The OpenMOC Method of Characteristics Neutral Particle Transport Code

William Boyd, Samuel Shaner, Lulu Li, Benoit Forget, Kord Smith

Massachusetts Institute of Technology, Department of Nuclear Science and Engineering, 77 Massachusetts Avenue, Building 24, Cambridge, MA 02139, United States

Abstract

The method of characteristics (MOC) is a numerical integration technique for partial differential equations, and has seen widespread use for reactor physics lattice calculations. The exponential growth in computing power has finally brought the possibility for high-fidelity full core MOC calculations within reach. The OpenMOC code is being developed at the Massachusetts Institute of Technology to investigate algorithmic acceleration techniques and parallel algorithms for MOC. OpenMOC is a free, open source code written using modern software languages such as C/C++ and CUDA with an emphasis on extensible design principles for code developers and an easy to use Python interface for code users. The present work describes the OpenMOC code and illustrates its ability to model large problems accurately and efficiently.

Keywords: Method of characteristics, neutron transport, criticality, high performance computing, nonlinear diffusion acceleration, open source

1. Introduction

A new general purpose open source 2D method of characteristics reactor physics code called OpenMOC is currently under development at the Massachusetts Institute of Technology. OpenMOC has been developed as a general purpose platform to deploy new numerical acceleration techniques and parallel algorithms - two of the primary drivers for improved compute performance of scientific codes. Solvers for a variety of platforms, including multi-core CPUs and massively parallel GPUs, are available in OpenMOC to exploit the increasingly heterogenous nature of present day as well as future computing hardware. OpenMOC has been developed using modern software development standards to enhance its value as a tool for research and collaboration.

2. Method of Characteristics

The method of characteristics (MOC) is a widely used technique for solving partial differential equations, including the Boltzmann form of the neutron transport equation (J. R. Askew, 1972). MOC is used to solve the transport equation in 2D by discretizing both polar and azimuthal angles and integrating the characteristic form of the equation for a particular azimuthal and polar angle quadrature (W. Boyd, 2014; L. Li, 2013). The MOC integration method is based on the multi-group, steady-state form of the neutron transport equation:

$$\Omega \cdot \nabla \Psi_g(\mathbf{r}, \mathbf{\Omega}) + \Sigma_g^T(\mathbf{r}) \Psi_g(\mathbf{r}, \mathbf{\Omega}) = Q_g(\mathbf{r}) \tag{1}$$

where g is the energy group index, $\mathbf{r}$ is the spatial position vector, $\mathbf{\Omega}$ is the angular direction vector, $\Psi_g(\mathbf{r}, \mathbf{\Omega})$ is the angular flux, $\Sigma_g^T(\mathbf{r})$ is the total cross-section and $Q_g(\mathbf{r})$ is the source term. In MOC, this equation is transformed through a parametrization along discrete *tracks* $k \in K$, resulting in the characteristic form of the transport equation:

$$\frac{d\Psi_{k,g}(s)}{ds} + \Sigma_g^T(s) \Psi_{k,g}(s) = Q_g(s) \tag{2}$$

In addition, the geometry is generally discretized into *flat source regions*, or *FSRs* denoted by index i, to give the final system of ODEs solved in MOC:

$$\frac{d\Psi_{k,i,g}(s)}{ds} + \Sigma_{i,g}^T \Psi_{k,i,g}(s) = Q_{i,g} \tag{3}$$

The source term $Q_{i,g}$ is defined in terms of both fission and isotropic scattering from the area-averaged scalar flux $\Phi_{i,g}$ in each FSR:

$$Q_{i,g} = \frac{1}{4\pi} \left(\sum_{g'=1}^{G} \Sigma_{i,g' \to g}^S \Phi_{i,g'} + \frac{\chi_{i,g}}{k_{eff}} \sum_{g'=1}^{G} \nu \Sigma_{i,g'}^F \Phi_{i,g'} \right) \tag{4}$$

where $\Sigma_{i,g' \to g}^S$ is the scattering cross-section for group g' to group g, $\Sigma_{i,g}^F$ is the fission cross-section for group g, ν is the average number of neutrons produced from fission, $\chi_{i,g}$ is the fraction of neutrons produced in group g from fission and k_{eff} is the multiplication factor. Each track is discretized into *segments* across individual FSRs. Equation 3 can be integrated for a segment across an FSR from its entry point at s' to exit point at s'' using an integrating factor:

$$\Psi_{k,g}(s'') = \Psi_{k,g}(s') e^{-\tau_{k,i,g}} + \frac{Q_{i,g}}{\Sigma_{i,g}^T} (1 - e^{-\tau_{k,i,g}}) \tag{5}$$

with the optical path length defined as $\tau_{k,i,g} = \Sigma_{i,g}^T (s'' - s')$. With minor algebraic rearrangment, the change in the angular flux along the characteristic k is given by the following:

Email addresses: wboyd@mit.edu (William Boyd), shaner@mit.edu (Samuel Shaner), lululi@mit.edu (Lulu Li), bforget@mit.edu (Benoit Forget), kord@mit.edu (Kord Smith)

$$\begin{aligned}\Delta\Psi_{k,g} &= \Psi_{k,g}(s') - \Psi_{k,g}(s'') \\ &= \left(\Psi_{k,g}(s') - \frac{Q_{i,g}}{\Sigma^T_{i,g}}\right)\left(1 - e^{-\tau_{k,i,g}}\right)\end{aligned} \tag{6}$$

By defining $l_{k,i} = s'' - s'$, the average angular flux contribution to FSR i from track k is the following integral:

$$\overline{\Psi}_{k,i,g} = \frac{1}{l_{k,i}} \int_{s'}^{s''} \Psi_{k,i,g}(s)\, ds \tag{7}$$

Upon evaluating the integral, the average flux can be reduced to the following algebraic expression:

$$\begin{aligned}\overline{\Psi}_{k,i,g} = \frac{1}{l_{k,i}} \Bigg[&\frac{\Psi_{k,g}(s')}{\Sigma^T_{i,g}}(1 - e^{-\tau_{k,i,g}}) + \\ &\frac{l_{k,i} Q_{i,g}}{\Sigma^T_{i,g}} \left(1 - \frac{(1 - e^{-\tau_{k,i,g}})}{\tau_{k,i,g}}\right)\Bigg]\end{aligned} \tag{8}$$

The area-averaged scalar flux in FSR i with area A_i can be found by integrating the angular flux over azimuthal and polar angles m and p using quadrature rules for each track segment in A_i:

$$\Phi_{i,g} = \frac{4\pi}{A_i} \sum_{k \in A_i} \sum_{p=1}^{P} \omega_{m(k)} \omega_p \omega_k l_{k,i} \overline{\Psi}_{k,i,g,p} \tag{9}$$

where $m(k)$ represents the azimuthal angle for track k, $\omega_{m(k)}$ and ω_p are the azimuthal and polar quadrature weights, and ω_k represents the *track width* for track k. The final form of the scalar flux can be found by substituting the expression for the average angular flux from Equation 8 into Equation 9 and rearranging in terms of the change in angular flux along the characteristic from Equation 6:

$$\Phi_{i,g} = \frac{4\pi}{\Sigma_{i,g}} \left[Q_{i,g} + \frac{1}{A_i} \sum_{k \in A_i} \sum_{p=1}^{P} \omega_{m(k)} \omega_p \omega_k \sin\theta_p \Delta\Psi_{k,i,g,p} \right] \tag{10}$$

This is the form of the transport equation used in the MOC formulation presented in this paper. The azimuthal angle quadrature is chosen to ensure cyclic track wrapping at the boundaries as discussed in the following section. The quadrature recommended by Yamamoto (A. Yamamoto, M. Tabuchi, N. Sugimura, T. Ushio and M. Mori, 2007) is used for the polar angles and weights.

