An Improved Parallelism Scheme for Deterministic Discrete Ordinates Transport

Tom Deakin*, Simon McIntosh-Smith†, and Matt Martineau‡

Department of Computer Science, University of Bristol, Bristol, UK

Wayne Gaudin§

High Performance Computing, UK Atomic Weapons Establishment, Aldermaston, UK

July 1, 2016

Abstract

In this paper we demonstrate techniques for increasing the node-level parallelism of a deterministic discrete ordinates neutral particle transport algorithm on a structured mesh to exploit many-core technologies. Transport calculations form a large part of the computational workload of physical simulations and so good performance is vital for the simulations to complete in reasonable time. We will demonstrate our approach utilizing the SNAP mini-app, which gives a simplified implementation of the full transport algorithm but remains similar enough to the real algorithm to act as a useful proxy for research purposes.

We present an OpenCL implementation of our improved algorithm which achieves a speedup of up to 2.5× on a many-core GPGPU device compared to a state-of-the-art multi-core node for the transport sweep, and up to 4× compared to the multi-core CPUs in the largest GPU enabled supercomputer; the first time this scale of speedup has been achieved for algorithms of this class. We then discuss ways to express our scheme in OpenMP 4.0 and demonstrate the performance on an Intel Knights Corner Xeon Phi compared to the original scheme.

*tom.deakin@bristol.ac.uk
†simonm@cs.bris.ac.uk
‡m.martineau@bristol.ac.uk
§Wayne.Gaudin@awe.co.uk

1 Introduction

Deterministic discrete ordinates transport models the movement and interaction of neutral particles through a mesh of media with varying properties. The interactions consist of fission, absorption and scattering; as the particles move within the material, they may collide with the material atoms and change direction (elastic) and/or energy (inelastic). They may also be absorbed by an atom which in turn may cause fission which results in the loss or gain of neutral particles respectively. It is the net balance of neutral particles that is governed by the Boltzmann transport equation [15].

Transport simulations consume 50–80% of time on the United States Department of Energy High Performance Computing systems [13]. It is therefore critical that transport codes perform well on both current and future architectures. Recently it has been shown that transport codes should scale well at large node counts [4, 12, 1]. However it is the on-node performance that will govern the speed at which the data for communication to neighbor nodes is made available. The current trend towards Exascale computing is to leverage the increased rate of floating point operations, memory bandwidth and performance per watt of many-core accelerator devices, such as GPUs or Intel's Xeon Phi.

Deterministic transport is a memory bandwidth bound problem with a computational intensity of $26/(8 \times 15) = 0.22$ floating point operations per byte of data loaded in the sweep kernel [23]. Therefore the speed at which the data can be loaded from memory will be the dominant factor of the computational solve.

We present a novel node-level implementation of a structured grid deterministic transport mini-app, which leverages the parallelism in the algorithm to enable better performance on many-core devices than previous approaches. This knowledge is used to design an implementation of the SNAP mini-app which can effectively exploit many-core technologies.

1.1 Contributions

The contributions of the work presented in this paper include:

1. We have developed techniques which express a higher degree of parallelism in the discrete ordinate transport algorithm than the state-of-the-art.

2. We have developed an OpenCL implementation of the deterministic discrete ordinates transport mini-app from Los Alamos National Laboratory: SNAP.

3. We present results comparing our improved scheme versus the state-of-the-art, exploiting OpenCL's cross-platform capability to show data from a diverse range of hardware. Our new scheme delivers a 44% performance increase over the state of the art on the same hardware.

4. We present an OpenMP 4.0 implementation of our improved scheme using the `target` directives to offload computation to an Intel Knights Corner Xeon Phi.

The rest of the paper is structured as follows. We begin with a short discussion of related work in Section 2. We present an introduction to deterministic transport and the SNAP mini-app in Section 3. In Section 4 we describe the hardware inspired methodology behind our design and details of the OpenCL implementation. The performance of our OpenCL implementation on a variety of accelerator devices is detailed in Section 5 and our version utilizing OpenMP 4.0 is shown in Section 6. Section 7 concludes and discusses avenues for future work.

2 Related Work

There has been some recent work on porting related wavefront applications to GPGPUs [18, 21, 20], however this was based on the much simpler computational workload of the Lower-Upper Gauss-Seidel solver (LU) benchmark [3]. The LU benchmark does not have as many problem dimensions as deterministic transport — only the three spatial dimensions are available for parallelization. For both the LU benchmark and transport, $O(N^2)$ cells of an N^3 grid can be processed in parallel. The number of cells in the plane, or *wavefront*, increases according to a sequence of triangle numbers in a cube shaped grid until a vertex is reached. As such, if only spatial parallelism is employed, a many-core device remains underutilized until the size of the wavefront saturates the device with useful compute.

A direct port of the SNAP mini-app using the CUDA framework by P. Wang et al. exists and uses a similar parallelization scheme to the original code [22]: energy groups are batched together and each cell is allocated to a thread-block with angles assigned to threads. Some device-specific tricks are employed to synchronize between wavefronts without host interaction [24], however this limits the number of concurrent cells to the number of thread-blocks on the device thus limiting the concurrency. Testing of this implementation on an NVIDIA K40 GPU show that this port only achieves similar performance to our benchmark CPU, even though the algorithm is memory bandwidth bound, and the GPU has 1.6× the memory bandwidth as measured using the STREAM benchmark on both.

The Denovo discrete ordinates transport code for reactor assembly calculations has been parallelized using CUDA [9]. Angles are mapped to threads, and the energy groups are fully decoupled giving a thread count equal to the product of angles and energy groups. Sub-blocks of cells are mapped to thread-blocks and KBA-style sweeps with the required synchronization are used within blocks. The paper states a speed up of 2× when compared to 8 of the 16 cores of their test system CPU nodes (Titan). Only half of the cores were used due to the lack of RAM on the host system for the size of MPI tasks used. The quoted speedup is in reality approximately parity speedup when compared against a

full node. By comparison, our implementation of the SNAP mini-app obtains approximately 4× speedup on an NVIDIA K20 GPU compared to a *full* 16 core node of Titan running the original SNAP mini-app (see Table 3 later for more details).

The performance of a shared memory node has also been investigated [17]. Intel Thread Building Block (TBB) tasks were used to maintain a task dependency graph of cells within a wavefront. Automatic compiler vectorization of angles within the octant was also used. This approach shows good strong scaling for within node performance, implying that *intra-node* spatial parallelism is important to exploit in addition to the inter-node spatial decomposition.

