189 greater than the near surface distance. The neutron is then placed on the
 190 boundary, the material information is updated for the material the neutron

191 is entering, and the interaction distance is sampled again using the same
192 direction of flight as before. WARP will use an algorithm to determine the
193 entering material number by using ray tracing, since all the geometric infor-
194 mation is stored in the OptiX context. This also means the material query
195 will be able to take advantage of the OptiX acceleration structures and should
196 scale well.

197 The material query algorithm is shown in Figure 2. An ordered list of
198 surface intersections is generated by iteratively ray tracing and adding the
199 closest surface number to the hit list. Tracing is terminated when a predefined
200 outer cell (that contains all other cells) is intersected. Since all surfaces are
201 closed, the ray will intersect any cell surface twice that it isn't nested in.
202 When the list is made, the double entries are removed, which yields a list of
203 cells the neutron is nested in. The first entry will be its current cell and the
204 second entry will be the cell it is entering into.

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

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

228 The last problem that can result from this algorithm is when cells have

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

This algorithm is very similar to the ray casting point-in-polygon (PIP) algorithm, which determines whether a point is inside or outside of an arbitrary polygon by counting the number of times it crosses a surface. An even number means it is outside and an odd number means it is inside. This algorithm is almost the same, but keeps track of cell numbers instead of binary logic for one surface. This way, the nesting of a neutron can be determined and the entering cell can be found. Both of these algorithms take advantage of the cell being closed, and in this sense are similar to Gauss's Law or the divergence theorem, which states the flux integrated around a surface will be nonzero only if the surface contains a source. This material query algorithm is like a discrete, single field line version where the neutron's position is the source point and the cell boundaries are the integrating surfaces.

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

2.2. *Instancing*

Nuclear reactors can have very complicated geometries, but many rely on simple shapes that are repeated in arrays. There are a few ways in which identical cells can be instanced in OptiX. The first, and most convenient, is to define a single primitive in its own coordinate system, then use a transform node in the OptiX node graph to transform the primitive to its actual position via an affine transformation matrix [11]. The resulting node graph

is shown in Figure 3 for a scene that has three boxes in it. Note, a GeometryInstance object ties a hit program to the spatial geometry data in the geometry primitive object, acceleration objects are attached to all group objects, and transform objects can only have a *single child* that must be either a Group or GeometryGroup object [11]. This is why each GeometryInstance must have its own GeometryGroup.

Figure 3: The OptiX node graph using transform instancing

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

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

Figure 4: The OptiX node graph using mesh primitive instancing.

282 written into the OptiX buffer as separate elements.

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

290 2.3. Test Geometries

291 Figure 5 shows the geometry for the interleaved assembly and the large
 292 assembly cases used in the first scaling study. These figures were created
 293 by OptiX by performing a cell number query as prescribed in the previous
 294 subsection. The interleaved assembly consists of three hexagonal arrays, two
 295 cylinders, and one hexagonal prism. The arrays are seven elements on a
 296 side, which corresponds to 127 elements each for a total of 383 elements
 297 (including the large hex cell around the arrays and the outer box cell). The
 298 large assembly has 25 cylinders on a side, for a total of 1803 elements. The
 299 smaller assembly is 15 cylinders on a side, for a total of 633 elements.

300 The cell numbers are mapped to colors in Figure 5. The images appear
 301 correct upon visual inspection and no tracking errors were generated when

(a) The interleaved assembly

(b) The large assembly

Figure 5: x - y cross sections of the geometry created by using the PIP cell/material query algorithm

302 creating these images, it therefore appears the routine is working and the
 303 algorithm can calculate the cell numbers. A helpful feature of OptiX is that
 304 variables can be attached to each geometry primitive. In addition to the cell
 305 ID number, the material ID is also attached, so the material query can be
 306 done directly by OptiX rather than by determining the cell number and then
 307 having to do an additional lookup or hash.

Figure 6: The geometries used in the scaling study from least number of objects on the left to most on the right.

308 Figure 6 shows the geometries used in the second scaling study. The
 309 hexagonal cell has the same dimensions as the assembly-like geometry case
 310 in the first scaling study, but each successive case has smaller and more
 311 numerous cylinders in a hexagonal array inside the hexagonal cell. The most
 312 sparse scene only has a single primitive in it, shown on the left side of Figure
 313 6, and the most complex scene, shown on the right side of the figure, has
 314 42,843 primitives in it.

3. Unionized Cross Section Data Layout

The nuclear data required in Monte Carlo simulations are very heterogeneous, such as, for example, data stored in cross section libraries, and the way the data is accessed in a Monte Carlo simulation is random. This section describes the methods used to deal with the data heterogeneity and how data access in WARP has been optimized for the GPU. A material's macroscopic cross sections dictate the reaction probabilities as a neutron travels through it. In order for the reaction probabilities to be computed from the macroscopic cross sections, the material's total macroscopic cross section (the sum of all the isotopic macroscopic cross sections of a material) must be known. This value is used to normalize the isotopic macroscopic cross sections to one, thus making them probabilities.

The atomic densities change according to the material the neutron is traveling through, and WARP recomputes a material's total macroscopic cross section at every interaction. This is done in order to not store preprocessed macroscopic cross sections for each material. If macroscopic cross sections are computed on-the-fly, a single set of microscopic cross sections and a small vector of material number densities can be stored instead. Of course, recomputing the macroscopic cross sections at every inner transport iteration adds additional work, but if the material densities are stored in the GPU's shared memory, they can be accessed quickly and an equivalent number of global loads are needed for loading the microscopic cross sections as would be need for loading the preprocessed macroscopic cross sections. If the material total cross section is precomputed and stored, this may save global loads at the cost of increased memory usage.

Since cross sections stored in the data libraries need to be interpolated between energy points, It is particularly troublesome that each cross section has its own independent energy grid. Using point-wise data for continuous energy simulations requires an interpolation be performed between points, and to do this, the code must somehow scan the energy grid array to find the points between which the neutron's energy falls. If every isotope has its own grid, this search must be done for every isotope, and can become very expensive. This is why a unionized energy grid structure, like that implemented in Serpent, is used in WARP [17].