The spatial shape and energy distribution of the flux across FSRs is iteratively computed by transport sweeps and source updates until the scalar flux for each FSR has converged. Each transport sweep integrates the flux (from the previous iteration) along each track for each energy group while tallying a new flux contribution to each flat source region (Equation 6 and Equation 10). The fission source and the absorption rate for each flat source region is updated and used to compute k_{eff} and the total source is updated based on the flux tallies (Equation 4). These steps are repeated until each region's source has converged.

3. Implementation

3.1. Geometry Representation

The geometry is treated by a constructive solid geometry formulation (CSG) similar to OpenMC (P. K. Romano and B. Forget, 2013) using nested universes, cells, lattices and quadratic surface *primitives*. This general methodology can represent a wide range of complicated reactor geometries with minimal memory requirements.

An example of a 4 × 4 lattice of identical pin cells is illustrated in Figure 1. To reduce memory storage costs, only unique universes are stored in memory and the appropriate coordinate transformations between the local coordinates of each universe and the global coordinate system are made during ray tracing. Although this formulation works well for ray tracing, the MOC solver requires repeated cells throughout a core geometry to be represented by a unique FSR for flux tallying. To accomplish this, a recursive algorithm for identifying unique FSRs within a geometry built using CSG primitives was developed and is implemented in OpenMOC (W. Boyd, 2014; D. Lax, W. Boyd and N. Horelik, 2014). OpenMOC also provides the capability to subdivide pin cells into rings and angular sectors to better resolve the angular and radial flux and power gradients.

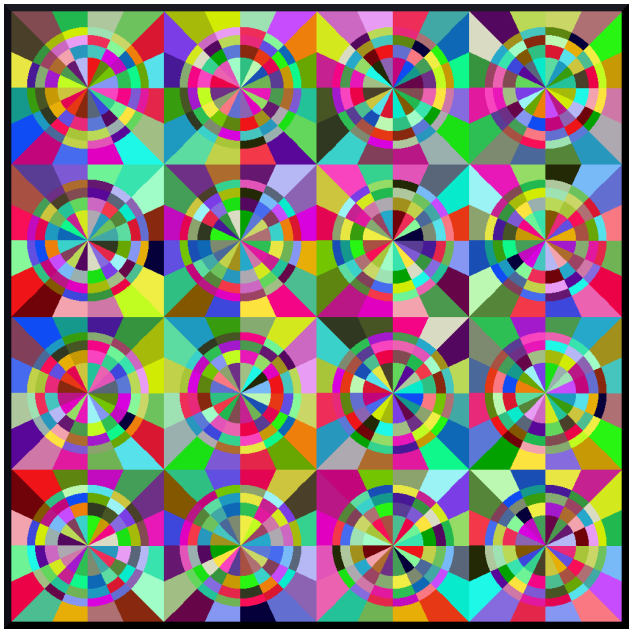

Figure 1. A 4 × 4 lattice with 3 rings and 16 angular divisions per pin cell. Unique colors identify each flat source region.

3.2. Global Cyclic Tracking

OpenMOC uses *global tracking* such that each track spans the entire extent of the simulation domain. By symmetry, tracks only need to be represented for azimuthal angles in the range $\phi \in [0, \pi]$ since a track with angle ϕ will have the same start and end points as a track with angle $\phi + \pi$. *Complementary angles* are pairs of angles (ϕ, α) in the azimuthal quadrature where $\alpha = \pi - \phi$. Tracks for complementary

angles have a *track spacing* such that they intersect at the boundaries of the geometry and form closed cycles. An illustration of OpenMOC's track layout for eight azimuthal angles is given in Figure 2.

Cyclic tracking is important since it allows for a simple treatment of reflective boundary conditions. Boundary conditions must be respected for the track outgoing fluxes at each iteration. Vacuum boundary conditions on a surface can be implemented for the MOC formulation by ensuring that each track originating along that surface has zero incoming flux, while tallying outgoing flux as leakage. Reflective boundary conditions are easily implemented since complementary angles ensure that tracks wrap around the geometry in closed cycles. Hence, no approximations need to be made for each track's incoming flux since it is exactly the outgoing flux of another track in the cycle.

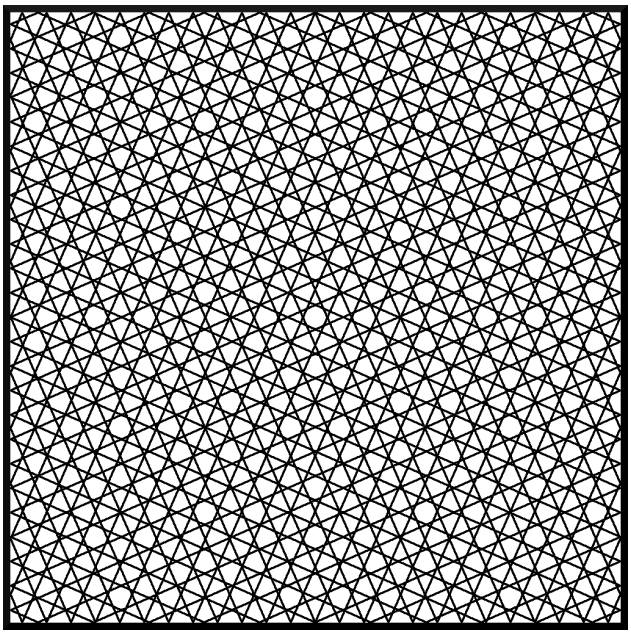

Figure 2. Tracks for eight azimuthal angles across a square demonstrate cyclic track wrapping.

3.3. MOC Solver

OpenMOC includes a solver implementation which integrates the angular flux across the geometry for each track as described in Section 2. A single transport sweep involves five nested loops over azimuthal angles, tracks, segments in different FSRs, energy groups and polar angles. A description of OpenMOC solver's transport sweep is given by Algorithm 1 along with the methodology to update the source in Algorithm 2. The sets of all azimuthal angles, tracks, track segments, FSRs, energy groups and polar angles are denoted by M, K, S, I, G and P, respectively. For notational simplicity, the subset of tracks for azimuthal angle m is denoted by $K(m)$, the subset of segments for track k is given by $S(k)$, and the FSR for segment s is represented as $I(s)$. The leakage tally for vacuum boundary conditions is designated as L.

3.4. Shared Memory Parallelism

For over fifty years, Moore's Law (G. E. Moore, 1965) has dictated a doubling in computational performance every 18 months. Since the mid-2000s, however, this trend has been challenged by the physics implications of ever smaller and denser transistors. Instead, hardware vendors have relied on parallelism to maintain steady performance improvements in accordance to Moore's Law. As a result, scientific computing software must now take advantage of parallel algorithms in order to realize the performance gains from new hardware.