3 SNAP: Sn Application Proxy

The S_n (Discrete Ordinates) Application Proxy (SNAP) [25] is a proxy application from Los Alamos National Laboratory based on their code PARTISN. The computation in the mini-app mimics the 3D deterministic S_n transport equation. The transport equation (1) operates over seven dimensions: time, three-dimensional space, two angular (S_n discretization) and groups of energy. The net movement of neutral particles in a media of varying properties in these dimensions is called the *angular flux* ψ. Reduction over the angular dimension yields the *scalar flux* ϕ.

$$\left[\frac{1}{v}\frac{\partial}{\partial t} + \hat{\Omega}\cdot\vec{\nabla} + \sigma(\vec{r},E)\right]\psi(\vec{r},\hat{\Omega},E,t) = \tag{1a}$$

$$q_{\text{ex}}(\vec{r},\hat{\Omega},E,t) + \tag{1b}$$

$$\int dE' \int d\Omega' \sigma_{\text{s}}(\vec{r},E'\rightarrow E,\hat{\Omega}'\cdot\hat{\Omega})\psi(\vec{r},\hat{\Omega}',E',t) + \tag{1c}$$

$$\chi(E)\int dE' v\sigma(\vec{r},E')\int d\Omega'\psi(\vec{r},\hat{\Omega}',E',t) \tag{1d}$$

SNAP is designed to perform both time-dependent and time-independent deterministic transport solves. For the purposes of this paper we consider a time-dependent solution and therefore require storage of the angular flux which imposes a significant memory requirement, a characteristic which may be an issue for systems with discrete accelerators.

3.1 SNAP iteration structure

For every time-step we calculate the angular flux. Within the time-step loop, there is a double nested iteration structure for the numerical solve known as 'iteration on the scattering source' [15].

Let ψ^l give the angular flux value for iteration l. The iteration relationship is formed by replacing the angular flux ψ in (1a) by ψ^{l+1}, and in (1b), (1c) and (1d) by ψ^l. The right-hand side is termed the *scattering source* and the left-hand

```
for Time-steps do
    for Outer iterations do
        Update source: fission term
        for Inner iterations do
            Update source: external and scattering terms
            Perform a sweep
                                        ▷ i.e. invert collision-streaming operator
            Check inner convergence of ψ
        end for
        Check outer convergence ψ
    end for
end for
```

Figure 1: Transport equation iteration overview

side the *streaming-collision operator*. This allows an update to the guess of the angular flux in terms of previous results. An initial guess of the solution is taken, $\psi^0 = 0$. ψ^1 can then be calculated by inverting the streaming-collision operator. This inversion requires a *sweep* across the grid in all directions (originating at the four corners of a square in 2D, 8 points of a cube in 3D). This update-invert process, called *inner* iterations, repeats until the value of ψ converges. During each inner iteration, only the external source (1b) and group-to-group scattering (1c) terms are updated with the new value of the angular flux.

The converged value of ψ found from the inner iterations is then used as a new initial guess which is more accurate than the original. At this stage the fission term (1d) in the scattering source is updated with the converged ψ from the inners. Note that the scattering source depends on ψ for the fission and scattering terms. Convergence at this level is checked, forming *outer* iterations which take into account all source terms. The whole procedure can be summarized by the algorithm in Fig. 1.

During the calculation, nonphysical negative values of the angular flux may occur due to the numerical scheme. It is important to correct these values in a process called *negative flux fix-up*. This involves setting negative fluxes to zero and recalculating the central flux value. The edge flux values are then recalculated to check they do not turn negative as a result of the recalculated central flux. This process is repeated until all the values are positive. This procedure cannot be post processed due to the data dependency.

3.2 Data dependency of the sweep

The data dependency illustrated in Fig. 2 given by the finite difference spatial discretization forms a *wavefront* (hyperplane) consisting of cells available to compute given availability of the values already calculated. This is illustrated in Fig. 3, where cells of the same color are in the same wavefront. If two cells share a face they are *neighbors*. For a given cell, neighboring cells in the opposite

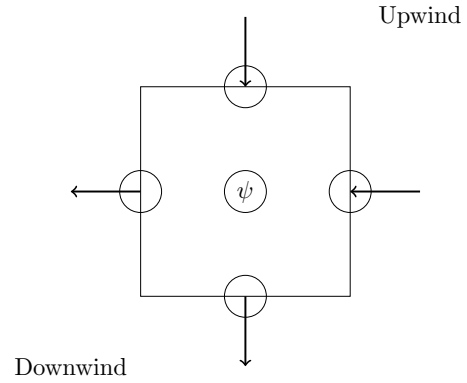

Figure 2: Data dependency of the sweep algorithm

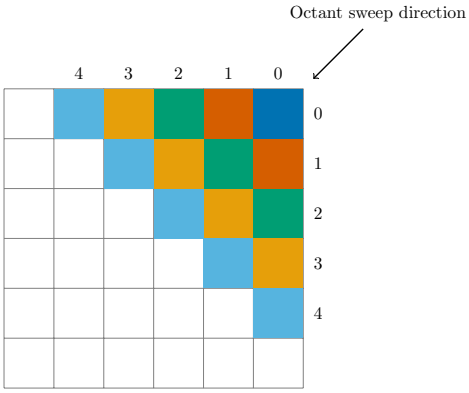

Figure 3: Wavefront sweep across a 2D grid

direction to the sweep are called *upwind* neighbors. Neighboring cells which rely on a given cell's value are called *downwind* neighbors. A value in a cell can only be calculated if all of its upwind neighbors have already been calculated.

Cells of equal co-ordinate sum are in the same wavefront. For example, the cell (i, j, k) belongs to wavefront $i + j + k = c$. In a cube with sides of length N, there are $3N - 2$ wavefronts. In general the maximum number of cells in the wavefront is the product of the two largest spatial dimensions. For the cube with sides of length N wavefronts contain $O(N^2)$ cells with the upper bound being obtained once. Note that this wavefront pattern gives rise to potential for concurrency in the spatial dimensions of cells within a wavefront.

3.3 The original parallelization scheme

SNAP is written in Fortran 90/95 and parallelized with the OpenMP [8] and MPI [10] APIs. A 2D decomposition of the 3D mesh according to the KBA

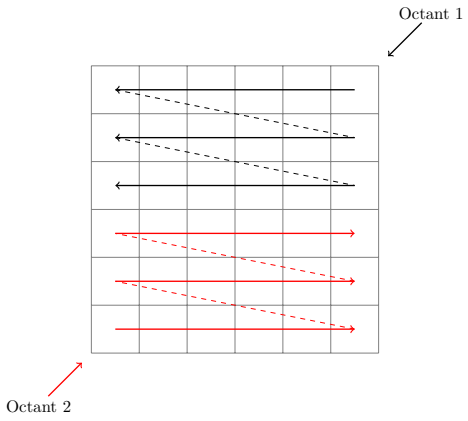

Figure 4: Serial sweep of the grid in original code

scheme is used [14]. The modification of decoupling angles and energy groups is employed so that angles within an octant are considered as a single vectorized batch rather than overlapped in multiple sweeps of individual angles [5].

SNAP uses OpenMP threads to run in parallel over energy groups and automatic compiler vectorization over angles within an octant (resulting in SSE/AVX SIMD instructions depending on the target CPU architecture). The energy groups are divided into bins and each OpenMP thread is allocated one bin.

