MPI + OpenACC: Accelerating Radiation Transport Mini-Application, Minisweep, on Heterogeneous Systems

Robert Searles[a], Sunita Chandrasekaran[a,*], Wayne Joubert[b], Oscar Hernandez[b]

[a] *University of Delaware, 101 Smith Hall, Newark, DE 19716*
[b] *Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, TN 37831-6173*

Abstract

Architectures are rapidly evolving, and exascale machines are expected to offer billion-way concurrency. We need to rethink algorithms, languages and programming models among other components in order to migrate large scale applications and explore parallelism on these machines. Although directive-based programming models allow programmers to worry less about programming and more about science, expressing complex parallel patterns in these models can be a daunting task especially when the goal is to match the performance that the hardware platforms can offer. One such pattern is wavefront. This paper extensively studies a wavefront-based miniapplication for Denovo, a production code for nuclear reactor modeling. We parallelize the Koch-Baker-Alcouffe (KBA) parallel-wavefront sweep algorithm in the main kernel of Minisweep (the miniapplication) using CUDA 9.0, OpenMP 4.0 (`SIMD`) and OpenACC 2.6. Our OpenACC implementation running on NVIDIA's next-generation Volta GPU boasts an 85.06x speedup over serial code, which is larger than CUDA's 83.72x speedup over the same serial implementation. We also explore the scalability of our solution using MPI to decompose our simulation domain, allowing us to run on many nodes and accelerators present in state-of-the-art HPC systems. Our parallelization effort across platforms also motivated us to define an abstract parallelism model that is architecture independent, with a goal of creating software abstractions

*Corresponding author.
E-mail address: schandra@udel.edu

that can be used by applications employing the wavefront sweep motif.

Keywords: Radiation Transport; MPI+OpenACC; Summit; Acceleration; Minisweep

---

Program Title: Minisweep
Licensing provisions(please choose one): BSD 2-clause
Programming language: C

Supplementary material:
Journal reference of previous version:
Does the new version supersede the previous version?:
Reasons for the new version:
*Summary of revisions:**

Nature of problem(approx. 50-250 words):
The Minisweep proxy application [1] is part of the Profugus radiation transport miniapp project [2] that reproduces the computational pattern of the sweep kernel of the Denovo S_n radiation transport code [3]. The sweep kernel is responsible for most of the computational expense (80-99%) of Denovo. Denovo, a production code for nuclear reactor neutronics modeling, is in use by a current DOE INCITE project to model the International Thermonuclear Experimental Reactor (ITER) fusion reactor [4]. The many runs of this code required to perform reactor simulations at high node counts makes it an important target for efficient mapping to accelerated architectures.

Solution method(approx. 50-250 words):
This work proposes an abstract parallelism model for efficiently mapping wavefront application to modern HPC architectures. Minisweep is used as a case study for evaluating this technique. Our evaluation is performed using OpenACC to target many architectures.

Additional comments including Restrictions and Unusual features (approx. 50-250 words):

1. Introduction

Hardware architectures are rapidly evolving. High performance computing nodes are becoming increasingly heterogeneous. The current and anticipated

exascale accelerated node architectures are heterogeneous [5]. They are expected to contain a mix of throughput and latency optimized cores [6]. Such a balanced mixture of cores is expected to manage different types of parallelism available in an algorithm. Memory has advanced as well. 3-D memory stacking with memory moving on-socket provides increased bandwidth and faster communication.

Such diverse architectures require their own code optimization strategies, while the application developers prefer a "write-once" code development strategy in which a single code will execute efficiently on all targeted architectures. In addition to considering the underlying hardware, a programming model is also expected to address requirements of applications and their algorithms. The programming language that implements the model should provide the right abstractions to improve the productivity of scientific developers. Programmers often resort to a trade-off between achieving portability and high performance. Why? The issue is two-fold. Adequate application parallelism will not be exposed to the hardware architecture if the algorithm is structured in a way that limits the level of concurrency that a programming model can benefit from. Secondly, such a single code representation is possible only if the programming abstractions are carefully crafted for the programming models to provide informative hints to the compilers to generate optimized code across platforms.

Directives allow us to abstract the rich feature set of hardware architectures and incrementally improve, port, and maintain the codebase across platforms. However, there still remains a gap in the way that they do not adequately expose and parallelize some of the complex algorithms often found in applications. One such case is a wavefront-based algorithm that is of critical importance to solving scientific problems in multiple science domains. Wavefront algorithms are useful for problems for which the computed result values have dependencies, requiring that results be computed in stages (wavefronts) for which each stage's results depends on results computed in previous stages.

In this paper, our objective is to study this type of algorithm and identify challenges in exposing parallelism using high-level abstractions that can be lowered to de facto parallel programming languages. We study the Minisweep proxy application [1], that is illustrative of the complexities in a wavefront-based algorithm. Parallelizing Minisweep using current directive-based APIs revealed the gaps in their expressivity and features. We address this issue by designing and envisioning an abstract parallelism model that highlights these

gaps with a combination of notations [7]. Having these proposed notations in a programming language are key to exposing and mapping wavefront-based parallelism to multiple architectures.

We enhance our original work, described in [7], in a couple of ways. First, we utilize asynchronous parallel regions in OpenACC to modify our OpenACC implementation of Minisweep to sweep in eight directions in parallel. This is the intended behavior of the original code, but due to the difficulty of implementation, our original work simply used 8 sweeps in a single direction to model the performance of the code. Extending this to support the proper behavior of kernel launches in Minisweep allowed us to explore an additional layer of parallelism in OpenACC that is not widely used. Secondly, we also utilize MPI to decompose our simulation's spatial domain across devices in an effort to examine the scalability of our proposed abstract parallelism model. State-of-the-art HPC systems (such as ORNL's Summit machine) are comprised of many nodes equipped with many accelerators, so a single-node/single-accelerator approach is not representative of how an application like this would be used in practice. To that end, we felt it necessary to explore the scalability of parallelization across many GPUs.

1.1. Application Under Study

The Minisweep proxy application [1] is part of the Profugus radiation transport miniapp project [2] that reproduces the computational pattern of the sweep kernel of the Denovo S_n radiation transport code [3]. The sweep kernel is responsible for most of the computational expense (80-99%) of Denovo. Denovo, a production code for nuclear reactor neutronics modeling, is in use by a current DOE INCITE project to model the International Thermonuclear Experimental Reactor (ITER) fusion reactor [4]. The many runs of this code required to perform reactor simulations at high node counts makes it an important target for efficient mapping to accelerated architectures.

This study involves S_n radiation transport algorithms for solving the linear Boltzmann equation [8]. Here, a continuum model is used to simulate the density of particles of a given energy and direction of motion within a 3-D volume. The approach yields a six dimensional problem (3-D in space, 2-D in angular particle direction and 1-D in particle energy) that is appropriately discretized in each dimension. Minisweep includes neutronics calculations for nuclear reactor [9] and fusion reactor [4] design, radiation shielding, nuclear forensics and radiation detection. The large number of problem dimensions available in the S_n transport algorithm affords significant opportunities for

parallelism on manycore parallel systems. However, the recursive nature of the wavefront calculation in the spatial dimensions is a challenge to efficient parallelization.

Denovo was one of six applications selected for early application readiness on ORNL's Titan system under the Center for Accelerated Application Readiness (CAAR) project [10] and is part of the Exnihilo code suite which received an R&D 100 award for modeling the Westinghouse AP1000 reactor [11]. Minisweep can be considered a successor to the well-known Sweep3D benchmark [12] and is similar to other S_n wavefront codes including Kripke [13], SN (Discrete Ordinates) Application Proxy (SNAP) [14] and PARTISN [15].

Minisweep is used as a vehicle to examine parallelization of wavefront algorithms in general. However, it has multiple computational motifs (dense and sparse linear algebra, structured grids) and parallelism requirements (halo communications, hierarchical synchronizations, atomic updates) which make the study of this algorithm relevant to a much broader spectrum of codes.

For Minisweep, we use OpenACC 2.6 and OpenMP 4.0 (`SIMD`) along with CUDA 9.0. OpenACC, since its inception in 2012, is being widely used to port large scale applications spanning several domains such as ANSYS [16], GAUSSIAN [17], and Icosahedral non-hydrostatic (ICON) [18] to massively parallel architectures. Similarly, OpenMP 4.5 is being deployed to applications such as Pseudo-Spectral Direct Numerical Simulation-Combined Compact Difference (PSDNS-CCD3D) [19] and a CFD code for turbulent flow simulation, Quicksilver [20], a Monte Carlo Transport code.

1.2. Contributions

Our work presents the following contributions:

- An abstract representation elucidating architectural, memory and threading challenges to programming models for such complex wavefront algorithms as used in Minisweep that can be broadly applicable to applications with a similar computational motif. More details in Section 1.1 and Section 8.
- Parallelizing the sweep across eight directions using OpenACC, a challenge in transport algorithms.
- A description of the challenges in existing programming models, and extensions that will allow programmers to overcome the obstacle of recursivity in the spatial dimensions of wavefront algorithms without requiring large modifications to the code base.