Unionizing the cross sections means that the energy grids of the cross sections are all unionized into a single, larger structure that contains the energy points of every isotope. Including every energy point ensures that no

352 information is lost in the unionizing process. Since the unionized grid is then
 353 used to index every isotope’s cross section vectors, there are energy points
 354 which do not correspond to values in the original nuclear data. These holes in
 355 the data are filled by linear interpolating between the closest two grid points
 356 that have values. The unionization process transforms many 1D reaction
 357 cross section vectors into a single 2D matrix indexed by incoming neutron
 358 energy and MT reaction number. Figure 7 shows the unionization process
 359 with two small, arbitrary energy grids and their corresponding cross section
 360 vectors. The colors in the unionized energy grid represent which isotope the
 361 grid value came from, the green cross section data blocks are interpolated
 362 values, and the red blocks are placeholder zeros to preserve thresholds. It
 363 is clear that the resulting dataset is larger than the sum of the individual
 364 cross sections since it contains redundant data, but it will be much easier to
 365 search.

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

374 Because the cross sections vectors are unionized in this way, the cross
 375 section values for every isotope can be read along a single row once the
 376 bounding energy indices, $E_i < E < E_{i+1}$, are found for a given interaction.
 377 This format reduces the number of energy searches needed and promotes
 378 data locality by keeping all the data points for all isotopes in a given energy
 379 range adjacent to one another in memory (if row-major matrix format is
 380 used). On CPUs, the data for an energy value can be read as a cache line
 381 and accessed quickly by the cores. On a GPU, since a thread needs to read
 382 the entire energy line (one energy over all isotopes) in sequence in order to
 383 compute the macroscopic cross section on-the-fly, the memory loads are not
 384 coalesced and memory bandwidth is wasted (adjacent threads do not access
 385 adjacent memory at the same time).

386 To mitigate this inefficiency, the data can be recast to use the float4
 387 datatype, which is a vector type that is 16 bytes long. When requested,
 388 the entire 16 bytes will be loaded in one transaction rather than 4 separate
 389 4-byte transactions, maximizing bandwidth and minimizing the number of

Figure 7: Unionizing two cross section vectors.

global requests (and reducing the impact of latency). Also, since all of the total cross sections for a material are all needed at the same time, they are stored together in the first columns of the unionized data array. After this block of total cross section columns, the individual reaction cross section are stored as blocks in the same material order as the total cross section block (all the cross sections for an isotope are in a single block). This is done so the macroscopic or microscopic kernel needs only scan a single, contiguous block of data in the array. Figure 8 shows an illustration of the unionized cross section dataset structure used in WARP.

3.1. Energy and Angle Probability Distribution Data as a Linked List

The cross section data can be regularized with the unionized grid method, but there are other data distributions that WARP needs to conduct accurate

		Matrix Column Index										...
		0	1	2	3	4	5	6	7	8	9	
		Unionized Energy Grid	Isotope 1 Total	Isotope 2 Total	Isotope 3 Total	Isotope 1 Rxn 1	Isotope 1 Rxn 2	Isotope 2 Rxn 1	Isotope 2 Rxn 2	Isotope 2 Rxn 3	Isotope 3 Rxn 1	
Matrix Row Index	0	E_0	Cross Section Values	Cross Section Values	Cross Section Values	Cross Section Values	Cross Section Values	Cross Section Values	Cross Section Values	Cross Section Values	Cross Section Values	...
	1	E_1										
	2	E_2										
	3	E_3										
...												

Figure 8: The unionized cross section layout used in WARP.

simulations. The most prevalent of which are scattering matrices that give tabulated probability distributions for outgoing center-of-mass (CM) angle μ and incident neutron energy. Since these matrices have their own incoming energy grid, these values can be unionized into the cross section dataset as well. This transforms many 2D scattering matrices into a 3D matrix indexed by incoming energy, outgoing μ , and reaction MT number.

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

After determining that unionizing with the main energy grid was an unacceptable method, another approach was tested where the energy grids of the scattering matrices were unionized between themselves, but not with the main energy grid. This would reduce the amount of unnecessary data, but would require an additional search be made on the unionized scattering matrix energy grid. The memory usage of this method with ^{235}U was reduced to 12 MB (103 energy grid points, 598 μ CDF points, 51 reactions). When tested with ^{235}U and ^{238}U , usage for the scattering data increased to 68 MB

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

Unified Energy Grid	Isotope 1 Rxn 1	Isotope 2 Rxn 1	Isotope 2 Rxn 2
1e-11	pointer to Dist 1	pointer to Dist 1	pointer to Dist 1
1e-8	pointer to Dist 1	pointer to Dist 1	pointer to Dist 1
1e-6	pointer to Dist 1	pointer to Dist 1	pointer to Dist 1
1e-3	pointer to Dist 1	pointer to Dist 1	pointer to Dist 1
1e-2	pointer to Dist 1	pointer to Dist 2	pointer to Dist 1
1e-1	pointer to Dist 1	pointer to Dist 2	pointer to Dist 1
5e-1	pointer to Dist 1	pointer to Dist 2	pointer to Dist 1
1e0	pointer to Dist 1	pointer to Dist 2	pointer to Dist 1
5e0	pointer to Dist 1	pointer to Dist 2	pointer to Dist 1
1e1	pointer to Dist 1	pointer to Dist 2	pointer to Dist 2
2e1	pointer to Dist 1	pointer to Dist 2	pointer to Dist 2

Isotope 1 Rxn 1	mu = -1	-0.713	0.713	1
Energy = 1e-11	0.0	0.12	0.90	1

Isotope 2 Rxn 1	mu = -1	-0.713	0.713	1
Energy = 1e-11	0.0	0.21	0.86	1
Energy = 1e-2	0.0	0.18	0.67	1

Isotope 2 Rxn 2	mu = -1	-0.82	-0.32	0.32	0.82	1
Energy = 1e-11	0.0	0.12	0.45	0.67	0.93	1
Energy = 1e1	0.0	0.19	0.30	0.56	0.87	1

Figure 9: Making a link to distribution data in the unionized dataset.

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

439 is similar to a linked list, where there is a pointer at the end of an object
440 pointing to the next object in a list.

441 Structuring the arrays like this means once a reaction is sampled to occur,
442 there is a pointer readily available at the same index in a second matrix to
443 the appropriate distributions, and no searching needs to be done. Since the
444 distribution PDFs need to be read serially by each thread, the float4 format
445 can also be of use here. If there are no distributions for a particular reaction
446 or energy, a null pointer is inserted into the matrix. For fission reactions
447 that do not have scattering distributions (neutron emission can be assumed to
448 be isotropic [18]), the ν value for the grid energy is stored in the scattering
449 pointer matrix instead of a pointer. This way a search for the appropriate ν
450 value does not have to be performed either. Figure 9 shows a pointer matrix
451 for the unionized cross section matrix in Figure 7.