The spatial domain is split using MPI, but the sweeps within each MPI task are conducted in serial, as shown in Fig. 4. Each sweep traverses the full grid in an octant direction and computes the angular flux in every cell for all energy groups for all angles within the octant. This is then repeated for each octant in turn.

4 Concurrency For Many-core

We present a new way to express the maximum potential for parallelism in the transport sweep algorithm. Our new approach exposes more exploitable parallelism for many-core technologies.

Applications which exhibit a wavefront sweep across the grid naturally allow for parallelization over the cells within the wavefront, as shown in Fig. 5. Note that a wavefront must finish before the next wavefront can begin due to the data dependency discussed in Section 3.2. This is an improvement on the original code which exhibits a serial sweep, as in Fig. 4, without the use of nested OpenMP threads for spatial concurrency. In testing we found that this nested behavior performed poorly increasing the single sweep run time by about 50% on a dual socket Xeon CPU; note that we spawn 16 threads in total (one per core) to prevent over-subscription of threads to cores which reduces the performance

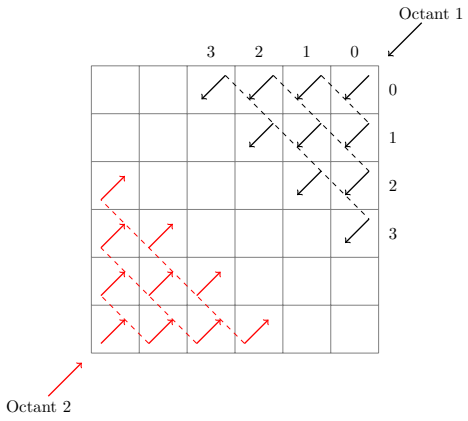

Figure 5: Parallel sweep of the grid

further.

We build on this natural spatial parallelism for all wavefront algorithms by using knowledge of the S_n transport algorithm itself. Additionally S_n transport allows for parallelization of angles within an octant (in general this is true in a structured grid, however this might change based on types of symmetric boundary conditions) and of energy groups (at the cost of potentially increased iteration counts [18]). Pipelining of the octants is also possible for certain boundary conditions. For example, in the case of only vacuum boundaries, all eight octants may be started simultaneously. We do not take this final pipelining into account to allow for reflective boundary conditions in the future where it would become unavailable.

The intuition behind our approach to improve upon the natural spatial parallelism with the details of deterministic transport is to leverage as many dimensions of parallelism as possible in order for the sweep to perform well on a many-core device and to exploit coalesced memory reads and writes as much as possible. The level of parallelism in specific transport problems may be reduced due to angular dependency, or complex un-structured grids. It is important that in this simple structured case of maximum potential concurrency that the parallelism is fully exploited.

We therefore take a single angular flux calculation for a given cell (i, j, k), angle a in octant o and energy group g, $\psi(a, g, i, j, k, o)$ as our unit of work. Here a cell consists of the angles in the octant organized in energy groups.

Note that the original SNAP threads over the energy groups. The coupling of energy groups in the original code is flexible as the binning depends on the available threads, and so energy groups can be considered fully decoupled.

At any given point in the wavefront sweep we have

$$N_{\text{angles in octant}} \times N_{\text{energy}} \tag{2}$$

units of work per cell. We also have at most

$$O\left(\frac{N_x N_y N_z}{\max\{N_x, N_y, N_z\}}\right) \tag{3}$$

concurrent cells in each wavefront; the product of the smallest two dimensions. We batch all of this work together for each wavefront plane for concurrent execution on a many-core device. This means we launch work proportional to the product of (2) and (3).

With this approach it is possible to provide enough work for a many-core device even for the first wavefront consisting of a single cell. Take an example problem with 50 energy groups and discrete ordinates set S_{32} which results in 136 angles per octant. This means we have $136 \times 50 = 6,800$ units of work per cell. The NVIDIA K40 GPU has $2,880$ processing elements (CUDA cores), resulting in even the first cell/wavefront nearly saturating the compute device with enough work. Cells within the wavefront are also included in the available work, and so it is clear that as we progress across the grid the device is given sufficient work to schedule for good performance. This is in contrast to the previous approaches, which do not exploit all dimensions of available parallelism.

Another approach could be to construct a dependency graph where each graph node is a cell. We specify a dependency for each cell on its upwind neighbors. We use the same unit of work as above for concurrent work within the cell. If a particular cell took longer to compute, due to negative flux fix-up for example, the other cells within the wavefront could continue marching through the grid without waiting for the expensive fix-up to occur. While this approach might at first seem attractive, we discuss its drawbacks in Section 4.2.2.

In order to calculate the scalar flux from the angular flux we must perform a reduction over all angles within each cell and energy group. The reductions are only within a cell and energy group and so there are many independent reductions, rather than one large reduction over the whole grid to a single number as is typical for other Boltzmann equations. Remember that the scalar flux is defined in terms of three spatial and one energy group dimension, per time-step. The reduction must occur for every inner as the result is required for the source updates and convergence tests.

We use the OpenCL framework to implement a port of the SNAP mini-app to test this parallelization scheme. OpenCL gives us the ability to express all the elements in our scheme, including out-of-order execution models and task dependency graphs for the task-based approach suggested. Additionally, OpenCL allows for cross-platform testing of this approach for a variety of multi- and many-core devices.

4.1 OpenCL

OpenCL [19] is a parallel programming framework for heterogeneous platforms. The basic model consists of a host and one or more multi-/many-core devices such as a GPU, CPU, FPGA, etc. The host CPU and device have separate

memory regions and the location of data in this memory hierarchy must be explicitly managed by the programmer.

The unit of work for compute, called work-items, are defined in a kernel. The work-items are mapped onto the vector-lanes of the device architecture. In general these lanes are called processing elements, and operate in lock-step groups called compute units.

An N-Dimensional Range (NDRange) defines the problem size (number of work-items) when the kernel is launched, and, if required, so too is the work-group size — a work-group is a collection of work-items. Synchronization of work-items is only possible within a work-group on the device and between work-groups via the host at the end of kernels. In a CPU, each work-group would be assigned to a core, and the work-items a vector lane within that core. The kernel is enqueued on (offloaded to) the device as a whole and the OpenCL runtime schedules the computation of work-groups onto compute units. The host controls the compute offload and memory transfer via a command queue.

OpenCL therefore allows the programmer to prescribe a very fine grained parallel programming notation of concurrent work. It is through OpenCL that we can express the parallel workload to leverage performance from many-core architectures for deterministic transport.

4.2 Implementation details

SNAP itself is written in Fortran 90/95, and our new OpenCL implementation has been written in C. The iteration loops and reductions have also been ported to C, along with the source update and initialization routines. This means that the data can remain resident on the device for the full simulation, avoiding expensive host to device data transfers in the innermost loops.

