# Comparison of nested geometry treatments within GPU-based Monte Carlo neutron transport simulations of fission reactors [a]

The International Journal of High Performance Computing Applications
XX(X):2–39

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE



**Elliott Biondo, Thomas Evans, Seth Johnson, and Steven Hamilton**

*Prepared using* sagej.cls

**Abstract**

Monte Carlo (MC) neutron transport provides detailed estimates of radiological quantities within fission reactors. This involves tracking individual neutrons through a computational geometry. CPU-based MC codes use multiple polymorphic tracker types with different tracking algorithms to exploit the repeated configurations of reactors, but virtual function calls have high overhead on the GPU. The Shift MC code was modified to support GPU-based tracking with three strategies: dynamic polymorphism with virtual functions, static polymorphism, and a single tracker type with tree-based acceleration. On the Frontier supercomputer these methods achieve 77.8%, 91.2%, and 83.4%, respectively, of the tracking rate obtained using a specialized tracker optimized for rectilinear-grid-based reactors. This indicates that all three methods are suitable for typical reactor problems in which tracking does not dominate runtime. The flexibility of the single tracker method is highlighted with a hexagonal-grid microreactor problem, performed without hexagonal-grid-specific tracking routines, providing a 2.19× speedup over CPU execution.

**Keywords**

Monte Carlo, radiation transport, GPU computing, nuclear reactor analysis, computational geometry

## 1 Introduction

Nuclear reactors account for nearly 20% of electricity production in the United States, with lifecycle greenhouse gas emissions 17–29× less than coal-fired power plants per unit of energy generated (Center for Sustainable Systems 2022). These reactors derive energy from induced nuclear fission, a process in which a free neutron is captured by a heavy *fuel* nucleus, causing it to split into multiple lighter nuclei. Each fission releases heat, as well as additional free neutrons that may induce subsequent fissions, perpetuating the process. The pressurized water reactor (PWR)—the most common reactor design worldwide—consists of a *core* composed of *assemblies*, each of which is composed of fuel *rods*, as shown in Figure 1. Water is pumped through the core to

Oak Ridge National Laboratory, Oak Ridge, TN, USA

**Corresponding author:**
Elliott Biondo, Oak Ridge National Laboratory, 1 Bethel Valley Rd., Oak Ridge, TN, 37830, USA.
Email: veb@ornl.gov

*Prepared using* sagej.cls *[Version: 2017/01/17 v1.20]*

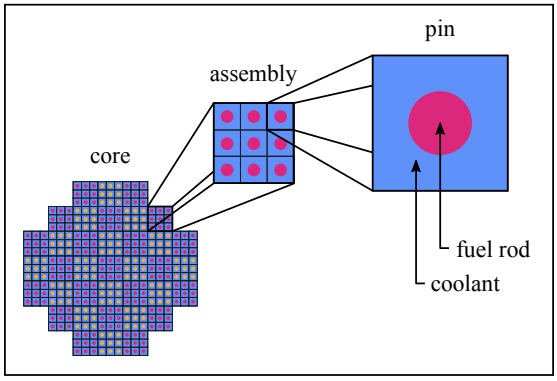



**Figure 1.** Simplified diagram of a generic PWR core shown as a 2D slice. Fuel rods are shown in red and yellow, ostensibly with different fuel compositions.

extract heat, and electricity is generated with a thermodynamic power cycle via the expansion of steam through a turbine.

Significant computational modeling and simulation is required for reactor design, licensing, and operation. Monte Carlo (MC) neutron transport is the preeminent method for obtaining high-fidelity estimates of radiological quantities because of its continuous (i.e., non-discrete) treatment of space, direction, and energy dimensions. This stochastic method involves simulating neutron *histories*—the circuitous paths that individual neutrons take within a reactor—using a random walk technique. By simulating a large number of histories, accurate statistical estimates of radiological quantities can be deduced. A key radiological quantity is the effective neutron multiplication factor ($k_{\text{eff}}$), defined as the average number of neutrons born from fission that induce subsequent fissions. A $k_{\text{eff}}$ of 1 indicates that the reactor is *critical*, i.e., operating at steady state, whereas $k_{\text{eff}} < 1$ and $k_{\text{eff}} > 1$ indicate that the fission rate will decrease or increase over time, respectively. Estimates of $k_{\text{eff}}$ are obtained using MC via a power iteration scheme (Lieberoth 1968).

Simulating neutron histories requires *tracking* the positions of neutrons within a computational representation of the reactor, usually a constructive solid geometry (CSG) model. This is accomplished in a fashion similar to the ray tracing techniques used in computer graphics rendering. With CSG, models are constructed from surface primitives (e.g., planes, cylindrical shells, spherical shells) and Boolean logic operations to form *cells*, i.e., closed regions with uniform material properties, as demonstrated in Figure 2. Tracking operations such as those summarized in Table 1 are implemented by querying the surfaces that comprise each cell.

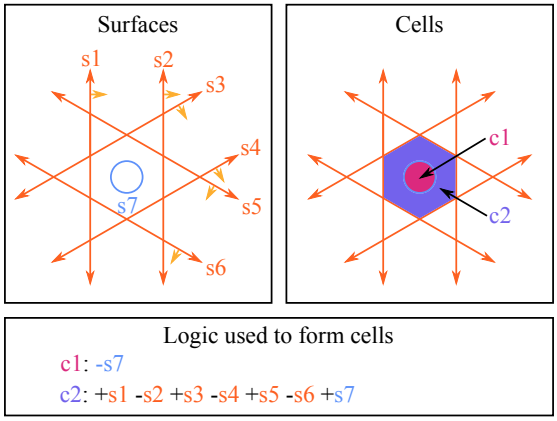



**Figure 2.** Demonstration of the CSG construction process, shown as a 2D slice. Plane surfaces (orange) have surface normals (yellow) denoting which side of the plane is positive. The outside of the blue cylindrical surface is considered positive. Cell construction logic is used to form cells 1 and 2. The cell 2 logic can be read as "the intersection of the space on the positive side of surface 1, the negative side of surface 2, the positive side of surface 3, ... etc."

**Table 1.** High-level geometry tracking operations required by MC transport codes.

| Tracking operation | Inputs | Output |
|---|---|---|
| find_cell | (1) position | (1) cell containing the position |
| distance_to_surface | (1) position<br>(2) direction | (1) distance to the next surface<br>(2) next surface |
| move_within_cell | (1) position<br>(2) direction<br>(3) distance | (1) new position |
| cross_surface | (1) position<br>(2) current cell<br>(3) current surface | (1) new cell after crossing the surface |
| change_direction | (1) new direction | - |

The nested and repeated structures that comprise reactors receive special treatment for performance and user convenience. A CSG *universe* is a contiguous geometric region composed of one or more cells that can be embedded within one or more *parent* cells (West et al. 1979). *Array* universes are special universe types in which cells comprise a structured mesh. The cells in an array universe must be filled with other universes which may be either CSG universes or other array universes. Examples of CSG universes embedded in rectilinear and hexagonal array universes are shown in Figure 3. Some fission reactors, including PWRs, can be modeled with three *levels* of

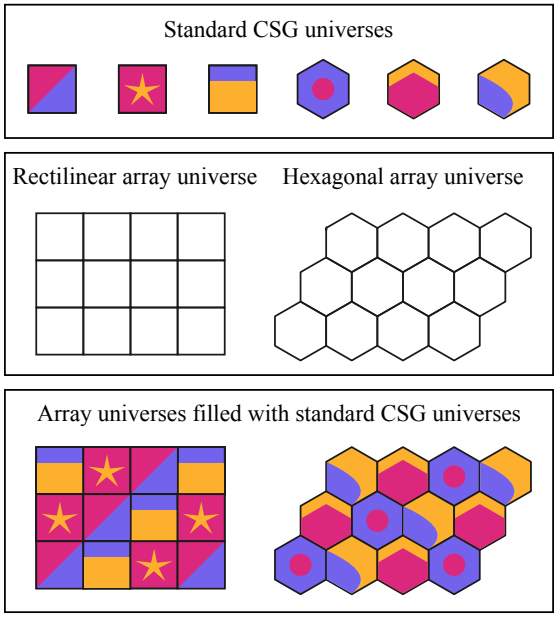



**Figure 3.** Demonstration of CSG universes nested within rectilinear and hexagonal array universes, shown as a 2D slice. The rectilinear array universe is shown with uniform spacing for simplicity, but non-uniform spacing is required to support typical reactor applications.

universes: an array universe representing the core, with each core array cell filled with an assembly array universe, and each assembly array cell filled with a CSG universe containing fuel rods surrounded by coolant (referred to as a *pin* within this work).

The use of array universes in concert with CSG universes provides tracking performance benefits. When determining what cell contains a given point, the hierarchical configuration of nested universes can be exploited by first finding which assembly contains the point, then which pin within the assembly, and finally which cell within the pin. This is loosely analogous to standard ray tracing acceleration structures such as bounding volume hierarchies (BVHs) (Ericson 2004) or $k$-d trees (Bentley 1975). In addition, because CSG and array universes represent geometry differently, the tracking operations summarized in Table 1 are implemented separately for each universe type, allowing each to be optimized. Within this work, objects that carry out these tracking operations are referred to as *trackers*. Thus, a CSG universe has a corresponding CSG tracker, a rectilinear array universe has a rectilinear array tracker, etc. Array trackers can implement tracking operations significantly more efficiently than CSG trackers by exploiting the regular structure of array universes.

When using multiple tracker types, polymorphism can be employed to call tracking functions. On the CPU with C++, this is accomplished by implementing tracker functions as virtual functions of a tracker base class. However, using GPU programming models (e.g., HIP, CUDA), it is not clear whether the performance benefits of different tracker types outweigh the overhead associated with virtual function calls (Zhang et al. 2021).

*Shift* is a general-purpose MC radiation transport code developed at Oak Ridge National Laboratory (ORNL) capable of simulating the behavior of neutrons and high-energy photons for fission, fusion, and national security applications (Pandya et al. 2016). Shift, written in C++ with abstracted HIP/CUDA device programming models, supports both CPU and GPU execution (Hamilton and Evans 2019) and is designed to scale effectively from laptops to leadership-class supercomputers. On the CPU, Shift uses the Oak Ridge Adaptable Nested Geometry Engine (ORANGE) (Johnson et al. 2023),* which includes CSG, rectilinear array, and hexagonal array trackers, leveraged via virtual functions. In Shift, GPU polymorphism has previously been avoided by supporting only a single universe/tracker type.

Prior to this work, Shift only supported the Reactor Tool Kit (RTK) universe type on the GPU. RTK is a special-purpose universe type that models a full reactor core, and is separate from the ORANGE package. A single RTK universe always consists of (1) a rectilinear array representing the core, (2) rectilinear arrays representing assemblies, and (3) cuboids containing concentric cylinders, representing pins. The RTK tracker is optimized to track neutrons through this specific configuration. As a result, the RTK tracker is expected to provide the best possible performance. However, RTK has limited applicability because not all fission reactors can be modeled with nested rectilinear grids. Numerous reactor designs consist of hexagonal assemblies (Habush and Harris 1968; Hejzlar et al. 2013; Betzler et al. 2020), and pebble bed reactors consist of irregular configurations of spherical fuel elements (Andreades et al. 2016; Mulder and Boyes 2020). In addition, MC neutron and photon transport is used for other applications such as nuclear fusion reactors (Juarez et al. 2021; Kos et al. 2023) and accelerator devices (Radel and Van Abel 2016; Nelson et al. 2022), which involve geometries with complexity far beyond rectilinear grids.