452 Serpent stores pointers in its unionized cross section layout [13]. This
453 method is referred to as “double-indexing,” and is used by the OpenMC
454 Monte Carlo neutron transport code as well [17, 19, 18]. The values in a
455 double-indexed layout are pointers to arrays that contain both cross
456 section and distribution data for an isotope. This way, only a single unionized
457 array needs to be stored as opposed to the three needed by WARP. Since
458 Serpent’s layout only contains pointers, however, the pointer needs to be
459 loaded before any cross section values can be loaded. This means that for
460 every cross section loaded, a pointer needed to be loaded as well. WARP’s
461 layout stored the cross section values directly in the layout, so only cross
462 sections are loaded during the isotope and reaction sampling routine instead
463 of pointers and cross sections. A pair of pointers is only loaded when the
464 reaction has been selected and the angular or energy distribution data is
465 needed. Such a layout may use more space, but reduces the amount of
466 memory traffic needed to sample reactions.

467 3.2. *Embedded Python*

468 Now that the data layout has been discussed, how the data is loaded
469 or reformatted from ACE-formatted data files will be explained. An initial
470 effort was made to write an ACE-parsing script in C from scratch, but this
471 was abandoned in favor of using the existing ACE module of the PyNE
472 (Python for Nuclear Engineering) package [14] (why reinvent the wheel?).
473 The PyNE package “is a suite of tools to aid in computational nuclear science
474 & engineering. PyNE seeks to provide native implementations of common
475 nuclear algorithms, as well as Python bindings and I/O support for other

476 industry standard nuclear codes.” [14]. PyNE contains a Python module
477 that can parse ACE data files into Python objects that can be more easily
478 handled than flat data arrays. This module was written by Paul Romano as
479 a preliminary project for OpenMC, whose ACE library was later written in
480 Fortran based on the methods developed in this Python module [14, 18].

481 Some parts of PyNE have a C++ application programming interfaces
482 (APIs) as well as a Python API, but this is not the case for the ACE module.
483 It was originally written in Python, not C++, so it has no C++ API. This
484 created a problem for WARP since WARP is written in C/C++ and could not
485 directly make use of the ACE module. Fortunately, there is a very effective C
486 API for Python! This API allows one to initialize and run a Python instance
487 from a C program. In WARP, this API is used to start a Python instance
488 where PyNE can be used to load cross sections from ACE data files. NumPy
489 [20] is then used to unionize the energy grids of the requested isotopes and
490 perform the linear interpolation to fill in the gaps. Once this is done, the
491 Python instance returns the NumPy array to WARP as a C data structure.
492 The data is copied to an internal array and the Python object is cleared.

493 WARP then loops through the unionized array, requesting distribution
494 data from the Python instance. The scattering and energy distribution data
495 are also copied for the requested energy range in a similar fashion, and a
496 device pointer for the distribution is written into the scattering and energy
497 pointer matrices. When all the needed data has been copied over to WARP,
498 the Python instance is terminated and its memory freed. Since the unionized
499 energy grid is an index of the matrix that needs to be searched before a row of
500 the unionized cross section matrix can be accessed, the energy grid is stored
501 as its own contiguous array rather than a column of the matrix. Figures 7,
502 8, and 9 show the energy grid as a column simply for illustrative purposes.

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

510 Both the constant and texture memory spaces are cached, so the random
511 access inherent in Monte Carlo simulations may make cached access worse
512 than non-cached because of cache miss penalties. This effect was not stud-
513 ied in the initial development of WARP, however. Nelson reports using the

constant memory space in his simulations, with little-to-no performance improvement [5]. Using the texture memory to store the cross section matrix might benefit from the free linear interpolation that can be performed with the texture element load, however. With all of these points considered, it was decided to store all data, whether it be for history data or cross sections, in global memory in WARP.

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

4. CUDA Kernels and Data-Parallel Tasks

CUDA was first released in 2006 as NVIDIA’s proprietary GPU programming platform. It makes minimal additions to C/C++, and any C programmer would be very comfortable programming in CUDA [21]. It was chosen over OpenCL, the open source GPU programming platform, for WARP due to CUDA’s greater feature support, stability, ease of programming, wider community usage, and ability to use new, cutting-edge features in NVIDIA GPUs that OpenCL is not.

The overarching theme for CUDA is that a host CPU thread directs the GPU’s operation. Data must be transferred from the host to the GPU’s global memory over the PCIe bus, *kernels* are launched to perform computations based on the transferred data, and then control returns to the host thread. “Kernels” are CUDA’s parallel programs that each thread carries out over the data. Kernels must specify independent tasks for each GPU thread from the standpoint that thread execution order does not matter. There can be barriers where threads will wait for all other threads to arrive, but the order in which the threads execute to get to such a barrier is arbitrary and handled by the GPU hardware [21]. Blocks of threads have the same interleaving requirement. The order in which they execute is unspecified, and cannot be relied on for calculating values. Some newer NVIDIA GPUs have a “dynamic parallelism” feature where a CUDA kernel can launch additional kernels, so control does not need to be returned to the CPU host until the entire simulation is complete [21].

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

Figure 10: WARP inner transport loop that is executed until all neutrons in a batch are completed.

558 The inner loop, shown in Figure 10, actually transports the neutrons
 559 through the problem geometry and samples the reaction CDFs. The blocks
 560 in the figure represent independent kernel launches, and are executed left-
 561 to-right. The quantities listed under the routine names are what the routine
 562 calculates. The block color corresponds to the library used to perform the
 563 task. The first step of the inner loop is using OptiX to perform the material
 564 query, since material information is needed to query the proper cross sections
 565 and to determine the distance to the nearest surface along the neutron's

566 path of travel. Once this is known, a kernel is launched to do a search
 567 on the unionized energy grid to determine the grid index i that satisfies
 568 $E_i < E < E_{i+1}$. Next, the macroscopic kernel is launched. This kernel
 569 computes the total macroscopic cross section for the material, samples the
 570 interaction distance, samples which isotope the neutron interacts with, moves
 571 the neutron to the nearer of the interaction/intersection distance, and sets
 572 the neutron's reaction number to the resample flag if the intersection is closer.
 573 Since the macroscopic cross section has been computed at this point, the tally
 574 kernel is launched and scores a specified flux tally. After the flux is scored,
 575 the microscopic kernel is launched to determine which reaction type in the
 576 determined isotope occurs. The radix sort is done next since the reaction
 577 has been determined, and this operation sorts the neutrons by reaction type
 578 and updates the values in the remapping vector. The next four kernels are
 579 launched concurrently, as indicated by being in the same horizontal position
 580 in Figure 10. Each of these kernels performs the specific functions necessary
 581 to model their different reactions, and since a neutron cannot undergo two
 582 reactions simultaneously, the data they access does not overlap and these
 583 kernels can be launched concurrently. If a neutron's reaction number is the
 584 resample flag, it skips the microscopic and reaction kernels and isn't processed
 585 again until the material query, which resets the reaction number to zero. This
 586 loop cycles until all neutrons in a batch are terminated through absorption
 587 or leakage.

