MDPI

Article

The Two-Dimensional Swept Rule Applied on Heterogeneous Architecture

Anthony Walker and Kyle E. Niemeyer

School of Mechanical, Industrial, and Manufacturing Engineering
Oregon State University
Corvallis, Oregon, USA

Abstract: The partial differential equations describing compressible fluid flows can be notoriously difficult to resolve on a pragmatic scale and often require the use of high performance computing systems and/or accelerators. However, these systems face scaling issues such as latency, the fixed cost of communicating information between devices in the system. The swept rule is a technique designed to minimize these costs by obtaining a solution to unsteady equations at as many possible spatial locations and times prior to communicating. In this study, we implemented and tested the swept rule for solving two-dimensional problems on heterogeneous computing systems across two distinct systems. Our solver showed a speedup range of 0.22–2.71 for the heat diffusion equation and 0.52–1.46 for the compressible Euler equations. We can conclude from this study that the swept rule offers both potential for speedups and slowdowns and that care should be taken when designing such a solver to maximize benefits. These results can help make decisions to maximize these benefits and inform designs.

Keywords: Latency; GPU; CPU; PDE

Citation: Walker, A.; Niemeyer, K.E. The Two-Dimensional Swept Rule Applied on Heterogeneous Architecture. *Journal Not Specified* **2021**, *1*, 0. https://doi.org/

Received:
Accepted:
Published:

1. Introduction

Partial differential equations (PDEs) are used to model many important phenomena in science and engineering. Among these, fluid mechanics and heat transfer are which can be notoriously difficult to solve on pragmatic scales. Wildfire is a good example of an expensive numerical simulation because it involves fluid mechanics, heat transfer, and chemical reactions over massive areas of impact. Wildfire is also an unsteady phenomenon—so it changes over time. The ability to accurately model and predict the properties of such phenomena in real time would allow for better responses, but there are many challenges associated with making these predictions.

Unsteady multi-dimensional PDEs often require using of distributed-memory computing systems to obtain a solution with practical grid resolution or scale in a reasonable time frame. Advancing the solution at any point in the grid inherently depends on the neighboring points in each spatial dimension. This dependence requires communication between computing nodes, to transfer data associated with the boundary locations. Each of these communications incurs a minimum cost regardless of the amount of information communicated—this is network latency. In contrast, bandwidth is the variable cost associated with the amount of data transferred. The total latency cost can significantly restrict the solution's time to completion, especially when using distributed systems. This barrier to scaling is referred to as the "latency barrier" and is impacts large-scale simulations that involve advancing many time steps, i.e., ones that require a large amount of communication [1]. The barrier is a bottle neck in the system which can limit the performance regardless of the architecture.

Graphics processing units (GPUs) are powerful tools for scientific computing because they are well suited for parallel applications and large quantities of simultaneous calculations. Modern computing clusters often have GPUs in addition to CPUs because

of their potential to accelerate simulations and, similar to CPUs, they perform best when communication is minimized. These modern clusters are often referred to as heterogeneous systems or systems with multiple processor types—CPUs and GPUs, in this case. Ideally, a heterogeneous application will minimize communication between the GPU and CPU which effectively minimizes latency costs [2]. Minimizing latency in high performance computing is one of the barriers to exascale computing which requires the implementation of novel techniques to improve [3].

This study presents our implementation and testing of a two-dimensional heterogeneous solver for unsteady partial differential equations that employs a technique to help overcome the latency barrier—the swept rule [1]. In other words, the swept rule is a latency reduction technique that focuses on obtaining a solution to unsteady PDEs at as many possible locations and times prior to communicating with other computing nodes (ranks). In this article, we first discuss related work, distinguish our work from prior swept studies, and provide more motivation in section 1. Next, we describe implementation details, objectives, study parameters, design decisions, methods, and tests in section 2. As expected, this is followed with results and conclusions in sections 3 and 5.

Surmounting the latency barrier has been approached in many ways; the most closely related to this study include prior swept rule studies which involve multiple dimensions and architectures but not the combination of the two [1,4–6]. Parallel-in-time methods are also related to the swept rule but differ in the sense that they iteratively parallelize the temporal direction, while the swept rule minimizes communication [7]. There are many examples of parallel-in-time methods which are all variations of the core concept [8–14]. For example, Parareal is one of the more popular parallel-in-time methods; it works by iterating over a series of coarse and fine grids with initial guesses to parallelize the problem [9]. However, there are some stability concerns with Parareal that are addressed by local time integrators [11]. These methods have the same goal but achieve that goal differently. Similarly, cache optimization techniques have the same objective, but they achieve it differently by optimizing communication not avoiding it [15].

Finally, communication avoidance techniques closely emulate the swept rule but involve overlapping parts or redundant operations. The GPU implementation particularly blurs this difference because it solves an extra block to avoid intra-node communication but no extra blocks are solved for inter-node communication. The swept rule also differs because it particularly focuses on solving PDEs. There are varying degrees of communication algorithms that tend to focus on linear algebra [16–20].

