EPJ Web of Conferences **302**, 02009 (2024) https://doi.org/10.1051/epjconf/202430202009
SNA + MC 2024

Efficient sweep kernels on shared-memory architectures for the discrete ordinates neutron transport equation on Cartesian and hexagonal geometries

Gabriel Suau[1,*], *Ansar* Calloo[2], *Rémi* Baron[2], *Romain* Le Tellier[3], and *Thierry* Gautier[4]

[1]Université Paris-Saclay, CEA, Service d'Études des Réacteurs et de Mathématiques Appliquées, F-91191, Gif-sur-Yvette, France
[2]Université Paris-Saclay, CEA, Service de Génie Logiciel pour la Simulation, F-91191, Gif-sur-Yvette, France
[3]CEA, DES, IRESNE, DTN, Cadarache, F-13108 Saint-Paul-Lez-Durance, France
[4]INRIA / AVALON / LIP - ENS Lyon

Abstract. This paper describes the implementation of DONUT, a small multi-group S_N-DG transport solver that aims at providing efficient and portable sweep kernels on shared-memory architectures for Cartesian and hexagonal geometries. DONUT heavily relies on the Kokkos C++ library for portability and genericity. First encouraging performance results are presented for multicore CPU architectures.

1 Introduction

The discrete ordinates (S_N) method [1] is a popular method to discretise the angular variable of the multigroup neutron transport equation (NTE) as it provides a good balance between accuracy and computational cost in many applications including reactor core physics. It is often used in conjunction with an upwind Discontinuous Galerkin (DG) spatial discretisation [2] to handle structured (Cartesian, hexagonal) or unstructured meshes, with arbitrarily high-order elements. Due to the high dimensionality of the problem, resolution is achieved by nested iterative algorithms [3]. For each inner iteration, the advection-reaction operator is inverted by sweeping the domain for a set of discrete directions and energy groups. The transport sweep is the most time-consuming kernel, thus it is crucial to provide an efficient parallel implementation.

Massively parallel computing architectures exhibit three levels of parallelism: multiple computing nodes with their own memory interconnected through a network, multicore CPUs on each node sharing the same memory, and vector units in each core that operate on vector registers containing multiple values (SIMD). Modern architectures also exhibit heterogeneous computing units at the node level, with one or several GPUs per node. While distributed parallelism is generally tackled using the MPI standard and multicore CPU parallelism is traditionally harnessed using the OpenMP programming model, the use of GPUs requires specific programming models such as CUDA for NVIDIA GPUs or HIP for AMD GPUs.

For transport sweep on Cartesian grids, the Koch-Baker-Alcouffe (KBA) algorithm [4] is widely used in conjunction with MPI to parallelise over the spatial variable and its efficiency

*e-mail: gabriel.suau@cea.fr

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

has been proven both theoretically and experimentally [5]. In addition to flat MPI strategies that use one MPI process per core, Hybrid MPI+X and task-based parallelisation strategies have also been developed to make efficient use of multicore processors [6]. In [7], explicit vectorisation on directions *via* the use of SIMD types also showed good performance for CPUs supporting standard vector instruction sets. With the growing heterogeneity of massively parallel clusters, several research works have been carried out to develop efficient GPU transport sweep kernels [8–11], but with mixed results for high-order DG schemes.

In this paper, we present our first steps towards an implementation of the transport sweep kernel for a high-order DG spatial scheme on Cartesian and hexagonal grids that can achieve good performance on CPUs.

2 Numerical Background

2.1 Energy and angular discretisations

In this work, we consider the stationary NTE discretised with a multigroup formulation in energy and a discrete ordinates (S_N) method [1] for the angular variable. For each energy group $g \in [\![1, n_g]\!]$ and discrete angular direction $d \in [\![1, n_d]\!]$, the semi-discrete neutron transport equation can be written

$$\begin{aligned}\vec{\Omega}_d \cdot \vec{\nabla}\psi_d^g(\vec{r}) + \Sigma_t^g(\vec{r})\psi_d^g(\vec{r}) = \frac{1}{2\pi}\sum_{g'}\sum_{l=0}^{L}\frac{2l+1}{2}\Sigma_{sl}^{g\leftarrow g'}(\vec{r})\sum_{m=-l}^{+l}\mathcal{R}_l^m(\vec{\Omega}_d)\phi_{lm}^{g'}(\vec{r}) \\ + \frac{\chi^g(\vec{r})}{4\pi k_{eff}}\sum_{g'}\nu\Sigma_f^{g'}(\vec{r})\phi_{00}^{g'}(\vec{r}).\end{aligned} \quad (1)$$