Figure 11: WARP outer transport loop that is executed in between neutron batches for criticality source runs.

588 The outer loop, shown in Figure 11, uses the result from the inner loop
 589 to set up the next batch of neutrons. The inner loop serves to determine the
 590 yields of the batch’s neutrons. The yield array is then reduced to determine
 591 the multiplication factor, k_{eff} , of the batch. The yields are then divided by
 592 k_{eff} to make the yield as close to one as possible without biasing the fission
 593 source distribution. Since dividing by k_{eff} is unlikely to yield an integer, the
 594 rebased yield is sampled between the bounding integers to preserve the mean
 595 across many neutrons.

596 After the yields are rebased, a prefix sum (also called a scan) is performed
 597 on the yield array. The value for each particle’s element of the prefix sum
 598 is the total number of fission neutrons before it. Since this is known, the
 599 “pop” routine can insert the sampled fission neutrons into the next batch’s
 600 history vector with no risk of writing into the same element. Since the yields
 601 were rebased to make $k_{\text{eff}} \approx 1$, the yield vector will contain enough secondary
 602 neutrons to fill the next generation’s entire history vector. The pop kernel
 603 inserts neutrons into the history vector at points where fissions occurred, but
 604 with individually sampled energies and (isotropic) directions. After the pop
 605 completes, the inner loop transports this newly initialized batch of neutrons.

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

613 4.1. Grid Search Kernel

614 Much effort has gone into ensuring that only one search on the main
 615 unionized energy grid is required to find the index i that satisfies $E_i < E <$
 616 E_{i+1} , but an algorithm to perform the search itself has not yet been estab-
 617 lished. It is convenient that the values in the energy grid are monotonically
 618 increasing, but since the array is composed of floating point values with ar-
 619 bitrary spacing, an inverse function cannot be made to calculate an index in
 620 $O(1)$ time. Conversely, a naïve approach would be to scan the array from
 621 beginning to end, performing comparisons along the way, and therefore cal-
 622 culating the cross section array index in $O(n)$ time, where n is the number
 623 of elements in the array. This method does not scale well and will not be
 624 considered.

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

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

642 The binary search is the classic search algorithm for searching irregular
 643 data. Instead of progressing through the array from beginning to end, it is
 644 continually bisected until the correct interval is found. In other words, the
 645 search begins at the center of the vector. If the searched-for value is less
 646 than the middle value, the next loaded value is the middle value of the first
 647 bisection. If the searched-for value is greater than the middle value, the next
 648 loaded value is the middle value of the second bisection (of the first bisection).
 649 This process repeats iteratively until the value is found. The binary search
 650 algorithm is attractive because it runs in $O(\log_2(n))$ loads/comparisons in
 651 the worst case, and uses no storage except for the loaded and comparing
 652 values [22]. WARP currently uses a simple binary search algorithm because
 653 of this algorithm's logarithmic scaling and ease of implementation. It will be
 654 determined if the grid search is a significant portion of the simulation time,
 655 and if it is, WARP could benefit by implementing more advanced searching
 656 algorithms in the future.

657 4.2. Pseudo-Random Numbers

658 Random numbers are the basis of the Monte Carlo method, and generat-
 659 ing them must be done efficiently. Since these numbers must be computed,
 660 they are not strictly random, but rather pseudo-random since they are ran-
 661 domly distributed but can be deterministically computed. NVIDIA provides

662 a random number library called CURAND that can generate large arrays
 663 of randomly-distributed numbers [10]. Initially, WARP precomputed all the
 664 random numbers it would need for a single transport loop iteration. This
 665 could be done since direct sampling methods were used everywhere (which
 666 was incorrect for the target velocity in scattering) and there was a known
 667 maximum number of random numbers required. The threads would simply
 668 access random number values stored in this giant array (about 20 numbers
 669 were needed per thread). This is a very bandwidth-intensive way to ac-
 670 cess random numbers since they are always written to and read from global
 671 memory. In these early implementations, the random number generation and
 672 access was taking about a quarter of the transport loop time. In addition,
 673 the introduction of the correct rejection sampling method for the target ve-
 674 locity disqualifies precomputation since there was no longer a predetermined
 675 number of numbers needed.

676 Taking these points into consideration, it was decided to use a linear con-
 677 gruential random number generator (LCRNG) for computing random num-
 678 bers in the transport kernels. It allows the kernels to generate random num-
 679 bers with the recursion relations shown in Eq. (1), where x_{n-1} is the previous
 680 random number. The values a , c , and m are taken from Pierre L’Ecuyer’s
 681 manual for configurations with a high figure of merit [23].

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

682 OpenMC uses a similar LCRNG implementation but with different values
 683 for a , m , and c since it uses double rather than single precision. The values
 684 were chosen close to the full 32-bit range for maximum floating point reso-
 685 lution. Having a modulus as a power of two is very convenient as well since
 686 the modulus can be performed by bit truncation rather than a full modulus
 687 operation [18]. In other words, the operation can be done by a bitwise AND
 688 operation between $ax_{n-1} + c$ and $2^{30} - 1$, which in binary is all ones below
 689 the 30th bit.

690 Using a LCRNG keeps global access down and improves performance
 691 greatly, since a single seed value is loaded at the start of the transport loop,
 692 stored in fast registers, then written back at the end of the loop. CURAND

693 is used outside of the LCRNG cycle to advance the seed bank with the
 694 Mersenne Twister algorithm (which has a period of $2^{19937} - 1$) to make sure
 695 no correlated numbers are used as intermediate seeds.

696 4.3. Macroscopic Cross Section Kernel

697 The macroscopic kernel in WARP computes all values related to macro-
 698 scopic cross sections. These include total macroscopic cross section for the
 699 material, interaction distance, which isotope the neutron interacts with, mov-
 700 ing the neutron to the nearer of the interaction/intersection distance, and
 701 setting the neutron's reaction number to the resample flag if the intersection
 702 is closer.