In this work, three methods for GPU-based multi-universe tracking were implemented in ORANGE to assess the trade-offs between single and multiple tracker

*The ORANGE package is shared between Shift and the Celeritas MC code, used for high-energy physics detector analysis (Johnson et al. 2021).

types. The timing results for Shift $k_{\text{eff}}$ calculations on the Summit (Oak Ridge Leadership Computing Facility 2023b) and Frontier (Oak Ridge Leadership Computing Facility 2023a) supercomputers at ORNL were recorded for each of the three methods and compared to RTK timing results. The three methods are outlined as follows. The dynamic polymorphism (DP) method employs multiple tracker types with virtual functions. The static polymorphism (SP) method employs multiple tracker types with switch statements. The single tracker (ST) method avoids polymorphism by using a single tracker type for all universe types. The ST method is accomplished using *pseudo-array* universes (a term specific to this work), which are arrays modeled as CSG universes and tracked upon with a CSG tracker. Although this approach forfeits the aforementioned benefits of array trackers, this is counteracted with a bounding interval hierarchy (BIH) acceleration structure (Wächter and Keller 2006).

The remainder of this work proceeds as follows. Section 2 provides background on how Shift solves for $k_{\text{eff}}$, including the role of tracking operations. Section 3 introduces tracking algorithms for CSG and array universes, multi-universe geometries, and RTK universes. Section 4 describes the implementation of each of the three experimental multi-universe GPU tracking methods. Section 5 describes the computer hardware on Summit and Frontier used for this work. Section 6 provides a performance comparison of the three methods using a model of the NuScale small modular reactor (SMR) (NuScale Power 2023), composed of rectilinear assemblies. Section 7 demonstrates the flexibility of the ST method on the Empire microreactor benchmark problem (Lee et al. 2020; Matthews et al. 2021) composed of hexagonal assemblies.

## 2 Background

This section provides background on how Shift solves for $k_{\text{eff}}$. Section 2.1 describes the neutron transport equation formulated with $k_{\text{eff}}$ as the dominant eigenvalue. Section 2.2 describes the power iteration solution method and the role of tracking operations in the MC random walk process. Section 2.3 describes the Shift implementation of MC power iteration on the GPU.

### *2.1 Neutron transport equation*

Neutrons interact with fuel and non-fuel nuclei via a variety of different mechanisms which are broadly organized into three categories:

1. absorption: a neutron is captured by a nucleus,
2. fission: a neutron is captured by a nucleus, which subsequently breaks apart,

3. scattering: a neutron interacts with a nucleus, effectively changing the neutron's kinetic energy and direction.

As a neutron travels through a material, the probability that it will undergo one of these interactions is quantified using nuclear *cross sections*, functions which depend on the energy of the neutron, the types of nuclei within the material, and the density and temperature of the material. Absorption and fission cross sections tend to be negatively correlated with neutron energy. As a result, within a reactor core, neutrons typically scatter $\sim 10^1$ times before being terminated via absorption/fission. These effects are quantified in the neutron transport equation:

$$(\hat{\mathrm{T}} - \hat{\mathrm{S}})\psi = \frac{1}{k_{\mathrm{eff}}}\hat{\chi}\hat{\mathrm{F}}\psi \,, \tag{1}$$

where $\hat{\mathrm{T}}$ is the transport operator, $\hat{\mathrm{S}}$ is the scattering operator, $\hat{\chi}$ is the energy spectrum of neutrons born from fission, $\hat{\mathrm{F}}$ is the fission operator, and $\psi$ is the neutron flux, which describes the number of neutrons that pass through a 2D area per unit time per unit angle. The $\hat{\mathrm{T}}$, $\hat{\mathrm{S}}$, and $\hat{\mathrm{F}}$ operators depend on nuclear cross sections, but these relationships are omitted for brevity. By collecting the operators in a single term,

$$\hat{\mathrm{A}} \equiv \hat{\mathrm{F}}(\hat{\mathrm{T}} - \hat{\mathrm{S}})^{-1}\hat{\chi} \,, \tag{2}$$

Equation 1 can be formulated as a standard eigenvalue problem,

$$k_{\mathrm{eff}}\mathrm{f} = \hat{\mathrm{A}}\mathrm{f}, \tag{3}$$

where $k_{\mathrm{eff}}$ is the dominant eigenvalue of $\hat{\mathrm{A}}$ and f, the eigenvector, is the fission source given by

$$\mathrm{f} = \hat{\mathrm{F}}\psi. \tag{4}$$

Physically, f is a probability density function describing the spatial distribution of fission neutrons. Like $k_{\mathrm{eff}}$, this distribution is not known a priori. Due to the complexity of the geometry and cross sections, Equation 3 must be solved numerically for most practical cases.

## 2.2 Monte Carlo power iteration

The eigenvalue problem presented in Equation 3 can be solved with MC power iteration. With this method, $k_{\mathrm{eff}}$ and f are solved iteratively by applying the standard

power iteration eigenvalue algorithm (von Mises and Pollaczek-Geiringer 1929) to Equation 3,

$$\mathrm{f}^{(n+1)} = \frac{1}{k_{\mathrm{eff}}^{(n)}} \hat{\mathrm{A}} \mathrm{f}^{(n)}, \tag{5}$$

$$k_{\mathrm{eff}}^{(n+1)} = k_{\mathrm{eff}}^{(n)} \frac{\langle \mathrm{f}^{(n+1)} \rangle}{\langle \mathrm{f}^{(n)} \rangle}, \tag{6}$$

where $\langle \cdot \rangle$ denotes integration over space, energy, and angle. This iteration scheme is carried out using Algorithm 1, where run_histories is an MC neutron transport simulation. As seen in Algorithm 1, iterations of Equations 5 and 6, referred to as *cycles*, are carried out in two stages. First, a sufficient number of *inactive* cycles are run to converge the shape of f, with intermediate estimates of $k_{\mathrm{eff}}$ discarded. Then *active* cycles are run, accumulating $k_{\mathrm{eff}}$ estimates to produce the final $k_{\mathrm{eff}}$ with statistical uncertainty. Statistical estimates of f and other radiological quantities such as neutron flux or specific reaction rates are also obtained from active cycles (omitted from Algorithms 1 and 2 for brevity). These estimates, often calculated within geometry cells or the volume elements of a superimposed mesh, are called *tallies*. Active cycles typically require longer compute times than inactive cycles because of the additional overhead associated with tallies.

**Algorithm 1.** MC power iteration algorithm for calculating the converged $k_{\mathrm{eff}}$ using $m_i$ inactive cycles, $m_a$ active cycles, and $n$ histories per cycle.

```
1 procedure MC_POWER_ITERATION(m_i, m_a, n)
2     Set f to initial guess
3     for inactive_cycle ∈ [1, m_i]
4         k_eff, f = run_histories(f, n)
5     for active_cycle ∈ [1, m_a]
6         k_eff, f = run_histories(f, n)
7         accumulate k_eff
8     return average of all stored k_eff
```

The run_histories MC transport function used in Algorithm 1 is shown in Algorithm 2. For each of $n$ histories, a neutron birth position is sampled from the supplied guess for the fission source distribution. A random walk technique is then used to move the neutron through a computational representation of the reactor. This algorithm makes use of the geometry tracking operations defined in Table 1. Tracking operations typically account for 20% or less of the total runtime within an MC simulation (Hamilton and Evans 2019). Cross section calculations make up the

plurality of runtime because of the large number of memory fetch operations. By definition, the random walk process imposes divergent neutron paths across histories, resulting in random memory access. As a result, MC simulations tend to be latency bound.

**Algorithm 2.** MC neutron transport algorithm for simulating $n$ neutron histories born from f, in order to update estimates of $k_{\text{eff}}$ and f. This algorithm uses the geometric tracking operations specified in Table 1.

```
1  procedure RUN_HISTORIES(f, n)
2      for history ∈ [1, n]
3          sample position (r) from f
4          sample energy (E) from fission energy spectrum
5          sample direction (Ω) isotropically
6          cell (c) = find_cell(r)
7          while true
8              calculate the cross section (Σ) in c
9              distance (d), surface (s) = distance_to_surface(r, Ω)
10             sample # of mean free paths (τ) before event
11             while d < τ/Σ
12                 τ = τ - Σ × d
13                 r = move_within_cell(r, Ω, d)
14                 c = cross_surface(r, c, s)
15                 calculate Σ in c
16                 d, s = distance_to_surface(r, Ω)
17             r = move_within_cell(r, Ω, τ/Σ)
18             sample event type (ξ)
19             if ξ == scatter
20                 update E and Ω
21             else if ξ == fission
22                 calculate and store an estimate of k_eff
23                 store fission sites
24                 break
25             else if ξ == absorption
26                 break
27     calculate updated k_eff from stored k_eff values
28     calculate updated f from stored fission sites
29     return updated k_eff, updated f
```

## 2.3 GPU-based implementation in Shift

On the CPU, Shift carries out MC power iteration using the MC neutron transport simulation algorithm shown in Algorithm 2. On the GPU, Shift uses an *event-based* MC algorithm. As in Algorithm 2, the event-based algorithm involves simulating $n$

histories per cycle; however, operations are reordered to take advantage of single instruction, multiple threads (SIMT) parallelism. All operations—including birth, tracking operations, and collisions—are performed on a vector of histories. This vector is masked in order to only perform operations on applicable histories. Each of the tracking operations in Table 1 is called for a vector of histories via a kernel of the same name. Full details of the event-based transport algorithm are found in Hamilton and Evans (2019). This approach leads to smaller kernel sizes and therefore increased occupancy and higher tracking rates.

## 3 Tracking algorithms

This section discusses the implementation of the Table 1 tracking operations for different universe types. For brevity, only the two most algorithmically interesting tracking operations are discussed: find_cell and cross_surface. Section 3.1 provides possible implementations of these operations for CSG and rectilinear array universes, where the latter are shown to have significantly lower time complexity. Hexagonal array universe tracking algorithms can be implemented using the same strategy as that used for rectilinear arrays, with the same time complexity. However, hexagonal array indexing is considerably more complicated, so these algorithms are omitted here for simplicity. Section 3.2 provides the implementation of multi-universe tracking on the CPU within ORANGE. Section 3.3 provides the tracking algorithms for the reactor-specific RTK universe type, used by Shift on the GPU.

### *3.1 Single universe tracking*

Possible implementations of find_cell and cross_surface for a CSG tracker are shown in Algorithms 3 and 4. Because CSG universes have no underlying structure, find_cell involves conducting an $O(N)$ search over all of the cells within the universe. The cross_surface implementation relies on a list of *neighbor* cells connected to each surface, which can be generated as a preprocessing step. This neighbor list can then be searched in $O(N)$ time.

**Algorithm 3.** Possible CSG tracker version of find_cell.

```
1 procedure FIND_CELL(pos)
2     for cell ∈ cells
3         if cell contains pos
4             return cell
```

**Algorithm 4.** Possible CSG tracker version of cross_surface.

```
1 procedure CROSS_SURFACE(pos, cell, surf)
2     for new_cell ∈ neighbors(surf)
3         if new_cell == cell
4             continue
5         if new_cell contains pos
6             return new_cell
```

Possible implementations of find_cell and cross_surface for a rectilinear array tracker are shown in Algorithms 5 and 6. These algorithms have favorable time complexity relative to their CSG counterparts. By conducting a binary search over the mesh divisions, find_cell can be performed in $O(\log N)$ time. [†] The cross_surface operation is performed in $O(1)$ time because the surface of each cell is known to have only one neighbor. It requires the current cell, the $i/j/k$ dimension of the surface being crossed, and whether the surface is being crossed in the positive or negative direction.