Initializing data in this way has the benefit that we can choose the data layout to match the parallelization scheme and initialize the memory directly on the device. It should be noted that the memory layout chosen differs from the original Fortran, which defines the layout of the angular flux as angle/x/y/z/octant/energy group; angles have unit stride, and energy groups have the largest stride. In our new implementation we adopt the layout: angle/energy group/x/y/z/octant. This results in angles having unit stride, and octants with the largest stride. Our new layout means that we can achieve coalesced memory reads in our inner most vector unit over angles/energy groups.

The OpenCL implementation is added into the original version of the code, meaning both versions run in sequence, so that the results can be compared element-wise to ensure both implementations obtain the same results. At present the implementation is for a single device (one MPI task).

4.2.1 Octant sweep implementation

We assign each angular flux calculation for a given cell (the unit of work) to a work-item. The kernel defines the number of work-items launched on the device and we consider this in two-dimensions. The first describes the number

of angle/energy group units in a single cell; the cross product of angles and energy groups. The second describes the number of cells within the wavefront. The mapping of cell index to wavefront number is computed in advance for one sweep direction and offsets are calculated in the kernel to access the correct cell index for the current sweep direction. We do not specify a work-group size for this kernel.

4.2.2 Cell dependency graph version

A full dependency graph of cells was built using OpenCL events and an out-of-order command queue. We define the kernel as all of the angular flux calculations for a single cell. We specify a dependency on each cell/kernel to the upwind cells, hence constructing a dependency graph in the OpenCL runtime.

We discovered that this DAG-like scheme resulted in lower performance, due to less work being scheduled at a time (less than the full wavefront that would be possible), coupled with an increase in the number of kernel enqueues and a high number of OpenCL events for the run-time to track and manage (one event per cell). If the performance of handling events and out-of-order queues in the OpenCL runtime were improved, perhaps with the advent of drivers optimized with HSA [11], then it would be possible to increase the available parallelism to *any cell* with satisfied upwind dependency, rather than just those cells in a wavefront. Until run-times have significantly improved to support these finer-grained, asynchronous styles of parallelism, these approaches will not perform as well as the simpler, batch parallel approach.

4.2.3 Reduction implementation

In the OpenCL implementation, the reduction is designed with angles within a single cell and a single energy group assigned to work-items within one work-group. The reduction of angles over octants is performed in serial by each work-item summing the eight values and storing the value into local memory, a special part of the OpenCL memory hierarchy designed for fast access by work-items in the same work-group. This kernel is run as a post-processing step after all the octant sweeps have completed, so the mapping of work-group number to cell index is straightforward.

The reduction is then a commutative tree-based reduction within a work-group in local memory [2]. Note that this requires the work-group to be a power-of-two sized, and the number of angles in an octant may not be. We therefore set the work-group size to the *closest* power to two greater than or equal to the number of angles, and perform any extra angles in a serial reduction step similar to the summation of the octants.

This reduction scheme performs poorly on a CPU as it does not vectorize, so a simpler reduction was implemented for CPUs which does auto-vectorize. We assign a work-item to each cell, each performing the reduction in serial. Threads (work-groups) then have good data locality within their cache and have a large stride between them to avoid cache-thrashing. The memory access of this

reduction on the CPU will not be coalesced, and so vector gather instructions will be issued. However it performs well enough for our focus on the sweep.

The reduction must occur every inner iteration as the resulting scalar flux is used for both the inner and outer convergence tests and source updates. A similar reduction is implemented for the scalar flux moments reduction array present in SNAP. These two reductions are considered as a single unit in all timing results.

The tree-based reduction scheme may be combined into the sweep kernel directly by ensuring the same work-group conditions, avoiding loading the (large) angular flux twice.

5 OpenCL Results

Timing results for the OpenCL implementation of our improved version of SNAP are presented for both the eight octant sweep and the scalar flux reduction parts of our algorithm, across a range of devices. This performance is compared against a baseline result from a high-end 32-core dual-socket CPU system. We also show the memory bandwidth achieved by the sweep kernel. Memory bandwidth is the preferred performance metric for this memory bandwidth bound problem.

Additionally, a longer simulation is tested demonstrating the total time to solution of the OpenCL implementation, which includes the source updates as well as the sweep itself.

Note that there has been no manual performance tuning of the simulation for the different architectures, i.e. exactly the same code is executed on all devices, in order to evaluate the performance portability of the OpenCL implementation. In particular, the default run-time decided local work-group size was used for the sweep kernel. For the reduction we enforce a 1D power-of-two sized work-group to ensure the reduction works correctly.

5.1 Experimental platforms

In order to test the effectiveness of this parallelization scheme it is necessary to test on a wide variety of many-core devices. For the ones used in this test, see Table 1. STREAM [16] benchmark runs on the CPUs were performed using an appropriate array size for the device cache according to the benchmark rules, and results for the GPUs were gained by using GPU-STREAM [6] The systems the devices are installed within are detailed in Table 2. The chosen devices represent the current state-of-the-art of available HPC accelerator devices.

5.2 Sweep performance

The performance of the sweep was tested independently of the iteration structure. The sweep routine typically dominates the runtime of transport solvers

Table 1: Specifications of target devices used for testing

Platform	RAM (GBytes)	Memory B/W (GBytes/s)	STREAM benchmark (GBytes/s)	D.P. TFLOP/s
Intel Xeon E5-2670 ×2	64	52.2 × 2 = 104.4	76	0.33
Intel Xeon E5-2698 v3 ×2	128	68 × 2 = 136	118	1.18
AMD Opteron Processor 6274	32	51.2	32	0.14
Intel Xeon Phi Co-processor SE10P	8	320	137	1.07
NVIDIA Tesla K40	12.288	288 (ECC off)	194 (ECC on)	1.43
NVIDIA Tesla K20c	5.12	208 (ECC off)	152 (ECC on)	1.17
AMD FirePro S9150	16	320 (ECC off)	273 (ECC off)	2.53

Table 2: Benchmark system specifications

	Test system 1	**Test system 2**	**Test system 3**
Device(s)	None	Xeon Phi	NVIDIA and AMD GPUs
CPU Model	Intel Xeon E5-2698 v3 (×2)	Intel Xeon E5-2670 (×2)	Intel Core i5-3550 (×1)
CPU Specs	Haswell, 16 cores (×2), 2.3 GHz	8 cores (×2), 2.6 GHz	4 cores (×1), 3.3 GHz
Memory	128 GB	64 GB	16/32 GB
OS	SUSE Linux Enterprise Server 11	Scientific Linux release 6.4	Ubuntu 14.04.1 LTS
Compiler/Driver	Cray 8.4.1	Intel 15.0.2 / NVIDIA 331.75	GNU 4.8.2 / NVIDIA 49.12 and AMD 1573.4

Table 3: Solve time for a single sweep and reductions

Code	**Device**	**Time (s)**
Original	E5-2698 v3	0.37
	E5-2670	0.50
	Xeon Phi	0.68
	Opteron 6274	1.24
P. Wang et al.	K40	0.32
	K20c	0.44
OpenCL	K40	0.24
	K20c	0.30
	S9150	0.15
	E5-2670	0.68
	Xeon Phi	2.68

and so the performance of the solve is crucial. The following problem size was used:

- Grid size: $16 \times 16 \times 16$
- 50 energy groups
- S_{32}: 136 angles per octant
- Expansion order of 2 in the scalar flux moments

This results in a memory requirement of 3.6 GBytes for both copies of the angular flux. These parameters have been chosen to reflect typical problem characteristics (per node), while fitting within the memories of all the devices under test.