The swept rule was originally developed by Alhubail et al [1]. who first developed a one-dimensional CPU-only swept PDE solver which was tested by solving the Kuramoto-Sivashinsky equation and the compressible Euler equations. They concluded that in each case a number of integrations can be performed during the latency time of a communication. Their analysis showed that integration can be made faster by latency reduction and increasing computing power of the computing nodes [1]. Alhubail et al. followed this work with a two-dimensional CPU only swept PDE solver which reported speedups of up to three times compared to classical methods when solving the wave and Euler equations [4]. These studies differ from our study most prominently by the dimensionality and intended architecture.

Magee et al. created a one-dimensional GPU swept solver and a one-dimensional heterogeneous solver and applied both to solving the compressible Euler equations [5,6]. They concluded that their shared memory approach typically performed better than alternative approaches, but speedup was not obtained in all cases for either study. Varying performance results were attributed to greater usage of lower-level memory, which limits the performance benefits of the swept rule depending on the problem [5]. Our current study extends upon the swept rule for heterogeneous architectures, but it differs in the dimensionality. Our implementation also attempts to use and extend some of the implementation strategies that showed promise in the aforementioned studies.

The effect of added dimensionality on performance is a pragmatic interest and can be considered from multiple perspectives. The primary goal is speeding up simulations requiring high performance computing by reducing network latency. The swept rule is motivated by reducing the time to obtain solutions of problems involving complicated phenomena frequently requiring the use of high performance computing systems. While many simplifications exist to reduce the dimensionality of fluid dynamics problems, most realistic problems are three-dimensional. Our solver is a step towards more realistic simulations by considering two spatial dimensions, which can provide insight into multi-dimensional performance and constraints. This insight can offer the chance to optimize system usage and promote faster design and prototype of thermal fluid systems.

In the event that computation time is not the primary concern, available resources or resource costs are other important considerations. The ability to execute a simulation on a high performance computing system depends on access to such systems. In the case of Amazon EC2, simulation time can be purchased at different hourly rates depending on the application [21]. This quickly becomes expensive for applications that require large numbers of computing hours. Network latency is aggrandized in applications requiring a lot of communication because each communication event takes a finite amount of time regardless of the data size. It is also possible to obtain and show performance benefits on smaller systems. This claim is supported by findings from Magee et al.; they showed speedup on a work station with a single GPU and CPU [5]. While this is not the primary focus, an optimized solver that reduces latency would require less computing resources and more practical applications could potentially be solved on smaller, less costly computing clusters. Hopefully, it is clear at this point that latency reduction is important in high performance computing and scientific applications as this is the intention of this work.

2. Materials and Methods

2.1. Implementation & Objectives

We call our implementation of the two-dimensional swept rule `PySweep`[1]—it consists of two core solvers: `Swept` and `Standard`. `Swept` minimizes communication during the simulation via the swept rule. `Standard` is a traditional solver that communicates as is necessary to complete a timestep, and serves as a baseline to the swept rule. Both solver use the same decomposition, process handling, and work allocation code so that a performance comparison between them is representative of swept rule performance. However, `Swept` does require additional calculations prior to solving which are penalties of this swept rule implementation.

We implemented `PySweep` using Python and CUDA; the parallelism relies primarily on `mpi4py` [22] and `pycuda` [23]. Each process spawned by MPI is capable of managing a GPU and a CPU process, e.g., 20 processes can handle up to 20 GPUs and 20 CPU processes. Consequently, the aforementioned implementation allowed us to meet the objectives of this study on the swept rule, which include understanding:

1. its performance on distributed heterogeneous computing systems,
2. its performance with simple and complex numerical problems on heterogeneous systems,
3. the impact of different computing hardware on its performance, and
4. the impact of input parameters on its performance.

2.2. Parameters & Testing

GPUs execute code on a "block-wise" basis, i.e., they solve all the points of a given three-dimensional block simultaneously. We refer to the dimensions of these blocks as `block size` or b—a single integer that represents the x and y dimension. The z dimension of the block was always unity because the solver is two-dimensional. The

[1] `PySweep`is openly available at https://github.com/anthony-walker/pysweep-git

block size is a parameter of interest because it affects the performance of the swept rule by limiting the number of steps between communications. It also provides a natural basis for the decomposition of the data and imposes some further restrictions in the process.

The swept solution process restricts the block size to the interval $(2n, b_{max}]$ where b_{max} is the maximum block size allowed by the hardware and n is the maximum number of points on either side of any point j used to calculate the derivatives. We defined the b as Equation 1 because it allowed us to prove that the maximum number of steps in time based on the stencil is $k-1$ steps. It also allowed us to prove that the block's shortest dimension limits the number of steps for the swept rule. As such, we also restricted block size to being square ($x = y$).

$$b = 2nk, \quad k \in \mathbb{Z}^+ \ni 2n < b \leq b_{max} \tag{1}$$

Blocks provide a natural unit for describing the amount of work, e.g, an 16×16 array has four 8×8 blocks to solve. As such, the work load of each process' GPU and CPU was determined by the GPU share, s, on a block-wise basis. A share of 1 corresponds to the GPU handling 100% of the work. This is expressed mathematically in equation 2 where G, C, and W represent the number of GPU blocks, CPU blocks, and total blocks respectively. This parameter is of interest because it directly impacts the performance of the solvers.

$$s = \frac{G}{W} = 1 - \frac{C}{W} \tag{2}$$

In this implementation, the share does not account for the number of GPU or CPU cores available but simply divides the given input array based on the number of blocks in the x direction e.g. if the array contains 10 blocks and $s = 0.5$ then 5 blocks will be deemed as GPU work and the remainder as CPU work. These portions of work would then be divided amongst available resources of each type.