**Algorithm 5.** Possible rectilinear array tracker version of find_cell.

```
1 procedure FIND_CELL(pos)
2     for dim ∈ [i, j, k]
3         ijk[dim] = binary_search(mesh[dim], pos[dim])
4     cell = ijk_to_cell(ijk)
5     return cell
```

**Algorithm 6.** Possible rectilinear array tracker version of cross_surface.

```
1 procedure CROSS_SURFACE(cell, dim, dir_sign)
2     ijk = cell_to_ijk(cell)
3     if dir_sign is positive
4         ijk[dim] = ijk[dim] + 1
5     else
6         ijk[dim] = ijk[dim] - 1
7     next_cell = ijk_to_cell(ijk)
8     return next_cell
```

[†]For rectilinear grids with uniform spacing, find_cell can be performed in $O(1)$ time by calculating the array indices directly in each dimension. However, the presence of gaps between adjacent assemblies in most reactors necessitates the use of non-uniform spacing and therefore a full binary search.

### *3.2 Multi-universe CPU tracking in ORANGE*

The algorithms shown in Section 3.1 show how possible implementations of CSG and array trackers operate within single universes. Here, tracking algorithms for geometries consisting of multiple nested universes are shown, as implemented on the CPU in ORANGE. The ORANGE CPU version of find_cell and cross_surface are shown in Algorithms 7 and 8. The find_cell algorithm recursively finds the cell within daughter universes until reaching the bottom-most (i.e., most embedded) level. In cross_surface, a surf_universe argument is supplied that denotes the top-most (i.e., least embedded) level for which the neutron is on a surface, noting that lower-level coincident boundary surfaces are removed during preprocessing. The cell on the other side of the surface is then found by recursing through daughters starting at this level. Both algorithms assume the existence of a polymorphic get_tracker function which returns a tracker object for a given universe, depending on its type.

**Algorithm 7.** Multi-universe version of find_cell.

```
1 procedure FIND_CELL(pos)
2     tracker = get_tracker(root_universe)
3     cell = tracker.find_cell(pos)
4     while cell.daughter
5         tracker = get_tracker(cell.daughter)
6         cell = tracker.find_cell(pos)
7     return cell
```

**Algorithm 8.** Multi-universe version of cross_surface. For simplicity, args can be assumed to be a struct containing the union of the arguments to the standard CSG and array versions of cross_surface.

```
1 procedure CROSS_SURFACE(surf_universe, args)
2     tracker = get_tracker(surf_universe)
3     next_cell = tracker.cross_surface(args)
4     while next_cell.daughter
5         tracker = get_tracker(next_cell.daughter)
6         next_cell = tracker.find_cell(args.pos)
7     return next_cell
```

### *3.3 RTK tracking*

As mentioned in Section 1, RTK is the reactor-specific universe type currently employed by Shift for GPU execution. RTK is a template universe type explicitly instantiated to contain the three nested levels of universes necessary to model reactors

such as PWRs with rectilinear configurations. An RTK universe consists of a rectilinear array core universe populated with rectilinear array assembly universes, each populated with pin universes. The pin universe is not a full, general-purpose CSG universe, but rather a limited CSG universe consisting of concentric cylinders within a rectangular cuboid cell.

The RTK version of find_cell and cross_surface are shown in Algorithms 9 and 10. These algorithms resemble the standard multi-universe tracking algorithms shown in Section 3.2, with two key differences. First, because there are exactly three levels of universes, the *while* loops over daughter universes can be unrolled (in practice, this is achieved through C++ template recursion). Second, the types of universes at each level are known at compile time. Whereas the multi-universe tracking algorithms shown in Section 3.2 rely on a polymorphic get_tracker function, Algorithms 9 and 10 can call non-polymorphic get_rect_tracker and get_csg_tracker functions. As a result of these simplifications, RTK is expected to be the most performant tracker type. However, the clear limitations on geometric complexity imposed by RTK motivate the implementation of general-purpose GPU tracking capabilities.

**Algorithm 9.** RTK version of find_cell.

```
1  procedure FIND_CELL(pos)
2      core_tracker = get_rect_tracker(core_universe)
3      core_cell = core_tracker.find_cell(pos)
4      assm_universe = core_cell.daughter
5      assm_tracker = get_rect_tracker(assm_universe)
6      assm_cell = assm_tracker.find_cell(pos)
7      pin_universe = assm_cell.daughter
8      pin_tracker = get_csg_tracker(pin_universe)
9      cell = pin_tracker.find_cell(pos)
10     return cell
```

## 4 Methodology: Multi-universe GPU tracking methods

This section describes the ORANGE implementation of the three experimental GPU-based multi-universe tracking methods explored in this work. The DP and SP methods, which rely on universe-specific tracker types, were implemented only for CSG and rectilinear array universes for the purposes of this work. The ST method, which relies on only a single tracker, supports CSG, rectilinear array, and hexagonal array universe types.

**Algorithm 10.** RTK version of cross_surface.

```
1  procedure CROSS_SURFACE(surf_universe, args)
2      if surf_universe.level == core
3          core_tracker = get_rect_tracker(surf_universe)
4          core_cell = core_tracker.cross_surface(args)
5          assm_universe = core_cell.daughter
6          assm_tracker
7                  = get_rect_tracker(assm_universe)
8          assm_cell = assm_tracker.find_cell(args.pos)
9          pin_universe = assm_cell.daughter
10         pin_tracker = get_csg_tracker(pin_universe)
11         new_cell = pin_tracker.find_cell(args.pos)
12     else if surface_universe.level == assembly
13         assm_tracker = get_arr_tracker(surf_universe)
14         assm_cell = assm_tracker.find_cell(args.pos)
15         pin_universe = assm_cell.daughter
16         pin_tracker = get_csg_tracker(pin_universe)
17         new_cell = pin_tracker.find_cell(args.pos)
18     else if surface_universe.level = pin
19         pin_tracker = get_csg_tracker(surf_universe)
20         new_cell = pin_tracker.find_cell(args.pos)
21     return new_cell
```

### *4.1 Dynamic polymorphism (DP) method*

This first method uses the standard multi-universe tracking algorithms put forth in Section 3.2, i.e., the approach used on the CPU in ORANGE. A tracker base class is created, defining the tracking operations listed in Table 1 as pure virtual methods. CSG and rectilinear array tracker classes inherit from the base class and implement the virtual functions. A polymorphic get_tracker function returns a pointer to either a CSG or rectilinear array tracker object. For a fair comparison, within CSG universes, the DP method uses BIH acceleration, which is described in Section 4.3.

### *4.2 Static polymorphism (SP) method*

Like the DP method, the SP method uses separate tracker types for CSG and rectilinear array universes, but this is achieved with static polymorphism. Using this approach, each tracker operation involves a switch statement predicated on the type of the current universe (represented by an enumeration, `UType`). For example, a possible implementation of find_cell is shown in Listing 1. This function takes three arguments: the universe identifier (`uid`) of the current universe, the position, and a set of geometry parameters which supply a mapping between `uid` and `UType`. A CSG or rectilinear

tracker is created based on the `UType` of the universe specified by the `uid`. The find_cell method is then called on the tracker, and the resulting cell is returned.

**Listing 1**. Possible C++ implementation of find_cell using static polymorphism. This code block has equivalent logic to the code in Listings 2, 3, and 4.

```
__host__ __device__
Cell find_cell(UniverseId uid,
               Position pos,
               Params params)
{
  switch(params.types[uid])
  {
    case UType::CSG:
        auto tracker
            = CSGTracker(uid, params)
        return tracker.find_cell(pos);
    case UType::RectArray:
        auto tracker
            = RectArrayTracker(uid,
                               params)
        return tracker.find_cell(pos);
  }
}
```

In ORANGE, this behavior is achieved via a template metaprogramming approach which allows a single set of switch statement logic to be used by all tracking operations, as shown in Listings 2, 3, and 4. In Listing 2, a `Traits` struct is templated on `UType`, and template specializations are used to define the corresponding tracker type for each universe type. A `visit_universe_type` function returns the output of a given functor, which takes a `Traits` object as an argument. Listing 3 provides a `visit_tracker` function which returns the output of a supplied functor `f`, which takes a tracker of arbitrary type as an argument. This is achieved by wrapping `f` in a second functor—`f2`, which calls `f` for a given `Traits` object—and then calling `visit_universe_type` with this second functor.

Listing 4 shows how the `visit_tracker` function can be used to achieve polymorphism. A find_cell functor is first created. This functor, along with a `uid` and `params`, is passed to `visit_tracker`, and the resulting cell is returned. Upon compilation, code in Listings 2, 3, and 4 produce bitcode equivalent to that of Listing 1. The other Table 1 tracking operations are implemented in an identical fashion to the find_cell example in Listing 4. As was the case with the DP method,

CSG universes in the SP method also use BIH acceleration, which is described in Section 4.3. Alternatives to this template metaprogramming approach include the curiously recurring template pattern (CRTP) or `std::variant` with `std::visit`, which would likely perform similarly.

**Listing 2**. Simplified C++ function for visiting different universe types.

```
template<UType U>
struct Traits;
template<>
struct Traits<UType::CSG>
{
  using tracker_type = CSGTracker;
}
template<>
struct Traits<UType::RectArray>
{
  using tracker_type = RectArrayTracker;
}

template<class F>
inline constexpr
__host__ __device__ decltype(auto)
visit_universe_type(F&& f, UType ut)
{
  switch (ut)
  {
  case UType::CSG:
    return f(Traits<UType::CSG>{})
  case UType::RectArray:
    return f(Traits<UType::RectArray>{})
  }
}
```

**Listing 3**. Simplified C++ function for visiting different tracker types.

```
template<class F>
 __host__ __device__ decltype(auto)
visit_tracker(F&& f,
              UniverseId uid,
              Params params)
{
  auto f2 = [&](auto traits){
    return f(traits::tracker_type(
               uid,
               params));
    }

  return visit_universe_type(
           f2,
           params.universe_types[uid]);
}
```

**Listing 4**. C++ implementation of find_cell using the visit_tracker function shown in Listing 3.

```
auto find_cell = [&pos](auto&& tracker){
    return tracker.find_cell(pos);
    }
Cell cell =  visit_tracker(find_cell,
                           uid,
                           params)
```

### 4.3 Single tracker (ST) method

While the DP and SP methods benefit from tracking algorithms optimized for specific universe types, it is not clear that these benefits offset the potential performance pitfalls of polymorphism, i.e., virtual function calls in the case of the DP method and divergent execution paths in the case of the DP and SP methods. For comparison, instead of employing multiple polymorphic tracker types, the final approach uses a single CSG tracker for CSG, rectilinear array, and hexagonal array universes. This is accomplished by converting rectilinear and hexagonal array universes into CSG universes. To do so, array cells are explicitly modeled as CSG cells using the method shown in Figure 2. This conversion is done automatically in ORANGE, and the resulting CSG universes

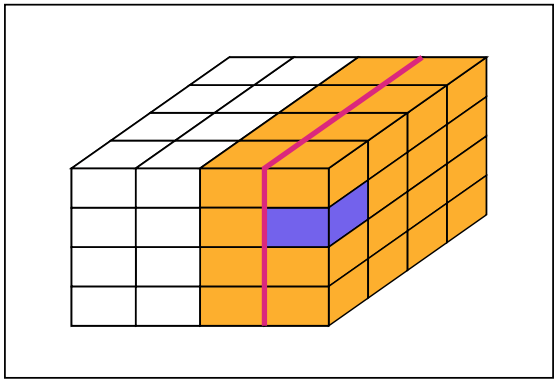


**Figure 4.** Example of a pseudo-rectilinear-array neighbor list. A neutron in the purple cell crossing the magenta surface has all 31 gold cells as neighbors.

are referred to as "pseudo-array universes," a term specific to this work. Modeling arrays in this fashion is not a new approach: it is the simplest way of modeling an array within an MC code without specialized array universe types.