The method of characteristics is one neutron transport algorithm which can be highly parallelized and vectorized due to its nested loop structure (see Algorithm 1). OpenMOC has been developed to take advantage of parallel algorithms on conventional shared memory architectures using OpenMP (OpenMP Architecture Review Board, 2013), on Single Instruction Multiple Data (SIMD) vector units using vector intrinsics (Intel, 2012), and on massively parallel and heterogeneous graphics processing units (GPUs) using NVIDIA's CUDA programming language (NVIDIA, 2013).

3.4.1. OpenMP

A shared memory multi-threaded implementation of the CPU-based solver has been implemented using the OpenMP framework. Parallelization is generally most efficient when implemented at the coarsest level possible while still providing enough degrees of concurrency to keep the hardware busy. For multi-core systems, both of these objectives are achieved by parallelizing the outermost loops over azimuthal angles and tracks in Algorithm 1 which allows for thread launch overhead to be amortized most effectively (W. Boyd, K. Smith, B. Forget and A. Siegel, 2014).

3.4.2. SIMD Vectorization

Much of performance improvement in next generation processors will come from more powerful Vector Processing Units (VPUs) for Single Instruction Multiple Data (SIMD) algorithms. Intel's Haswell processor line and the Intel Xeon Phi (Knights Corner) Coprocessors provide 8-wide and 16-wide (single precision) VPUs for each core, respectively (Intel, 2013a). OpenMOC includes solvers which utilizes Intel's Math Kernel Library (MKL) (Intel, 2013b) and leverages the auto-vectorization capabilities of Intel's C++ compiler to vectorize the inner loop over energy groups in Algorithm 1.

3.4.3. Graphics Processing Units (GPUs)

A GPU-based solver written in NVIDIA's CUDA programming language (W. Boyd, K. Smith and B. Forget, 2013) is included in OpenMOC. The MOC transport sweep was massively parallelized for GPUs by taking advantage of the nested loop structure of the algorithm as shown in Algorithm 1.

The OpenMOC framework for I/O and ray tracing is kept intact as these functions are performed on the CPU. Following ray tracing, all tracks, segments and FSRs are transformed into arrays of corresponding CUDA structures and copied to the GPU's memory. After the sources and fluxes are converged

Algorithm 1 Transport sweep for OpenMOC

$\Phi_{i,g} \leftarrow 0 \quad \forall\, i, g \in \{I, G\}$ # Initialize FSR scalar fluxes to zero
while $\Phi_{i,g}$ $\forall\, i$ not converged **do**
  for all $m \in M$ **do** # Loop over azimuthal angles
    for all $k \in K(m)$ **do** # Loop over tracks
      for all $s \in S(k)$ **do** # Loop over segments
        for all $g \in G$ **do** # Loop over energy groups
          for all $p \in P$ **do** # Loop over polar angles
            $i \leftarrow I(s)$ # Get FSR for this segment
            $\Delta\Psi_{k,i,g,p} \leftarrow \left(\Psi_{k,g,p} - \frac{Q_{i,g}}{\Sigma^T_{i,g}}\right)(1 - e^{-\tau_{k,i,g,p}})$ # Compute angular flux change along segment
            $\Phi_{i,g} \leftarrow \Phi_{i,g} + \frac{4\pi}{A_i}\omega_m\omega_p\omega_k \sin\theta_p l_{k,i}\Delta\Psi_{k,i,g,p}$ # Increment FSR scalar flux
            $\Psi_{k,g,p} \leftarrow \Psi_{k,g,p} - \Delta\Psi_{k,g,p}$ # Update track outgoing flux
          end for
        end for
      end for
    end for
    if B.C. are reflective **then** # Set incoming flux for outgoing track
      $\Psi_{k',g,p}(0) \leftarrow \Psi_{k,g,p}$ # Reflective B.C.'s
    else
      $\Psi_{k',g,p}(0) \leftarrow 0$ # Vacuum B.C.'s
      $L \leftarrow L + \Psi_{k,g,p}$ # Increment leakage tally
    end if
  end for
  Update k_{eff} and FSR sources $Q_{i,g}$ $\forall\, i$ # Algorithm 2
end while

Algorithm 2 FSR source update for OpenMOC

for all $i \in I$ **do** # Loop over FSRs
  for all $g \in G$ **do** # Loop over energy groups
    $Q^{(n+1)}_{i,g} \leftarrow \frac{\chi_{i,g}}{k_{eff}} \nu\Sigma^F_{i,g}\Phi^{(n)}_{i,g}$ # Initialize new total source with fission
    for all $g' \in G$ **do** # Loop over energy groups
      $Q^{(n+1)}_{i,g} \leftarrow Q^{(n+1)}_{i,g} + \Sigma^S_{i,g'\rightarrow g}\Phi^{(n)}_{i,g'}$ # Increment total source with scattering
    end for
  end for
end for

by the GPU solver, the FSRs are copied back to the CPU to generate relevant output and data visualizations using the same routines applied to output data from the CPU solver.

During a transport sweep, each GPU thread integrates the flux for one energy group of a track across the entire geometry while updating the FSR scalar fluxes for each segment. The loop over energy groups is unrolled across GPU threads to provide sufficient parallel concurrency to keep all of the GPU cores busy and to reduce thread warp divergence. To achieve good performance on the GPU, OpenMOC makes extensive use of the sophisticated GPU memory hierarchy. In particular, the solver uses fast shared memory for frequently updated values such as FSR sources and scalar fluxes, and cacheable constant memory for fixed value scalar variables such as loop termination conditions.

3.5. Nonlinear Diffusion Acceleration

In addition to providing speedup through parallelization and hardware acceleration, OpenMOC also includes a nonlinear diffusion acceleration (NDA) scheme to solve the neutron transport equation. Acceleration schemes, such as NDA, are necessary when solving full-core problems which require thousands of power iterations due to a high dominance ratio.

The NDA algorithm of choice is the Coarse Mesh Finite Difference (CMFD) method. CMFD was first proposed by Smith (K. S. Smith, 1983) and has been widely used in accelerating neutron diffusion and transport problems for many years (J. Y. Cho, H. G. Joo, K. S. Kim and S. Q. Zee, 2002; Z. Zhong, T. J. Downar, Y. Xu, M. D. DeHart and K. T. Clarno, 2008). In particular, it has been shown that CMFD acceleration gives >100x speedups on large LWR problems (K. S. Smith and J. D. Rhodes, 2002).