Array size is another parameter of interest because it demonstrates how performance scales with problem size. We restricted them to being evenly divisible by the block size for ease of decomposition. Square arrays were also only considered—so array size is represented by a single number which is the number of points in the x and y directions. Finally, array sizes were chosen as multiples of the largest block size; this can result in unemployed processes if there are not enough blocks. Potential for unemployed processes is, however, consistent across solvers and still provides a valuable comparison.

The numerical scheme used to solve a problem directly affects the performance of the swept rule as it limits the number of steps that can be taken in time (t) before communication. As such, the aforementioned parameters were applied to solving the unsteady compressible Euler equations and the unsteady heat diffusion equation as was done in other works regarding the swept rule [1,4–6]. The compressible Euler equations were applied to the isentropic Euler vortex and solved using a second order Runge-Kutta in time and a five point finite volume method in each spatial direction with a minmod flux limiter and Roe approximate Reimann solver [24,25]. The heat diffusion equation was applied to an analytical problem and solved using forward Euler method in time and a three point finite difference in each spatial direction. Further details on these methods and the associated problems is provided in A and B.

To summarize the performance parameters, array size, block size, share, and the hardware were all varied in this study. Array sizes of [320, 480, 640, 800, 960, 1120] were used to make the total number of points span a couple orders of magnitude. Block sizes of [8, 12, 16, 24, 32] were used based on hardware constraints. Share was varied from 0 to 100% at intervals of 10%. Along with these parameters, we advanced each problem 500 time steps on two different sets of hardware.

Hardware was selected based on the available resources at Oregon State University to analyze performance differences of the swept rule based on differing hardware. `PySweep` was tested separately with 32 processes on two sets of hardware. The first two

nodes each have Nvidia Tesla V100-DGXS-32GB GPUs and Intel E5-2698v4 CPUs, and the second two nodes each have Nvidia GeForce GTX 1080 Ti GPUs and Intel Skylake Silver 4114 CPUs. As a convention, parameters with respect to the first set of hardware will be referred to with a subscript 1 and likewise a subscript 2 for the second set. Further information on these devices is located in C or the sources [26–29]. Each of these sets was used to solve both the compressible Euler equations and heat diffusion equation for 500 time steps. The performance data we collected from this evaluation is presented in section 3.

The results of this study raised some questions about the scalability of the algorithms used. Accordingly, weak scalability was considered with up to three nodes consistent with hardware set 2. Each node was given an array size of 960 and solved both problems for 500 time steps, e.g, the case with three nodes solved an array with 2880^2 total points. A share of 80% and block size of 16 were used in this test.

2.3. Swept Solution Process

To recap, the swept rule is a latency reduction technique that focuses on obtaining a solution to unsteady PDEs at as many possible locations and times prior to communicating. Discussion of the swept rule can at times be convoluted as there are time steps, swept steps, and other actions that can be referred to as steps. To simplify this, we refer to the two primary actions of the code as communication and calculation. We refer to the specific type of calculation as the phase. Finally, we refer to individual time steps and intermediate time steps as so—such as those that occur during multi-step schemes.

We considered communication to be when the code was transferring data to other ranks. Node based rank communication between the CPU and GPUs happens via the use of shared memory; a capability implemented in MPICH-3 or later [30]. Inter-node communication was performed with typical MPI constructs such as send and receive. A shared GPU memory approach was implemented by Magee et al. which showed benefit [5]; this concept lead to the idea of using shared memory on each node for node communication and primary processes for inter-node communication.

We considered calculation to be specifically when the code was developing the solution in time. The specifics of CPU and GPU calculation depended on the particular phase the simulation was in. In general, the GPU calculation proceeded by copying the allocated section of the shared array to GPU memory and launched the appropriate kernel; the nature of GPU computing handled the rest. The CPU calculation began by disseminating predetermined blocks amongst the available processes. Each block was a portion of the shared memory which each process is allowed to modify.