- A performance-portable implementation of these abstractions using OpenACC to offload portions of an application to a variety of parallel architectures.
- An exploration of the scalability of our solution using MPI to decompose our simulation domain, allowing us to run on many nodes and accelerators present in state-of-the-art HPC systems.

2. Overview of Sweep Algorithm

The S_n transport sweep algorithm possesses features common to wavefront algorithms in general yet has structure specific to the requirements of S_n transport. It can be considered in two parts: first, a wavefront algorithm relating the computations between gridcells of a 3-D grid, and second the computations performed on a single gridcell within this wavefront sweep across a grid.

2.1. Grid-level computations

We consider here a 3-D structured grid, with locally connected gridcells; see Figure 1. Importantly, the result computed at a gridcell is dependent on the results computed at the three neighbor gridcells in the upstream x, y and z directions; thus the computation is described by a four-point stencil. This dependency puts a restriction on the order in which results can be computed. One possible ordering is a series of wavefronts described by a sequence of planes of gridcells starting at a corner of the grid and sweeping through the whole grid (Figure 1). Other orderings are allowed as well, as long as the dependencies are satisfied.

Modeling the physical problem requires modeling particle flux in all directions. To accomplish this, an execution instance of the algorithm performs a total of eight sweeps, one starting at each corner of the domain. These directions are referred to as "octants." The results of all eight octant sweeps are added together form the final result.

2.2. Gridcell-level computations

To further describe the algorithm, we define array v_s with dimensions $v_s(n_x, n_y, n_z, n_u, n_e, n_a, n_o)$. The n_x, n_y, n_z dimensions refer to the spatial grid size. The dimension $n_o = 8$ is the octants axis, across which results are summed for the final result. The value n_a is a set of angular directions,

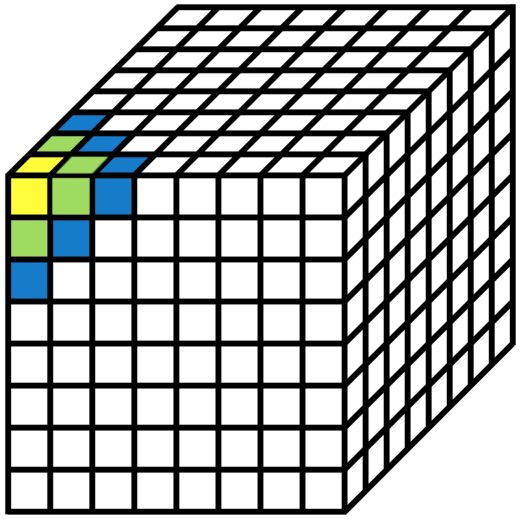

Figure 1: Wavefront computational pattern

and n_e is the number of energy groups, each representing a decoupled instance of the problem. Finally, n_u represents a set of unknowns for each gridcell based on the spatial discretization, e.g., finite element coefficients for a gridcell. Though v_s is relevant to key computations of the algorithm, the arrays that actually hold the input and output of the algorithm have the form $v(n_x, n_y, n_z, n_u, n_e, n_m)$. Here v_s and v are related by the fact that the n_a, n_o axes are compressed into n_m moments to form v from v_s. The computation at a gridcell then is composed of the following steps:

1. Moments-to-angles conversion. For a given octant and energy group the input $v_{in}(i_x, i_y, i_z, *, i_e, *)$ is transformed into array $v_{in,s}(i_x, i_y, i_z, *, i_e, *, i_o)$ by a small matrix-vector product that relates the n_m moments to the n_a angles. The matrix depends on the octant but is independent of spatial location, energy group and unknown.
2. Face contribution. The upstream components from the sweep are added to $v_{in,s}(i_x, i_y, i_z, *, i_e, *, i_o)$.
3. Solve. An operation is performed on the array values $v_{in,s}(i_x, i_y, i_z, *, i_e, *, i_o)$ which is coupled between the n_u unknowns but decoupled in all other dimensions. The result is $v_{out,s}(i_x, i_y, i_z, *, i_e, *, i_o)$.
4. Face update. The values of $v_{out,s}(i_x, i_y, i_z, *, i_e, *, i_o)$ are stored for use downstream by the sweep.

5. Angles-to-moments conversion. Array
   $v_{out,s}(i_x, i_y, i_z, *, i_e, *, i_o)$ is transformed and added to
   $v_{out}(i_x, i_y, i_z, *, i_e, *)$ with another matrix-vector product depending on octant only.

The computation can be represented mathematically as follows. To simplify we assume a single octant direction (+x,+y,+z) and a single energy group. Given matrix $M_{AM} \in R^{n_a \times n_m}$ we have

$$(v_{in,s;i_x,i_y,i_z})_u = M_{AM} \cdot (v_{in;i_x,i_y,i_z})_u,$$

where i_x, i_y, i_z is the gridcell spatial coordinate and $(\cdot)_u$ denotes the restriction of the vector to gridcell unknown u. Then

$$(v_{out,s;i_x,i_y,i_z})_a = S((v_{in,s;i_x,i_y,i_z} - v_{out,s;i_x-1,i_y,i_z} - v_{out,s;i_x,i_y-1,i_z} - v_{out,s;i_x,i_y,i_z-1})_a)$$

where $(\cdot)_a$ is the restriction of the vector to angle a. Notice this represents the wavefront recursion proper. Here the function $S : R^{n_u} \to R^{n_u}$ is a solve process which for actual transport solvers is related to the spatial discretization used but for Minisweep is a synthetic function chosen to give a known analytic solution for correctness checking; in either case, the computational cost is minor, thus the actual choice is not material to algorithm performance. Finally, for matrix $M_{MA} \in R^{n_m \times n_a}$,

$$(v_{out;i_x,i_y,i_z})_u = M_{MA} \cdot (v_{out,s;i_x,i_y,i_z})_u.$$

2.3. Summary of problem axes

The problem dimensions and their respective couplings are thus summarized as follows:

- Space: in the x, y and z dimensions, each gridcell depends on results from three upstream cells, based on octant direction, resulting in a wavefront problem.
- Octant: results for different octants can be calculated independently, the only dependency being that results from different octants at the same entry of v'_{out} are added together and thus are a race hazard, depending on how the computation is done.
- Energy: computations for different energy groups have no coupling and can be considered separate problem instances, enabling flexible parallelization.

- Moment, angle: for fixed energy, octant and spatial location, the moments and angles are interrelated by small (dense) matrix-vector products.
- Unknown: when the problem is represented as angles, a computation is performed which may couple (only) the unknowns within the gridcell; for all other parts of the computation, elements on this axis are fully independent.

3. Parallelizing the Sweep Algorithm

To map the sweep algorithm to a parallel system, it is of paramount importance to minimize data motion as well as maximize parallelism, these being increasingly critical for high performance on exascale systems. As a result, the general guiding principle is that spatial dimensions must be the outermost loops, due to their sparse coupling, whereas moment, angle and unknown loops must be innermost due to the strong all-to-all couplings. The specific approach to parallelizing each axis is as follows:

Space: Spatial parallelism is based on the Koch-Baker-Alcouffe (KBA) algorithm [21]. Here the 3-D structured grid is decomposed to processors with a 2-D tiling in x and y (Figure 2). Each processor's part of the grid is decomposed into blocks along the z axis. Then a block wavefront process is applied starting at the corner block of the domain. Processors proceed in a series of parallel steps, with one block wavefront computed at each step and block face information communicated between consecutive steps. For a GPU or other accelerated processor, the KBA block described above is further decomposed into subblocks, and the computation is arranged into a series of subblock wavefronts, which are then mapped to parallel threads.

Octant: Minisweep assigns compute threads to the eight wavefronts corresponding to the eight octant directions. The wavefronts are independent; however there is a potential race condition when two or more threads are updating the same KBA block on the same KBA block wavefront step. The solution used in Minisweep is a grid coloring approach. The KBA block is split in half along each dimension resulting in eight "semiblocks." Eight semiblock steps are taken, and for each step every one of the (up to) eight active wavefronts is assigned to a different semiblock. This is arranged so that the wavefront dependencies are satisfied. A synchronization and thread fence are required between consecutive semiblock steps.