where $\psi_d^g(\vec{r})$ is the neutron angular flux, k_{eff} is the effective multiplication factor and the angular flux moments over the real spherical harmonics $\mathcal{R}_l^m(\vec{\Omega})$ are defined as

$$\phi_{lm}^g(\vec{r}) = \int_{\mathbb{S}^2} d\vec{\Omega}\,\mathcal{R}_l^m(\vec{\Omega})\psi^g(\vec{r},\vec{\Omega}) \simeq \sum_{d=1}^{n_d}\omega_d\mathcal{R}_l^m(\vec{\Omega}_d)\psi_d^g(\vec{r}), \quad (2)$$

where ω_d is the weight associated to $\vec{\Omega}_d$ in the angular quadrature formula. With these notations, $\phi_{00}^g(\vec{r})$ is the scalar flux. The neutron transport equation becomes a system of $n_d \times n_g$ equations coupled through the fission and scattering terms. It can be written in operator form as

$$\mathbf{L}\underline{\psi}(\vec{r}) = \mathbf{MSD}\underline{\psi}(\vec{r}) + \frac{1}{k_{eff}}\mathbf{MF}(\mathbf{D}\underline{\psi})_{00}, \quad (3)$$

where

$$\begin{aligned}&\mathbf{L} = \mathrm{diag}\Big[\vec{\Omega}_d \cdot \vec{\nabla} + \Sigma_t^g(\vec{r})\Big]_{(g,d)} \quad \mathbf{S} = \Big[\Sigma_{sl}^{g\leftarrow g'}(\vec{r})\Big]_{(g,lm),(g',lm)} \quad \mathbf{M} = \Big[\frac{2l+1}{4\pi}\mathcal{R}_l^m(\vec{\Omega}_d)\Big]_{(g,d),(g,lm)} \\ &\mathbf{D} = \Big[\omega_d\mathcal{R}_l^m(\vec{\Omega}_d)\Big]_{(g,lm),(g,d)} \quad \mathbf{F} = \Big[\chi^g(\vec{r})\nu\Sigma_f^{g'}(\vec{r})\Big]_{(g,1),(g',1)} \quad \underline{\psi} = \Big[\psi_d^g(\vec{r})\Big]_{(g,d)}.\end{aligned} \quad (4)$$

2.2 Iterative resolution

Equation (3) is a generalised eigenvalue problem, whose solution is the fundamental pair $\{k_{eff}, \underline{\psi}\}$, that is solved using a power iteration method. At each iteration, a fixed source

problem is solved of the form

$$\Big(\mathbf{L} - \mathbf{MSD}\Big)\underline{\psi}^{(p)} = \underline{q}_f^{(p-1)} = \frac{1}{k_{eff}^{(p-1)}}\mathbf{MF}(\mathbf{D}\underline{\psi}^{(p-1)}), \tag{5}$$

where p denotes the power iteration index. The scattering term is then splitted as $\mathbf{S} = \mathbf{A} + \mathbf{B}$, and a fixed-point iterative algorithm is used to compute the flux

$$\Big(\mathbf{L} - \mathbf{MAD}\Big)\underline{\psi}^{(p,m)} = \underline{q}_m^{(m-1)} = \underline{q}_f^{(p-1)} + \mathbf{MBD}\underline{\psi}^{(p,m-1)}, \tag{6}$$

where m denotes the multigroup iteration index. Two classical splitting choices are