There are four phases that occur during the swept solution process. In the code, we refer to them as `Up-Pyramid`, `Bridge` (`X` or `Y`), `Octahedron`, and `Down-Pyramid`—that convention is continued here. We named them this way because these are the shapes produced during the solution procedure if you visualize the solution's progression in 3 dimensions (x,y,t); this is demonstrated in later figures for a general case. However, the specific shape that forms in each phase is a function of the spatial stencil and time scheme chosen to solve a given problem. As mentioned in section 2.2, the maximum steps of the swept rule based on block size is $k-1$. The height or number of steps in time that each phase except `Octahedron` can take is defined by equation 3. The `Octahedron` phase is of height $2h$.

$$h = k - 1 \tag{3}$$

The first phase of calculation is the `Up-Pyramid` shown in Figure 1 (top left). A dynamic programming approach was used on the CPU portion to calculate the `Up-Pyramid` in lieu of conditional statements. Specifically, the indices to develop the `Up-Pyramid` were stored in a set which was accessed as needed. The GPU portion implemented a conditional statement to accomplish this same task because its speed typically dwarfed that of the CPU regardless. If the code were to eventually be optimized, it could be beneficial to consider a dynamic programming approach such as that implemented on

the CPU. In both cases, conditional or dynamic, the code removed $2n$ points from each boundary after every time step. If the time scheme required multiple intermediate time steps, these points were removed after each intermediate step. The initial boundaries of the `Up-Pyramid` were the given block size and the final boundaries were found as $2n$ using equation 1 for $k = 1$.

The next phase in the process is referred to as the Bridge which occurs independently in each dimension. The solution procedure for the bridges on the CPU and GPU follows the same paradigm as the `Up-Pyramid` but twice as many bridges are formed because there is one in each direction. The dimension in which the Bridge grows as time passes is its reference dimension e.g., the Bridge that grows in the y dimension as time progresses is referred to as the `Y-Bridge`. The initial boundaries of each Bridge were determined from n and b. The Bridges grew by $2n$ from these initial boundaries in their reference dimensions and shrank by $2n$ in the other dimension which allowed the appropriate calculation points to be determined conditionally or prior to calculation.

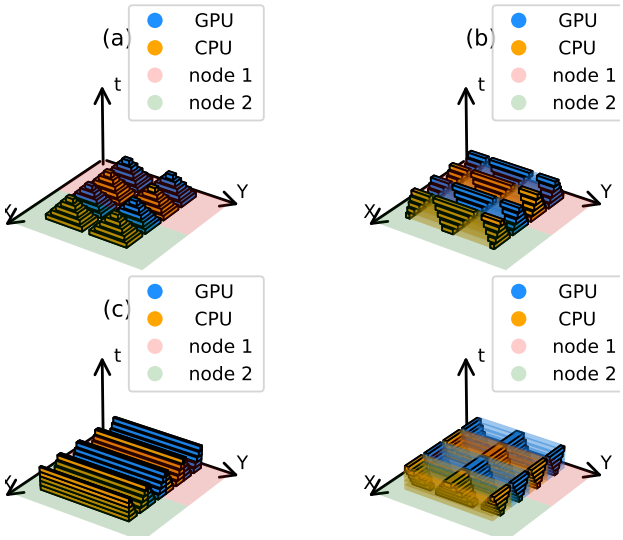

Figure 1. The first four steps in the swept solution process: `Up-Pyramid` (top left), `Y-Bridge` (top right), `Communication` (bottom left), `X-Bridge` (bottom right)

Depending on the decomposition strategy, a large portion of the Bridges can be performed prior to inter-node communication. In this implementation, the `Y-Bridge`—which is shown in Figure 1 (top right)—was computed prior to the first communication. The first inter-node communication then occurred by shifting nodal data $b/2$ points in the positive x direction. Any data that exceeded the boundaries of its respective shared array was communicated to the appropriate adjacent node. This process is demonstrated in Figure 1 (bottom left). The shift in data allowed previously determined sets and blocks to be used in the upcoming calculation phases; so the `X-Bridge` proceeded directly after the communication as shown in Figure 1 (bottom right).

The first three phases in the swept solution process demonstrated in Figure 1 were followed by the `Octahedron` phase shown in Figure 2 (top left). This phase is a superposition of the `Down-Pyramid` and `Up-Pyramid`. The `Down-Pyramid`—shown in Figure 2 (bottom right)–always began with boundaries that were $2n$ and grew by $2n$ on each boundary with every passing time step. The start was a natural consequence of removing these points during the `Up-Pyramid` phase. The `Down-Pyramid` was completed upon passing the top of the previous `Up-Pyramid` or `Octahedron` at which time the upward portion of `Octahedron` phase began. The entire `Octahedron` was calculated in the same fashion as the `Up-Pyramid` on both CPUs and GPUs. While the steps are described separately for clarity, they were performed in a single calculation step without communication between ranks.

The `Octahedron` was always followed by the `Y-Bridge`, Communicate, `X-Bridge` sequence. However, the communication varied in direction as the shift and communication of data was always the opposite of the former communication. We repeated this series of events as many times as was necessary to reach the final desired time of the simulation. The final phase is the aforementioned `Down-Pyramid` which occurred only once at the end of the simulation. We show the ending sequence—minus the communication—in its entirety in Figure 2.

To summarize the progression of swept phases, the `Up-Pyramid` was calculated a single time; The Bridge and `Octahedron` phases were then repeated until the simulation reached a value greater than or equal to that of the desired final time. Finally, the `Down-Pyramid` was executed finally to fill in the remaining portion of the solution.