Because pseudo-array universes are CSG universes, they use CSG tracking algorithms rather than the array versions that have improved time complexity. In other words, for pseudo-arrays, naive implementations of find_cell and cross_surface would each use linear searches over all cells or all neighbors, respectively. The cost of the simple neighbor-based approach is exacerbated by the large number of neighbor cells for pseudo-array surfaces, since planar surfaces in CSG geometries are unbounded. Figure 4 demonstrates the potential inefficiency: when crossing from the purple cell through the magenta surface, which bounds all 32 colored cells, the neighbor list includes all 31 gold cells. It is noted that this issue does not arise with cell-based neighbor lists, an alternative approach in which a mapping between each cell and its neighbor cells is generated dynamically during runtime (Harper et al. 2020).

To improve the performance of tracking operations for CSG universes, including pseudo-array universes, a BIH acceleration structure was implemented. The BIH construction process is demonstrated in Figure 5. First, axes-aligned bounding boxes are generated for each cell using a simple method that involves successively truncating the universe bounding box using the bounding planes associated with each of the cell's surfaces. Although the nascent implementation of this method in ORANGE does not yet guarantee minimum bounding boxes for arbitrarily complex cells, it is effective for the simple geometric shapes found within the reactor models in this work. Once bounding boxes are ascertained, a partition plane is then chosen, and bounding boxes

are partitioned into two sets according to the location of each bounding box center. For each of the two sets of partitioned bounding boxes, bounding planes are created by moving the partition to fully enclose all bounding boxes. By performing this process recursively on each set of bounding boxes, a binary tree structure is created, with edges specifying the half-spaces which contain all children.

One key feature of BIH trees is that for a given node, the half-spaces created by the bounding planes may overlap. This guarantees that each cell appears in the BIH tree exactly once, unlike $k$-D trees that must store a cell twice if its bounding box is bisected by a partition. This provides an advantage in the pseudo-array use case in which adjacent cells may have bounding boxes that overlap slightly because of floating point error. BIH trees can handle this case without significant performance consequences. BVH trees can also handle this overlap case. However, BVH trees may require more memory because they store six planes per node, significantly more than the two planes required by BIH trees. An open research question within the BIH construction process is how to choose partitions. For the purposes of this work, a standard surface area heuristic (SAH) partitioning scheme (Wald 2007) was used. This was implemented by evaluating three equally spaced candidate partitions per axes at each partitioning step. Candidate partitions were evaluated using a cost function balancing the number of bounding boxes and the total surface area of bounding boxes appearing on each side of the partition.

The find_cell and cross_surface functions are accelerated by traversing the BIH tree. For find_cell, traversal is terminated when a cell is found that contains the supplied point. For cross_surface, the traversal is terminated when a cell is found that contains the supplied point, excluding the cell in which the neutron originated. When traversing the BIH tree, both edge conditions must be tested at each node because the half-spaces are allowed to overlap. BIH trees are enabled for all CSG universes—not just pseudo-array universes. As mentioned in Sections 4.1 and 4.2, BIH trees are enabled for all CSG universes in the DP and SP methods as well.

## 5 Hardware description

All simulations were performed on the Summit and Frontier supercomputers at ORNL. Summit has 4608 compute nodes, each consisting of two 22-core IBM Power9 CPUs, and six NVIDIA Tesla V100 GPUs, each consisting of a single graphics complex die (GCD). One core per Power9 is reserved for system tasks, leaving 42 usable CPU cores per node. On the CPU side, 512 GB of RAM are available, and each V100 has 16 GB of RAM. Within this work, code was compiled on Summit with CUDA 11.5.2.

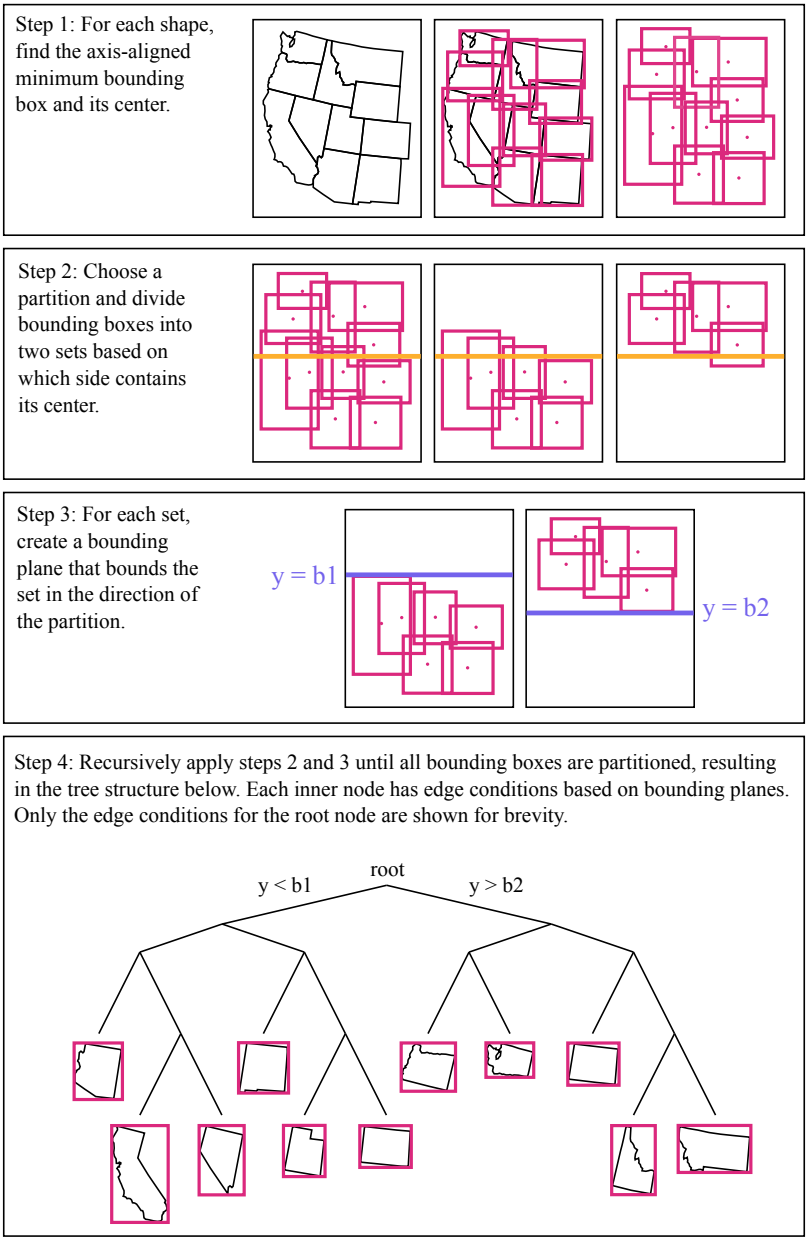



**Figure 5.** BIH construction process demonstrated in 2D using the states of the American West. Map outline from OpenClipart (2023).

Frontier has 9408 compute nodes, each consisting of a 64-core AMD 3rd Gen EPYC CPU, and four AMD Radeon Instinct MI250X, each consisting of two GCDs. Eight

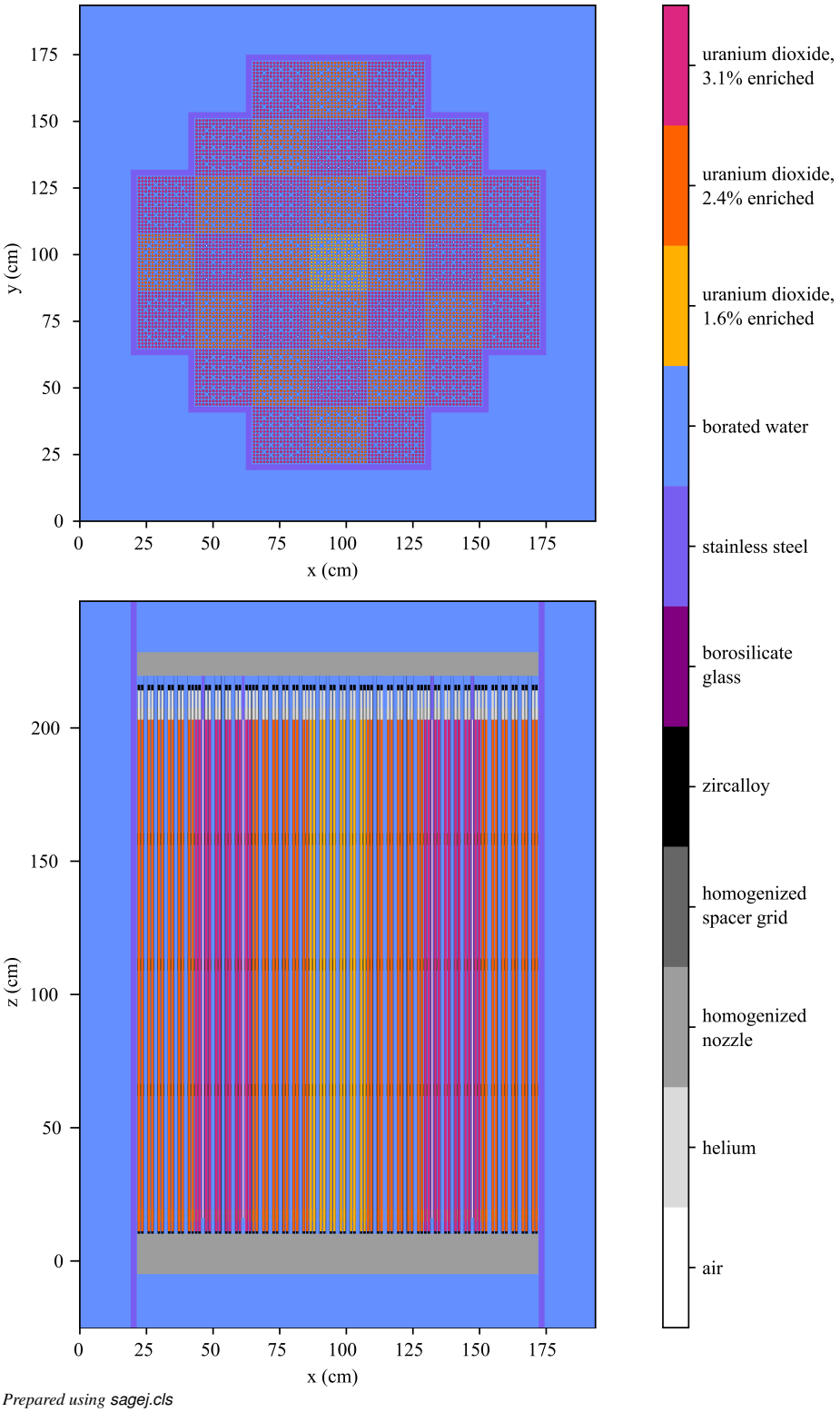
CPU cores are reserved for system tasks, leaving 56 usable CPU cores per node. On the CPU side, Frontier also has 512 GB of RAM, but each MI250X has 128 GB of RAM. As a result, Frontier has four times as much GPU RAM per GCD compared to Summit (64 GB vs. 16 GB). Within this work, code was compiled on Frontier with ROCm 5.6.0.