CMFD acceleration is implemented in OpenMOC by over-

laying a Cartesian coarse mesh on top of the unstructured flat source region mesh. During a MOC transport sweep, OpenMOC tallies the partial currents across the surfaces of each mesh cell, designated by J^{I+} for the positive sides of coarse mesh cell I and J^{I-} for the negative sides of I. At the end of the transport sweep, OpenMOC proceeds to calculate the following terms for each mesh cell I and energy group g:

Table 1. Variables in CMFD

Variable	Description
$\bar{\phi}_{g,I}$	Volume-averaged scalar flux
$\bar{\Sigma}^{x}_{g,I}$	Flux-weighted cross section for reaction x
$\bar{D}_{g,I}$	Flux-weighted diffusion coefficient
$\hat{D}_{g,I\pm}$	Finite-difference diffusion coefficient coupling mesh cell I and $I \pm 1$
$\tilde{D}_{g,I\pm}$	Finite-difference-like nonlinear diffusion coefficient coupling mesh cells I and $I \pm 1$

The nonlinear diffusion coefficient coupling terms lie at the heart of CMFD, and are computed from the $J^{I\pm}$ tallied during the MOC sweep using the net current equation:

$$J_g^{I\pm} = -\hat{D}_g^{I\pm}(\bar{\phi}_g^{I\pm1} - \bar{\phi}_g^{I}) - \tilde{D}_g^{I\pm}(\bar{\phi}_g^{I\pm1} + \bar{\phi}_g^{I}) \qquad (11)$$

There are two subtle points in computing the nonlinear coupling coefficients $\tilde{D}_g^{I\pm}$. First, the condition $|\tilde{D}_g^{I\pm}| < |\hat{D}_g^{I\pm}|$ must be met in order to guarantee the diagonal dominancy in the destruction matrix. Otherwise, OpenMOC will re-compute two terms that are equal in magnitude and satisfy Equation 11. Furthermore, under-relaxation of the nonlinear correction factor is used to accelerate and maintain stability of the eigenvalue convergence rate for large, heterogeneous geometries. OpenMOC does so by applying a fixed damping factor on the $\tilde{D}_g^{I\pm}$ terms.

Upon computing the above terms, OpenMOC sets up the standard production and destruction matrices for the diffusion eigenvalue problem with addition of nonlinear coupling coefficients in the destruction matrix:

$$\mathbf{A}\phi = \frac{1}{k_{eff}}\mathbf{M}\phi \qquad (12)$$

where $\mathbf{A}$ is the destruction matrix whose main diagonal contains absorption, out-scattering and leakage terms (from this cell to the four adjacent cells) and off-diagonals contain in-scattering and leakage terms (from the adjacent cells), $\mathbf{M}$ is the construction matrix filled with the fission source terms, and ϕ is a vector containing scalar fluxes for each mesh cell and energy group.

Within each CMFD iteration, OpenMOC uses power iterations to solve the generalized non-Hermitian eigenvalue problem. In each power iteration, the linear system is solved using a parallel (red-black) implementation of the successive over-relaxation method.

Upon convergence of the CMFD diffusion problem, OpenMOC performs prolongation by multiplying each FSR's scalar flux by the ratio of the converged coarse mesh scalar flux to the initial coarse mesh scalar flux in the acceleration step:

$$\bar{\phi}_{g,i} = \bar{\phi}_{g,i}\frac{\bar{\phi}_{g,I}}{\bar{\phi}_{g,I,0}} \qquad \forall i \in I \qquad (13)$$

where $\bar{\phi}_{g,i}$ is the FSR scalar flux for energy group g and FSR i in coarse mesh cell I, $\bar{\phi}_{g,I,0}$ is the initial scalar flux for the CMFD solver, and $\bar{\phi}_{g,I}$ is the converged CMFD scalar flux for mesh cell I.

4. Modern Software Design

4.1. Programming Languages

One of the motivations behind OpenMOC was to develop a code with balanced design criteria for maximal compute performance, maintainability, extensibility, and portability. In addition, it was required that OpenMOC be both user and developer friendly with a relatively small learning curve. To achieve these goals, the code was developed to conform to modern software development practices.

OpenMOC is designed using the object-oriented programming paradigm, a standard for software development for over two decades. In addition, OpenMOC uses a compiled language coupled with a scripting language "glue" (M. F. Sanner, 1999), a methodology that has increasingly gained traction across scientific and engineering disciplines since it enables both usability and performance.

The majority of the source code is written in C/C++ as it is the most robust and well supported general purpose, high performance, compiled programming language with object-oriented features. In addition, OpenMOC's solver routines for the GPU are written in NVIDIA's CUDA programming language (NVIDIA, 2013) - a compiled language with similar syntax to C/C++. The widely adopted *Simplified Wrapper Interface Generator (SWIG)* (D. M. Beazley, 2003) is deployed to expose the C/C++/CUDA classes and routines to the Python scripting language (see Figure 3). OpenMOC's Python interface allows for rapid prototyping and testing of code features and tight integration with the rich ecosystem of powerful data processing and visualization tools developed for Python.

OpenMOC uses the Git revision control system and an open source distribution is hosted on GitHub at `https://github.com/mit-crpg/OpenMOC`. The build system and configuration management for OpenMOC is handled using Python's *Distutils* package, which is provided by default with all modern Python distributions.

4.2. User Input

OpenMOC's Python interface makes it relatively easy to create complicated reactor models with modest effort. Generating input for an OpenMOC simulation does not involve writing an input file in the traditional sense. OpenMOC leverages the flexiblity provided by Python to allow users complete control to build their inputs in one or more scripts just as

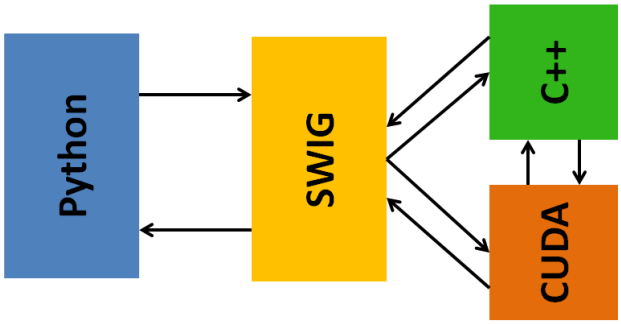

Figure 3. The model used in OpenMOC to couple the compiled C/C++/CUDA code to the Python scripting language.

one may do for any Python program. The user imports the necessary OpenMOC modules (see Table 2) into Python and "builds" a simulation using only those classes and routines which are needed.

Table 2. OpenMOC's Python modules

Module	Description
`openmoc`	Main module for OpenMOC
`openmoc.options`	Command line options
`openmoc.log`	Level-based logging messages
`openmoc.materialize`	Import multi-group nuclear data
`openmoc.plotter`	Visualizations for geometry, flux, etc.
`openmoc.process`	Data processing
`openmoc.cuda`	Routines for NVIDIA GPUs

Figure 4 illustrates a simple OpenMOC Python script to model a simple pin cell lattice. The script first defines some of the key simulation parameters for an OpenMOC simulation, including the track spacing and azimuthal angle quadrature order. With respect to materials data, such as multi-group cross-sections, the user can manually create arrays of the data within the input script and assign the data to unique *`Material`* class objects. The *`openmoc.materialize`* module provides routines to write nuclear data to a binary format, such as HDF5 (S. Koranne, 2011), and easily retrieve the data at runtime as shown in Figure 4. To construct the geometry using the CSG formulation as described in Section 3.1, the user defines *`Surface`*, *`Cell`*, *`Universe`* and *`Lattice`* objects from the main *`openmoc`* module and adds them to a *`Geometry`* object. Once the simulation parameters have been defined, all materials data has been imported, and the *`Geometry`* has been built, the user may instantiate one of the *`Solver`* class objects provided with OpenMOC for CPUs or NVIDIA GPUs.

The open source distribution of OpenMOC provides a number of sample input files of varying complexity which may serve as templates for new users and developers wishing to learn how to use the code.