- $\mathbf{A} = \mathbf{L}_S + \mathbf{D}_S$ and $\mathbf{B} = \mathbf{U}_S$, leading to the Gauss-Seidel method;
- $\mathbf{A} = \mathbf{D}_S$ and $\mathbf{B} = \mathbf{L}_S + \mathbf{U}_S$, leading to the Jacobi method;

where $\mathbf{L}_S$, $\mathbf{U}_S$ and $\mathbf{D}_S$ are respectively the lower-triangular, upper-triangular and diagonal parts of $\mathbf{S}$. While Gauss-Seidel converges with few iterations, especially for down-scattering dominated problems, it is a purely sequential method for which each group g can only be solved once all groups $g' < g$ have been solved. On the contrary, the Jacobi method needs more iterations to converge, but effectively decouples groups and allows for a parallel-in-energy resolution. Finally, the self-scattering source is converged in each group with a Richardson fixed-point algorithm

$$\mathbf{L}_{g,d}\psi_{g,d}^{(p,m,s)}(\vec{r}) = \Big[\underline{q}_s^{(s-1)}\Big]_{g,d} = \Big[\underline{q}_m^{(m-1)} + \mathbf{MD}_S\mathbf{D}\underline{\psi}^{(p,m,s-1)}\Big]_{g,d}, \tag{7}$$

where s denotes the self-scattering iteration index. For Gauss-Seidel, the multigroup source is computed and the self-scattering problem is solved in each group before going to the next group. For Jacobi, the multigroup source is assembled once for all groups, then the n_g self-scattering problems are solved and converged together.

2.3 Space discretisation

Inside a group g, for each inner iteration, one needs to solve $\forall d \in [\![1, n_d]\!]$

$$\vec{\Omega}_d \cdot \vec{\nabla}\psi_d^g(\vec{r}) + \Sigma_t^g(\vec{r})\psi_d^g(\vec{r}) = q_d^g(\vec{r}), \qquad \forall \vec{r} \in \mathcal{D}, \tag{8}$$

where $\mathcal{D}$ is a spatial 2D or 3D domain, and q_d^g gathers all source contributions (fission, cross- and self-scattering) to group g and direction d. Let us denote $\mathcal{M}_h$ a meshing of $\mathcal{D}$. In this work, $\mathcal{D}$ is either a rectangular domain meshed with rectangular cells or a hexagonal lattice of hexagonal cells in 2D and extruded versions of those in 3D. The spatial variable is discretised using the upwind DG scheme described in [2]. In cell K, the flux is expanded on a $n(p)$-size basis of polynomials $\underline{v}_K = (v_K^1, \ldots, v_K^{n(p)})$ of order p or less

$$\psi_d^g(\vec{r}) = \sum_{i=1}^{n(p)} \psi_{d,i}^g v_K^i(\vec{r}) = \underline{\psi_d^g} \cdot \underline{v}_K. \tag{9}$$

and the local weak formulation is projected on each basis function. Thus, denoting ∂K_d^- the set of incoming faces (3D) or edges (2D) of K for direction $\vec{\Omega}_d = \mu_d\vec{e}_x + \eta_d\vec{e}_y + \xi_d\vec{e}_z$ (as illustrated in figure 1), the flux spatial DOFs are the solution of the local linear system

$$\Big(\mu_d\mathbf{A}_K^x + \eta_d\mathbf{A}_K^y + \xi_d\mathbf{A}_K^z + \Sigma_t^g\mathbf{M}_K - \sum_{F\in\partial K_d^-}(\vec{\Omega}_d\cdot\vec{n}_F)\mathbf{M}_{K,F}^+\Big)\underline{\psi}_{d,K}^g = \mathbf{M}_K\underline{q}_{K,d}^g - \sum_{F\in\partial K_d^-}(\vec{\Omega}_d\cdot\vec{n}_F)\mathbf{M}_{K,F}^-\underline{\psi}_{d,K^{F,-}}^g \tag{10}$$

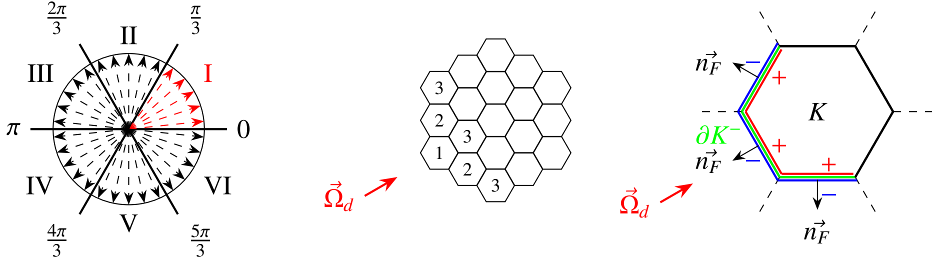


Figure 1: Angular sectors segmentation (left), cell ordering for a transport sweep in a direction in the angular sector I (middle), and associated cell incoming boundary (right) for a 2D hexagonal heometry.