## 6 Performance testing

The performance of the three multi-universe GPU tracking methods described in Section 4 relative to RTK (described in Section 3.3) was assessed by obtaining timing results for a rectilinear-array-based reactor problem. A full-core model of the NuScale SMR (Smith 2017), shown in Figure 6, was chosen for this purpose. The small size of the NuScale design—about one-eighth the size of a typical PWR—is economically attractive because of its low capital cost (Black et al. 2019), and also permits detailed full-core MC analysis. Likewise, this problem served as the challenge problem for the ExaSMR project within the Exascale Computing Project, in which Shift was used for GPU-based coupled neutron transport / thermal hydraulics analysis (Merzari et al. 2023).

The NuScale design consists of 37 assemblies arranged in a rectilinear grid. Each assembly is a $17\times17$ array of pins and contains uranium dioxide fuel with a $^{235}$U enrichment of either 1.6%, 2.4%, or 3.1% (by mass). The 3.1%-enriched fuel assemblies in the inner circle contain borosilicate glass burnable neutron absorber rods. Spacer grids and nozzles have been homogenized into slabs for simplicity. For this analysis, the fresh (i.e., non-depleted) fuel version of the problem was used. Fresh fuel contains many fewer nuclides than depleted fuel, thereby minimizing the time required to calculate cross sections and maximizing the relative time spent on tracking operations. As a result, inactive cycles within this problem (where no time is spent on tallies) represent the scenario in which tracking operations are expected to comprise the largest fraction of runtime.

The computational representation of this reactor takes on several different forms at runtime when testing the four tracking methods. With the RTK method, the entire core consists of a single RTK universe. With the DP and SP methods, the core consists of an array universe with embedded array universes representing assemblies, each containing CSG universe pins. BIH acceleration is used within these CSG pin universes. This is beneficial because each pin consists of a large number of cells ($\sim 10^2$); due to the complexity of the model, pins are split into 40 axial regions. With the ST method,




*Prepared using* sagej.cls

**Figure 6.** Midplane radial (top) and axial (bottom) slices of a full core model of the NuScale SMR from Smith (2017).

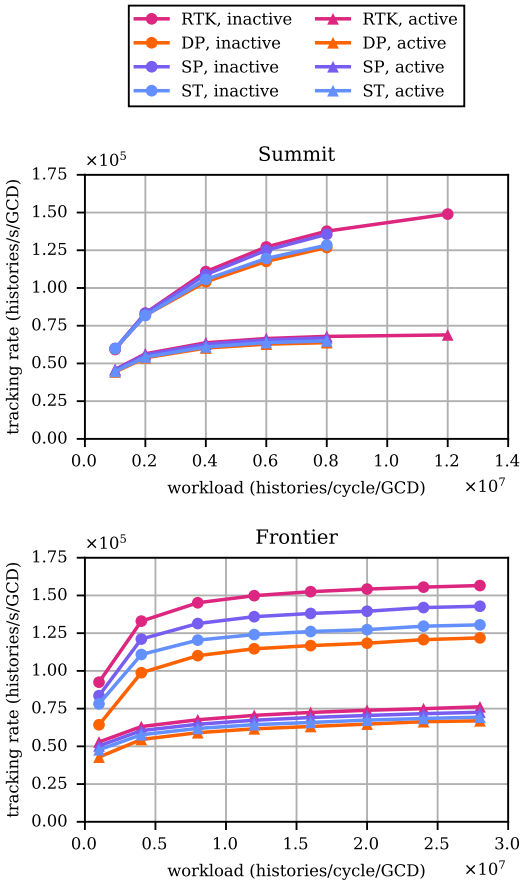



**Figure 7.** Neutron tracking rates as a function of the number of histories run by each GPU GCD on Summit and Frontier for the NuScale SMR problem.

the core, assemblies, and pins are all CSG universes, and each benefits from BIH acceleration.

For each of the four tracking methods, performance testing was conducted by measuring the tracking rate, i.e., the number of histories simulated per unit time, over a sweep of *workloads*. Here, workload refers to the number of histories per cycle assigned to each GPU GCD. Each trial consisted of 10 inactive cycles and 10 active cycles using all available GPU GCDs on a single node of Summit or Frontier. During active cycles, the neutron flux was tallied on a $119 \times 119 \times 30$ superimposed rectilinear mesh (425,830 mesh volume elements).

### *6.1 Tracking rate results*

Tracking rate results are shown in Figure 7, noting that both plots have different $x$ scales but the same $y$ scale. As expected, the tracking rate increased with workload for all four methods, because large workloads allow the GPU to more effectively hide the latency of memory fetches associated with cross section calculations. Inactive cycle tracking rates are higher because no time is spent processing tally results. On Frontier, tracking rates for all four methods are near their asymptotic limit at a workload of $2.8 \times 10^7$ histories/cycle/GCD, well before running out of memory, which was found to occur at $5 \times 10^7$ histories/cycle/GCD for RTK and $3.7 \times 10^7$ histories/cycle/GCD for the DP, SP, and ST methods. However, on Summit, which has $4\times$ less RAM/GCD, all four methods ran out of memory prior to nearing an asymptotic limit, with RTK running out of memory at $1.2 \times 10^7$ histories/cycle/GCD and the DP, SP, and ST methods running out of memory at $8 \times 10^6$ histories/cycle/GCD.

Aside from the $1 \times 10^6$ histories/s/GCD inactive cycles, in which the low workloads cause the kernel launch overhead to wash out any differences between the methods, the relationships between the performance of the four methods remained constant over all workloads. The highest tracking rates were achieved with RTK, as expected, followed by the SP method, the ST method, and the DP method. Further analysis was performed with the $8 \times 10^6$ histories/cycle/GCD trials on Summit and $2.8 \times 10^7$ histories/cycle/GCD trials on Frontier. Tracking rates for each of the four methods are shown in Table 2. Results show that all three experimental methods achieve over 90% of the RTK tracking rate on Summit. On Frontier, the DP, SP, and ST methods achieve 77.8%, 91.2%, and 83.4% of the RTK tracking rate, respectively, during inactive cycles, with all methods achieving at least 87.9% of the RTK tracking rate during active cycles. These results show that using universe-specific tracker types provides better performance than using a single tracker type for all universe types, provided that this can be done without virtual function calls. However, although the SP method consistently provides the best performance, no method incurred a significant performance penalty.

### *6.2 Performance of individual tracking kernels*

The percentages of the total GPU runtime (inactive and active cycles) spent in tracking kernels are shown in Table 3. These values vary from 12.1–26.1%, indicating that tracking operations do not dominate runtime, as expected. Likewise, much larger differences in the performance of the four methods are observed when only considering the time spent within tracking kernels. Figure 8 shows the total time spent within

**Table 2.** GPU tracking rates for the NuScale SMR problem, run on Summit with $8 \times 10^6$ histories/cycle/GCD and Frontier with $2.8 \times 10^7$ histories/cycle/GCD.

| | Method | Summit | | Frontier | |
|---|---|---|---|---|---|
| | | GPU tracking rate (histories/s/GCD) | Fraction of RTK tracking rate (%) | GPU tracking rate (histories/s/GCD) | Fraction of RTK tracking rate (%) |
| Inactive | RTK | $1.38 \times 10^5$ | 100 | $1.57 \times 10^5$ | 100 |
| | DP | $1.27 \times 10^5$ | 92.2 | $1.22 \times 10^5$ | 77.8 |
| | SP | $1.36 \times 10^5$ | 98.5 | $1.43 \times 10^5$ | 91.2 |
| | ST | $1.28 \times 10^5$ | 93.3 | $1.31 \times 10^5$ | 83.4 |
| Active | RTK | $6.79 \times 10^4$ | 100 | $7.62 \times 10^4$ | 100 |
| | DP | $6.38 \times 10^4$ | 93.9 | $6.69 \times 10^4$ | 87.9 |
| | SP | $6.69 \times 10^4$ | 98.6 | $7.26 \times 10^4$ | 95.3 |
| | ST | $6.49 \times 10^4$ | 95.5 | $6.93 \times 10^4$ | 91.0 |

**Table 3.** Fraction of total GPU runtime (inactive and active cycles) spent in geometry tracking kernels for the NuScale SMR problem. Summit results are for $8 \times 10^6$ histories/cycle/GCD and Frontier results are for $2.8 \times 10^7$ histories/cycle/GCD.

| | Runtime fraction (%) | |
|---|---|---|
| Method | Summit | Frontier |
| RTK | 12.1 | 13.3 |
| DP | 19.2 | 26.1 |
| SP | 13.6 | 18.3 |
| ST | 17.2 | 23.3 |

the five principal tracking kernels introduced in Table 1. For a fair comparison, both Summit and Frontier results are from workloads of $8 \times 10^6$ histories/cycle/GCD. This figure shows that the DP, SP, and ST methods spent 1.70×, 1.14×, and 1.50× more time, respectively, in tracking kernels compared to RTK on Summit, and 2.45×, 1.5×, and 2.02× more time, respectively, compared to RTK on Frontier. These results highlight the supremacy of the SP method, almost achieving parity with RTK on Summit.

## *6.3 GPU to CPU performance comparison*

Whereas overall tracking rates were slightly higher on Frontier, significantly more time was spent in tracking kernels on Frontier compared to Summit. Frontier also lags behind Summit when comparing GPU performance to CPU performance. Table 4 compares GPU to CPU tracking rates in terms of (1) the number of CPU cores required to match a single GPU GCD, and (2) the *speedup*, defined as the ratio of the GPU tracking rate using all GPUs on a single node to the CPU tracking rate

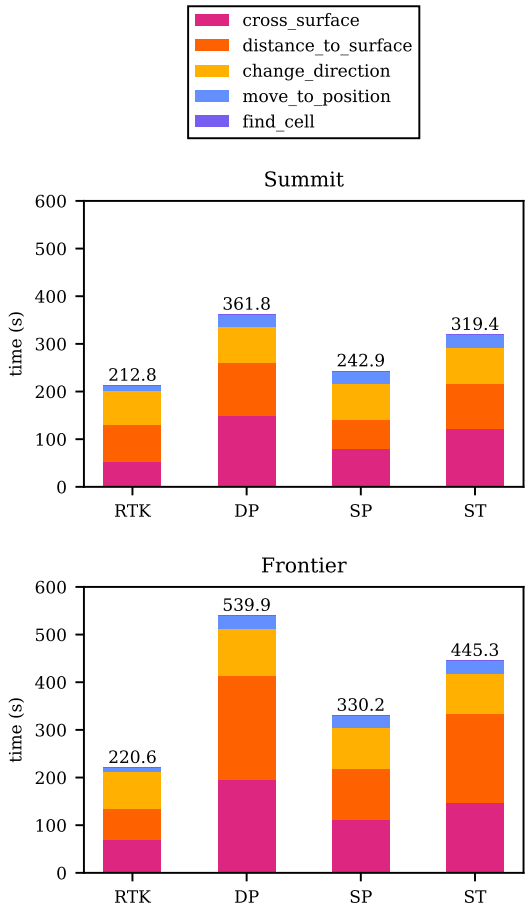



**Figure 8.** Total time spent in the 5 principal tracking kernels on Summit and Frontier for the NuScale SMR problem, both with $8 \times 10^6$ histories/cycle/GCD. All tracking kernels not listed here accounted for less than 0.3% of the total tracking time.

using all CPU cores on a single node. Noting that both GPU and CPU resources differ between these two machines, Summit has higher CPU core equivalence and speedups compared to Frontier in all cases. This can be attributed to the fact that significantly better CPU performance is achieved on Frontier, where a tracking rate of $4.94 \times 10^3$ histories/s/core was achieved for inactive cycles, compared to $2.82 \times 10^3$ histories/s/core for inactive cycles on Summit.