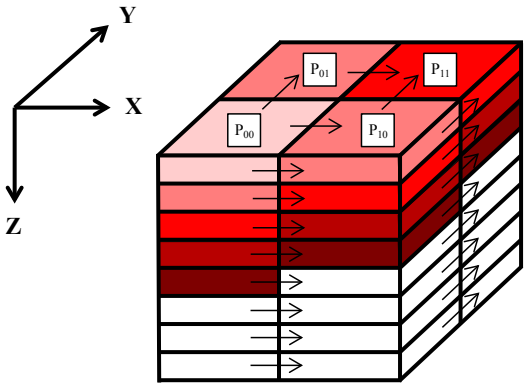

Figure 2: KBA parallel wavefront algorithm

Energy: Since the algorithm is embarrassingly parallel along the energy axis, this problem axis can be decomposed in any way: across nodes, across threads in a node, in a core or vector unit, or any combination.

Moment, Angle: The moment and angle axes are coupled to each other in an all-to-all fashion via two small matrix-vector products. The moment, angle, and unknown dimensions of the relevant arrays are ordered to be most rapidly varying, enabling efficient stride-1 memory access. When possible, the matrix-vector products are arranged to fit entirely within a vector unit. If n_a or n_m exceed the vector unit size, a blocking strategy is used with the computation fitting within the vector unit. Importantly, the moments-to-angles transform is threaded in angle and the angles-to-moments transform is threaded in moment (otherwise a reduction across threads and/or vector lanes is required). Because threads must be reassigned between moment and angle threads, a synchronization and memory fence is required between these operations.

Unknown: No couplings exist along the unknowns axis when the small matrix-vector products are performed, therefore this axis permits some opportunity for threading here. However, for the inner solve computation in angle, each unknown may require different kinds of computations. To prevent poor use of vector units, these computations are kept serial.

4. Abstract Parallelism Model

Minisweep defines and implements a set of high level abstractions to describe the parallelism of its algorithm such that these abstractions can be used by any similar application implementing the sweep motif. The goal of these

abstractions is to achieve productivity and performance portability across different architectures (GPUs, Xeon Phi, CPUs, etc). These abstractions can be instantiated using general purpose parallel programming languages like CUDA, OpenMP, and OpenACC. The need for these abstractions is also suggestive of possible shortcomings in parallel programming languages and suggests the need for extensions to support applications of this type.

The large number of problem dimensions inherent in S_n transport solves makes the need for managing thread parallelism axes and hierarchical memories via use of abstraction layers acute. However, the techniques described here are applicable to many other problems requiring multidimensional parallelism, for example, batched dense linear algebra, block sparse linear solvers, and others.

Abstract machine model: Modern compute node hardware has an execution hierarchy. For example, a compute node may be composed of multiple GPUs, each with multiple cores possessing hardware threads and employing vector units composed of vector lanes. Some of these have co-located memories, for example node main memory, GPU high bandwidth memory or GPU shared memory associated with a streaming multiprocessor (SM) core. Execution threads are also associated with each level: for NVIDIA GPUs, in-warp threads execute in lock-step within a warp, in-threadblock threads are associated with an SM, and the thread grid is associated with the GPU. One can thus view a node as a hierarchy of execution units, with local memory and compute threads, and in particular hardware threads can be thought of as indexed as a tuple depending on the location in the hierarchy. Threads also have characteristics based on location, e.g., thread synchronization across different cores of a node may be impossible or much slower compared to on-core synchronization. Likewise memories at different levels have different speeds, and thread access to memories may have NUMA effects depending on the level. Note that these concepts readily apply to heterogeneous node as well as homogeneous systems.

To abstract the characteristics of heterogeneous / homogeneous architectures, we use the concept of "place," borrowing ideas from X10 and Chapel ("locales"). A place is an abstract location (in our case, a node of a computer) where work can execute using local executions units with local memories where threads can possibly synchronize with each other (e.g., barriers, memory fences) and access its memory. What we want to explore is how to generalize a flat model of places (e.g., that originally X10 and Chapel used to abstract an entire system) to more local hierarchical abstractions with

concepts such as Hierarchical Tree Places [22] or Chapel (Sub-locales) [23] to abstract the new trends of complex memory hierarchies and accelerators. An architecture can be described as a set of "hierarchical places," where at the higher level places communicate with each other via message passing or remote puts/gets. In the node, places can be nested in order to abstract the memory hierarchy of an architecture and its local execution units. In our abstract machine model, a nested place can access the memory of its parent place, but sometimes cannot synchronize with sibling places. This restriction is due to the way a child place can be mapped to GPU SMs, across which it is not possible to synchronize.

We use the term "place threads" to refer to execution threads associated with each of the specific places. In Minisweep, place threads are created during execution via an adaptor function which instantiates or destroys the threads for the requested places in the underlying architecture; in practice, this is implemented for example by launching a CUDA kernel or entering an OpenMP parallel region, depending on programming language implementation on the given architecture.

Abstract arrays: In Minisweep, an "abstract (multidimensional) array" is defined as an object that consists of a list of dimensions and a base pointer associated with a memory place in the hierarchy. Array elements are accessed using a multi-index through an indexing function. In this way the memory layout is controlled by an abstraction layer that can be easily modified based on the architecture. An abstract array thus has a local view within the place (and its children) at which it is allocated.

Abstract threads: Each independent variable of the science problem is assigned an abstract "threading axis" of abstract thread indices assigned to the corresponding problem axis. For example, the axis of n_e energy group values is assigned a set of (n_e or fewer) abstract compute thread indices used to compute those values. The collection of these abstract thread indices (which can be used to describe threads applied to the problem dimensions of energy, octant, y location, z location, etc.) form a tuple or thread multi-index, which will be later bound to a place thread.

Instantiation of the abstract threaded region: A fundamental operation for multi-threaded or accelerated codes is entry into a fork-join parallel region. In Minisweep this is abstracted as a multi-threaded region that instantiates the abstract threads. These regions can be nested and mapped to the same or different places.

Binding of abstract threads to places: A mapping of the abstract

thread multi-index to a place thread multi-index is made based on the type of parallelism required. For example, since spatial wavefronts of the sweep algorithm require barriers between wavefronts, the spatial y and z dimensions must necessarily be mapped to threads within a place (e.g., threadblock on a GPU) that allows synchronization. By comparison, energy groups are fully decoupled, thus no restrictions are placed on where the abstract energy thread axis is mapped.

Parallel worksharing construct: This construct schedules work along problem axes to a set of abstract threads and executes the work in parallel. The array index values for a given problem axis are distributed to the abstract thread indices via a block decomposition. For example, the full set of n_e energy groups is partitioned into blocks which are in turn assigned to abstract energy threads. Table 1 describes the mapping of problem axes and associated abstract threads to the place thread hierarchy for the algorithm discussed in Section 2. It is evident that the ability to synchronize a subset of threads, akin to a barrier within an MPI sub-communicator, would be of benefit, since synchronization does not scale well to large thread counts.

problem dimension	dependency type	GPU threading	Intel Phi threading
energy	(none)	grid	OpenMP thread
octant	coloring	threadblock	CPU thread
spatial y	wavefront	threadblock	CPU thread
spatial z	wavefront	threadblock	CPU thread
moment	all-to-all	warp, serial	vector, serial
angle	all-to-all	warp, serial	vector, serial
unknown	all-to-all	warp, serial	vector, serial

Table 1: Problem dimensions mapping to thread hierarchy.

Figure 3 is a simplified version of S_n sweep parallelization using the abstract parallelism model. The pseudocode shows the allocation of the required arrays and definition of the hierarchical thread regions, followed by nested parallel loop over energy groups, serial loop over wavefronts, parallel loop over gridcells in the wavefront, and then the three threaded operations of moment-to-angles, solve and angles-to-moments.

```
// Abstract Arrays Allocation
AbstractArrayAllocation(vin(nx,ny,nz,ne,nm,nu): place_main)
AbstractArrayAllocation(vout(nx,ny,nz,ne,nm,nu): place_main)
AbstractArrayAllocation(neighbors(num_neighbors,ne,na,nu): place_main)
// Multithreaded Regions for Abstract Threads
AbstractMultithreadedRegion(abstract_threads_e: place_main) {
AbstractMultithreadedRegion(abstract_threads_a,
                    abstracts_thread_xy: place_local) {
// Do_All Parallel Worksharing
 Do_All(e in range(0,ne); abstract_thread_e) {
  AbstractArrayAllocation(vs(na,nu): place_local)
  do(w in range(0,w_max)) {
   // Do_All Parallel Worksharing
   Do_All((x,y) in wavefront(w); abstract_thread_xy) {
    z = z_coord(x,y,w)
    //Do_All Parallel Worksharing Matrix-Vector Product
    Do_All(a in range(0,na); abstract_thread_a) {
     do(u in range(0,nu)) {
      vs(a,u) = 0
      do(m in range(0,nm)) {
       vs(a,u) += a_from_m(a,m) * vin(x,y,z,e,m,u)
      }}
    } // End of Do_All(a)
    // Do_All Parallel Worksharing
    Do_All(a in range(0,na); abstract_thread_a) {
     // Apply upstream wavefront dependencies
     do(i neighbor of (x,y,z) in wavefront(w-1)) {
      do(u in range(0,nu)) {
       vs(a,u) -= neighbors(i,e,a,u)
      }}
     // Computation based on unknowns
     solve(vs,a)
     // Save downstream wavefront dependencies
     do(i neighbor of (x,y,z) in wavefront(w+1)) {
      do(u in range(0,nu)) {
       neighbors(i,e,a,u) = vs(a,u)
      }}
    } // End of Do_All(a)
    //Do_All Parallel Worksharing Matrix-Vector Product
    Do_All(m in range(0,nm); abstract_thread_m) {
      do(u in range(0,nu)) {
       vout(x,y,z,e,m,u) = 0
       do(a in range(0,na)) {
        vout(x,y,z,e,m,u) += m_from_a(a,m) * vs(a,u)
       }}}
   } // End of wavefront loop w
  AbstractArrayFree(vs)
 } // End of Do_All(e)
}} // End of AbstractMultithreadedRegions
AbstractArrayFree(vin, vout, neighbors)
```

Figure 3: Abstract representation of wavefront algorithm

5. Translation of Abstract Parallelism Model

This section shows flavors of how different models, CUDA, OpenMP and OpenACC parallelize Minisweep. The narrative also discusses what we need (referring to Section 4) and what the models lack discussed in Section 6.