703 Since the isotopes' total microscopic cross sections are stored together
 704 in the unionized cross section matrix, the total macroscopic cross section
 705 can scan the contiguous row of this part of the matrix, interpolating the
 706 value between i and $i + 1$ (which has been determined by the grid search
 707 kernel). The interpolated values are multiplied by the number density (in
 708 atoms/barn-cm, stored in its own vector) and the sum is accumulated as
 709 shown in Eq. (2), where M_k is the number density and $\sigma_{t,k}(E)$ is the total
 710 microscopic cross section of isotope k at energy E .

$$\Sigma_t(E) = \sum_{k=0}^{N_{\text{isotopes}}} M_k \sigma_{t,k}(E) \quad (2)$$

711 When the material vector has been completely scanned and total cross
 712 section is computed, the array is scanned again, but this time a random
 713 number is used to determined which isotope the neutron interacts with via
 714 Eq. (3), where k is the isotope index.

$$PDF_k(E) = \frac{\Sigma_{t,k}(E)}{\Sigma_t(E)} \Rightarrow CDF_k(E) = \frac{1}{\Sigma_t(E)} \sum_{n=1}^k \Sigma_{t,n}(E) \quad (3)$$

715 The material's total macroscopic cross section is written to an array so
 716 the flux tally routine does not have to recompute it, which would take many
 717 more global memory transactions than loading it from the array. Next, the
 718 interaction distance is sampled and is compared against the nearest surface
 719 distance, which was computed with OptiX in the first step of the transport
 720 loop. If the interaction distance is nearer, the neutron's coordinates are

721 changed to this location. If the surface is closer, the neutron is moved to
 722 the surface and its reaction number is changed to the resample flag value,
 723 which will cause it to skip all other kernels until OptiX determines the new
 724 material and resets its reaction flag. As stated in the beginning of this
 725 section, preprocessed values are not stored in memory and WARP recomputes
 726 macroscopic cross section at every iteration of the inner transport loop.

727 4.4. Flux Tally Kernel

728 The flux tally kernel scores a collision in a predefined cell into a bin via
 729 Eq. (4), where $\Sigma_{t,j}(E)$ is the total macroscopic cross section of the material
 730 in volume j at energy $E_i < E < E_{i+1}$.

$$E_i < E < E_{i+1}$$

$$\bar{\phi}_{i,j} = \frac{1}{V_j} \sum_{i=1}^{N_{i,j}} \frac{1}{\Sigma_{t,j}(E)} \quad (4)$$

731 Currently, WARP only allows evenly spaced logarithmic bins since their
 732 indices can be calculated analytically (as opposed to an arbitrary input grid).
 733 The tally index, j , out of N_{tally} bins for a neutron with energy E is calcu-
 734 lated via Eq. (5). Since multiple threads could be adding to the element
 735 located at j , “atomic” operations are used to perform the sum, specifically
 736 the atomicAdd() function. Atomic functions perform the load-compute-store
 737 operations in a single, uninterruptible transaction. This eliminates the risk
 738 of data races occurring.

$$j = \text{floor} \left(\frac{\log \left(\frac{E}{E_{\min}} \right)}{\log \left(\frac{E_{\max}}{E_{\min}} \right)} (N_{\text{tally}} - 1) \right) \quad (5)$$

739 Since WARP uses single precision floats and integers, buffer overflow can
 740 be a problem when a tally is scored frequently (or at least more of one than
 741 if double precision were used). Because of this, the tallies are copied from
 742 the device during each outer loop iteration. The tally values are divided by
 743 the total number of source neutrons and are accumulated into double or long
 744 integer arrays, as appropriate, on the host. After copying, the device arrays
 745 are zeroed. Dividing and accumulating also serves to increase the accuracy of
 746 the arithmetic since the tally values are kept within a smaller range of values.
 747 Roundoff error is exacerbated by adding small floating point numbers to large

ones, which is more likely when simply accumulating all of the tally data without periodically normalizing by particle count. The required intermittent copying during transport increases the device-host communication, but this impact is minimal and can be overlapped with other routines since it is only done once in the outer loop.

4.5. Microscopic Cross Section Kernel

Unlike the macroscopic kernel, the microscopic kernel only has one job - to determine which reaction the neutron undergoes in the isotope it has been sampled to interact with. This is done by scanning the isotope's subrow in the unionized cross section matrix as shown in Eq. (6). $N_{\text{start},k}$ is the starting index of the row for isotope k , $N_{\text{end},k}$ is the index of the isotope's last reaction cross section, and $\sigma_{t,k}(E)$ is isotope k 's total microscopic cross section. The starting and ending indices for an isotope are precomputed and stored in a separate array. For most runs, this array and the material isotope density array can be stored in fast shared memory.

$$PDF_m = \frac{1}{\sigma_{t,k}(E)} \sum_{z=N_{\text{start},k}}^{z=m} \sigma_{i,z}(E) \quad (6)$$

Once a new reaction m is sampled, the sorting routine is launched to remap data references and sweep completed histories out, then reaction kernels can be launched to carry out the individual reactions.

4.6. Interaction Kernels

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

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

778 4.7. Concurrent Kernels

779 As mentioned previously, the reaction kernels should never access over-
780 lapping data so that they can all be launched concurrently. NVIDIA cards
781 with a compute capacity of 2.0 and higher support up to 16 concurrent kernel
782 launches [21]. Kernel launches are submitted to nonzero execution streams
783 by declaring additional streams to be present and passing the stream objects
784 to the kernel launches as an extra launch parameter. Kernels launched on
785 different streams can be executed concurrently and their blocks can be in-
786 terleaved, which is why it is very important for them to act on independent
787 data. Launching the reaction kernels concurrently means they are launched
788 in parallel and they do not need to wait for each other to complete. If not
789 launched concurrently, kernels must be launched sequentially, and the inelas-
790 tic scattering kernel would have to wait for the elastic scattering kernel to
791 complete before it could be launched, for instance.

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

801 4.8. Reference Remapping with Radix Sort

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

809 Another benefit pushing all of the completed references to the end of
810 the vector is that, as neutrons start to complete, the length of the sort can
811 be reduced and it will be computed faster. Figure 12 shows how the radix
812 sort will arrange the remap vector and how threads will be mapped to the
813 data. CUDPP's radix sort conveniently sorts for key-value pairs, and WARP

Table 1: Reaction number encodings in WARP.

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

814 uses the reaction numbers as keys and the data index as values, effectively
815 creating a remap vector.

Figure 12: A radix key-value sort creating a remapping vector.