**Table 4.** GPU to CPU performance comparison for the NuScale SMR problem, run on Summit with $8 \times 10^6$ histories/cycle/GCD and Frontier with $2.8 \times 10^7$ histories/cycle/GCD. Speedup is defined as the ratio of the GPU tracking rate using all GPUs on a single node to the CPU tracking rate using all CPU cores on a single node.

| | | Summit | | Frontier | |
|---|---|---|---|---|---|
| | Method | CPU core equivs. per GCD | Speedup | CPU core equivs. per GCD | Speedup |
| Inactive | RTK | 48.9 | 6.98 | 31.7 | 4.53 |
| | DP | 45.0 | 6.43 | 24.7 | 3.53 |
| | SP | 48.1 | 6.87 | 28.9 | 4.13 |
| | ST | 45.6 | 6.51 | 26.4 | 3.78 |
| Active | RTK | 28.1 | 4.01 | 15.3 | 2.19 |
| | DP | 26.3 | 3.76 | 13.4 | 1.92 |
| | SP | 27.7 | 3.95 | 14.6 | 2.08 |
| | ST | 26.8 | 3.83 | 13.9 | 1.99 |

## *6.4 Profiling*

In order to better understand the performance differences between the four methods, profiling was done using NVIDIA NSight Compute on a V100 (i.e., a Summit GPU) and AMD rocProf on an MI250X (i.e., a Frontier GPU) using a workload of $8 \times 10^6$ histories/cycle/GCD. Results appear in Table 5. On both the V100 and MI250X, RTK had the highest tracking rates and also the highest theoretical occupancies for the two most expensive kernels (cross_surface and distance_to_surface). The DP method had the lowest tracking rates on the V100 and MI250X and also the lowest theoretical occupancies for cross_surface and distance_to_surface on the V100. On the V100, all kernels except move_within_cell had higher theoretical occupancy with SP compared to DP, which was confirmed to be reflected proportionally in register usage. The fact that the DP and SP methods are identical other than their polymorphism implementation suggests that the virtual functions themselves increase register usage and therefore decrease occupancy. Achieved occupancy was observed to be strongly correlated with theoretical occupancy and did not provide any further insights. On the MI250X, in contrast to the V100, the DP method had the same theoretical occupancy as SP for all kernels except change_direction.

On both the V100 and MI250X, there were no differences in theoretical occupancy between SP and ST because these two methods were run using the exact same kernels within the same executable. This was done because the SP method still needs the ST code for BIH acceleration within CSG universes. While the ST method does not need to be compiled with the SP code in place, stripping out the relatively small and simple

SP code did not result in an increase in the ST tracking rate. Consequently, a single executable was maintained for simplicity.

On the V100, only minor differences were observed in the L1 and L2 cache hit rate, and these were not correlated with tracking rate results. This is expected because in all cases, cache performance is likely governed by cross section lookups occurring within the physics kernel launches interspersed with the geometry kernel launches. The differences in branch efficiency between the methods were also not correlated with tracking rate results. Similar results were obtained on the MI250X, noting that (1) only the L2 hit rate can be obtained with rocProf, and (2) instead of branch efficiency, rocProf provides vector arithmetic logic unit (VALU) utilization, which is correlated with branch efficiency.

Finally, on the V100, the number of warp stalls during instruction fetches correlated strongly with tracking rate results. [‡] The DP method resulted in about $3\times$ as many warp stalls within the cross_surface and distance_to_surface kernels, suggesting that virtual function calls put additional pressure on the instruction cache.

### *6.5 Verification*

Single node performance testing trials, which were run with only 10 inactive and 10 active cycles, were not sufficient to produce converged $k_{\text{eff}}$ results. To verify that all four tracking methods produce the same solution, a final trial was performed for each method on 50 nodes of Frontier with a workload of $2.5 \times 10^5$ histories/cycle/GCD, for a total of $10^8$ histories per cycle. For each trial, 200 inactive cycles and 400 active cycles were run. A lower workload was necessary to complete all 600 cycles within Frontier's walltime limit. Converged $k_{\text{eff}}$ values agreed closely, as shown in Table 6. Neutron flux results for all 4 methods also matched expectations. Figure 9 shows the converged flux for the SP method trial. This figure clearly shows the depression on the flux within the borosilicate glass rods, which act as neutron absorbers.

## 7 Demonstration problem

In Section 6.1, the SP method consistently provided the best performance, but all three methods achieved tracking rates reasonably close to that of RTK. From a software engineering perspective, when considering multiple approaches, the trade-off between performance and other factors such as code simplicity and maintainability must be assessed. The ST method achieves over 90% of the SP method tracking rate in all cases

[‡] Obtained via the `smsp__pcsamp_warps_issue_stalled_no_instructions` metric.

**Table 5.** Profiling results on NVIDIA V100 and AMD MI250X GPUs using a workload of $8 \times 10^6$ histories/cycle/GCD. Highlighted columns show the metrics correlated with performance results.

| | | V100 | | | | | MI250X | | |
|---|---|---|---|---|---|---|---|---|---|
| Method | Kernel | Theoretical occupancy (%) | L1 hit rate (%) | L2 hit rate (%) | Branch efficiency (%) | Warp stalls from instruction fetches (#) | Theoretical occupancy (%) | L2 hit rate (%) | VALU utilization (%) |
| RTK | cross_surface | 75 | 57.8 | 71.9 | 89.9 | 657 | 100 | 70.4 | 86.5 |
| | distance_to_surface | 50 | 58.0 | 66.5 | 85.9 | 326 | 100 | 53.7 | 89.6 |
| | change_direction | 100 | 45.0 | 50.4 | 100 | 34 | 100 | 82.1 | 88.4 |
| | move_within_cell | 100 | 66.7 | 58.8 | 0.0 | 27 | 100 | 63.4 | 86.3 |
| | find_cell | 100 | 53.7 | 69.7 | 78.8 | 398 | 100 | 58.1 | 91.7 |
| DP | cross_surface | 25 | 59.9 | 64.4 | 97.3 | 4851 | 50 | 69.4 | 87.6 |
| | distance_to_surface | 12.5 | 59.6 | 67.8 | 89.6 | 10,031 | 50 | 54.5 | 89.6 |
| | change_direction | 25 | 45.0 | 52.4 | 100 | 70 | 50 | 51.8 | 89.6 |
| | move_within_cell | 100 | 72.0 | 69.9 | 100 | 23 | 100 | 66.2 | 84.0 |
| | find_cell | 25 | 57.1 | 65.2 | 92.9 | 8796 | 62.5 | 58.5 | 91.7 |
| SP | cross_surface | 37.5 | 68.3 | 72.7 | 97.9 | 1153 | 50 | 69.0 | 87.6 |
| | distance_to_surface | 25 | 72.4 | 48.9 | 89.4 | 3741 | 50 | 54.5 | 89.6 |
| | change_direction | 50 | 44.1 | 50.3 | 100 | 36 | 100 | 52.3 | 89.6 |
| | move_within_cell | 100 | 72.0 | 69.9 | 100 | 13 | 100 | 65.4 | 84.0 |
| | find_cell | 50 | 63.8 | 75.4 | 92.0 | 3710 | 62.5 | 58.4 | 91.7 |
| ST | cross_surface | 37.5 | 70.3 | 74.1 | 98.7 | 6110 | 50 | 69.2 | 89.2 |
| | distance_to_surface | 25 | 71.7 | 59.5 | 87.8 | 6523 | 50 | 54.4 | 89.6 |
| | change_direction | 50 | 41.3 | 51.6 | 100 | 55 | 100 | 51.5 | 89.6 |
| | move_within_cell | 100 | 71.95 | 69.8 | 100 | 45 | 100 | 65.7 | 84.0 |
| | find_cell | 50 | 66.2 | 79.2 | 94.0 | 9995 | 62.5 | 58.2 | 91.7 |

**Table 6.** Converged $k_{\text{eff}}$ values and $1\sigma$ statistical uncertainties for the NuScale SMR problem, obtained on 50 nodes of Frontier with $2.5 \times 10^5$ histories/cycle/GCD, with 200 inactive and 400 active cycles.

| Method | $k_{\text{eff}}$ |
|---|---|
| RTK | $1.046076 \pm 0.000005$ |
| DP | $1.046076 \pm 0.000005$ |
| SP | $1.046067 \pm 0.000005$ |
| ST | $1.046072 \pm 0.000005$ |

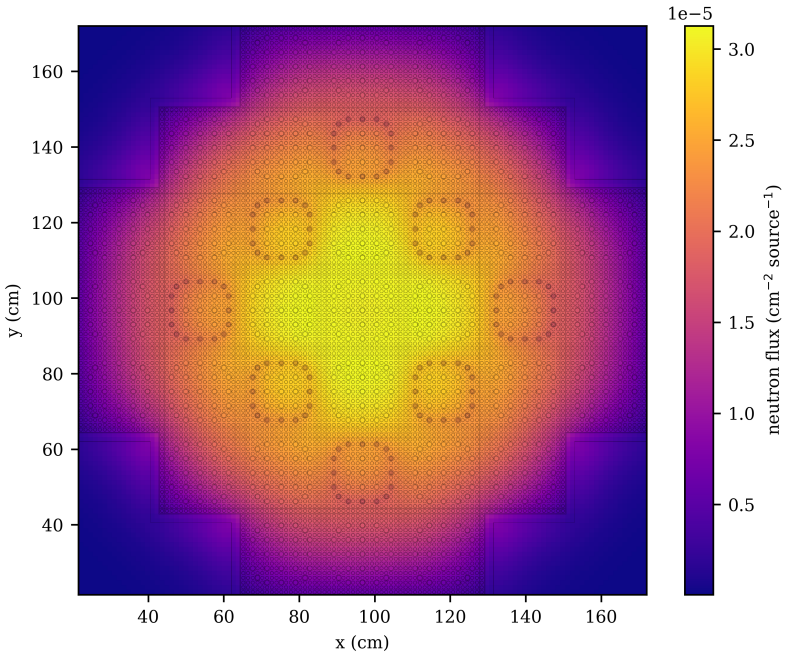



**Figure 9.** Converged neutron flux distribution on the midplane of the NuScale SMR problem using the SP method, obtained on 50 nodes of Frontier with $2.5 \times 10^5$ histories/cycle/GCD, with 200 inactive and 400 active cycles. The $1\sigma$ statistical uncertainties in the flux on this slice are all less than 0.15% within the core region.

shown in Table 2 and has the added benefit of not requiring an additional hexagonal array tracker. This is advantageous for hexagonal-array-based reactors, which cannot be represented using RTK, and would otherwise require the development of a GPU-based hexagonal array tracker for use with the SP method. To demonstrate this benefit, the ST method was used to perform a $k_{\text{eff}}$ calculation on the Empire microreactor benchmark problem that consists of nested hexagonal arrays, as shown in Figure 10. Microreactors, even smaller than SMRs, are designed to be factory assembled and

easily transported in order to supply power for remote areas, disaster relief, and space applications.