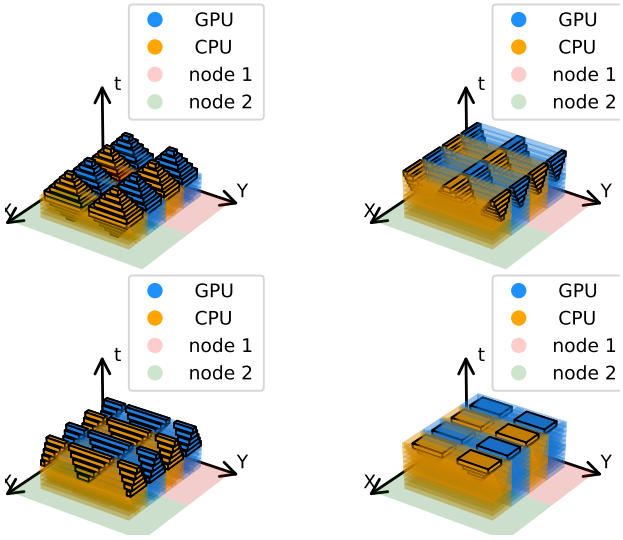

Figure 2. The intermediate and final steps of the swept solution:

The final number of time steps taken by a swept simulation was determined by the number of `Octahedron` phases ($2h$ time steps) that most accurately captured the specified number of time steps. This is a consequence of the swept rule; the exact number of steps is not always achievable in some cases because the simulation only stops after the completion of a phase. These phases occur on both the GPU and CPU with respect to the given share. In between each calculation step, a communication step occurs which consists of shared memory data management and writing to disk.

The shared memory data management of the communication step as well as the writing to disk involve a couple of nuances worth mentioning. It includes shifting of the data which is a strategy implemented for handling boundary blocks in the array. `PySweep` was implemented with periodic boundary conditions based on the targeted test problems. The boundary blocks of the array form half of the shape it would normally form in the direction orthogonal to the boundary (e.g. During the `Octahedron` phase on the boundary where $x = 0$, only half the `Octahedron` will be formed in the x direction). As expected, the corner will form a fourth of the respective shape. In lieu of added logic for handling these varying shapes, a data shifting strategy was implemented which allows the majority of the same functions and kernels to be used. The boundary points are able to be solved as if they were in the center of a block with this strategy. This strategy comes at the expense of moving the data in memory. In hindsight, changing the perspective of each process, i.e. the data it sees and manages, may be a more optimal way to implement this strategy.

`PySweep` writes to disk during every communication as it is the ideal time. The code uses parallel HDF5 (h5py) so that each rank can write its data to disk independently of other ranks [31]. The shared memory array is the height of an `Octahedron` in time plus

the number of intermediate steps of the time scheme so that the intermediate steps of the scheme may be used for future steps if necessary. The appropriate fully solved steps are written to disk. The data is then moved down in the time dimension of the array and so that the next phase can be calculated in the existing space.

In summary of this swept rule solution process, `PySweep` is an extensible PDE solver that implements the swept rule in 2 dimensions on heterogeneous architecture. It consists of two primary steps which are calculation and communication. The calculation computes points based on a predictable patterns referenced as the phases of the simulation. The communication occurs after the `Y-Bridge` forms along with shifting of the data so that the same logic may be used to calculate all of the points in the array. This shifting strategy provides simple communication logic at a minimal expense. The four phases—`Up-Pyramid`, Bridge (X or Y), `Octahedron`, and `Down-Pyramid`—are the predictable shapes used to solve the problem in time. These shapes are available in Figures 1 and 2.

The solver has a few restrictions based on architecture and implementation which have been previously described. It is currently implemented for periodic boundary conditions but can be modified to suit other conditions using the same strategy. The solver is also capable of handling given CPU functions and GPU kernels so that it may be used for any desired application that falls within the guidelines presented here. `PySweep` could certainly be further optimized but it is presently sufficient to test and determine the performance capabilities of the two-dimensional heterogeneous swept rule in comparison to a traditional solver.

3. Results

3.1. Heat Diffusion Equation

The heat diffusion equation was solved numerically using Forward Euler in time and a three point central difference in space. We validated it against an analytical solution developed for the two-dimensional heat diffusion equation with periodic boundary conditions—the problem formulation is shown in Appendix A. The numerical and analytical solutions are graphically compared in Figure 3 and behave as expected.