4.3. Simulation Output

The Python framework for OpenMOC makes it convenient to leverage the extensive data processing and visualization capabilities available in the scientific Python ecosystem. The OpenMOC code includes mechanisms to generate output in the form of text-based console output, visualizations, and binary data files.

The *`openmoc.log`* module uses a level-based logging module that is unified between the C/C++/CUDA and Python source codes. Messages written using the *`openmoc.log`* module's routines are displayed in the console and written to a persistent logfile for each simulation run. Furthermore, OpenMOC's *`openmoc.process`* module includes the functionality needed to store simulation data - such as k_{eff}, flat source region fluxes, etc. - to binary output file(s) and to retrieve simulation data for data processing at a later time.

The *`openmoc.plotter`* module contains routines to create visualizations using the popular matplotlib Python package (J. D. Hunter, 2007). The types of plots which may be generated include diagrams of the geometry color-coded by material, cell, or flat source region (see Figure 1). In addition, it is often instructive to plot a diagram of the tracks and track segments (see Figure 2). Furthermore, the *`openmoc.plotter`* module may be used to plot the flux by energy group as well as the normalized pin and assembly powers.

An example code snippet utilizing the *`openmoc.plotter`* and *`openmoc.process`* modules is shown in Figure 5.

5. Results

5.1. C5G7 2D Benchmark

A series of test cases were run for the 7-group 2D C5G7 benchmark problem (E. E. Lewis, G. Palmiotti, T. A. Taiwo, R. N. Blomquist, M. A. Smith and N. Tsoulfanidis, 2003). The C5G7 problem was developed as a modern benchmark for deterministic neutron transport methods without spatial homogenization. The problem contains four 17 × 17 pin cell assemblies with vacuum boundary conditions on the right and bottom and reflective boundary conditions on the left and top boundaries. Each pin has a 0.54 cm radius with a pitch of 1.26 cm. The bundles on the top left and bottom right contain UO_2 fuel while the ones on the opposite two corners are MOX assemblies, as depicted in Figure 6. Each assembly contains a pattern of fuel pin cells of varying enrichments, guide tubes and fission chambers.

A model of the C5G7 geometry was built for OpenMOC with three rings and eight angular divisions per pin cell. A 1.26 mm × 1.26 mm mesh was used for the moderator region adjacent to the bundles (of width 13.86 cm) to capture the thermal flux gradient in this region. A coarser 1.26 cm × 1.26 cm mesh was used for the remaining 7.56 cm of moderator on the outermost edge of the geometry. A total of 142,964 flat source regions were represented for the entire geometry. OpenMOC was used to converge the source distribution to 1E-5 with 12 threads on two Intel Xeon processors, each with six cores. The thermal flux distribution is shown in Figure 7.

The converged eigenvalues for OpenMOC's solvers are presented in Table 3 for 4 - 128 azimuthal angles with 0.01 cm track spacing, and in Table 4 for 0.1 - 0.01 cm track spacing

```
from openmoc import *
from openmoc.materialize import *

num_threads = 4                 # Number of OpenMP threads
track_spacing = 0.1             # Preferred track spacing in cm
num_azim = 128                  # Azimuthal angle quadrature order
tolerance = 1E-5                # Convergence criterion on the source

# Import materials data from HDF5 file
materials = materialize('materials.hdf5')

# Extract the unique ID for each Material of interest
uo2_id = materials['UO2'].getId()
water_id = materials['Water'].getId()

# Create bounding Surfaces for a pin cell
circle = Circle(x=0.0, y=0.0, radius=0.45)
left = XPlane(x=-0.63)
right = XPlane(x=0.63)
top = YPlane(y=0.63)
bottom = YPlane(y=-0.63)

# Create bounded Cells
cells = []
cells.append(CellBasic(universe=1, material=uo2_id))    # Fuel
cells.append(CellBasic(universe=1, material=water_id))  # Moderator
cells.append(CellFill(universe=0, universe_fill=2))     # Pin cell

# Add bounding Surfaces to each Cell
cells[0].addSurface(halfspace=-1, surface=circle)
cells[1].addSurface(halfspace=+1, surface=circle)
cells[2].addSurface(halfspace=+1, surface=left)
cells[2].addSurface(halfspace=-1, surface=right)
cells[2].addSurface(halfspace=+1, surface=bottom)
cells[2].addSurface(halfspace=-1, surface=top)

# Create simple 2 x 2 Lattice
lattice = Lattice(id=2, width_x=4.0, width_y=4.0)
lattice.setLatticeCells([[1, 1],
                         [1, 1]])

# Create the Geometry
geometry = Geometry()

# Add Materials, Cells and Lattices to the Geometry
for material in materials.values(): geometry.addMaterial(material)
for cell in cells: geometry.addCell(cell)
geometry.addLattice(lattice)

# Create the TrackGenerator and perform ray tracing
track_generator = TrackGenerator(geometry, num_azim, track_spacing)

# Create a Solver for the CPU and converge the source
solver = CPUSolver(geometry, track_generator)
solver.setConvergenceThreshold(tolerance)
solver.setNumThreads(num_threads)
solver.convergeSource(max_iters)
```

Figure 4. A sample OpenMOC Python script to model a 2 × 2 fuel pin lattice. First, the necessary OpenMOC Python modules are imported and variables are defined for some of the parameters in an OpenMOC simulation. Multi-group nuclear data is imported into OpenMOC from an HDF5 binary file. A `Geometry` composed of `Surfaces`, `Universes`, `Cells` and `Lattices` is constructed. Finally, a `Solver` class object for the CPU is instantiated and used to converge the source distribution.

```
import openmoc.plotter as plotter
import openmoc.process as process

# Generate plots using Matplotlib
plotter.plotMaterials(geometry)
plotter.plotCells(geometry)
plotter.plotFlatSourceRegions(geometry)
plotter.plotFluxes(geometry, solver, energy_groups=[1,3,5,7])

# Store the simulation data in an HDF5 output file
process.storeSimulationState(solver, filename='pin-cell-data', extension='hdf5')
```

Figure 5. Routines to plot the materials, geometry, FSRs and more are provided in the *openmoc.plotter* module. The *openmoc.process* module enables users to store/retrieve simulation data, such as k_{eff} and FSR scalar fluxes, to binary files.

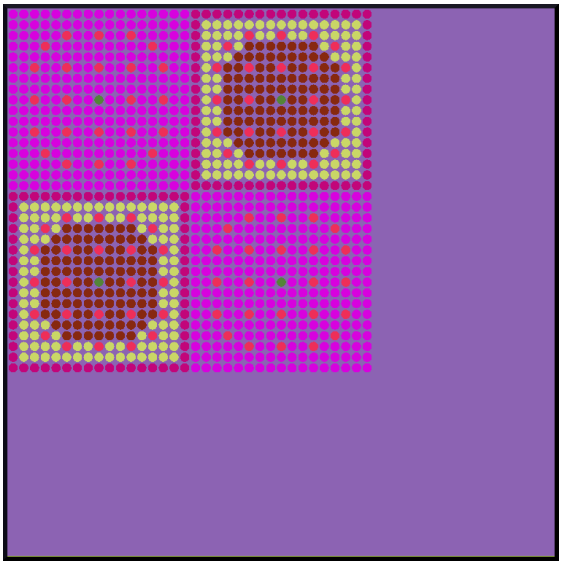

Figure 6. The C5G7 benchmark problem.