The Empire microreactor consists of 18 hexagonal assemblies arranged around a central void region. Each assembly contains a total of 217 pins: 60 uranium nitride fuel pins with 16.05% $^{235}$U enrichment (by mass), 96 yttrium hydride moderator pins, and 61 stainless steel heat pipes filled with liquid sodium. Within this model, material within the heat pipe pins is homogenized for simplicity. The core is surrounded by 12 control drums which contain a europium boride neutron absorber on one side. These drums can be rotated to change the absorber configuration to control the reactivity within the core. For this analysis, the "drums-in" configuration was used, with all absorbers directly facing the core.

A Cartesian mesh tally would require a prohibitively large number of elements in order to conform to the hexagonal pins and assemblies found in the Empire geometry, and Shift does not yet support hexagonal mesh tallies on the GPU. Thus, cell-based tallies were used to tally the flux and fission source density within each fuel pin. For high-resolution results, each fuel pin was subdivided into 360 cells, with 3 radial divisions, 4 circumferential divisions, and 30 axial divisions. This resulted in a total of 388,800 cell tallies, similar to the number of elements in the Cartesian mesh tally used for the NuScale SMR problem.

The problem was run on 100 Frontier nodes using all eight GCDs on each node, with a workload of $10^6$ histories/cycle/GCD, for a total of $8 \times 10^8$ histories/cycle. To achieve converged results, 120 inactive cycles and 120 active cycles were performed. For comparison, CPU results were obtained on a single node of Frontier with $2 \times 10^6$ histories/cycle for 120 inactive cycles and 120 active cycles.

Converged flux results are shown in Figure 11. As expected, the highest flux occurs in an annular region around the center of the reactor, i.e., the region where leakage to the outside of the reactor and inner void region are minimized. Figure 12 shows the converged fission source. As expected, the highest fission source density occurs around the edges of fuel rods. For rods in the outer parts of assemblies, the fission source density is observed to be highest on the inward-facing edge. A converged $k_{\text{eff}}$ of $1.026772 \pm 0.000003$ was obtained.

Table 7 shows tracking rates, as well as GPU vs. CPU comparison results, in the same format as that used in Table 4. For inactive cycles, the tracking rate of $4.74 \times 10^4$ histories/s/GCD is 36.2% of the ST method tracking rate for the NuScale SMR problem on Frontier. For active cycles, the tracking rate of $4.33 \times 10^4$ histories/s/GCD is 62.5% of the ST method tracking rate for the NuScale SMR problem on Frontier. The 8.65% degradation in tracking rate between inactive and active cycles for this problem is much

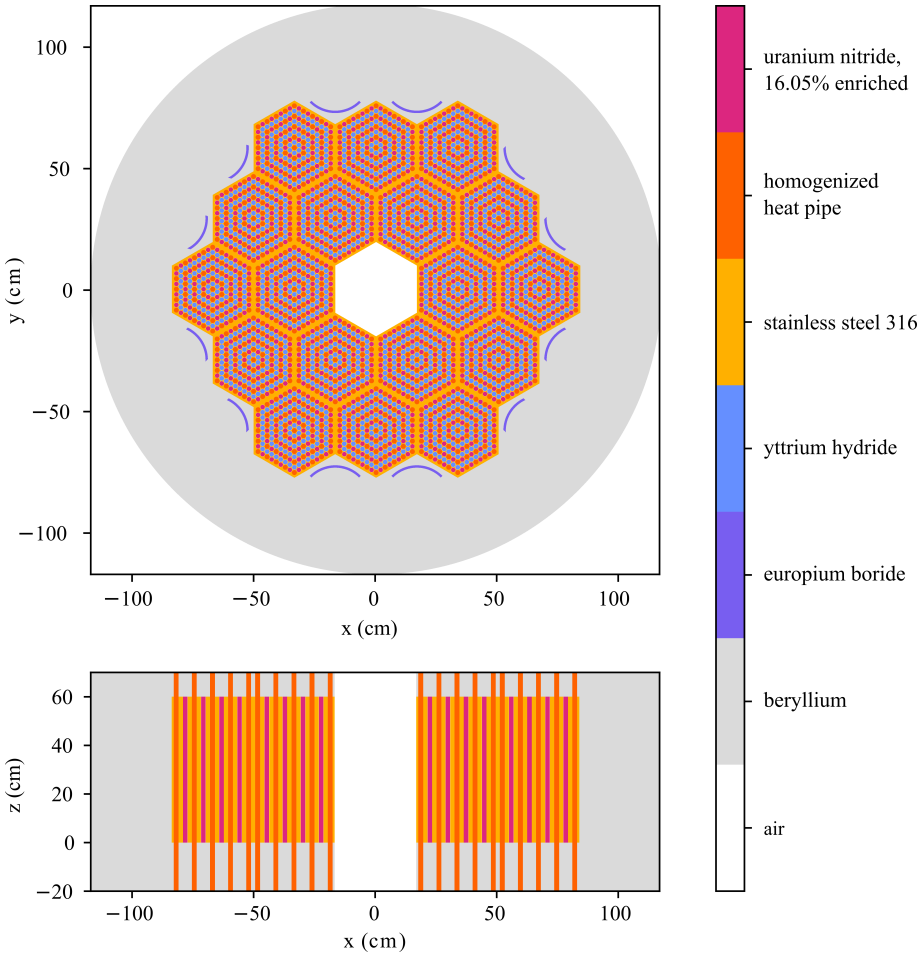



**Figure 10.** Midplane radial (top) and axial (bottom) slices of a full core model of the Empire microreactor in "drums-in" configuration.

less than the 47.1% degradation observed in the NuScale SMR problem. This is due to the fact that the cell tallies used in this problem have significantly less overhead than tracking on the superimposed mesh tally for the NuScale SMR problem. When considering both inactive and active cycles, an overall speedup of 2.19× was obtained.

## 8 Conclusion

In this work, three methods for GPU-based neutron tracking within multi-universe geometries were tested and compared to RTK. For the NuScale SMR problem, the

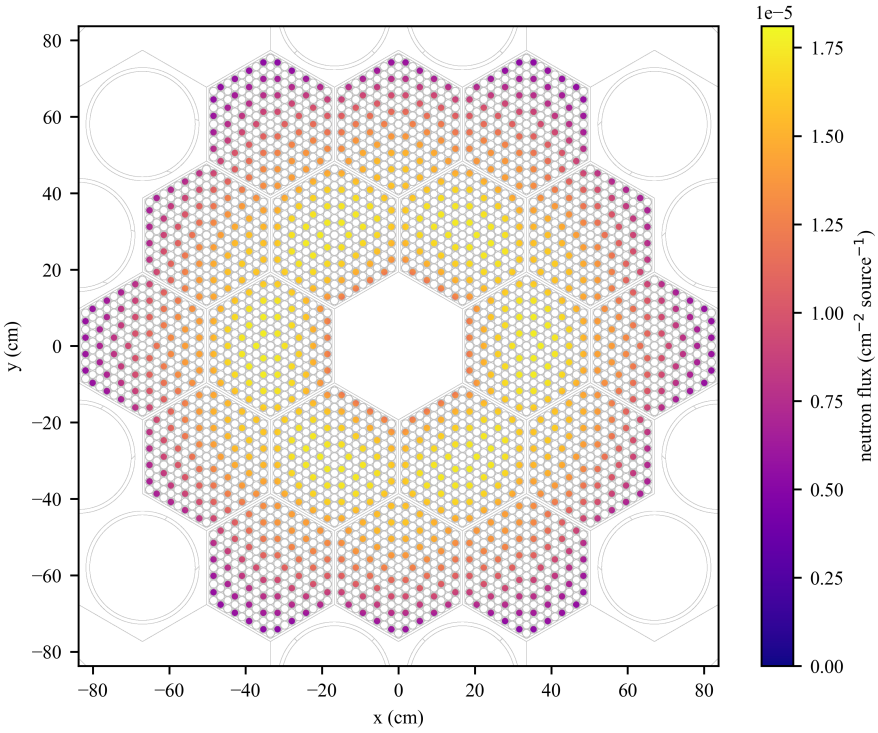



**Figure 11.** Converged neutron flux distribution on the midplane of the Empire problem using the ST method, obtained on 100 nodes of Frontier with $10^6$ histories/cycle/GCD, with 120 inactive and 120 active cycles. Statistical uncertainties on this slice are all less than 0.10%.

**Table 7.** GPU to CPU performance comparison for the Empire problem on Frontier, with $10^6$ histories/cycle/GCD. Speedup is defined as the ratio of the GPU tracking rate using all GPUs on a single node to the CPU tracking rate using all CPU cores on a single node.

| Cycle type | GPU tracking rate (histories/s/GCD) | CPU core equivs. per GCD | Speedup |
|---|---|---|---|
| Inactive | $4.74 \times 10^4$ | 15.9 | 2.27 |
| Active | $4.33 \times 10^4$ | 14.8 | 2.12 |

DP, SP, and ST methods spent 2.45×, 1.50×, and 2.02× more time, respectively, in tracking kernels in inactive cycles on Frontier compared to RTK. However, although inactive cycles on this fresh fuel problem represent the maximum time spent performing tracking operations, these methods still achieved 77.8%, 91.2%, and 83.4%, respectively, of the RTK tracking rate on Frontier. On the NVIDIA V100 (i.e., a