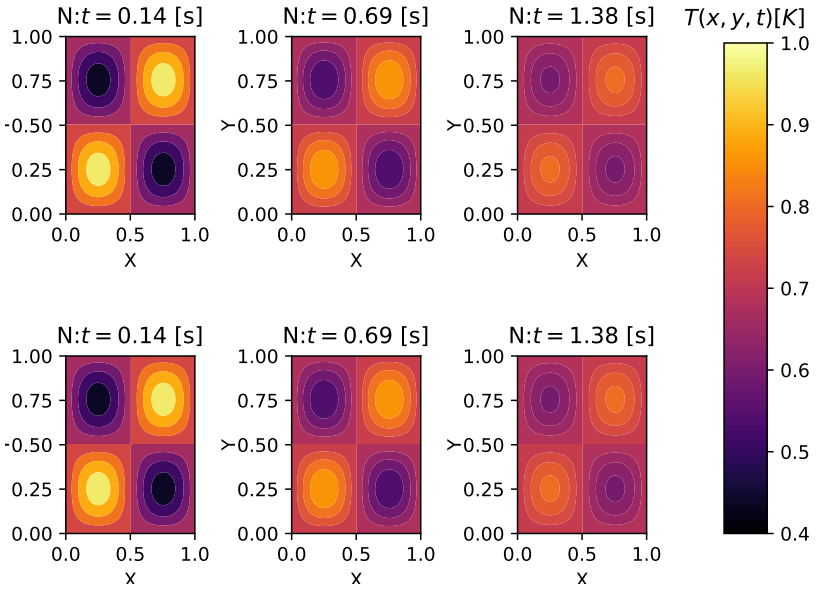

Figure 3. Heat equation contours over 2000 time steps

Our performance metric of choice for the swept rule was speedup, S, as a function of array size, block size, and share. It was calculated based on the run time, R_i, of a simulation i as

$$S = \frac{R_{standard}}{R_{swept}}. \tag{4}$$

The performance results produced by these simulations are shown in Figures 4 and 5 for the two sets of hardware as described in section 2.2. A third contour of speedup is shown in Figure 6 to compare the performance across the differing hardware. This speedup is found as $R_{swept,2}/R_{swept,1}$. In these figures, a black dot with a white border represents the worst case and a white dot with a black border represents the best case.

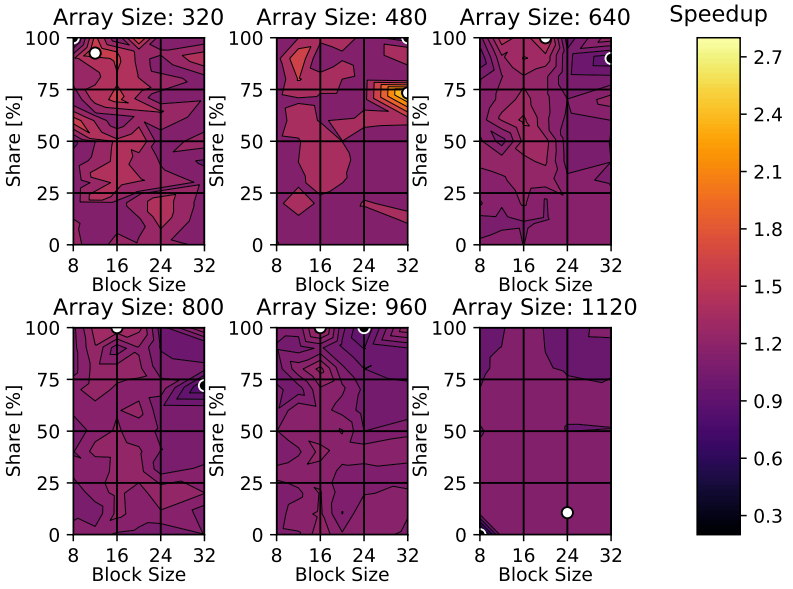

Figure 4. Swept rule speedup results for the Heat Diffusion Equation with Nvidia Tesla V100-DGXS-32GB GPUs and Intel E5-2698v4 CPUs.

In Figure 4 there is an average speedup of 1.19. It is clear that the performance of the swept rule diminishes as the array size is increased. The largest areas of performance increases generally exist above a share of 50% and along the block size of 12-20. The majority of best cases lie between 90-100% share and the 12-20 block size range. It is quite interesting that the worst cases lie very close to the optimal cases. They lie between 90-100% share but outside the block size limits 12-20.

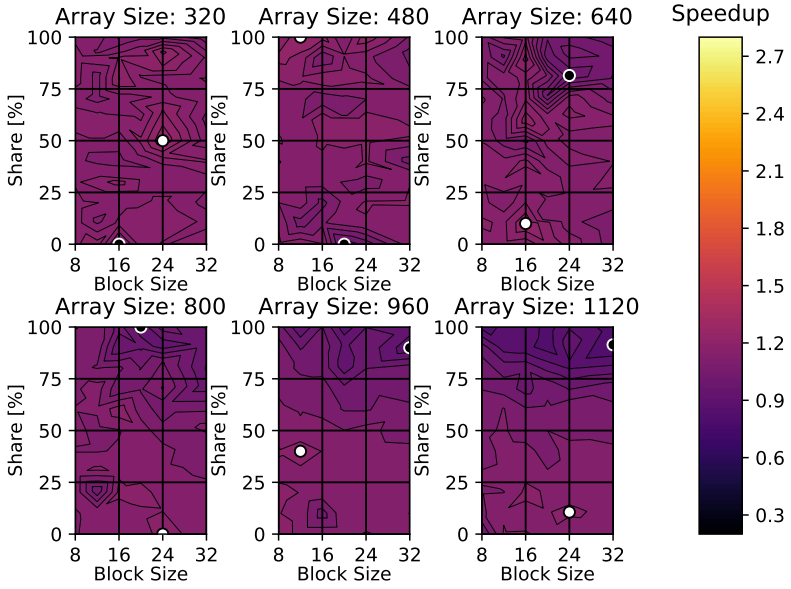

Figure 5. Swept rule speedup results for the Heat Diffusion Equation with Nvidia GeForce GTX 1080 Ti GPUs and Intel Skylake Silver 4114 CPUs.