5.1. CUDA

The main sweep function, `Sweeper_sweep()`, of Minisweep has a KBA pipeline loop to support the KBA block sweep calculations and related asynchronous face communication between nodes using MPI. For the CUDA

case, faces and KBA blocks are also transferred to and from the GPU asynchronously; for systems not requiring offload, these calls do nothing. A mirrored array datatype, resembling the underlying mechanisms of OpenACC, maintains copies of an array on CPU and GPU and manages transfers; an accessor function returns the CPU or GPU pointer depending on where the computation takes place. For details, see [24]. The key kernel operation of Minisweep is the block sweep operation, in function `Sweeper_sweep_block()`. This is launched as a CUDA kernel or alternatively is initiated as a parallel region in OpenMP, as described above in the abstract model. Since energy groups are independent, the energy thread axis is mapped off-threadblock into the GPU thread grid; all other axes are mapped in-threadblock due to coupling requirements as described earlier. As discussed in Section 2, an execution instance of the algorithm performs a total of eight sweeps and the CUDA port of minisweep performs these sweeps.

5.2. OpenMP

OpenMP is used to run Minisweep natively on a multicore or manycore processor with OpenMP 3.1 parallel directives and the OpenMP 4.0 `simd` directive. The model also implements KBA. OpenMP runs with a single thread of execution until the block sweep function is encountered, at which point threads are spawned in energy, octant and the y and z spatial dimensions. The temporary arrays placed in GPU shared memory for the CUDA case are now CPU arrays, with one part of the array reserved for each compute thread. Since the OpenMP model uses a SIMD loop rather than thread numbers to access vector lanes, loops are placed in the code for the angle, moment and unknown dimensions, and each of these dimensions is assigned only a single thread; for the CUDA case, however, these axes receive multiple threads and the SIMD for loop is removed. The OpenMP port models the particle flux in all directions, i.e. performs a total of eight sweeps using the tasks concept. Note: We have not ported the code to OpenMP4.5 as part of this work. We believe that OpenMP4.5 could be an alternate solution to explore minisweep on GPUs.

5.3. OpenACC

Our OpenACC proof-of-concept for our abstract parallelism model consists of two parts: parallelizing the initialization of faces at the beginning of the sweep and parallelizing the sweep itself. The sweep itself consists of three parts: 8 octant directions originating from each corner of the 3-dimensional

space being examined, a sweep across the 3-dimensional space for the octant in question, and the in-gridcell computations that happen inside of each gridcell of the space.

Parallelization of the face initializations involves five nested loops (spatial decomposition of gridcells, unknowns within gridcell, energy groups and angles). However, it is worth noting that PGI's OpenACC compiler only provides us with two levels of parallelism currently: gang (block) and vector (thread). The compiler is yet to thoroughly exploit the worker level of parallelism. So, in order to map a loop nest of five loops onto the accelerator to achieve full parallelization, we utilized OpenACC's `collapse` clause to collapse a specified number of nested loops into one large loop, which we can then map at either the gang or vector level. For Minisweep's face initializations, we collapse the outer three loops (corresponding to the unknowns and two spatial dimensions of the gridcells) and execute at the gang level. We also collapse the inner two loops (corresponding to the energy groups and angles) and execute at the vector level.

Parallelization of the main sweep component in Minisweep is not as trivial, as there are data dependencies between gridcells, as mentioned in Section 2.1. To that end, we utilize the KBA parallel sweep algorithm (discussed in Section 3) in order to exploit *gang*-level parallelism across the x, y, and z gridcell loops. Since there is currently no existing high-level language that provides functionality for implementing this type of parallel sweep, the programmer must modify the loop nest manually in order to achieve the desired behavior. This involves creating an outer *wavefront* loop that iterates over the wavefront decomposition, as discussed in Section 2.1 and shown in Figure 1. The computations within these wavefronts can be parallelized, albeit not trivially. First we must parallelize across the inner two dimensions: y and x. This spawns a number of threads on the GPU. Within each of these threads, we calculate our z value based off of the thread's y and x values and the wavefront iteration number. Then, we can perform a bounds check to determine whether that z value is within the bounds of the wavefront being examined (denoted by the current wavefront iteration number). This allows us to exploit parallelism across gridcells, while still accounting for data-dependencies between wavefront iterations.

The embarrassingly parallel in-gridcell computations are performed for each energy group within each gridcell. We mark these computations for execution at the *vector* level. A representation of the result is shown in Figure 4. Note that this code snippet is also the serial code if one were to simply

remove all the directives.

Transport algorithm sweeps are particularly challenging. After parallelizing this sweep kernel using only two levels of parallelism, we faced an additional obstacle. Minisweep doesn't run just a single sweep each timestep; it runs eight. Each of these sweeps represents a different *octant*, which refers to the direction the sweep iterates through the 3-dimensional simulation space. Each octant starts at a different corner of the space and iterates diagonally to the opposing corner. This complicates our calculation of the z dimension value, since we have to consider the direction that a given sweep is moving in along each axis, as well as the bounds check to make sure that the calculated z value lies along the current wavefront. If that isn't already a daunting enough task, these *octant* sweeps are also parallelizable, but OpenACC doesn't provide us another explicit layer of parallelism. Our solution to this predicament is to use asynchronous parallel regions within our *octant* loop. This effectively will launch kernels on the GPU asynchronously, which allows us to overlap computation across octants. This asynchronous behavior provides us with a third level of parallelism that we can use to saturate the GPU for a longer period of time, yielding optimal performance.

5.4. MPI Domain Decomposition

The goal of the domain scientists who use Minisweep is to explore the largest 3-dimensional space possible. This poses a challenge because we are limited by the amount of memory present on a single node or device. We can overcome this limitation by decomposing our simulation's spatial component across nodes and/or devices using MPI. Details about the data synchronization required to resolve dependencies between spatial blocks are presented in Section 5.1. In short, we decompose Minisweep's simulation domain across spatial dimensions (corresponding to gridcell-level computation). A subcomponent of the problem space is allocated on each device, which is bound to a single MPI rank. This includes the collection of gridcells that a given MPI rank is responsible for computing, as well as the few neighbors that it may need to read data from (despite not computing). After each wavefront iteration, a synchronization is used to update the neighbors that lie along the edge of the rank's portion of the simulation space. This incurs a small amount of overhead that is outweighed by the computational benefit of utilizing many accelerators.

Minisweep allows the user to control the behavior of this decomposition by providing two command line flags: *nproc_x* and *nproc_y*. The product of the

```
/*--- Loop over wavefronts ---*/
for (wavefront = 0; wavefront < num_wavefronts; wavefront+=1) {

   /*---KBA threading---*/
#pragma acc loop independent gang, collapse(2)
     for( iy=0; iy<dim_y; ++iy )
     for( ix=0; ix<dim_x; ++ix ) {

       int iz = wavefront - (ix + iy);
       if (iz >= 0 && iz <= wavefront && iz < dim_z) {

               /*---moments to angles---*/
#pragma acc loop independent vector, collapse(3)
         for( ie=0; ie<dim_ne; ++ie )
               for( iu=0; iu<NU; ++iu )
               for( ia=0; ia<dim_na; ++ia ) {
                  P result = (P)0;
#pragma acc loop seq
                   for( im=0; im < dim_nm; ++im )
                    { /*---moments to angles conversion---*/ }
               }

         /*---solve---*/
         #pragma acc loop independent vector, collapse(2)
         for( ie=0; ie<dim_ne; ++ie )
               for( ia=0; ia<dim_na; ++ia )
                { /*---solve calculation---*/ }

               /*---angles to moments---*/
#pragma acc loop independent vector, collapse(3)
         for( ie=0; ie<dim_ne; ++ie )
               for( iu=0; iu<NU; ++iu )
               for( im=0; im<dim_nm; ++im ) {
#pragma acc loop seq
                  P result = (P)0;
                  for( ia=0; ia<dim_na; ++ia )
                   { /*---angles to moments conversion---*/ }
         }
       }
     }
   }
```

Figure 4: Sweep loop nest with OpenACC annotations

values passed using these flags should equal the number of MPI ranks used. By using these additional arguments, we are able to decompose our problem across two spatial axes instead of one, and the user is given the ability to control to what extent the simulation is decomposed along each axis. For example, if we use 4 MPI ranks, we have the option to decompose either the x or y axis across 4 ranks, or we can set both *nproc_x* and *nproc_y* to 2, which decomposes the simulation across both axes simultaneously. When using larger simulation configurations and a larger number of MPI ranks, we can play with the *nproc_x* and *nproc_y* values to decompose the simulation in other ways and observe the impact on Minisweep's performance. An example of this would be using 16 MPI ranks and setting *nproc_x* to 8 and *nproc_y* to 2, or vice-versa. For the purposes of this work, we stick to an even decomposition across both axes in our experimental setup.