where the local matrices are given by

$$\left(\mathbf{M}_K\right)_{i,j} = \int_K v_K^i v_K^j dV, \qquad \left(\mathbf{A}_K^*\right)_{i,j} = \int_K v_K^i \frac{\partial v_K^j}{\partial *} dV,$$
$$\left(\mathbf{M}_{K,F}^+\right)_{i,j} = \int_F v_K^i v_K^j dS\,, \quad \left(\mathbf{M}_{K,F}^-\right)_{i,j} = \int_F v_K^i v_{K^{F,-}}^j dS\,. \tag{11}$$

denoting $v_{K^{F,-}}^j$ the basis function j in the neighbor cell of K across F.

The angular flux in a given cell K is the solution of a small local $n(p) \times n(p)$ linear system, whose right-hand side depends on the angular flux value in the immediate upwind neighbours. Therefore, the global solution can be computed by successive local resolutions following a propagation front *via* a *transport sweep*, where each cell's outgoing flux is the next cells' incoming flux (see figure 1).

3 Parallel transport sweep on shared memory architectures

3.1 Preliminary remarks

As stated in the introduction, a transport sweep exposes several levels of parallelism. First, in a given direction and energy group, cells lying on the same front can all be processed in parallel. Moreover, coupling between directions only occurs when computing the angular flux moments through the discrete-to-moments operator. Thus, directions can be considered independent during a transport sweep and can also be treated in parallel. Furthermore, since only Cartesian and hexagonal meshes are considered, angular directions can be grouped in angular sectors (see figure 1) such that the spatial sweeping order is the same for all directions in the same sector. In addition, for vacuum boundary conditions, angular sectors are also independent and may be treated in parallel. Finally, when using the Jacobi method for the multigroup iterations, the self-scattering problem is converged for all groups at the same time. Therefore, the transport sweep can also be applied for all groups at the same time, which enables another level of parallelism. Algorithm 1 summarises the loop nest of a generic transport sweep, indicating possibly parallel loops.

Algorithm 1: Pseudo-code for the space-angle-energy transport sweep

```
1  parallel for s ∈ ⟦1, n_s⟧ do
2  |  parallel for g ∈ ⟦1, n_g⟧ do
3  |  |  parallel for d ∈ ⟦1, n_d(s)⟧ do
4  |  |  |  for f ∈ fronts(s) do
5  |  |  |  |  parallel for K ∈ f do
6  |  |  |  |  |  C^g_{d,K} = f1(A^{x,y,z}_K, M_K, M^+_{K,F}, Ω_d, Σ^g_{t,K}); // Build local matrix
7  |  |  |  |  |  b^g_{d,K} = f2(M_K, M^-_{K,F}, Ω_d, q^g_{d,K}, ψ^g_{d,K^{F,-}});   // Build local RHS
8  |  |  |  |  |  Solve C^g_{d,K} ψ^g_{d,K} = b^g_{d,K};            // Gaussian elimination
9  |  |  |  |  end
10 |  |  |  end
11 |  |  end
12 |  end
13 end
```

As stated in [9], the loop on groups, the sector-wise loop on directions, and the loop nest on cells can be freely permuted without changing the result of the sweep operation. Therefore, there are 6 possible permutations of the loops that can be implemented to obtain the solution : GDK, GKD, DGK, DKG, KGD, KDG[1]. While not changing the result, the order of the loops in the transport sweep algorithm may affect performance in three ways:

1. For certain loop orders, some operations can be factorised, which reduces the global number of floating point operations, thus the runtime. For instance, with the KDG and DKG orders, the streaming and surface contributions to the local matrix in equation (10) can be computed knowing only the K and d indices, allowing to reduce the number of operations in the innermost loop on groups.

2. The internal memory layouts of the arrays storing the fluxes, the sources and the cross sections must change according to the order of the loops in order to maximise data locality and cache usage on CPU, and to achieve coalesced memory accesses on GPU.

3. The order of the loops directly impacts the mapping of logical work-items to parallel resources (OpenMP or CUDA threads).