In Figure 5, we see different trends than in Figure 4 with an average speedup of 1.11. The array size trend holds but trends based on the other two parameters are unclear. It seems that performance is better with lower shares in the case of this hardware and somewhat consistent across different block sizes. The majority of best cases occur at or below a 50% share and between block sizes 12-24. The majority of worst cases exist between 80-100% share and between block sizes of 20-32 with a couple cases being on the upper limit.

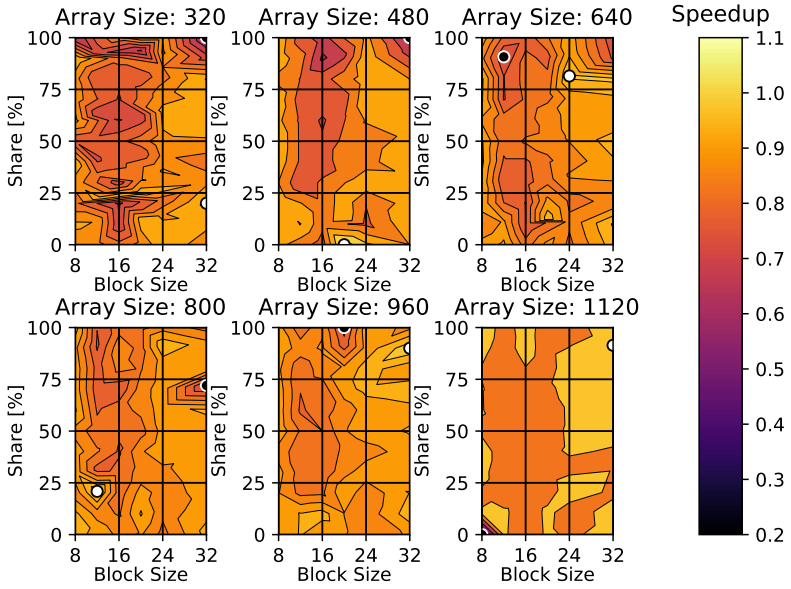

Figure 6. Hardware speedup comparison for the heat diffusion equation.

In Figure 6 we see that hardware set one is consistently slower than hardware set two with an average speedup of 0.85. Set one's best speedup is 1.08 and its worst is 0.20. Speedup is worst with shares above approximately 75% and block sizes outside 16-24. It is best above a share of 80% and below a share of 25% with a large range of block

sizes—a block size of 8 is the only point where an optimal case does not occur. However, three of the six best cases occur at a block size of 32.

3.2. Compressible Euler's Equations

The Euler equations were solved numerically using a second order Runge-Kutta in time and a five point central difference in space with a minmod flux limiter and Roe approximate Riemann solver which we validated against the isentropic Euler Vortex [24].

The same tests that we performed in Section 3.1 were repeated here. The performance results produced by these simulations are shown in Figures 8 and 9 for the two sets of hardware as described in section 2.2. A third contour of speedup is shown in Figure 10 to compare the performance across the differing hardware. The numerical and analytical solutions are graphically compared in Figure 7. The differences in this case are less apparent than in Figure 3 but, they behave as expected.