-

6. Programming Model Limitations

6.1. General

In all cases, inadequacies of current compilers required that some code be rewritten in an unnecessarily low-level fashion to obtain correctness and/or performance. This seems to be a systemic challenge, insofar as it is difficult for compiler teams to develop mature and performant compilers for frequently changing complex processor hardware. Programming models support vectorization in different ways, leading to portability challenges. CUDA treats vector lanes as threads, whereas OpenMP uses SIMD loops and OpenACC has a `vector` clause for parallel loops. Such differences can lead to increased use of undesirable `ifdef`s if it is required to support these multiple programming models. Developers would prefer a single highly performant programming model with a high level of abstraction targeting all architectures rather than the need to use multiple programming models within a code.

The Minisweep code requires in several places a thread synchronization or barrier over only a subset of threads. A barrier across fewer threads could potentially run much faster in current hardware. This feature is not currently supplied by any of the programming models, though in principle a barrier across a subset of OpenMP threads could be written, and the new CUDA 9 Cooperative Groups feature may be useful here.

The Minisweep design makes it easy to change the mapping of machine threads to abstract problem threads and problem dimensions. A more challenging goal is to allow easy modification of the execution hierarchy. Such a design would allow easy loop order permutation and other loop restructuring operations, loop blocking to optimize cache use or reduce loop overheads, and on-demand reassignment of loop axes either to parallel threads or alternatively serial execution. Such changes generally require motion of significant portions of code, e.g., to optimize for loop invariant quantities. Presently this must be done by hand, and is not directly supported by programming models or imperative programming languages as currently conceived. Likewise, the use of accessor functions in Minisweep permits easy modification of memory locale and layout for an array. One must still however schedule memory transfers across the memory hierarchy manually for peak performance. Automatic transfers via paging/caching such as the CUDA Unified Memory feature and similar functionality for Intel Phi on-package memory will simplify programming for this, however past experience has shown that

manual prefetching of data across the hierarchy is sometimes necessary to attain high performance. As memory layers proliferate, e.g., with inclusion of NVRAM, managing this will become more challenging.

6.2. CUDA

CUDA by nature provides a lower level programming model compared to directives-based methods. Though the CUDA runtime API provides a slightly higher abstraction level than the CUDA driver API, both cases require `ifdef`s to make a code portable between CUDA for GPUs and standard C/C++ for conventional architectures. CUDA has the advantage that vector lanes are addressed explicitly as threads, resulting in reliable vectorization. However, certain coding constructs can lead to losses in performance in unexpected ways. For example, in the course of developing the Denovo sweeper and Minisweep, it was observed that when loop bounds were passed into a CUDA kernel within a `struct`, performance was noticeably degraded compared to when passed in as scalars. Furthermore, in some cases a `for` loop that was provably one-trip at compile time ran slower than when the loop was altogether removed, necessitating use of an `ifdef` to make a single CUDA / OpenMP-SIMD code. CUDA additionally has limitations with repect to deep copy of structs and classes—since pointers in a host struct are invalid on the device—though this is improving with the support of GPU Unified Memory. In short, limitations of this nature can make it challenging to raise the abstraction level in CUDA codes and maintain performance portability with other platforms.

6.3. OpenMP

Intel Phi performance typically depends on the effective vectorization of loops, using the native `simd` directive or alternatively the OpenMP `simd` directive. In the process of porting Minisweep to the Intel Phi using the Intel compiler, challenges to loop vectorization were encountered. In one case an array accessor function needed to be flattened by removing its use of a `struct` in order to enable the loop to vectorize. In another case the compiler failed to remove a provably loop invariant quantity from a loop, inhibiting vectorization. Also, the compiler would not vectorize the outermost loop of a deep loop nest, though CUDA had no problem threading this loop. The differing treatment of vector lanes as threads by CUDA and by SIMD loops in OpenMP required the undesirable use of special case code to handle the differences. Also, CUDA generally favors larger kernels to minimize kernel

launch overhead and maximize data reuse, whereas with the Intel compiler it is difficult or impossible to vectorize large, complex loops in one piece. These differences made it challenging to support the different platforms without special case programming. Overall, the difficulty of predicting a priori when a complex loop would vectorize and the need at times to rewrite code at a lower abstraction level was detrimental to writing maintainable, performance portable code. We also explored converting current the OpenMP 3.1 code to 4.5 (only within the scope of the proposed ideas and not with respect to porting the full code to 4.5 to use GPUs). Adapting doacross for this type of wavefront problem would have been a potential direction to take. However doacross assumes a flat memory hierarchy (shared memory) but what we need for our type of case study is to map data objects to a memory hierarchy (e.g. place and child place) that would allow the wavefront computations to be more data-centric and be scheduled where the data is.

6.4. OpenACC

Similarly to OpenMP, we faced a number of challenges when implementing our parallelization strategy discussed in Section 5.3 in OpenACC. Our parallelization efforts also identified compiler bugs that we reported to the PGI team. The first issue was handling array accesses in the original Minisweep implementation. Accessor functions are used to calculate the address of the flattened 5-dimensional array accesses that occur throughout the `Sweeper_sweep` function, as described in Section 4. These functions returned the address of the array access in question, which was then dereferenced by the `Sweeper_sweep` function in order to perform the manipulation on the array element. OpenACC requires that we use the `routine` directive to convert these function calls to routines. However, the compiler was unable to properly generate routine code for functions utilizing external variables like these array accesses do. We had to eliminate the use of these functions and inline the calculation of the array address into the array accesses within the given loops, resulting in a more traditional array access. Unfortunately this is detrimental to efforts to raise the abstraction level of the code.

Another issue was related to loop bounds. In Minisweep, input parameters are stored in a globally defined struct. Since these values are representing the sizes of each dimension of the application, they are used later as loop bounds. However, while parallelizing, OpenACC does not assume that no aliasing is being done since this struct is defined globally (out of scope). There are two simple solutions to this issue. First, the compiler flag `-Msafeptr` can be

used to specify that there is no aliasing. However, this would not be the best option for this application as there is some sort of pointer aliasing present elsewhere. Instead, we simply extract the value of the dimension being used for a given loop bound and store it in an integer variable prior to the start of the accelerated loop.

The final issue we faced was identified as a compiler bug in PGI OpenACC 17.10. OpenACC can use `kernels` or `parallel` to generate code from an accelerated region. With `kernels`, the onus is on the compiler to check for dependencies and generate code, whereas with `parallel`, the onus is on the programmer; failure to do so will result in inaccurate results. In our case, we observed that even though we used the `parallel` directive, the compiler still performed dependency checks as if we had used `kernels` directive. We confirmed this behavior by parallelizing a loop with a known dependency. The compiler generated parallel code but showed incorrect results. We then added a collapse clause to the end of this loop directive, and we specified that it should be collapsed with the next loop in the loop nest, which contained no such dependencies. The result here was that the compiler reported that it parallelized this collapsed loop nest, but the results of the computation were accurate. Due to the increased runtime, we were able to conclude that the inner loop was indeed executing in parallel, but the outer loop was executing in serial despite what the compiler had reported. All of these issues were easily overcome in practice, but identifying them presented significant challenges along the way.