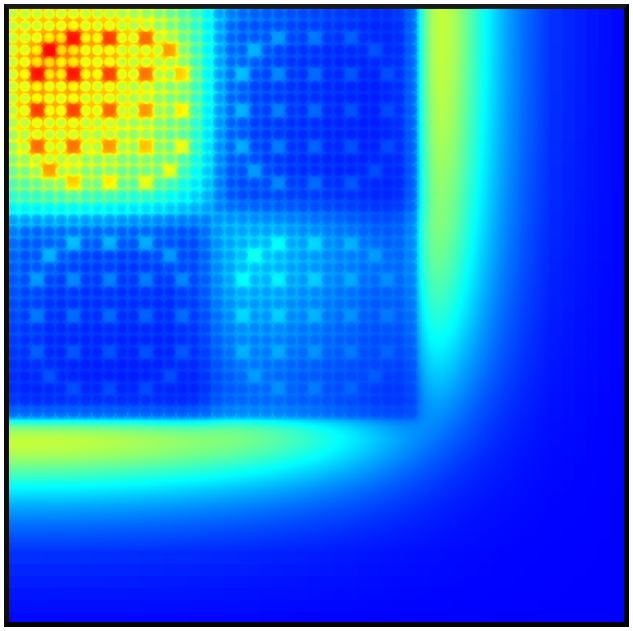

Figure 7. Thermal flux distribution for the C5G7 benchmark.

with 128 azimuthal angles. Each of these cases was run without use of CMFD acceleration. The absolute error in pcm for each converged solution with respect to the reference Monte Carlo eigenvalue solution of $k_{eff} = 1.18655 \pm 9.5$ pcm is given in each table. The results demonstrate excellent agreement between OpenMOC's converged eigenvalue to within the uncertainty of the reference Monte Carlo solution.

Table 3. OpenMOC converged eigenvalues with respect to azimuthal angle quadrature order for the C5G7 benchmark.

# Azimuthal Angles	Transport Sweeps	k_{eff}	$\Delta\rho$ [pcm]	Runtime [sec]
4	702	1.18543	-112	508
8	717	1.18456	-199	788
16	718	1.18499	-151	1,452
32	715	1.18625	-25	2,515
64	715	1.18650	-5	4,655
128	715	1.18663	+8	8,884

The percent relative pin power error for each pin in the C5G7 benchmark problem is illustrated in Figure 8. As expected, the maximum errors are for those pins nearest the moderator where the thermal flux gradient is greatest. The average and maximum relative pin power error dependence on azimuthal angle quadrature order is shown in Table 5 for cases run with 0.01 cm track spacing.

5.2. LRA 2D Diffusion Benchmark

The diffusion solver implemented in OpenMOC has been validated with the 2D Laboratorium für Reaktorregelung (LRA)

Table 4. OpenMOC converged eigenvalues with respect to track spacing for the C5G7 benchmark.

Track Spacing [cm]	Transport Sweeps	k_{eff}	$\Delta\rho$ [pcm]	Runtime [sec]
0.1	715	1.18664	+9	1,017
0.05	715	1.18662	+7	1,900
0.01	715	1.18663	+8	8,884

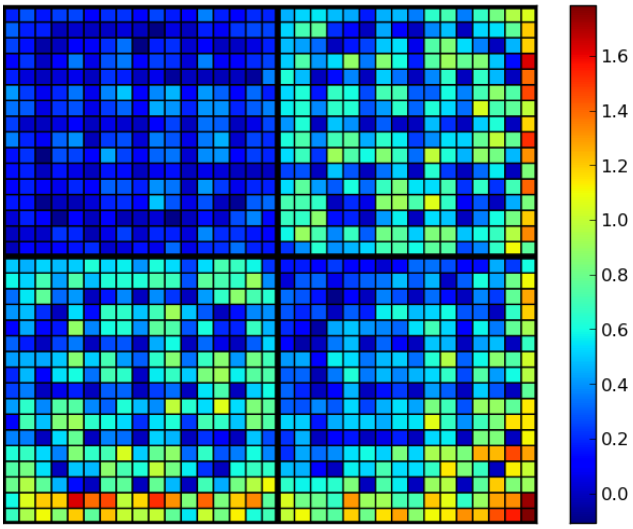

Figure 8. Percent relative pin power errors for the C5G7 benchmark.

Table 5. OpenMOC converged average and maximum percent relative pin power errors for the C5G7 benchmark.

# Azimuthal Angles	Avg. Relative % Error	Max. Relative % Error
4	1.791	6.558
8	0.404	1.659
16	0.368	1.720
32	0.436	1.734
64	0.451	1.772
128	0.456	1.785

benchmark initial steady state solution. The 2D LRA benchmark is a 2-group, quarter-core BWR blade drop transient problem. The geometry consists of 78 15 cm × 15 cm homogenized fuel assemblies (regions 1-4) surrounded by water cells (region 5) to fill the 165 cm × 165 cm geometry as depicted in Figure 9.

q

The reference eigenvalue for the initial steady state problem was taken to be $k_{eff} = 0.99637$, as reported by B. N. Aviles (1993). To validate the diffusion solver, a fine mesh was overlayed on the LRA geometry and successively refined until the eigenvalue converged. OpenMOC was used to converge the root-mean-square of the relative change in the successive iteration's energy-integrated fission source to 1E-8 with 12 threads on two Intel Xeon processors, each with six cores. The over-relaxation factor on the linear solve was set to 1.5 for all cases. As shown in Table 6, the OpenMOC diffusion solver eigenvalue agrees quite well with the reference solution. The diffusion solver for the nonlinear CMFD acceleration equations uses the same general diffusion solver with a non-symmetric form of the iteration matrix.

5.3. Parallel Scaling

The C5G7 benchmark problem was used in a series of scaling studies to profile OpenMOC's parallel performance (W. Boyd, K. Smith, B. Forget and A. Siegel, 2014). One type of study applied was a *strong scaling* study in which the problem size is fixed while the number of parallel threads is varied. In this case, the C5G7 problem was solved with 192 azimuthal angles and 0.1 cm track spacing with 1-12 threads on 12 Intel Xeon Sandy Bridge-EP cores with 24 GB of memory. Intel's icpc (version 13.1.0) and GNU's g++ (version 4.4.6) C++ compilers were evaluated for both single and double precision (SP and DP). The parallel speedup for the strong scaling study is shown in Figure 10. As shown in the Figure, all compiled versions of OpenMOC's multi-threaded solvers achieve nearly 11× speedup with 12 threads on 12 cores.

In addition, studies to compare performance results between the CPU and GPU solvers were performed (W. Boyd, 2014). Figure 11 illustrates the parallel speedup for the GPU as the number of azimuthal angles is scaled from 4 - 128 for four NVIDIA GPUs. The results demonstrate a nearly 50× speedup for the GPU-based solver over the sequential CPU solver. The advantage for the GPU grows with the problem size as the azimuthal angle quadrature order increases.