In [9], the authors showed that sweep performance can vary greatly depending on the loop order and the problem setting (number of groups, cells, directions), even for a simple Diamond-Differencing (DD) scheme (for which the local linear system is of size 1). Moreover, the various GPU implementations available in literature employ different loop orders and different mappings of physical dimensions to parallel resources. Thus, it is not clear whether one specific loop order is the best on CPUs and/or GPUs in the case of a high order DG implementation.

3.2 Implementation using the Kokkos library

In order to investigate the performance characteristics of each layout on several architectures, the six permutations were implemented in a mock-up neutron transport solver named DONUT, which heavily relies on the Kokkos framework [12].

[1]The leftmost letter designates the outermost loop, and the rightmost letter the innermost loop. G, D, K stand for groups, directions and cells loops respectively

Table 1: Mapping of Kokkos constructs to backend specific constructs.

	Kokkos	OpenMP	CUDA	HIP
	`(MD)RangePolicy`	Threads	Threads	Threads
Hierarchical	`TeamPolicy`	Threads	Blocks	Blocks
	`TeamThreadRange`	N/A	Threads	Threads
	`ThreadVectorRange`	`#pragma ivdep`	Warp	Wavefront
	`SIMD`	Intrinsics	N/A	N/A

Quick overview of Kokkos

Kokkos is a C++ library aimed at providing efficient data structures and abstractions to express shared-memory parallelism (*i.e.* GPUs and multicore CPUs) in an architecture-agnostic way. The Kokkos data structure to store multidimensional arrays is the `View`. It is a lightweight object holding a pointer to a memory chunk along with light metadata describing the type of the elements, the dimensionality of the array, the `Layout` (size and stride for each dimension) and the `MemorySpace` where the data resides (*i.e.* host or device memory). Kokkos offers three parallel patterns `parallel_for`, `parallel_reduce` and `parallel_scan` used in conjunction with execution policies that define the (possibly multidimensional) index space and the mapping of indices to backend-specific constructs (see Table 1). Kokkos also offers portable SIMD types, which are small packs of scalars that support classical arithmetic operations that map directly to hardware-specific vector instructions. The size and instruction set are chosen at compile-time depending on the target architecture.

Parallelisation and memory layout management

DONUT borrows the concept of groupsets as implemented in the Chi-Tech code developed at Texas A&M University [13]. Energy groups are lumped into groupsets, a Gauss-Seidel algorithm is employed for across-groupset scattering sources, and a Jacobi algorithm is used for within-groupset scattering sources. Therefore, groupsets are processed sequentially, but groups in a groupset are processed in parallel. The standard Gauss-Seidel method is equivalent to setting one group per groupset, and the standard Jacobi method is equivalent to setting one groupset containing all groups. Data is contiguous inside a groupset but may be discontiguous across groupsets. Inside a groupset, the loop nest in algorithm 1 (and its permutations) is parallelised using Kokkos hierarchical parallel constructs. A `TeamPolicy` is launched for the outermost loop, a `TeamThreadRange` for the middle loop, and a `ThreadVectorRange` for the innermost loop. The loop order is chosen at runtime along with the appropriate `View` memory layout which also depends on the target architecture (this version of DONUT targets multi-CPU architectures). There is no parallelisation on the spatial DOFs: the local linear system is assembled and solved using sequential `for` loops. Therefore, on CPUs, the contiguous dimension is the spatial DOFs to achieve cached memory accesses.

Theoretical performance bounds

In Algorithm 1 and in the current implementation, a synchronisation happens between each front (synchronous sweep). However, the computation of a given cell only depends on its immediate upstream neighbors, and could be started before the full resolution of the previous front (asynchronous sweep). For a synchronous sweep, assuming a homogeneous work-load for all cells, the minimum number of stages to complete a full-domain sweep with T threads

and the associated speedup can be computed as

$$N^{sync}_{stages}(T) = \sum_{f \in \text{fronts}} \left\lceil \frac{N_{cells}(f)}{T} \right\rceil \qquad \text{and} \qquad S^{sync}(T) = \frac{N^{sync}_{stages}(1)}{N^{sync}_{stages}(T)}. \tag{12}$$

For an asynchronous sweep, a minimal number of stages can be computed by simulating the sweep execution using a work queue updated at each step.