7. Evaluation & Results

Machine	CPU	GPU
NVIDIA PSG (V100)	2x Intel Xeon E5-2698 v3 (16 cores)	4x NVIDIA Tesla V100 (16GB HBM2)
NVIDIA PSG (P100)	2x Intel Xeon E5-2698 v3 (16 cores)	4x NVIDIA Tesla P100 (16GB HBM2)
NVIDIA PSG (K40)	2x Intel Xeon E5-2690 v2 (10 cores)	NVIDIA Tesla K40 (12GB GDDR5)
ORNL Titan	AMD Opteron 6274 (16 cores)	NVIDIA Tesla K20X (6GB GDDR5)
ORNL Summitdev	2x IBM POWER8 (10 cores)	4x NVIDIA Tesla P100 (16GB HBM2)
ORNL Summit	2x IBM POWER9 (21 cores)	6x NVIDIA Tesla V100 (16GB HBM2)
ORNL Percival	Intel KNL 7230 (64 cores)	N/A

Table 2: Specifications of the nodes in the systems we used to test different configurations of Minisweep.

As a validation of portability, Table 3 shows Minisweep results for one GPU of the Titan Cray XK7 system (CUDA), one GPU of the Summitdev

IBM Minsky system (CUDA) and one node of the Percival Cray XC40 KNL system (self-hosted OpenMP 4.0). The problem solved has $n_e = 64$, $n_a = 32$, and $n_u = 4$, with $n_x, n_y, n_z = 32$ The codes are not fully optimized, in particular one of the inner loops for the OpenMP-KNL case did not vectorize. However, all cases across different hardware and software environments attained a similar 4-5% of peak flop rate, a typical figure for this algorithm which has significant memory accesses, register usage and integer index calculations. This result suggests that the code is in fact performance portable, since reasonable performance is reached for all systems.

System	Cores (SMs)	GF/s peak	GF/s	% peak GF/s
Titan(K20X)	14	1311	55.9	4.26
Summitdev(P100)	56	5312	244.8	4.61
Percival(Phi7230)	64	2662	124.9	4.69

Table 3: Comparative performance on several platforms.

We evaluate the effectiveness of our abstract wavefront parallelism model by comparing the runtimes of our parallel implementations of Minisweep (described in Section 5) to the runtime of a serial version of the code on multiple HPC systems. Table 2 describes the hardware available on nodes of each system. Note that the NVIDIA Professional Service Group (PSG) machines and the ORNL Titan machine are existing state-of-the-art HPC systems, while ORNL Summitdev is a development cluster representative of the hardware that is now present on nodes in ORNL's next-gen supercomputer *Summit* [25]. We also utilized the PSG cluster's V100 nodes, which house NVIDIA's next-generation GPU that are present on nodes in *Summit*. We used PGI's 18.4 compiler to compile our OpenACC and OpenMP. We have also used GCC 6.3.0 and ICC 17.0 for OpenMP codes. Compiling the code using Intel's OpenMP compiler was not successful and required code restructuring to take advantage of SIMD in minisweep.

Our experimental configuration is a representative example of what a real run of Minisweep within the Denovo radiation transport code looks like. Our problem dimensions on a single node are designed to be as large as we can fit on a single GPU: $n_e = 64$, $n_a = 32$, and $n_u = 4$, with $n_x, n_y, n_z = 32$, on K20x/K40 and $n_x, n_y, n_z = 64$ on P100/V100. For MPI runs, we ran across 4 nodes on NVIDIA's PSG cluster, which are each equipped with 4 GPUs.

To that end, our in-gridcell sizes for n_e, n_a, and n_u remain unchanged, but we explore a larger 3-dimensional space using $n_x, n_y, n_z = 128$ on NVIDIA's PSG system.

Figure 5 presents the results when running different implementations of Minisweep using our single node configuration in the form of speedups over the baseline serial implementation on existing HPC systems. Note that the speedup results presented were obtained by calculating the average of a series of runs for each implementation. There are a few notable results. First, our multicore CPU GCC's OpenMP (3.1) and OpenACC implementations yield favorable speedups. Note that GCC's OpenMP performed better than PGI's OpenMP. As mentioned in Section 5.3, we have currently parallelized the in-gridcell computations, as well as the spatial decomposition utilizing the KBA parallel sweep algorithm to resolve data dependencies, as discussed in Section 2.1. This implementation boasts a larger speedup than our OpenMP GCC version, as well as our CUDA configuration when parallelized over the same problem dimensions. Our OpenACC KBA configurations yield an addition layer of parallelism across spatial dimensions and show a much larger speedup compared to configurations which only execute in-gridcell computations in parallel. This leads us to conclude that there is additional performance to be gained, albeit not trivial to implement. It is also worth noting that our OpenACC implementation running on NVIDIA's next-generation Volta GPU boasts an 85.06x speedup over serial code, which is larger than the 83.72x speedup over the same serial implementation achieved by CUDA. This supports our claim that our proposed extension to existing high-level programming models is worthwhile, both from a performance standpoint, as well as a programming productivity standpoint. Currently, without major code modification, this challenge cannot be overcome.

Absolute runtimes for GPU configurations utilizing the KBA parallel sweep algorithm are presented in Figure 6. As shown, our OpenACC GPU implementation performs well compared to its CUDA counterpart in all cases. In addition to its excellent GPU performance, it is worth noting that this same OpenACC implementation was used to obtain results on our multicore CPU platforms by simply recompiling and specifying a different target. No additional code modifications were necessary to achieve this demonstration of portability. We contend that this provides additional evidence for the importance of an extension that would allow us to parallelize the outer spatial dimensions, yielding additional parallelism across gridcells without requiring a major coding effort on the part of the programmer. As stated in Section 4,

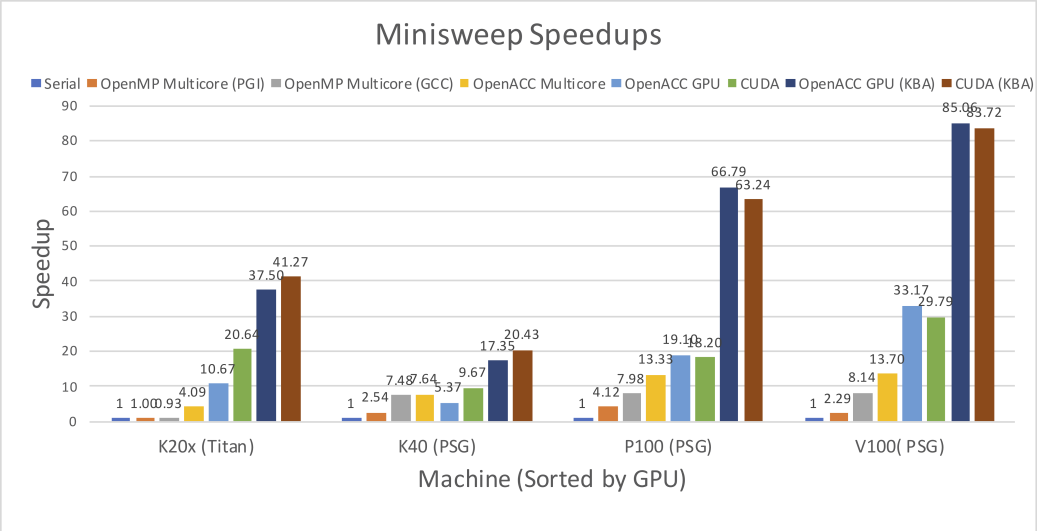

Figure 5: Minisweep's speedups over serial using different runtime configurations on a single node. The CUDA version is parallelized along the same dimensions as the OpenACC GPU configuration. The corresponding KBA configurations utilize the KBA blocking method for additional parallelism across spatial dimensions.

this type of abstraction will benefit any wavefront-type code that performs some type of spatial dimension sweep. Our OpenACC proof-of-concept of such an abstraction demonstrates this on a real-world wavefront-type application used at a major national laboratory. Since these types of codes are very common in computational scientific applications, we contend that this contribution has far-reaching implications for modern-day HPC applications.