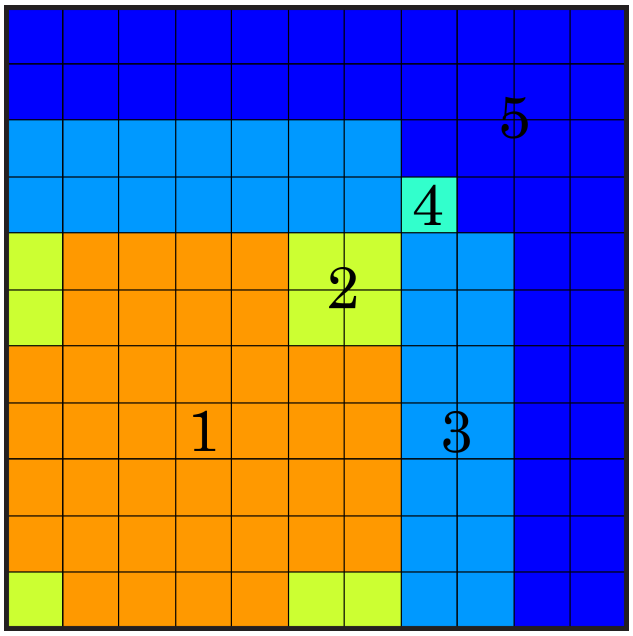

Figure 9. The LRA benchmark geometry.

Table 6. OpenMOC diffusion solver converged eigenvalues with respect to mesh cell size for the LRA benchmark.

Mesh Size [cm]	k_{eff}	$\Delta\rho$ [pcm]	Runtime [sec]
15	1.00389	755	0.4
7.5	0.99750	113	0.7
3.75	0.99623	-14	1.0
1.875	0.99625	-12	4.0
0.9375	0.99633	-4	38.0
0.46875	0.99636	-1	396.4
ref	0.99637	-	-

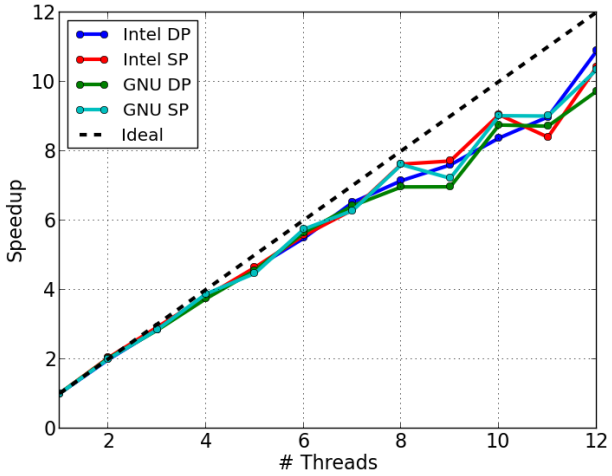

Figure 10. Strong scaling for the C5G7 benchmark with 1-12 threads on 12 cores.

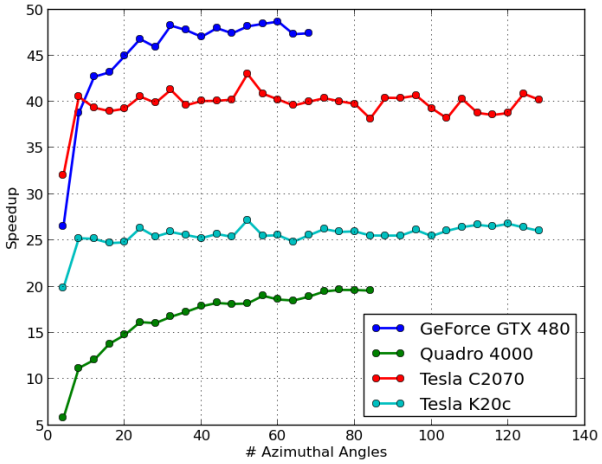

Figure 11. Speedup for NVIDIA GPUs relative to a single Intel Xeon CPU core for the C5G7 benchmark.

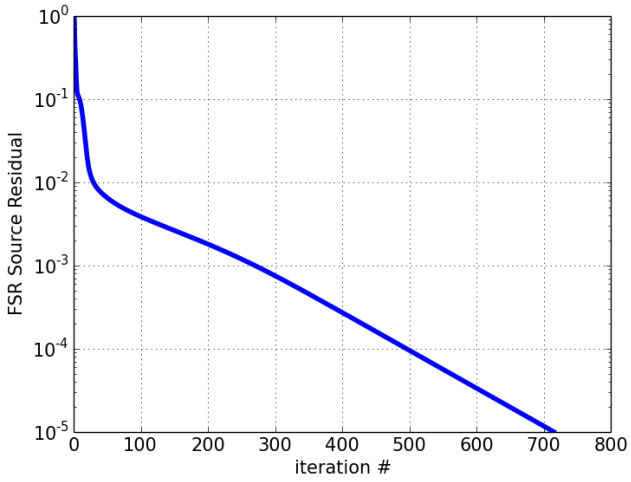

Figure 12. OpenMOC's convergence rate for the C5G7 benchmark without CMFD acceleration.

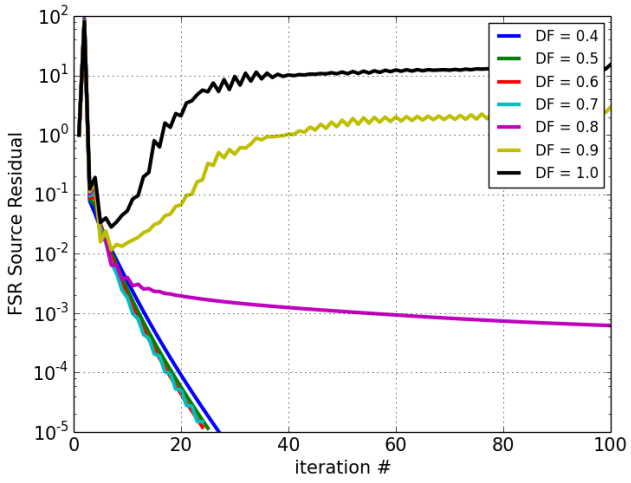

Figure 13. OpenMOC's convergence rate for the C5G7 benchmark using different CMFD damping factors.

5.4. Nonlinear Acceleration

To illustrate the effect of CMFD acceleration on the C5G7 benchmark problem, Figure 12 shows the root-mean-square of the relative change in the successive iteration's energy-integrated fission source for the unaccelerated OpenMOC calculation. As expected, the unaccelerated calculation takes hundreds of power iterations to solve a heterogeneous reactor problem with a high dominance ratio. In contrast, CMFD acceleration dramatically reduces the number of transport iterations and runtime, as shown in Table 7.

The effect of a fixed damping factor on the CMFD method is further investigated. Table 7 and Figure 13 show the number of transport sweeps required to converge the source distribution to 1E-5 for the C5G7 benchmark problem with 0.05 cm track spacing and 64 azimuthal angles using 12 threads on two Intel Xeon processors, each with six cores. As shown in Table 7 the CMFD method with a damping factor of less than 0.8 reduces the number of MOC transport sweeps by a factor of 30 and the runtime by a factor of 20. The optimal damping factor is about 0.7. For this specific case, no damping or using a damping factor of 0.8 or 0.9 would fail to converge the problem. The eigenvalues for the unaccelerated and accelerated cases agree quite well with a difference of less than 10 pcm. This difference can be expected for this problem and convergence criteria due to the high dominance ratio causing slightly premature convergence for the unaccelerated case.