We time eight complete octant sweeps across the grid — one inner iteration without the source update. Remember that the octants are computed *in turn* and are *not* pipelined. Table 3 shows the sweep timing including the reduction of the angular flux to the scalar flux and scalar flux moments.

Fig. 6 shows the speedup obtained for a single device across a variety of architectures compared with the original code running on 2×16 core Haswell CPUs in Test system 1 with 32 OpenMP threads (one per core). We were not able to run the OpenCL version of the code on the CPUs for a comparison because of the lack of the Intel OpenCL runtime on Test system 1. The NVIDIA K40 device has approximately twice the memory bandwidth (and similar D.P. FLOP/s) to the baseline CPUs in Test system 1, and achieves a speedup of 1.52×. The AMD FirePro S9150 has around 2.8× times the memory bandwidth of the CPUs and a comparable speedup of 2.8× is obtained. To the best of our

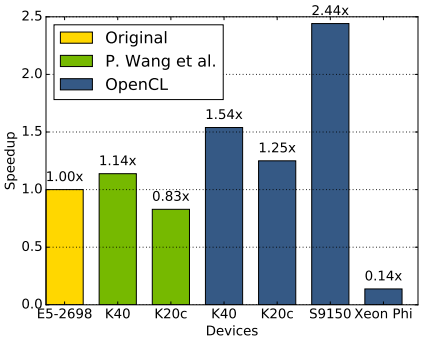

Figure 6: Speedup of Single sweep performance from the reference timing

knowledge, this is by far the largest speedup ever reported for SNAP or similar transport codes, when comparing against a highly optimized host code running on a leading edge multi-core system (32-cores of Haswell in this instance).

The sweep (excluding the reduction) in the OpenCL version on the E5-2670 CPUs runs around 15% slower than the original SNAP Fortran code. After profiling the OpenCL code on the CPU, it became apparent that there was a high cache miss rate, due to the data being in the wrong NUMA region. The Intel OpenCL runtime is implemented with TBB and work-groups are scheduled on sockets in a non-deterministic way meaning that data may be allocated in the wrong NUMA region.

The lack of performance on the Xeon Phi (Knights Corner) with our version is likely due to the runtime choosing poor work-group sizes and other OpenCL support issues on this device; the original implementation performs much better. The number of available threads in the original implementation is limited to the number of energy groups in the problem, and so the original implementation does not use all of the 60 available threads. Many-core architectures in general are increasing in the number of threads (specifically the maximum number of parallel execution units) and an algorithmic limit such as this may not be future-proof. The Xeon Phi appears to have a policy to generate around 1000 work-groups from the NDRange specified. This results in only a few work-items per work-group for the first few wavefronts. For larger wavefronts the numbers chosen by the run-time itself are not multiples of eight which is incongruous with the hardware vector width of up to 512 bits. See Section 6.3 for an implementation of the improved concurrency scheme in OpenMP 4.0.

5.2.1 Larger grid size

The grid size chosen for the results presented in Sec. 5.2 allows a wide range of GPUs to be tested, however may be considered small in comparison to physically relevant problems. We are limited by the memory capacity of the GPUs, however several devices in our collection have larger memory sizes. We therefore

Table 4: Single sweep and reduction timings for 24^3 grid

Code	**Device**	**Time (s)**	**Speedup**
Original	E5-2698 v3	1.25	1.0
	E5-2670	1.60	-
P. Wang et al.	K40	1.25	1.0
	K20c	2.35	0.53
OpenCL	S9150	0.46	2.72

also tried using a larger grid size of $24 \times 24 \times 24$, with the same number of energy groups and angles as above, for the devices in Table 1 that had enough enough memory to meet the 12.1 GBytes required for this problem size.

The timings for the various implementations for this larger grid size can be compared in Table 4. The AMD FirePro S9150 completed a single sweep and scalar flux reduction in 0.46s. Test system 1 running the original Fortran code completed the sweep in 1.25s. This leads to a speedup of 2.72x of device to multi-core node.

The P. Wang et al. version utilizes the shared virtual memory address space between device and host which is available in the version of CUDA on Test system 3, and we were able to collect timings for this implementation. The Xeon Phi does not have enough memory for this problem size. The NVIDIA GPUs do not have enough memory for our *memory-resident* scheme in OpenCL; using Shared Virtual Memory to mitigate this as done by P. Wang et al. was not supported by the NVIDIA OpenCL driver.

A typical transport simulation could be set up with around $O(10^5)$ cells per node. The memory footprint can be calculated by multiplying the number of cells by the number of angles ($O(10^3)$) and energy groups ($O(10)$). This results in $O(10^9)$ double precision values per node; a footprint of $O(10)$ GBytes for each copy of the angular flux (two copies are required). It is clear that for currently available GPUs, the amount of on-board RAM is lacking for production scale grid sizes, where it will not be possible to store both copies of the complete angular flux on the device along with the other data required.

The negative flux fix-up code within the kernel means we have private memory intensive kernels resulting in large register usage. The occupancy of the sweep kernel rises from 30% to 70% on the S9150 when the fix-up code is removed, demonstrating that the fix-up code was indeed responsible for much of the register pressure and resulting performance degradation. In the first inner iteration for this problem size, the negative flux fix-up is never called in the original code, and so the results remain the same even if we remove the fix-up for this iteration. Doing so results in a 10ms reduction in the stated runtime. Because this optimization does not work in the general case where fix-ups would be required, we left fix-up enabled in the version of the code used in the rest of this paper.

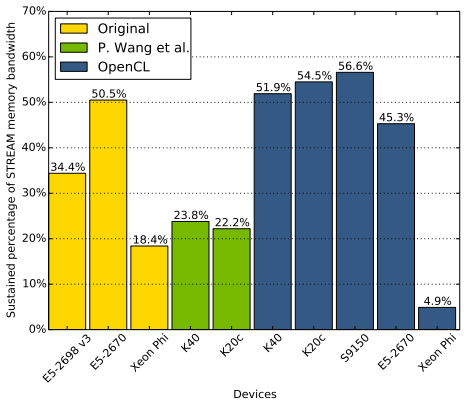

Figure 7: Sustained percentage of STREAM memory bandwidth for a single sweep and reduction

5.2.2 Memory bandwidth calculations

We can calculate the number of memory reads and writes of each sweep kernel in order to estimate the sustained memory bandwidth achieved. In the sweep operation there is reuse of data shared between work-items such as various coefficients. We assume a perfect cache model where each byte of data needs only to be read once per kernel.