Figure 7 takes our work a step further by introducing the additional layer of MPI communication into the KBA sweep component of the code. Using MPI, we can decompose the spatial domain across nodes, and in the case where a GPU architecture is targeted, across devices within a node. Here, we observe similar behavior to the results shown in Figure 5, as it relates to runtimes of our different configurations. It is worth noting that since we changed over from a single-directional sweep to Minisweep's true multi-directional octant sweeping method using asynchronous parallel regions, we lose a bit of performance when compiling for a multicore CPU target. This is due to the fact that PGI's OpenACC compiler will parallelize *gang* loops by default when compiled with a multicore target. The optimal configuration for Minisweep would be to actually parallelize our in-gridcell loops

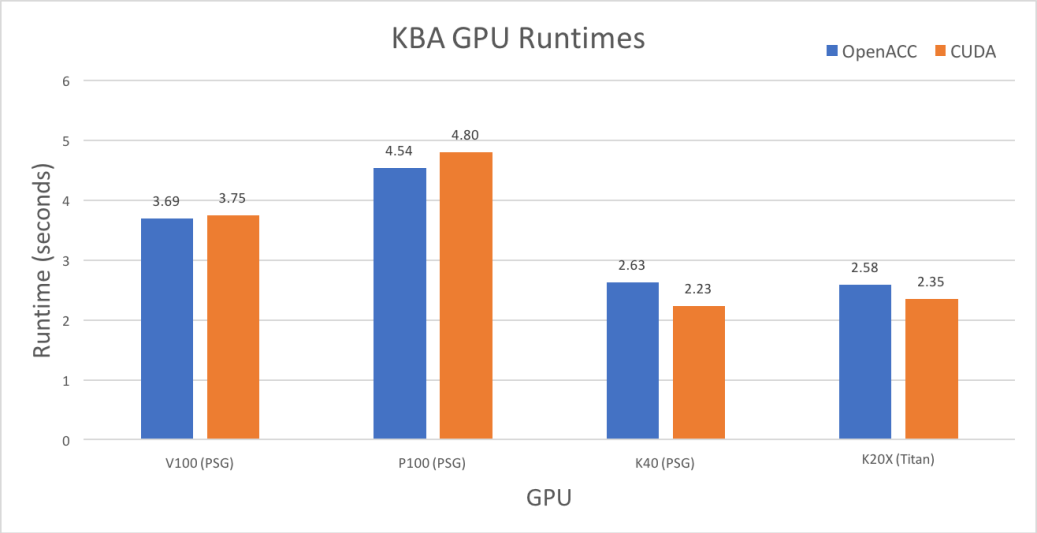

Figure 6: Absolute runtimes (sec) of OpenACC & CUDA experiments on all GPUs used. Note that the V100/P100 problem size is an order of magnitude > K40/K20x configuration, as mentioned earlier.

(currently annotated as *vector* parallel) because we have enough parallelism inside our gridcells to fully saturate most CPUs. Since we are unable to do this currently, we see OpenMP outperforming our OpenACC configuration. However, our GPU configurations yield very fruitful results when run across all 16 GPUs across 4 PSG nodes using 16 MPI ranks. Each rank is bound to a different GPU using the *acc_set_device_num* function. Here, we see OpenACC continuing to outperform CUDA, which serves as evidence that our OpenACC configuration will scale very well on larger, modern HPC systems.

We ran our GPU configurations of Minisweep on Summit itself. Constraints in accessing the system have limited the ability to collect complete results, hence we have not added our preliminary findings.

Scientifically, neutron flow simulation can take a considerable amount of computing time. If not for speeding up these simulation runs using accelerators, scientists will have to resort to simulating a model that is potentially less accurate leading to questionable results. This can be quite a risky task to rely on for scientists working in nuclear reactor facilities.

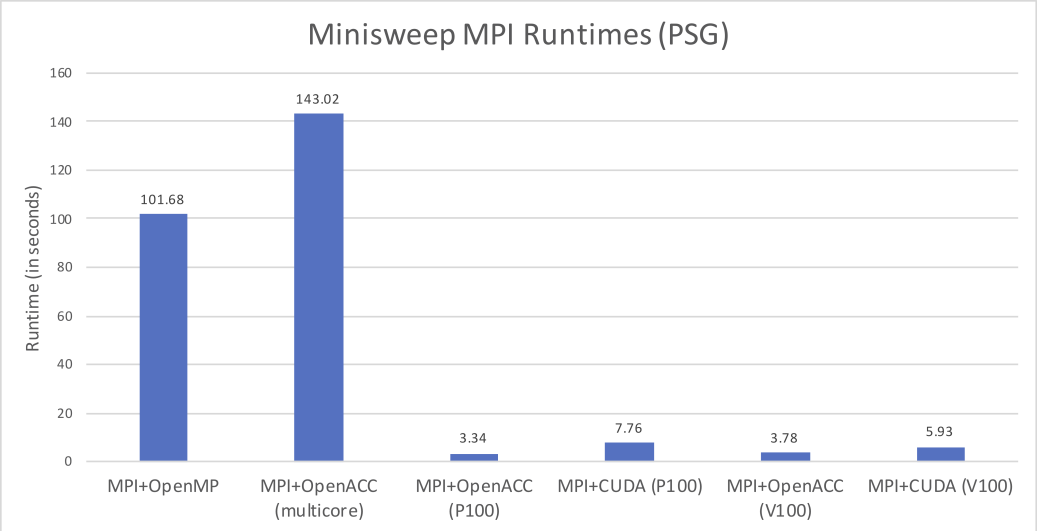

Figure 7: Minisweep's runtimes when running on 4 nodes (each with 4 GPUs) using 16 MPI ranks (1 rank per GPU). Lower is better. Note that the runtimes of OpenACC and CUDA are comparable even when run on multiple nodes. This reinforces our conclusions drawn from Figure 5.

8. Related Work

Considered as far back as 1974 by Lamport [26], wavefront computations are found in linear equation solvers [27, 28], gene sequence alignment [29] and radiation transport [21, 30], iterative solution methods [31], particle physics simulations [32], and parallel solution of triangular systems of linear equations [33]. Smith Waterman, a local sequence alignment algorithm, has mapped wavefront algorithm to GPUs [34], on Cell BE [35] and on FPGAs [36, 37]. ASCI Sweep3D wavefront application undergoes rapid succession of wavefront and solves a 1-group neutron transport problem on IBM Blue Gene/P machine [38] by using blocking techniques. Preliminary studies to use TBB, Cilk, CnC, and OpenMP 3.0 for wavefront in [39] indicate that a higher-level template is required for less experienced users. AWE Chimaera [40], NAS-LU [41] use a variation called 'hyperplane' algorithm [42] and [43] discusses acceleration of generalized pipeline wavefront applications.. Proxy apps such as Kripke [13], SNAP [14] (mimicking communication patterns of PARTISN [15] transport code) are wavefront codes investigating different data layout patterns and parallelism. Coarray Fortran-based Sweep3D's comparable performance to that of the MPI is discussed in [44].

Approaches using loop skewing [45] to derive the wavefront method of execution of nested loops is discussed in CHiLL [46], a polyhedral compiler transformation framework. Other related work include [47, 48]. Other approaches are High Productive Computing System (HPCS) languages: Chapel [23], X10 [49] and Fortress [50] but do not offer enough abstractions or vocabulary for heterogeneous platforms. HTP [22] proposed a hierarchical tree place to map to an architecture and schedules task to different different nodes in the tree.

However most of the above discussed strategies are solutions to specific problem types, or incurs steep learning curve thus making them not quite so favorable to easily adapt for large scale applications. Taking this up as a challenge, our work proposes solution using directives and demonstrate the parallelization of a multi-dimensional Minisweep using OpenACC on different platforms. The parallelization process revealed shortcomings in current directive-based model that we address by proposing an abstract model for expressing wavefront parallelism in programming models.

9. Conclusion & Future Work

This paper examines the challenges faced when porting a wavefront application to state-of-the-art HPC systems using directives using Minisweep as the case study. We present a performance-portable OpenACC implementation of Minisweep, as well as an analysis of this implementation's performance compared to optimized multicore CPU and GPU implementations of the same code using OpenMP and CUDA, respectively. Results demonstrate that utilizing high-level parallel programming abstractions, such as OpenACC, can achieve comparable performance to low-level, optimized parallel implementations. Our results also demonstrate the scalability of our solution by using MPI to compare the performance of our configurations when run across multiple nodes, and in the case of GPU configurations, accelerators within those nodes. Combining MPI for domain decomposition with OpenACC for device offloading yields favorable speedups when compared to MPI+CUDA configurations on state-of-the-art HPC systems. All of these designs and implementations are reflections of an abstract parallelism model that we propose for wavefront algorithms. This model motivates enhancing programming models with software abstractions to parallelize wavefront problems on multiple platforms, while minimizing programmer overhead.

As part of our ongoing work, we plan on applying our findings outlined

by this work on additional applications to further demonstrate the efficacy of our abstract parallelism model, formulating and automating the process of transforming loop nests into the KBA method discussed in Section 3, and investigating additional techniques for performing wavefront sweeps across multiple GPUs and/or nodes in HPC systems. As near future work, we will analyze results out of Summit and port the code to OpenMP 4.5.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of science, and this research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This material is based upon work supported by the National Science Foundation (NSF) under grant no. 1814609.

The authors would also like to thank NVIDIA for donating us a Titan Xp GPU, Tesla P40 and V100 GPUs, and providing us access to their Professional Service Group (PSG) machine that we have used for this work.

10. References

References

[1] B. Messer, E. D'Azevedo, J. Hill, W. Joubert, M. Berrill, C. Zimmer, Miniapps derived from production hpc applications using multiple programing models, The International Journal of High Performance Computing Applications 0 (0) (2016) 1094342016668241. arXiv:http://dx.doi.org/10.1177/1094342016668241, doi:10.1177/1094342016668241.
URL `http://dx.doi.org/10.1177/1094342016668241`

[2] Ornl-cees, `https://github.com/ORNL-CEES/Profugus` (2017).