Vectorisation using Kokkos SIMD types

Vectorisation on directions is achieved using a strategy adapted from [7]. This approach relies on the portable SIMD types provided by Kokkos. While originally developed for the DD scheme, it can be easily extended to the upwind DG scheme. With this strategy, vectorisation on direction is ensured on any CPU supporting standard vector instructions sets (SSE, AVX, AVX512) for any memory layout.

4 Numerical Results

4.1 Benchmarks

KAIST3A 2D benchmark

The KAIST3A benchmark is a small 2D heterogeneous PWR problem from a suite published by the KAIST Nuclear Reactor Analysis and Particle Transport Laboratory [14]. The problem is composed of a MOX-loaded core surrounded by a baffle and reflector. The detailed geometry description and the seven-group cross-sections can be found in [14]. In this work, only the All Rods Out (ARO) case is considered.

Takeda Model 4

The Takeda Model 4 benchmark is a small 3D FBR reactor with hexagonal geometry. The four-group cross-sections and the full geometry description can be found in [15]. In this work, only the ARO case is considered.

4.2 Description of the experiments

All tests are run on a single computing node of the Orcus cluster. The node is composed of two 24-cores AVX-enabled AMD EPYC 7352 processors, with 256GB of memory and 8 NUMA nodes. DONUT is compiled using single precision floating-point arithmetic with GCC 11.2.0 using `-O2 -march=native -mtune=native`, and uses Kokkos 4.2.0 configured with the OpenMP backend.

Experiment 1: strong scalability study

To assess the performance of the sweep implementation, a strong scalability study is performed for both benchmarks using Jacobi and Gauss-Seidel. For each case, 50 sweeps are run in each groupset, resulting in $50 \times n_g$ 1-group sweeps for Gauss-Seidel and 50 n_g-group sweeps for Jacobi. Details on the discretisation parameters are in Table 2. Since there are not enough directions or groups to fully utilise the CPU cores, only parallelisation on front cells can be expected to scale properly. Thus, only the KDG and KGD layouts are tested.

Table 2: Discrete setting for Experiment 1.

Case	Dir. per angular sector	Mesh	Order (DOFs/cell)	Groups
KAIST3A	12	170×170	5 (21)	7
TAKEDA-4	20	169×38	3 (20)	4

Table 3: Solver configuration for Experiment 2.

	No. max iter.	Stopping criterion
Power iterations	N/A	$\epsilon_{k_eff} = 10^{-6}$ $\epsilon_{S_f} = 10^{-5}$
Thermal iterations (Gauss-Seidel/Jacobi)	5	$\epsilon_\phi = 10^{-5}$
Inner iterations	2	$\epsilon_\phi = 10^{-5}$

Experiment 2: Gauss-Seidel/Jacobi comparison

The two benchmarks are run until convergence, for both Jacobi and Gauss-Seidel methods, on the 48 cores of the node, using the KDG layout. Discretisation parameters are the same as before, except for the spatial order for KAIST which is lowered to 3, resulting in 10 local DOFs per cell. The maximum number of iterations and convergence criteria are listed in Table 3.