We define the number of reads and writes in a kernel (which is a wavefront) as

$$aN_{\text{cells}} + b$$

where a is the number of reads and writes which depend on the cell index, b is the number of reads and writes which are shared between cells in the wavefront, and N_{cells} is the number of cells in the wavefront.

By counting data reads and writes in the kernel source code we observe $a = 61,400$ and $b = 866$, all to double precision (8 byte) data. By going through the number of cells in each wavefront in each of the eight octant sweeps we calculate the sweep operation reads and writes total ∼15 GBytes. We assume this is the same value for the original Fortran implementation of SNAP. Using the CodeXL profiler on the S9150 runs, we see 13.03 GBytes of data movement, also taking into account cache effects.

The achieved sustained bandwidth b can be calculated by dividing the total memory read/written by the total runtime. The percentage of achieved bandwidth for the device as measured by the STREAM benchmark (with ECC on as appropriate) β can be calculated as $b/\beta \times 100\%$. The sustained percentage of STREAM bandwidth obtained for the sweeps (excluding the reduction time in contrary to Table 3) for the various devices is shown in Fig. 7.

Our OpenCL implementation achieves a similar fraction of memory bandwidth on the E5-2670 CPU to the original version of the code. The slight drop

is likely to do with the same issues with NUMA cache misses as discussed above.

The GPUs demonstrate that we are leveraging a greater percentage of the available memory bandwidth than the original implementation, and in particular our scheme is performing considerably better than the P. Wang et al. scheme. Both the original and CUDA versions implement the same parallelism scheme and so we can conclude that the scheme presented in this paper leverages more of the available memory bandwidth than the original scheme. We also observe that our OpenCL implementation exhibits good performance portability, achieving very similar fractions of STREAM memory bandwidth across multiple GPUs from NVIDIA and AMD, and even a similar fraction on the CPU.

5.3 Performance of solve until convergence

In addition to looking at the performance of the sweep routine alone, we also consider the total application performance where the other, simple, routines are ported to the many-core device. This work does not include multi-node runs which would introduce the overhead of MPI communications and so may not be totally representative of a large scale simulation. This experiment will check that offloading the sweep routine to the device in the context of the other parts of the application is appropriate.

We take the above problem and run 14 time-steps, allowing up to five inner iterations per outer iteration, and up to 100 outer iterations per time-step. This allows us to see the performance of converging multiple timesteps while requiring the simulation to complete in reasonable time. The inner and outer source updates and recalculation of the denominator term and cross section expansion are performed directly on the OpenCL device allowing the data to remain resident on the device throughout the run as far as possible. The convergence test is completed by copying the scalar flux from the device to the host and checking convergence in serial on the host. This is to remove the need for an additional reduction on the device. The size of the data copied is approximately 1.5 MB for this problem size and the timing of both the copy and the convergence test is insignificant to the runtime. Table 5 shows the timings for this experiment.

The inner convergence test in the original SNAP code compares convergence of the scalar flux for batches of energy groups. The next inner iteration runs for the *un-converged* groups, and due to a reduced workload results in a faster time. Removing energy groups in this manner would not be possible if convergence of the *angular* flux was required. Note too that removing energy groups from the pool of available work will reduce the potential concurrency. The total work load for the OpenCL implementation remains constant for every inner iteration: all energy groups, all angles, all cells. As the convergence test is on the reduced scalar flux, the angular flux will not have converged to within the same tolerance and so the value for the angular flux will differ as some energy groups will be further converged than others, however the scalar flux will remain converged. If the original code is updated to always perform an inner iteration for all energy groups, as would be necessary on most many-core devices, our implementation achieves speedups to solution consistent with those for a single sweep.

Table 5: Solve time for 14 time-steps

Code	**Device**	**Time (s)**	**Speedup**
Original	E5-2698 v3	60.0	1.0
	E5-2670	87.5	-
	Xeon Phi	159.0	-
P. Wang et al.	K40	65.1	0.92
	K20c	89.1	0.67
OpenCL	K40	57.5	1.04
	K20c	56.8	1.06
	S9150	37.5	1.60
	E5-2670	155.38	-
	Xeon Phi	645.23	-

Table 6: Solve time for 14 time-steps calculating all energy groups for every inner iteration

Code	**Device**	**Time (s)**	**Speedup**
Original	E5-2698 v3	91.3	1.0
OpenCL	K40	57.5	1.59
(same timings	K20c	56.8	1.61
as Table 5)	S9150	37.5	2.43

6 An OpenMP 4.0 Approach

As shown in Section 5, we observe that the OpenCL implementation does not perform well on the Xeon Phi. The approach however exposes more parallelism than the original implementation and so should be able to utilize all the cores in the device. The original scheme only allows for one thread per energy group — this yields tens of threads for a device which is designed to perform well with hundreds. Whilst the original scheme did use nested OpenMP threads to increase the thread count, this did not produce any improvement in performance in testing. We now present an OpenMP 4.0 implementation that mimics our concurrency scheme as implemented in OpenCL, and show the performance on a Knights Corner Xeon Phi co-processor.

6.1 OpenMP

OpenMP [8] is a parallel programming API which follows the fork-join model. The body of work inside a loop is executed once for every iteration of the loop. The iterations, if independent, can be distributed across multiple threads in a shared memory system. The shared memory allows data to be accessed by all the threads and so no explicit communication is required between them. OpenMP describes a way through the use of `#pragma` compiler directives for the programmer to state that threads should be spawned and that the work inside the loop construct be spread over many threads. This concept is also extended to define independent tasks which can be allocated to available threads. It is usual to assign a thread to a single core.

OpenMP 4.0 extends this model to facilitate *offloading* work to an attached compute device. The work to be offloaded to the device is annotated with the `target` directive. The code written inside this body is then run on the device and the host waits until control is returned to it at the end of the `target` region. The usual work-sharing constructs are applied to loops within the `target` region to split the work across the computational units of the device.

Memory management is by default handled automatically by the implementation according to the memory model: the device inherits all the memory just as cores in a 'regular' CPU share memory. Clauses to the `target` directive can help reduce the number of data transfers by specifying whether the device access in the region is read-only, write-only or read-write.

6.2 Implementation variations

OpenMP 4.0 is used to implement our improved concurrency scheme as described in Section 4. However given the differences in the OpenMP 4.0 and OpenCL programming models, in particular differences in the host side control of the device, some implementation choices are available in expressing the parallelism in this API.

Note that we minimize the movement of memory between device and host by using a `target data` directive to initialize the device data environment so

that the data remains resident on the device for the duration of the simulation and is not automatically copied (mapped) back to the host at the end of each `target` region. This is much the same as our OpenCL implementation whereby buffers are created and initialized on the device and are explicitly copied back only as required.