[3] T. M. Evans, W. Joubert, S. P. Hamilton, S. R. Johnson, J. A. Turner, G. G. Davidson, T. M. Pandya, Three-Dimensional Discrete Ordinates Reactor Assembly Calculations on GPUs, in: ANS 2015, 2015.

[4] Quadrillions of calculations per second for fusion, ITER Newsline`https://www.iter.org/ajax/www/pop/wd_700/lang_/urldepth_0/id_newsline_ofinterest-670` [Online; accessed 15-August-2017].

[5] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. Kelly, et al., Abstract machine models and proxy architectures for exascale computing, in: Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014, IEEE, 2014, pp. 25–32.

[6] J. Shalf, Computer architecture for the next decade: Adjusting to the new normal for computing, EEHPC Workshop.

[7] R. Searles, S. Chandrasekaran, W. Joubert, O. Hernandez, Abstractions and directives for adapting wavefront algorithms to future architectures, in: The Platform for Advanced Scientific Computing (PASC), 2018, ACM, 2018.

[8] E. W. Larsen, J. E. Morel, Advances in discrete-ordinates methodology, in: Y. Azmy, E. Sartori (Eds.), Nuclear Computational Science: A Century in Review, Springer, New York, 2010, Ch. 1, pp. 1–84.

[9] Supercomputer team wins award for core work, World Nuclear News. http://www.world-nuclear-news.org/NN-Supercomputer-team-wins-award-for-core-work-0707147.html [Online; accessed 15-August-2017].

[10] C. G. Baker, G. G. Davidson, T. M. Evans, S. P. Hamilton, J. J. Jarrell, W. Joubert, High performance radiation transport simulations: Preparing for TITAN, in: Proceedings of Supercomputing Conference SC12, 2012.

[11] Scientists successfully test code that models neutrons in reactor core, R&D Magazinehttps://www.rdmag.com/news/2014/02/scientists-successfully-test-code-models-neutrons-reactor-core.

[12] The ASCI SWEEP3D README filehttp://www.ccs3.lanl.gov/PAL/software.shtml [Online; accessed 24-June-2014].

[13] Kripke. https://codesign.llnl.gov/kripke.php [Online; accessed 15-August-2017].

[14] Snap: Sn (discrete ordinates) application proxy. https://github.com/lanl/SNAP [Online; accessed 15-August-2017].

[15] R. S. Baker, Partisn on advanced/heterogeneous processing systemshttp://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-13-20948 [Online; accessed 24-June-2014].

[16] S. Sathe, Accelerating the ANSYS fluent r18.0 radiation solver with openacc, https://tinyurl.com/yacxh5g7 (2016).

[17] R. Gomperts, Quantum Chemistry on GPUs, http://images.nvidia.com/content/tesla/pdf/quantum-chemistry-may-2016-mb-slides.pdf (2016).

[18] W. Sawyer, G. Zaengl, L. Linardakis, Towards a multi-node openacc implementation of the icon model, in: EGU General Assembly Conference Abstracts, Vol. 16, 2014.

[19] M. P. Clay, D. Buaria, P. K. Yeung, Improving scalability and accelerating petascale turbulence simulations using openmp, http://openmpcon.org/conf2017/program/, to Appear (2017).

[20] D. F. Richards, R. C. Bleile, P. S. Brantley, S. A. Dawson, M. S. McKinley, M. J. O?Brien, Quicksilver: A proxy app for the monte carlo transport code mercury, in: Cluster Computing (CLUSTER), 2017 IEEE International Conference on, IEEE, 2017, pp. 866–873.

[21] K. Koch, R. Baker, R. Alcouffe, Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor, Transactions of the American Nuclear Society 65 (1992) 198–199.

[22] Y. Yan, J. Zhao, Y. Guo, V. Sarkar, Hierarchical place trees: A portable abstraction for task parallelism and data movement., in: LCPC, Vol. 10, Springer, 2009, pp. 172–187.

[23] B. L. Chamberlain, D. Callahan, H. P. Zima, Parallel programmability and the chapel language, The International Journal of High Performance Computing Applications 21 (3) (2007) 291–312.

[24] W. Joubert, Minisweep, `https://github.com/wdj/minisweep` (2017).

[25] O. R. N. Lab, Summit, `https://www.olcf.ornl.gov/summit/` (2017).

[26] L. Lamport, The parallel execution of DO loops, Communications of the ACM 17(2) (1974) 83–93.

[27] R. Li, Y. Saad, GPU-accelerated preconditioned iterative linear solvers, Journal of Supercomputing 63(2) (2013) 443–466, `http://link.springer.com/article/10.1007\%2Fs11227-012-0825-3` [Online; accessed 24-June-2014]. doi:10.1007/s11227-012-0825-3.

[28] S. Pennycook, S. Hammond, G. Mudalige, S. Wright, S. Jarvis, On the acceleration of wavefront applications using distributed many-core architectures, The Computer Journal 55 (2) (2012) 138–153, `http://comjnl.oxfordjournals.org/content/55/2/138` [Online; accessed 24-June-2014]. doi:10.1093/comjnl/bxr073.

[29] T. F. Smith, M. S. Waterman, Identication of common molecular subsequences, Journal of Molecular Biology 147(1) (1981) 195–197.

[30] C. Baker, G. Davidson, T. M. Evans, S. Hamilton, J. Jarrell, W. Joubert, High performance radiation transport simulations: preparing for titan, in: SC 2012 International Conference for, IEEE, 2012, pp. 1–10.

[31] F. Roddier, C. Roddier, Wavefront reconstruction using iterative fourier transforms, Applied Optics 30 (11) (1991) 1325–1327.

[32] K. R. Koch, R. S. Baker, R. E. Alcouffe, Solution of the first-order form of the 3-d discrete ordinates equation on a massively parallel processor, Transactions of the American Nuclear Society 65 (108) (1992) 198–199.

[33] M. T. Heath, C. H. Romine, Parallel solution of triangular systems on distributed-memory multiprocessors, SIAM Journal on Scientific and Statistical Computing 9 (3) (1988) 558–588.

[34] E. F. d. O. Sandes, A. C. M. de Melo, Smith-waterman alignment of huge sequences with gpu in linear space, in: IPDPS, 2011 IEEE International, IEEE, 2011, pp. 1199–1211.

[35] A. Wirawan, K. C. Keong, B. Schmidt, Parallel dna sequence alignment on the cell broadband engine, in: International Conference on PPAM, Springer, 2007, pp. 1249–1256.

[36] P. Zhang, G. Tan, G. R. Gao, Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform, in: Proc. of HPRCTA: held in conjunction with SC07, ACM, 2007, pp. 39–48.

[37] S. Chandrasekaran, S. Shanbagh, R. Jayaraman, D. L. Maskell, H. Y. Cheah, C2fpga?a dependency-timing graph design methodology, JPDC 73 (11) (2013) 1417–1429.

[38] A. S. C. Initiative, The asci sweep3d benchmark code (1995).

[39] A. J. Dios, A. G. Navarro, R. Asenjo, F. Corbera, E. L. Zapata, A case study of the task-based parallel wavefront pattern., in: PARCO, 2011, pp. 65–72.

[40] G. R. Mudalige, M. K. Vernon, S. A. Jarvis, A plug-and-play model for evaluating wavefront computations on parallel architectures, in: IPDPS. IEEE International Symposium on, IEEE, 2008, pp. 1–14.

[41] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al., The nas parallel benchmarks, The International Journal of Supercomputing Applications 5 (3) (1991) 63–73.

[42] L. Lamport, The parallel execution of do loops, Communications of the ACM 17 (2) (1974) 83–93.

[43] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Wright, S. A. Jarvis, On the acceleration of wavefront applications using distributed many-core architectures, The Computer Journal 55 (2) (2011) 138–153.

[44] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, Experiences with sweep3d implementations in co-array fortran, The Journal of Supercomputing 36 (2) (2006) 101–121.

[45] M. Wolfe, Loops skewing: The wavefront method revisited, International Journal of Parallel Programming 15 (4) (1986) 279–293.

[46] C. Chen, J. Chame, M. Hall, Chill: A framework for composing high-level loop transformations, Tech. rep., Technical Report 08-897, U. of Southern California (2008).

[47] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, I. T. Foster, Swift/t: Large-scale application composition via distributed-memory dataflow processing, in: Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on, IEEE, 2013, pp. 95–102.

[48] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, M. Hall, Automating wavefront parallelization for sparse matrix computations, in: Proc. of SC16, IEEE Press, 2016, p. 41.

[49] J. Milthorpe, V. Ganesh, A. P. Rendell, D. Grove, X10 as a parallel language for scientific computation: Practice and experience, in: IPDPS, 2011 IEEE International, IEEE, 2011, pp. 1080–1088.

[50] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al., The fortress language specification, Sun Microsystems 139 (140) (2005) 116.