816 There is one more piece of information that is needed to launch the re-
817 action kernels concurrently – the number of particles undergoing each type
818 of reaction. This is determined by launching a comparison kernel on the
819 sorted array, which detects the places where the reaction numbers change.
820 The algorithm to do this is a simple adjacent comparison. If there are N
821 elements in the array, $N - 1$ threads are launched that load values tid and
822 $tid+1$. If the values are different, tid is the last index of a reaction type. If
823 a transition is determined to be valid, its thread can write to a small array

824 and no data races should occur because there are coarse boundaries for the
 825 reaction kernels and each boundary is unique. There are only two boundaries
 826 for each reaction group, and writing the indices of the reaction block edges
 827 can be done without atomic operations since only one thread should ever
 828 write to an element.

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

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

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

852 Any reaction over 900 signifies a terminated history, so when the cross
 853 section data is loaded, any reaction larger than 102 has 1000 added to it to
 854 ensure it will be pushed out of the remapping vector upon sorting. Since
 855 fission also terminates particles but needs a yield value sampled after the
 856 microscopic kernel samples it to occur, any reactions in the MT=11-45 block
 857 have 800 added to their MT numbers. Therefore, after the sort is performed,
 858 the fission-like reactions are already near the end of the remapping vector.
 859 After the fission kernel processes these histories, their reaction number have
 860 100 added to them, making them part of the completed history block. The
 861 reaction numbers for scattering (MT=2, 51-91) are not changed. MT=11-

45 not including the fission numbers can usually be treated as continuum scattering (ENDF law 44), but may not be true for all isotopes. With the reactions laid out as such, a contiguous remapping vector can be formed and the total number of active neutrons can be calculated from a single, efficient radix sort operation.

4.9. Criticality Source

As was mentioned previously, when the simulation is run in criticality mode, the source points for neutron batches depend on the fission points in the previous batch. But if k_{eff} is not equal to one, each source neutron no longer corresponds to a secondary neutron. This is the same problem in the time-independent neutron transport equation, where getting rid of the time-dependent term causes the equation to be inconsistent if the multiplication factor is not one. To remedy this, the fission source term is divided by the multiplication factor to force the equations to be consistent (assuming the multiplication factor is known). Similarly, the fission yield vector of a batch of neutrons can be renormalized to artificially make the multiplication factor equal to unity via Eq. (7), where n is the batch number, N_d is the length of the dataset, y_j is the secondary neutron yield of particle j , and N_f is the total number of secondary neutrons.

$$\sum_{j=0}^{N_d} \frac{y_j}{k_{\text{eff},n}} = \frac{N_{f,n}}{k_{\text{eff},n}} \Rightarrow N_{s,n+1} = N_{s,n} \quad (7)$$

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

Once the yield vector is rebased by dividing by k_{eff} and stochastically rounding to a nearest integer, a prefix sum is done on the yield vector. The j^{th} value of an exclusive prefix sum, or exclusive scan as CUDPP calls it, is sum of the array values before index j as shown in Eq. (8). Such a vector can be used in the pop kernel, which writes the secondary particle data into the next batch's initial data based on the yield vector.

$$p_j = \sum_{i=0}^{j-1} x_i \quad (8)$$

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

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

908 All the necessary pieces have been outlined to perform Monte Carlo neu-
 909 tron transport in general 3D geometries using continuous energy cross sec-
 910 tions. Algorithms and libraries have been chosen so each step of neutron
 911 transport can be preformed efficiently on a GPU. The next chapter will dis-
 912 cuss the consequences of these choices.

913 5. Results

914 5.1. *OptiX Scaling Study Results*

915 Both of the scaling studies were run on an NVIDIA Tesla K20 card. In
 916 addition to finding the closest intersection point, cell queries were done with
 917 the PIP algorithm, and ray starting points were randomly chosen to be uni-
 918 formly distributed in 3D, and the directional vectors from those points were
 919 randomly sampled to be distributed isotropically in 4π . Figure 13 shows
 920 the ray trace rates versus number of particle starting points for twelve cases.
 921 The cases are combinations of the three test geometries (assembly, large,
 922 and interleaved), primitive and transform instancing (prim or xfrm), and split
 923 bounding volume hierarchy and regular bounding volume hierarchy (SBVH or
 924 BVH). K-d trees are not used since they require a mesh vertex buffer be pro-
 925 vided by the user, which implies that they only work for triangularly-meshed
 926 objects, not ones instanced by simple geometric primitives like spheres and

927 cubes. Handling geometries that are meshed in such a way may be a future
 928 area of development in WARP.

Figure 13: Trace rates of an NVIDIA Tesla K20 performing cell queries with the PIP algorithm.

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

934 The number of source points used in the second scaling study was set to
 935 10^6 and 10^8 to see if the trace rate scales with object number differently if
 936 done on the extreme edges of the trace rate plateau shown in Figure 13. A
 937 BVH acceleration structure was used since it showed the best performance in
 938 all cases. Figure 14 shows the results of the scaling test on an NVIDIA K20
 939 card. It can be seen that the ray trace rates for both primitive and transform
 940 instancing plateau after about 20 objects in the scene. Primitive instancing
 941 also always outperforms transform instancing. It should be noted that the
 942 trace rates for the two dataset sizes start quite far apart for one object
 943 then converge to equal performance quickly as more objects are included
 944 in the scene. This may be because the trace becomes more dependent on

945 acceleration traversal rather than output buffer access (where smaller, slower
 946 sizes makes a difference when there are few objects).

Figure 14: Trace rate scaling on an NVIDIA Tesla K20 performing cell queries with the PIP algorithm.

947 Using transform instancing fails for the 10⁸ dataset size when somewhere
 948 between 2613 and 10,623 objects were included in the scene. This is why the
 949 transform instancing trace stops at 2613 cells in Figure 14. The most likely
 950 reason is because transform instancing requires a large amount of memory,
 951 and the card runs out of memory when a large particle dataset is also present
 952 on the card. This is also most likely why performance of the transform
 953 instancing method drops at the same point for the 10⁶ dataset size. At
 954 this point, OptiX started paging GPU memory to the host memory, and
 955 this degraded performance significantly. Structure construction for transform
 956 instancing prior to the trace also became very slow compared to primitive
 957 instancing, presumably due to the independent acceleration objects attached
 958 to every instance and the overhead of computing all the affine transformations
 959 at acceleration structure build time.

960 From the scaling studies it can be concluded that OptiX can be used to
 961 handle the geometry representation in a Monte Carlo neutron transport code.
 962 Using a primitive-based geometry instancing method, a BVH acceleration

963 structure, and running as many parallel rays as possible provides the best
 964 performance. The OptiX ray trace rate also seems to be insensitive to the
 965 number of objects in a scene after about ten primitives.