The locations of the `target` regions are critical to the performance of the OpenMP implementation of our concurrent scheme, as we will show in Section 6.3. The natural approach is to replace the OpenCL kernel enqueues with loops, and place the OpenMP directive `#pragma omp target parallel for` as appropriate, mirroring our OpenCL structure, to offload the routine to the device and share the work of the loop in the same manner.

This approach results in the runtime implicitly performing a currently unavoidable host-device synchronization at the end of each of the parallel regions. Whilst no memory is copied, this synchronization does not occur in the OpenCL version as kernel enqueues are a non-blocking operation. OpenMP 4.0 does allow the `target` regions to be launched as tasks, and in the forthcoming OpenMP 4.5 specification `target` regions can be made tasks implicitly via clauses.

It is possible to place the equivalent of all the OpenCL 'kernel' code within one OpenMP 4.0 `target` region, resulting in a `target` region outside the time-step loop. This approach is similar to running in 'native mode' on the Xeon Phi, where the device is run without a host. Note however that it will not be possible to call MPI, for example, from within a `target` region, a severe limitation for real transport codes which run across many nodes to achieve the required performance.

A compromise between the two approaches (many target regions vs. one target region) is possible. Where a kernel is called multiple times via a loop construct, a `target` region can be placed before this loop, reducing the number of `target` regions, and consequently also reducing synchronization points. The work-sharing directive is left until the kernel loop itself, meaning that the device is still required to perform some single thread serial control flow code. In our implementation, by placing the target region outside the octant loop, all the sweep kernels can be enqueued without host interaction. The data dependency of the plane ordering is maintained due to the device-side synchronization guaranteed by the OpenMP 4.0 specification and the end of parallel regions. Because this approach uses multiple target regions, it is still possible to use it in combination with MPI for multi-node implementations.

Note that in a multi-node implementation, the code within a `target` region will not be able to make calls to the inter-node communication library such as MPI; this is reserved for the host to perform much in the same was as in OpenCL.

In the future, large systems constructed from many Xeon Phis operating in a self-hosted manner; i.e. there will be no controlling CPU with the Xeon Phi attached as a co-processor. Therefore the OpenMP 4.0 offload programming model using `target` regions will not be the method of programming these devices and the code will need to be developed as a native application similar to the approach taken by our 'single target' implementation.

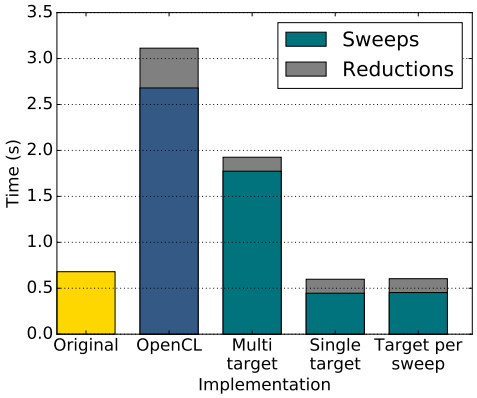

Figure 8: SNAP timings on a Knights Corner Xeon Phi (lower is better)

6.3 Results

We implement all three approaches discussed in the previous section in OpenMP 4.0 targeting a Xeon Phi: many target regions, one target region, and reduced target regions. We compare their performance to the original implementation of SNAP, and to our OpenCL version, in Figure 8. The reduction timings are highlighted on the bars separately for the OpenCL and OpenMP 4.0 versions as this part of the code runs after the sweeps, as discussed in Section 4. Because of the concurrency scheme implemented in the original version of SNAP, the reduction can be performed as part of the sweep routine and so timings are shown for the combined sweep and reduction in this case. It is instructive however to separate the sweep and scalar flux reductions in this manner for the OpenMP and OpenCL implementations. We observe that the reduction timings are consistent across the different OpenMP 4.0 implementations and the performance difference between the three OpenMP 4.0 versions is solely attributable to changes to the `target` regions surrounding the sweep routine.

All of the OpenMP 4.0 variations perform better than our OpenCL implementation on the Xeon Phi, even though they all use the same parallelism improved scheme. We previously explained that much of this difference is attributed to the sub-optimal, NUMA oblivious, TBB-based implementation of the OpenCL run-time on the Xeon Phi.

The bar labelled 'multi-target' shows the large penalty for synchronizing with the host after every kernel call. We can improve this by moving all the code into a singe target region (labelled 'single target') however this approach will not be performance portable when OpenMP 4.x compilers targeting GPUs are released due to the requirement to execute control code on the device. The compromise discussed above is shown as 'target per sweep' where there is a single `target` for the sweep, but synchronization occurs between the host and device before the reduction and the other offloaded routines, such as source updates. Note that the fastest sweep time occurs when there is only a single

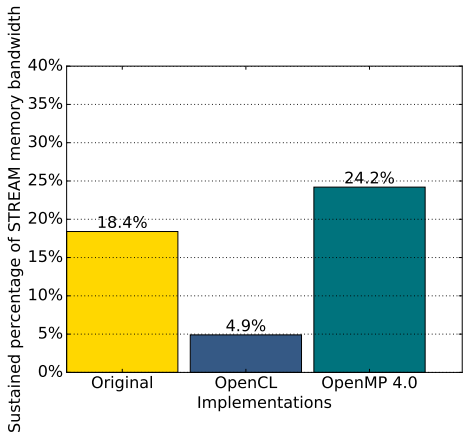

Figure 9: Achieved memory bandwidth of the sweep on Knights Corner Xeon Phi

`target` region, but limiting the target regions to just a few cases, one per inner for the sweep rather than one per plane, still provides close to best performance while maintaining the host-device relationship in the offload model.

The KNC-based Xeon Phi in our experiments has 60 compute cores (plus one to run the hosted operating system) and each is equipped to handle four hardware threads. This gives us a total of 240 available hardware threads. The original implementation only allows for one thread per energy group, resulting in tens of threads. Both our OpenCL and OpenMP 4.0 implementations utilize all 240 threads available without the need to resort to nested parallelism and the associated overheads.

As the transport sweep is memory bandwidth bound, the achieved memory bandwidth of an implementation demonstrates the effective use of the hardware by the implementation. As shown in Fig. 9 where we have taken the 'target per sweep' version as our OpenMP 4.0 implementation, the percentage of achievable bandwidth is higher with our concurrency scheme. Again note that the concurrency is *the same* in the OpenCL and OpenMP 4.0 implementations and so the performance difference must lie in the implementation of the different APIs.

From these results we conclude that our new approach to concurrency, as detailed in Section 4, does indeed enable the use of many-core devices to increase the performance of the transport sweep over the original concurrency scheme.

6.3.1 Performance portability of our approach

A native OpenMP implementation which uses the same parallelism scheme whereby the compiler pragmas are simply `parallel for` instead of `target teams distribute` can be simply generated. The use of `collapse` clauses enables the same concurrency to be expressed.