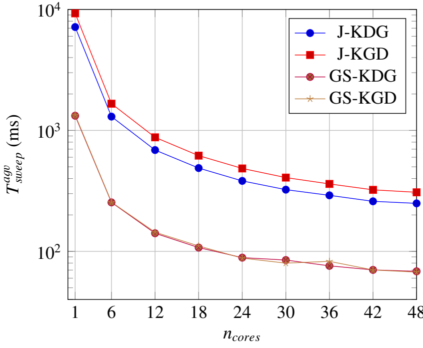
4.3 Results

Experiment 1

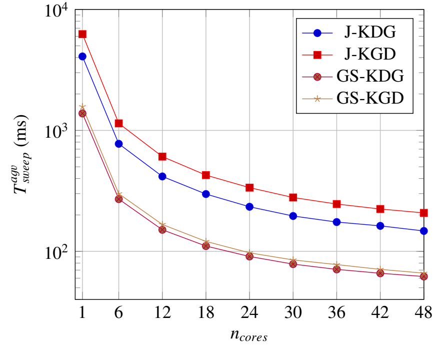
Measured average sweep times for Experiment 1 are plotted in Figure 2 from 1 to 48 cores. Speedups relative to a 1-core execution are also plotted, along with the maximum attainable speedups for ideal synchronous and asynchronous sweeps. First, it can be observed that when using Gauss-Seidel (one group per groupset), the KDG and KGD layouts have nearly the exact same performance. However, with Jacobi (all groups in one groupset), the KDG layout can be up to 35% faster than KGD because some operations can be factorised out of the innermost loop (see Section 3). Furthermore, it can be observed that the sweep scales better with Jacobi (*i.e.* when multiple groups are swept at the same time) than Gauss-Seidel. When threading on cells, sweeping more group at the same time offers more work per thread, thus less thread-management overhead and an overall better scaling. The Jacobi speedups are close to the ideal synchronous sweep speedup under 24 cores, but tend to degrade after. This could be due to either thread scheduling overhead that could be reduced with more groups, directions or DOFs/cell, or NUMA effects that could lead to non-local memory accesses. The implementation of an asynchronous task-based transport sweep could be a solution to improve scalability on CPUs by reducing the number of idle threads during the sweep.

Experiment 2

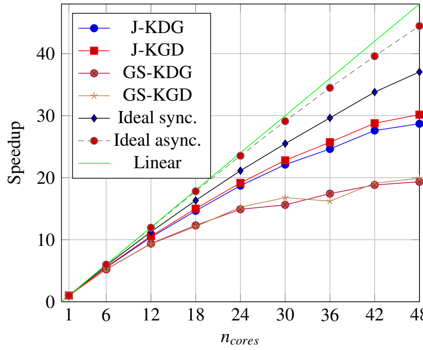
Results of this experiment are presented in Table 4. For both benchmarks, Gauss-Seidel and Jacobi give the same k_{eff} up to tenths of pcms ($\simeq 10^{-6}$), and their values are in line with the references. For KAIST3A, even though Jacobi needs more iterations to converge, it is more



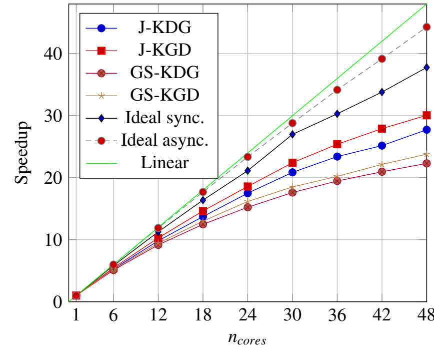
(a) KAIST3A average sweep time.



(b) Takeda-4 average sweep time.



(c) KAIST3A sweep speedups.



(d) Takeda-4 sweep speedups.

Figure 2: Results for Experiment 1. Experimental speedups are computed as Speedup(n_{cores}) = $T^{avg}_{sweep}(1)/T^{avg}_{sweep}(n_{cores})$. The Ideal sync. (resp. async.) curve is the maximum attainable speedup with an ideal synchronous (resp. asynchronous) sweep computed as described in Section 3.2.

than 2× faster than Gauss-Seidel. If we look at the last row, a 7-group sweep in Jacobi is only two times slower than a one-group sweep in Gauss-Seidel. In other words, a 7-group sweep is approximately 3.5× faster than seven 1-group sweeps. The same observations can be made for the Takeda 4-group benchmark, although to a lesser extent. Therefore, if there are enough groups, Jacobi's degraded convergence can be compensated for by the sweep speedup, and may perform better than Gauss-Seidel when using the KDG layout.

5 Conclusions and Future Work

In this work, we implemented parallel transport sweep kernels with different loop orders. The use of a Jacobi method in energy coupled with the KDG loop order and appropriate data layouts can lead to performance improvements on CPUs for Cartesian or hexagonal geometries. Improvements can still be made on CPUs, such as the implementation of an

Table 4: Results of experiment 2. GS stands for Gauss-Seidel and J for Jacobi. δk_{eff} is the relative discrepancy (in pcm) between the computed k_{eff} and the reference value k_{eff}^{ref}. For KAIST3A, $k_{eff}^{ref} = 1.13088$ (from [16]) and for Takeda-4, $k_{eff}^{ref} = 1.0951$ (from [15]). $N_{iter}^{p/th}$ is the total number of power/thermal iterations, N_{sweep} is the total number of sweeps, T^{tot} is the total simulation time, T_{sweep}^{avg} is the average sweep time.