966 5.2. The Effects of Reference Remapping

967 To measure the benefit of sorting and remapping references in WARP, a
 968 comparison to a non-remapping version of WARP must be made. Table 2
 969 shows the runtimes and multiplications factors of the criticality benchmarks
 970 for remapping and non-remapping versions of WARP. Four different criti-
 971 cality cases are compared. The “Jezebel” test is a bare plutonium sphere,
 972 and is a standard criticality test [24]. The fission neutron rate from ^{239}Pu is
 973 balanced by the leakage rate from the 5.1 cm radius to give a k_{eff} of approx-
 974 imately 1. The “Homogenized Block” test consists of a single 20x20x20cm
 975 cube. The cell contains a single material, but that material has multiple
 976 isotopes in it. This particular material is a mixture of 1% ^{235}U enriched UO_2
 977 and water at a 1:1 ratio. The “Pin Cell” test consists of a bare UO_2 cylinder
 978 surrounded by a block of water. This test now has two materials, each with
 979 multiple isotopes, and two cells. The water block dimensions are 10x10x50cm
 980 and the pin has a 1cm diameter and a 40cm length. The “Hex assembly” test
 981 consists of 631 bare UO_2 cylinders laid out in a hexagonal lattice surrounded
 982 by a 84x84x84cm cube of water. The material compositions, densities, and
 983 the cylinder dimensions are identical to the pin cell test case, but since this
 984 test has two orders of magnitude more objects, it serves to highlight the ef-
 985 fect of introducing many geometric objects into the problem and will further
 986 validate that the geometry processing routines work correctly if consistent
 987 results are obtained. All the cases have vacuum boundary conditions.

988 The remapping version performs much better than the non-remapping
 989 version when more complexity is added to the problem. For problems where
 990 neutrons die out very quickly, such as the Jezebel test, remapping does not
 991 benefit the simulation much because there are very few inner loop iterations
 992 done compared to a more complex problem, such as the assembly test. One
 993 of main benefits of remapping is that it eliminates completed neutrons from
 994 the GPU’s address space. If remapping is not done, the GPU does not know
 995 where the active neutrons are and must check if each neutron is done at
 996 every kernel launch. This leads to spending a lot of time checking already
 997 completed data. In problems where neutrons die out quickly, a neutron that
 998 is terminated on the first iteration may only be checked a small number
 999 of times compared to a problem where neutrons do not die out quickly. For

Table 2: Comparison of the non-remapping and remapping versions of WARP for the four criticality test cases. 20/40 discarded/active criticality cycles and 10^6 histories per cycle.

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

example, if a problem only takes five inner loop iterations to process an entire neutron batch, a neutron that completes in the first iteration is only checked 4 additional times and 80% of its data loads result in no work being done. In a problem that takes one hundred iterations, a neutron that completes in the first iteration is checked 99 additional times and 99% of its data loads results in no work being done. Remapping results in 100% of a neutrons loads being used for useful work, since no already completed data is ever loaded. This is a major benefit in slowly-attenuating batches where stale data could be loaded repeatedly, resulting in wasted time.

These effects of remapping are more clearly seen in the trends of Figure 15. This figure shows the rate at which each version is able to process neutrons vs. the number of active neutrons left in the batch for the homogenized block and the hex assembly. The data in the figure was quite noisy when there are few neutrons left to process since the iteration time is very small (~ 1 ms) then. The timer used had trouble resolving times at this rate, and the data shown in the figure has been smoothed with a window 11 data points wide, and linearly interpolated where the timings were too close together and gave Inf or NaN results for processing rates.

The non-remapping version is faster at large numbers of active neutrons (above $\sim 5 \times 10^5$) since accessing the entire dataset generally yields active

Figure 15: Neutron processing rate of WARP in the homogenized block and assembly test cases.

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

It therefore makes sense that the remapping version performs much better

Figure 16: Active neutrons per iteration in WARP in the homogenized block and assembly test cases.

when the simulation spends significant time in the sub-300k active neutron region on the processing rate curve (about where the lines cross). In the Jezebel, homogenized block, pin cell, and assembly tests, 90%, 77%, 91%, and 96% of the transport iterations occur in sub-300k region, respectively. It also makes sense that the relative number of *iterations* spent in the decades of Figure 15 is vaguely constant. The particles die out exponentially, and it takes roughly the same number of transport *iterations* to go from 10^7 to 10^6 active neutrons as it does to go from 10^2 to 10^1 active neutrons. This phenomenon manifests itself as the number of active neutrons per iteration being a straight line in Figure 16. The neutrons dying out exponentially is also why Figure 15 is plotted logarithmically with respect to active neutron count – to emphasize that equal spaces under the curves typically take the same number of iterations to complete. This shouldn't be confused for the amount of time the iterations take to complete, however. Iterations with more active neutrons typically take more time to complete, but have overall higher processing rates since the large neutron payload pays for overheads and latencies.

In a high leakage problem, like Jezebel, or a highly absorbing problem, like

Table 3: Summary table of the fraction of total kernel time spent in each WARP subroutine in criticality mode for each test case.

Remapping

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

Non-Remapping

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

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

Table 3 shows a breakdown of the amount of time spent in each subroutine in the remapping and non-remapping versions of WARP. The values shown are average values for an entire simulation. The values will also not add to 100% since there are many other kernels launched and memory copies made, but these are typically only done at problem startup and are negligible compared to those listed. The kernels included in the table are the compute kernels of both the inner and outer transport loops.

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

6. Conclusions

WARP has shown that GPUs are an effective platform for performing Monte Carlo neutron transport with continuous energy cross sections. Currently, WARP is the most detailed and feature-rich program in existence for performing continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs, but compared to production codes like Serpent

1092 and MCNP, WARP has limited capabilities. Despite WARP’s lack of fea-
1093 tures, its novel algorithm implementations show that high performance can
1094 be achieved on a GPU despite the inherently divergent program flow and
1095 sparse data access patterns. Remapping threads to active data is an effec-
1096 tive way of raising the processing rate when the number of active neutrons
1097 becomes small; this also reduces thread divergence in reaction kernels. Using
1098 a radix sort to do the remapping is effective since it segregates reactions into
1099 contiguous blocks, efficient since it can be done in place and in $O(kN)$ time,
1100 and can eliminate completed data from being accessed if slight modifications
1101 to the standard reaction number encodings are made.