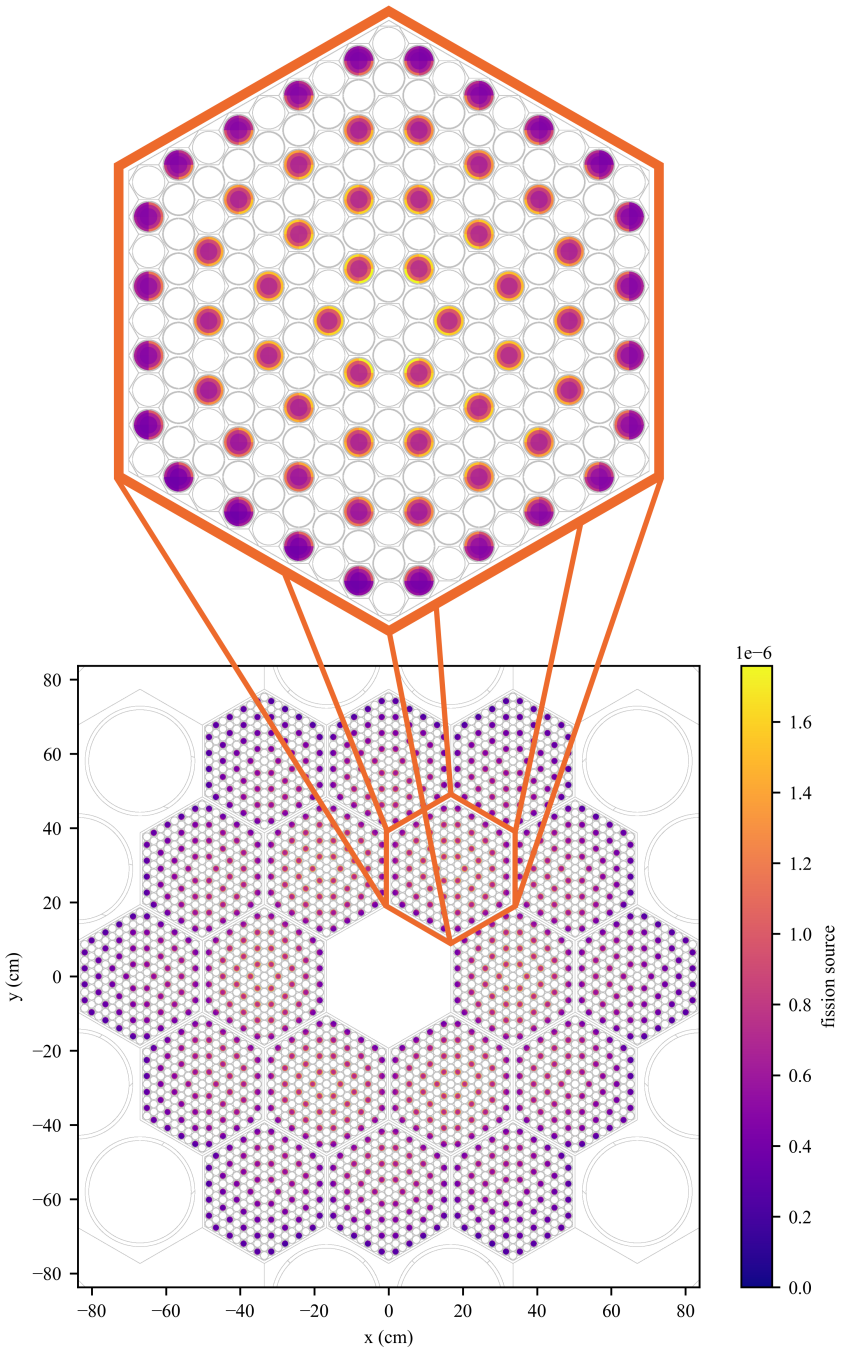



**Figure 12.** Converged fission source distribution on the midplane of the Empire problem using the ST method, obtained on 100 nodes of Frontier with $10^6$ histories/cycle/GCD, with 120 inactive and 120 active cycles. Statistical uncertainties on this slice are all less than 0.25%.

Summit GPU), it appears that performance differences between the methods can be explained by differences in device occupancy and pressure on the instruction cache. On the AMD MI250X (i.e., a Frontier GPU), occupancy was more weakly correlated to performance. It is concluded that any of these three methods—which unlike RTK can handle arbitrarily nested configurations—are suitable for production-level use, provided that the fraction of runtime required by tracking operations is consistent with the typical fission problems explored in this work (around 20%). For problems with geometric complexity far beyond these cases (perhaps outside of fission applications), tracking performance differences become more important, and the DP method should be avoided as it offers neither the performance of the SP method nor the convenience of the ST method. Broadly speaking, these results indicate that polymorphism can still be effectively employed on the GPU, provided that it can be done without virtual function calls. Improvements to GPU compilers may eliminate this limitation.

The SP method outperformed the DP and ST methods in nearly all cases. However, the ST method achieved over 90% of the SP method's tracking rate in all cases, and unlike the SP method, it requires only a single CSG tracker. The flexibility of this approach was demonstrated with the Empire microreactor, a hexagonal array-based reactor which cannot be represented using RTK, and would require an additional GPU-based hexagonal array tracker to be written in order to use the SP method. On Frontier, the ST method was used to produce converged $k_{\text{eff}}$, neutron flux, and fission source results, and an overall speedup of 2.19$\times$ over CPU execution was obtained. This work will facilitate GPU-based MC transport for reactor problems with geometric complexity beyond rectilinear arrays, as well as other non-reactor radiation transport problems with complex nested geometries.

## Funding

This research was supported by the Exascale Computing Project (ECP), project number 17-SC-20-SC. The ECP is a collaborative effort of two DOE organizations, the Office of Science and the National Nuclear Security Administration, that are responsible for the planning and preparation of a capable exascale ecosystem—including software, applications, hardware, advanced system engineering, and early testbed platforms—to support the nation's exascale computing imperative.

## Acknowledgements

The authors would like to thank Stuart Slattery and Friederike Bostelmann for their internal technical reviews.

## References

Andreades C, Cisneros AT, Choi JK, Chong AYK, Fratoni M, Hong S, Huddar LR, Huff KD, Kendrick J, Krumwiede DL, Laufer MR, Munk M, Scarlat RO and Zweibau N (2016) Design summary of the Mark-I pebble-bed, fluoride salt-cooled, high-temperature reactor commercial power plant. *Nuclear Technology* 195(3): 223–238. DOI:10.13182/NT16-2.

Bentley JL (1975) Multidimensional binary search trees used for associative searching. *Communications of the Association for Computing Machinery* 18(9): 509–517. DOI: 10.1145/361002.361007.

Betzler BR et al. (2020) Transformational Challenge Reactor preliminary core design report. Technical Report ORNL/TM-2020/1718, Oak Ridge National Laboratory.

Black GA, Aydogan F and Koerner CL (2019) Economic viability of light water small modular nuclear reactors: General methodology and vendor data. *Renewable and Sustainable Energy Reviews* 103: 248–258. DOI:10.1016/j.rser.2018.12.041.

Center for Sustainable Systems (2022) Nuclear energy factsheet. Technical Report Pub. No. CSS11-15., University of Michigan.

Ericson C (2004) *Real-Time Collision Detection*. The Morgan Kaufmann Series in Interactive 3D Technology. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Habush A and Harris A (1968) 330-MW(e) Fort St. Vrain high-temperature gas-cooled reactor. *Nuclear Engineering and Design* 7(4): 312–321. DOI:10.1016/0029-5493(68)90064-2.

Hamilton SP and Evans TM (2019) Continuous-energy Monte Carlo neutron transport on GPUs in the Shift code. *Annals of Nuclear Energy* 128: 236–247.

Harper SM, Romano PK, Forget B and Smith KS (2020) Threadsafe dynamic neighbor lists for Monte Carlo ray tracing. *Nuclear Science and Engineering* 194(11): 1009–1015. DOI: 10.1080/00295639.2020.1719765.

Hejzlar P, Petroski R, Cheatham J, Touran N, Cohen M, Truong B, Latta R, Werner M, Burke T, Tandy J, Garrett M, Johnson B, Ellis T, McWhirter J, Odedra A, Schweiger P, Adkisson D and Gilleland J (2013) Terrapower, LLC traveling wave reactor development program overview. *Nuclear Engineering and Technology* 45(6): 731–744. DOI:10.5516/NET.02. 2013.520.

Johnson S, Tognini S, Canal P, Evans T, Jun S, Lima G, Lund A and Pascuzzi V (2021) Novel features and GPU performance analysis for EM particle transport in the Celeritas code. *EPJ Web of Conferences* 251(03030). DOI:10.1051/epjconf/202125103030.

Johnson SR, Lefebvre R and Bekar K (2023) ORANGE: Oak Ridge Advanced Nested Geometry Engine. Technical Report ORNL/TM-2023/3190, Oak Ridge National Laboratory.

Juarez R, Pedroche G, Loughlin MJ, Pampin R, Martinez P, De Pietri M, Alguacil J, Ogando F, Sauvan P, Lopez-Revelles AJ, Kolšek A, Polunovskiy E, Fabbri M and Sanz J (2021) A full and heterogeneous model of the ITER tokamak for comprehensive nuclear analyses. *Nature Energy* 6(2): 150–157. DOI:10.1038/s41560-020-00753-x.

Kos B, Radulescu G, Grove R, Villari R and Batistoni P (2023) Comprehensive analysis of streaming and shutdown dose rate experiments at JET with ORNL fusion neutronics workflows. *Fusion Science and Technology* 79(3): 284–304. DOI:10.1080/15361055.2022.2129182.

Lee C, Jung YS, Zhong Z, Ortensi J, Laboure V, Wang Y and DeHart M (2020) Assessment of the Griffin reactor multiphysics application using the Empire micro reactor design concept. Technical Report ANL/NSE-20/23 and INL/LTD-20-59263, Argonne National Laboratory and Idaho National Laboratory.

Lieberoth J (1968) Monte Carlo technique to solve the static eigenvalue problem of the Boltzmann transport equation. Technical report, INTERATOM, Bensberg, Germany.

Matthews C, Laboure V, DeHart M, Hansel J, Andrs D, Wang Y, Ortensi J and Martineau RC (2021) Coupled multiphysics simulations of heat pipe microreactors using DireWolf. *Nuclear Technology* 207(7): 1142–1162. DOI:10.1080/00295450.2021.1906474.

Merzari E, Hamilton S, Evans T, Min M, Fischer P, Kerkemeier S, Fang J, Romano P, Lan YH, Phillips M, Biondo E, Royston K, Warburton T, Chalmers N and Rathnayake T (2023) Exascale multiphysics nuclear reactor simulations for advanced designs. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23. New York, NY, USA: Association for Computing Machinery. DOI:10.1145/3581784.3627038.

Mulder E and Boyes W (2020) Neutronics characteristics of a 165 MWth Xe-100 reactor. *Nuclear Engineering and Design* 357. DOI:10.1016/j.nucengdes.2019.110415.

Nelson NB, Smith MBR, Karriem Z, Navarro J, Denbrock CP, Wahlen RN and Grimm TL (2022) Radiation shielding analysis of Niowave's uranium target assembly 2 (UTA-2) facility for molybdenum-99 production. Technical Report ORNL/TM-2021/2269, Oak Ridge National Laboratory. DOI:10.2172/1878714.

NuScale Power L (2023) About us. URL https://www.nuscalepower.com/en/about.

Oak Ridge Leadership Computing Facility (2023a) Frontier. URL https://www.olcf.ornl.gov/olcf-resources/compute-systems/frontier.

Oak Ridge Leadership Computing Facility (2023b) Summit: Oak Ridge National Laboratory's 200 petaflop supercomputer. URL https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit.

OpenClipart (2023) Outline map of American states. URL https://freesvg.org/outline-map-of-american-states.

Pandya TM, Johnson SR, Evans TM, Davidson GG, Hamilton SP and Godfrey AT (2016) Implementation, capabilities, and benchmarking of Shift, a massively parallel Monte Carlo radiation transport code. *Journal of Computational Physics* 308: 239–272. DOI:10.1016/j.jcp.2015.12.037.

Radel T and Van Abel E (2016) Validation of MCNP5 for use in calculating temperature coefficients of reactivity for the SHINE system. *Transactions of the American Nuclear Society* 114(1).

Smith K (2017) NuScale small modular reactor (SMR) progression problems for the ExaSMR project. Milestone Report WBS 1.2.1.08 ECP-SE-08-43, Exascale Computing Project.

von Mises R and Pollaczek-Geiringer H (1929) Praktische verfahren der gleichungsauflösung. *Zamm-zeitschrift Fur Angewandte Mathematik Und Mechanik* 9: 152–164.

Wächter C and Keller A (2006) Instant ray tracing: The bounding interval hierarchy. In: Akenine-Moeller T and Heidrich W (eds.) *Symposium on Rendering*. The Eurographics Association. ISBN 3-905673-35-5. DOI:10.2312/EGWR/EGSR06/139-149.

Wald I (2007) On fast construction of SAH-based bounding volume hierarchies. In: *2007 IEEE Symposium on Interactive Ray Tracing*. pp. 33–40. DOI:10.1109/RT.2007.4342588.

West JT III, Petrie LM and Fraley SK (1979) KENO-IV/CG, the combinatorial geometry version of the Keno Monte Carlo criticality safety program. Technical Report NUREG/CR-0709, ORNL/NUREG/CSD-7, Oak Ridge National Laboratory.

Zhang M, Alawneh A and Rogers TG (2021) Characterizing massively parallel polymorphism. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. pp. 205–216. DOI:10.1109/ISPASS51385.2021.00037.