	KAIST3A		Takeda-4	
	GS	J	GS	J
k_{eff} (δk_{eff})	1.130874 (-0.5)	1.130873 (-0.6)	1.095250 (+13.7)	1.095243 (+13.1)
$N_{iter}^{p}/N_{iter}^{th}$	74/332	101/481	49/240	50/243
N_{sweep}	4402	958	1632	486
T^{tot} (s)	175.6	80.8	105.6	72.2
T_{sweep}^{avg} (ms)	37.1	74.3	61.1	145.8

asynchronous transport sweep kernel that would better exploit the spatial parallelism. The next step is to experiment on the performance portability of DONUT on GPUs with Kokkos.

References

[1] K.D. Lathrop, B.G. Carlson, *Discrete ordinates angular quadrature of the neutron transport equation* (1964), https://www.osti.gov/biblio/4666281

[2] W.H. Reed, T.R. Hill, Tech. Rep. LA-UR-73-479; CONF-730414-2, Los Alamos Scientific Lab., N.Mex. (USA) (1973), https://www.osti.gov/biblio/4491151

[3] E.E. Lewis, W.F. Miller, *Computational methods of neutron transport* (1984), publisher: John Wiley and Sons, Inc.,New York, NY, https://www.osti.gov/biblio/5538794

[4] R.S. Baker, K.R. Koch, Nuclear Science and Engineering **128**, 312 (1998), https://doi.org/10.13182/NSE98-1

[5] M.P. Adams, M.L. Adams, W.D. Hawkins, T. Smith, L. Rauchwerger, N.M. Amato, T.S. Bailey, R.D. Falgout, A. Kunen, P. Brown, Journal of Computational Physics **407**, 109234 (2020)

[6] S. Moustafa, M. Faverge, L. Plagne, P. Ramet, *3D Cartesian Transport Sweep for Massively Parallel Architectures with PaRSEC*, in *2015 IEEE International Parallel and Distributed Processing Symposium* (2015), pp. 581–590, https://ieeexplore.ieee.org/document/7161546

[7] S. Moustafa, I. Dutka-Malen, L. Plagne, A. Ponçot, P. Ramet, Annals of Nuclear Energy **82**, 179 (2015)

[8] A. Kunen, J. Loffeld, A. Black, R. Chen, P. Nowak, T. Haut, T. Bailey, P. Brown, S. Rennich, P. Maginot et al., *Porting 3D Discrete Ordinates Sweep Algorithm in Ardra to CUDA*, in *Proceedings of The International Conference on Mathematics and Computational Methods applied to Nuclear Science and Engineering* (American Nuclear Society, 2019), pp. 2585–2598

[9] A.J. Kunen, T.S. Bailey, P.N. Brown, *KRIPKE - A MASSIVELY PARALLEL TRANSPORT MINI-APP*, in *Joint International Conference on Mathematics and Computation, Supercomputing in Nuclear Applications, and the Monte Carlo Method* (American Nuclear Society, 2015)

[10] C. Baker, G. Davidson, T.M. Evans, S. Hamilton, J. Jarrell, W. Joubert, *High performance radiation transport simulations: Preparing for TITAN*, in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), pp. 1–10

[11] T. Deakin, S. McIntosh-Smith, J. Lovegrove, R. Smedley-Stevenson, A. Hagues, Journal of Computational and Theoretical Transport **49**, 121 (2020), `https://doi.org/10.1080/23324309.2020.1775096`

[12] C.R. Trott et al., IEEE Transactions on Parallel and Distributed Systems **33**, 805 (2022)

[13] J.I. Vermaak, J.C. Ragusa, M.L. Adams, J.E. Morel, Journal of Computational Physics **425**, 109892 (2021)

[14] N.Z. Cho, *Benchmark Problem 3A* (KAIST Nuclear Reactor Analysis and Particle Transport Laboratory, 2000), `https://github.com/nzcho/Nurapt-Archives/tree/master/KAIST-Benchmark-Problems`

[15] T. Takeda, H. Ikeda, Journal of Nuclear Science and Technology **28**, 656–669 (1991)

[16] Y. Zhang, X. Zhou, Frontiers in Energy Research **10** (2023)