1102 Most of the performance gain in remapping data references comes from
1103 being able to launch grids that are sized for only the active data rather than
1104 the entire dataset for both global and reaction kernels. A non-remapping
1105 algorithm does not keep track of where active data is, and therefore must
1106 launch a grid that covers the entire dataset. When the number of active
1107 neutrons drops below about 30% of the initial number, the overhead and
1108 memory bandwidth cost of launching these extra threads, which only load a
1109 “done” bit and return, is more than the cost of performing the radix sort and
1110 edge detection. The majority of the transport iterations occur while there
1111 are less than 30% of the initial neutrons left, and remapping references is
1112 usually worthwhile.

1113 Using the NVIDIA OptiX ray tracing framework was also shown to be an
1114 effective way to handle the geometry representation in WARP. OptiX is flex-
1115 ible, allows attachment of material and cell number to individual geometric
1116 primitives, can perform surface detection with a randomly-distributed and
1117 directed dataset, can incorporate the remapping vector created by a radix
1118 sort, and is fast enough to be used in WARP. The acceleration structures
1119 that OptiX can automatically build over the scene geometry was the initial
1120 reason for using it, and it was determined that the BVH builder and tra-
1121 verser provide the best performance as does using mesh primitive instancing
1122 rather than a transform node approach. The number of objects present in
1123 the scenes in reactors is small compared to many rendering scenes, and the
1124 SBVH acceleration structure does not perform as well. This is presumably
1125 due to some additional overhead related to traversing the objects that is not
1126 offset when few objects (less than a few hundred thousand) are present in
1127 the scene. Primitive instance provides better performance since using trans-
1128 form nodes requires traversing a deeper geometry tree, which also has more
1129 (redundant) data associated to it.

As mentioned in Section 1, a forthcoming companion paper is planned that will benchmark WARP against Serpent and MCNP using identical cross section libraries and problem geometries. In the paper, the relative accuracy and speed of WARP will be determined by comparing it to these production-level Monte Carlo neutron transport codes.

Acknowledgements

This research is based upon work partially supported by the U.S. Department of Energy National Nuclear Security Administration under Award Number DENA0000979 through the Nuclear Science and Security Consortium: <http://nssc.berkeley.edu>.

References

- [1] F. B. Brown, W. R. Martin, Monte carlo methods for radiation transport analysis on vector computers, *Progress in Nuclear Energy* 14 (1984) 269 – 299.
- [2] J. L. Vujic, W. R. Martin, Vectorization and parallelization of a production reactor assembly code, *Progress in Nuclear Energy* 26 (1991) 147 – 162.
- [3] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, K. Asanović, Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators, *SIGARCH Comput. Archit. News* 39 (2011) 129–140.
- [4] E. Z. Zhang, Y. Jiang, Z. Guo, X. Shen, On-the-fly elimination of dynamic irregularities for gpu computing, *SIGPLAN Not.* 46 (2011) 369–38.
- [5] A. G. Nelson, Monte carlo methods for neutron transport on graphics processing units using cuda, 2009. M.S. Thesis.
- [6] T. Liu, X. Du, W. Ji, X. G. Xu, F. B. Brown, A comparative study of history-based versus vectorized monte carlo methods in the gpu/cuda environment for a simple neutron eigenvalue problem, *Supercomputing in Nuclear Applications and Monte Carlo (SNA+MC)*, Paris, France.

- 1160 [7] Q. Xu, G. Yu, K. Wang, Research on gpu acceleration for monte carlo
1161 criticality calculation, Supercomputing in Nuclear Applications and
1162 Monte Carlo (SNA+MC), Paris, France.
- 1163 [8] Q. Xu, G. Yu, X. Wu, K. Wang, A gpu-based local acceleration strategy
1164 for monte carlo neutron transport, ANS Winter Meeting, San Diego,
1165 California.
- 1166 [9] M. Harris, S. Sengupta, J. D. Owens, Parallel prefix sum (scan) with
1167 cuda, in: H. Nguyen (Ed.), GPU Gems 3, Addison Wesley, 2007, pp.
1168 851 – 876.
- 1169 [10] CUDA Toolkit 5.0, CURAND Guide, NVIDIA Co, Santa Clara, CA,
1170 pg-05328-050_v02 edition, Sept. 2012.
- 1171 [11] OptiX Programming Guide, NVIDIA Co, Santa Clara, CA, v3.0 edition,
1172 Nov. 2012.
- 1173 [12] X.-. M. C. Team, MCNP - A General Monte Carlo N-Particle Transport
1174 Code, Version 5, Los Alamos National Laboratory,, Los Alamos, NM,
1175 volume i: overview and theory edition, 2003. (Revised 2/1/2008).
- 1176 [13] J. Leppänen, Development of a new monte carlo reactor physics code,
1177 2007. D.Sc. Dissertation.
- 1178 [14] A. Scopatz, S. Johnson, P. Romano, P. Wilson, N. Touran, K. Huff,
1179 C. Dembia, E. Relson, E. Biondo, M. Gidden, C. Bates, K. Manalo,
1180 Pyne: The nuclear engineering toolkit, 2014.
- 1181 [15] J. Levites, S. Jones, Inside kepler - world’s fastest and most efficient
1182 accelerator, Presented at the 2012 GPU Technology Conference, 2012.
- 1183 [16] E. Eshelman, Gtx titan, nvidia’s compute geforce - feb 2013, Presented
1184 at the Boston HPC & GPU Meetup Group, 2013.
- 1185 [17] J. Leppänen, Two practical methods for unionized energy grid construc-
1186 tion in continuous-energy monte carlo neutron transport calculation,
1187 Annals of Nuclear Energy 36 (2009) 878 – 885.
- 1188 [18] OpenMC Documentation, Cambridge, MA, 0.5.3 edition, 2013.
1189 [Http://mit-crp.github.io/openmc/index.html](http://mit-crp.github.io/openmc/index.html).

- 1190 [19] P. K. Romano, B. Forget, The openmc monte carlo particle transport
1191 code, *Annals of Nuclear Energy* 51 (2013) 274 – 281.
- 1192 [20] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific
1193 tools for Python, 2001–.
- 1194 [21] CUDA C Programming Guide, NVIDIA Co, Santa Clara, CA, pg-02829-
1195 001_v5.5 edition, 2013.
- 1196 [22] D. Knuth, Optimum binary search trees, *Acta Informatica* 1 (1971)
1197 14–25.
- 1198 [23] P. L’Ecuyer, Tables of linear congruential generators of different sizes
1199 and good lattice structure, *Mathematics of Computation* 68 (2009) 249–
1200 260.
- 1201 [24] O. N. E. Agency, International Handbook of Evaluated Criticality Safety
1202 Benchmark Experiments, Nuclear Energy Agency, OECD, 1995.