The single sweep and reduction time on the E5-2698 Haswell CPU for this native approach with the new implementation is 1.2 seconds. This is much slower than the original implementation which achieves 0.37 seconds. The concurrency scheme presented here however has memory access patterns that are no longer be stride one. Additionally, the data layout puts energy groups close together, and so for wavefront planes with small numbers of cells the threads are going to be updating elements of the angular flux array which reside close in memory, which will cause an element of cache thrashing.

A performance portable implementation of the transport sweep is therefore a major challenge as difference architectures require different loop and data orderings, and different concurrency schemes. The original scheme is optimized well for CPU architectures, and more parallelism is required for running on GPUs. However we can see that the way this parallelism is found is not appropriate for cache based CPU architectures.

7 Conclusion

Deterministic transport is a memory bandwidth bound problem, and whilst many-core devices can offer increased memory bandwidth, they require highly parallel workloads to be effective. Through unlocking more of the parallelism inherent in S_n transport it has been possible to achieve fast sweeps on many-core devices. Indeed we have achieved up to 2.5× speedup when compared to a state-of-the-art multi-core node for the 16^3 problem size, and around 2.8× on a 24^3 problem. This speedup is in line with the increase in memory bandwidth of the device and is significantly better than previous results reported in the literature.

Our approach is also shown to be sound for the Xeon Phi style many-core architecture as demonstrated by the OpenMP 4.0 implementation of our scheme.

If SNAP is a good proxy application of a production code such as PARTISN, then the authors believe that it would be straightforward to integrate the parallelization scheme presented here into the production code. The main concern would be using the presented memory layout *throughout* the production code to avoid wasteful memory transposes.

Our new implementation harnessed both the natural spatial parallelism of cells within a wavefront, as well as within cell parallelism in the transport equation. Taking advantage of the potential to parallelize over energy groups as well as angles within an octant was important in increasing the amount of available work to saturate the available processing elements on the device.

7.1 Future work

We intend to investigate schemes to leverage the heterogeneous *memory* hierarchy of nodes with $O(100)$ GBytes of host DRAM, coupled with $O(10)$ GBytes of high bandwidth device memory, to enable real-world problem sizes to be solved

much more quickly than has previously been possible. This approach is preferable to using a many more nodes of a much smaller memory size, just to be able to store the working data set.

We have also investigated spatial domain decomposition schemes via MPI [7]. We showed that our on-node implementation is performant at scale using existing MPI schedules.

Acknowledgments

This work has been financially supported by A.W.E. The authors would like to thank Randal Baker and Joe Zerr at Los Alamos National Laboratory; Richard Smedley-Stevenson and Iain Miller from A.W.E; the University of Bristol Intel Parallel Computing Center (IPCC), and the High Performance Computing Group; Intel for provision of the Xeon Phi; Thomas Bradley from NVIDIA for the CUDA version of SNAP; and Cray Inc. for providing access to the Cray XC40 supercomputer, swan. This work was carried out using the computational facilities of the Advanced Computing Research Centre, University of Bristol - http://www.bris.ac.uk/acrc/.

References

[1] Michael P Adams, Marvin L Adams, W Daryl Hawkins, Timmie Smith, Lawrence Rauchwerger, Nancy M Amato, Teresa S Bailey, and Robert D Falgout. Provably optimal parallel transport sweeps on regular grids. *International Conference on Mathematics, Computational Methods & Reactor Physics*, pages 2535–2553, 2013.

[2] AMD. OpenCL Optimization Case Study - Simple Reductions. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/.

[3] D. Bailey, E Barszcz, J Barton, D Browning, R Carter, L Dagum, R Fato Ohi, S Fineberg, P Frederickson, T Lasinski, R Schreiber, H Simon, V Venkatakrishnan, and S Weeratunga. The NAS Parallel Benchmarks. Technical report, NASA, RNR-94-007, 1994.

[4] T S Bailey and R D Falgout. Analysis of Massively Parallel Discrete-Ordinates Transport Sweep Algorithms with Collisions. In *International Conference on Mathematics, Computational Methods, and Reactor Physics*, pages 1–15, New York, New York, USA, 2009. American Nuclear Society.

[5] Randal S Baker. An Sn Algorithm for Modern Architectures. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, number ANS MC2015, Nashville, TN, 2015. American Nuclear Society.

[6] T Deakin and S McIntosh-Smith. GPU-STREAM: Benchmarking the achievable memory bandwidth of Graphics Processing Units (poster). In Supercomputing, 2015.

[7] T Deakin, S McIntosh-Smith, and W Gaudin. Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale. In High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings, M. J. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 429–448.

[8] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.1, 2011.

[9] Thomas M Evans, Wayne Joubert, Steven P Hamilton, Seth R Johnson, John A Turner, Gregory G Davidson, and Tara M Pandya. Three-Dimensional Discrete Ordinates Reactor Assembly Calculations on GPUs. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, number ANS MC2015, Nashville, Tennessee, 2015. American Nuclear Society.

[10] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 2012.

[11] Heterogeneous System Architecture Foundation. HSA Specification Library. http://www.hsafoundation.com.

[12] W Daryl Hawkins, Timmie Smith, and MP Adams. Efficient Massively Parallel Transport Sweeps. *Transactions of the American Nuclear Society*, 107, 2012.

[13] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications, 2000.

[14] K.R. Koch, R.S. Baker, and R.E. Alcouffe. Solution of the first-order form of three-dimensional discrete ordinates equations on a massively parallel machine. *Transactions of the American Nuclear Society*, 65:198–199, 1992.

[15] E.E. Lewis and W.F. Jr. Miller. *Computational methods of neutron transport*. American Nuclear Society, 1993.

[16] John D McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, dec 1995.

[17] Salli Moustafa, Ivan Dutka-Malen, Laurent Plagne, Angélique Ponçot, and Pierre Ramet. Shared memory parallelism for 3D Cartesian discrete ordinates solver. *Annals of Nuclear Energy*, 33, sep 2014.

[18] Gihan R. Mudalige, Mary K. Vernon, and Stephen A. Jarvis. A plug-and-play model for evaluating wavefront computations on parallel architectures. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, pages 1–14. IEEE, apr 2008.

[19] Aaftab Munshi. The OpenCL Specification, Version 1.1, 2011.

[20] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Wright, and S. A. Jarvis. On the acceleration of wavefront applications using distributed many-core architectures. *Computer Journal*, 55(2):138–153, jul 2012.

[21] Simon J. Pennycook, Simon D. Hammond, and Gihan R. Mudalige. Experiences with porting and modelling wavefront algorithms on many-core architectures. 2010.

[22] Oreste Villa, Daniel R. Johnson, Mike OConnor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, Stephen W. Keckler, and William J. Dally. Scaling the power wall: a path to exascale. In *Supercomputing*, 2014.

[23] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52:65–76, 2009.

[24] Shucai Xiao and Wu Chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010.

[25] Robert J. Zerr and Randal S. Baker. SNAP: SN (Discrete Ordinates) Application Proxy - Proxy Description. Technical report, LA-UR-13-21070, Los Alamos National Labratory, 2013.