6. Conclusions

A new method of characteristics code called OpenMOC has been developed for 2D neutron transport calculations. OpenMOC is an open source platform created to explore advanced algorithmic acceleration schemes and parallel algorithms for next generation heterogeneous computer architectures. The results presented in this paper verify the accuracy of the OpenMOC code with respect to other deterministic

Table 7. The eigenvalue, time, and number of transport sweeps required to convege the C5G7 benchmark with different CMFD damping factors.

CMFD	Damping Factor	Transport Sweeps	k_{eff}	Runtime [sec]
No	N/A	715	1.18650	4,655
Yes	0.4	24	1.18659	236
Yes	0.5	23	1.18659	216
Yes	0.6	22	1.18659	208
Yes	0.7	21	1.18659	207
Yes	0.8	-	-	-
Yes	0.9	-	-	-
Yes	1.0	-	-	-

neutron transport codes in solving the 2D C5G7 and LRA benchmark problems. OpenMOC has been shown to scale nearly perfectly on single-node shared memory CPUs, and to achieve nearly 50× speedups on NVIDIA GPUs with respect to a single-threaded CPU solver.

CMFD non-linear acceleration scheme has been implemented in OpenMOC and has been shown to achieve a factor of 30 reduction in the number of MOC transport sweeps required to obtain the same convergence as the unaccelerated method of characteristics results for the C5G7 example shown in this paper. An advanced low order acceleration scheme based on transport theory has also been implemented in the OpenMOC framework (L. Li, 2013).

The OpenMOC code is being actively pursued by the Computaional Reactor Physics Group at MIT and the DOE's Center for Exascale Simulation of Advanced Reactors. The OpenMOC collaboration intends to continue devloping the code with the intention of creating a tool for 3D full-core reactor physics calculations. To achieve that goal, future work will need to focus on distributed memory parallelism via spatial/angular domain decomposition to model larger and more complex problems. In addition, higher order source approximation schemes (R. Ferrer, J. Rhodes and K. Smith, 2012) will permit coarser spatial discretization which will reduce the memory requirements for large calculations. It is the hope of those involved with the OpenMOC collaboration that others in the nuclear engineering community will make use of and contribute to the open source codebase for their own research interests.

Acknowledgments

The software design principles employed for OpenMOC are in large part inspired by the legacy left behind by Paul Romano on the MIT Computational Reactor Physics Group.

The first author was supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374 in addition to the DOE's Center for Exascale Simulation of Advanced Reactors (CESAR). The second author is a recipient of the DOE Office of Nuclear Energy's Nuclear Energy University Programs Fellowship, and the third author is a Studsvik Scandpower Graduate Fellow. This work was also partially supported by the Office of Advanced Scientific Computing Research, Office of Science, US Department of Energy, under Contract DE-AC02-06CH11357.

References

A. Yamamoto, M. Tabuchi, N. Sugimura, T. Ushio and M. Mori, 2007. Derivation of Optimum Polar Angle Quadrature Set for the Method of Characteristics Based on Approximation Error for the Bickley Function. Journal of Nuclear Science and Engineering 44 (2), 129–136.

B. N. Aviles, 1993. Development of a Variable Time-Step Transient NEM Code: SPANDEX. pp. 425–427.

D. Lax, W. Boyd and N. Horelik, 2014. An Algorithm for Identifying Unique Regions in Constructive Solid Geometries. In: Submitted to the Proceedings of PHYSOR. Kyoto, Japan.

D. M. Beazley, 2003. Automated Scientific Software Scripting with SWIG. Future Generation Computer Systems 19 (5), 599–609.

E. E. Lewis, G. Palmiotti, T. A. Taiwo, R. N. Blomquist, M. A. Smith and N. Tsoulfanidis, 2003. Benchmark Specifications for Deterministic MOX Fuel Assembly Transport Calculations without Spatial Homogenization. , Organisation for Economic Co-operation and Development's Nuclear Energy Agency.

G. E. Moore, 1965. Cramming More Components onto Integrated Circuits. Electronics 38 (8), 114–117.

Intel, 2012. Intel C++ Intrinsic Reference. http://software.intel.com/en-us/articles/intel-intrinsics-guide, [Online; accessed 12/30/2013].

Intel, 2013a. Intel Architecture Instruction Set Extensions Programming Reference. http://software.intel.com/en-us/intel-isa-extensions, [Online; accessed 10/30/2013].

Intel, 2013b. Intel Math Kernel Library. http://software.intel.com/en-us/intel-mkl, [Online; accessed 10/30/2013].

J. D. Hunter, 2007. Matplotlib: A 2D Graphics Environment. Computing In Science & Engineering 9 (3), 90–95.

J. R. Askew, 1972. A Characteristics Formulation of the Neutron Transport Equation in Complicated Geometries. AAEW-M 1108, UK Atomic Energy Establishment.

J. Y. Cho, H. G. Joo, K. S. Kim and S. Q. Zee, 2002. Cell Based CMFD Formulation for Acceleration of Whole-Core Method of Characteristics Calculations. Journal of the Korean Nuclear Society 34 (3), 250–258.

K. S. Smith, 1983. Nodal Method Storage Reduction by Non-linear Iteration. Vol. 44.

K. S. Smith and J. D. Rhodes, 2002. Full-Core, 2-D, LWR Core Calculations with CASMO-4E. In: Proceedings of PHYSOR. Seoul, South Korea.

L. Li, 2013. A Low Order Acceleration Scheme for Solving the Neutron Transport Equation. M.S. Thesis, Massachusetts Institute of Technology.

M. F. Sanner, 1999. Python: A Programming Language for Software Integration and Development. Journal of Molecular Graphics and Modelling 17 (1), 57–61.

NVIDIA, 2013. NVIDIA CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, [Online; accessed 12/22/2013].

OpenMP Architecture Review Board, 2013. OpenMP Application Program Interface Version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, [Online; accessed 12/22/2013].

P. K. Romano and B. Forget, 2013. The OpenMC Monte Carlo Particle Transport Code. Annals of Nuclear Energy 51, 274–281.

R. Ferrer, J. Rhodes and K. Smith, 2012. Linear Source Approximation in CASMO5. In: Proceedings of PHYSOR. Knoxville, TN, USA.

S. Koranne, 2011. Hierarchical Data Format 5: HDF5. In: Handbook of Open Source Tools. Springer US, pp. 191–200.

W. Boyd, 2014. Massively Parallel Algorithms for Method of Characteristics Neutral Particle Transport on Shared Memory Computer Architectures. M.S. Thesis, Massachusetts Institute of Technology.

W. Boyd, K. Smith and B. Forget, 2013. A Massively Parallel Method of Characteristic Neutral Particle Transport Code for GPUs. In: Proceedings of the International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering. Sun Valley, ID, USA.

W. Boyd, K. Smith, B. Forget and A. Siegel, 2014. Parallel Performance Results for the OpenMOC Method of Characteristics Code on Multi-Core Platforms. In: Submitted to the Proceedings of PHYSOR. Kyoto, Japan.

Z. Zhong, T. J. Downar, Y. Xu, M. D. DeHart and K. T. Clarno, 2008. Implementation of Two-Level Coarse Mesh Finite Difference Acceleration in an Arbitrary Geometry, Two-Dimensional Discrete Ordinates Transport Method. Nuclear Science and Engineering 158 (3), 289–298.