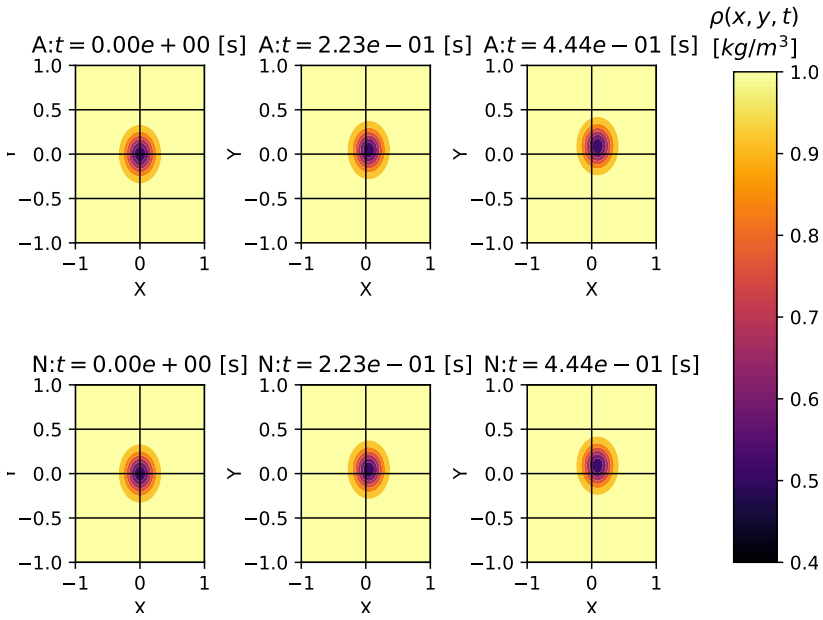

Figure 7. Euler equations contours over 500 time steps

The first hardware set results, Figure 8, show that the `Swept` solver is consistently slower than `Standard` with an average speedup of 0.95. Similar to the heat equation, performance declines with increasing array size but there is benefit in some cases. The majority of the best cases occur above approximately 90% share but there is no clear block size trend. The majority of the worst cases occur at 100% share but likewise a block size trend is unclear. However, in the three largest array sizes they always occur at 100% share with a block size of 8.

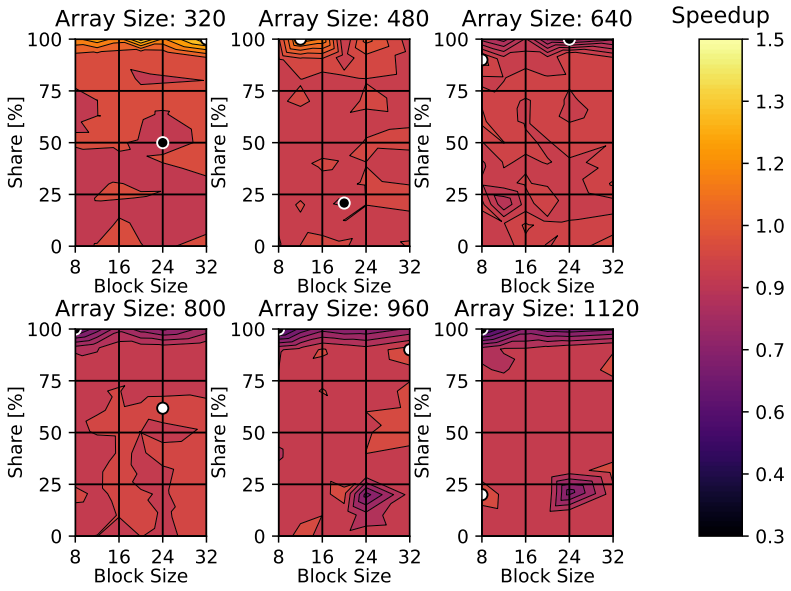

Figure 8. Swept rule speedup results for the compressible Euler equations with Nvidia Tesla V100-DGXS-32GB GPUs and Intel E5-2698v4 CPUs.

The second hardware set results, Figure 9, show that the `Swept` solver is consistently slower than `Standard` with an average speedup of 0.94. Again, performance declines with increasing array size but there is benefit in some cases. The majority of the best cases occur below approximately 50% share with a block size between 12-24. The majority of the worst cases occur at 100% share on the block size limits of 8 and 32.

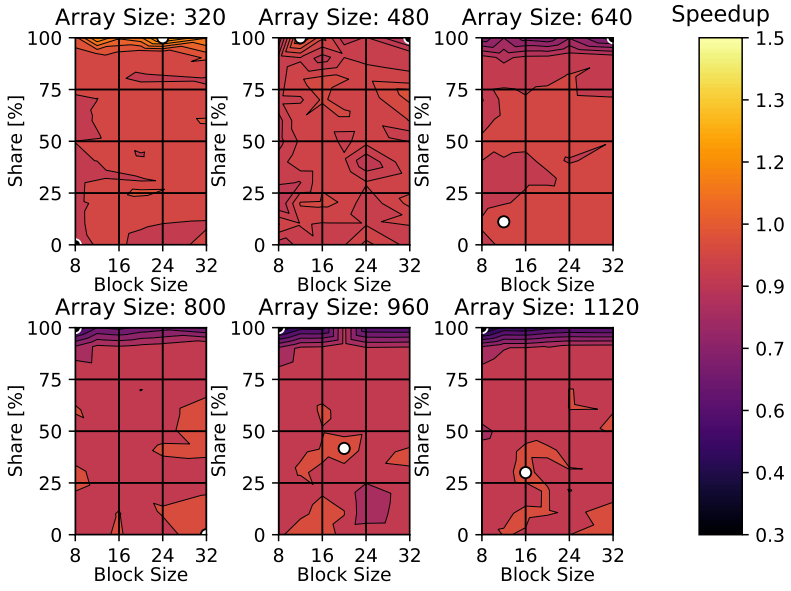

Figure 9. Swept rule speedup results for the compressible Euler equations with Nvidia GeForce GTX 1080 Ti GPUs and Intel Skylake Silver 4114 CPUs.

Figure 10 shows the comparison between the two hardware sets for the `Swept` solver. On average, the first set of hardware is slower than the second with an average speedup of 0.95. Set one's best speedup is 1.50 and its worst is 0.73. There are however regions in which performance benefits of the first set are evident. Shares above 90% across most block sizes and array sizes are evidence of this. Otherwise, the results are worse.

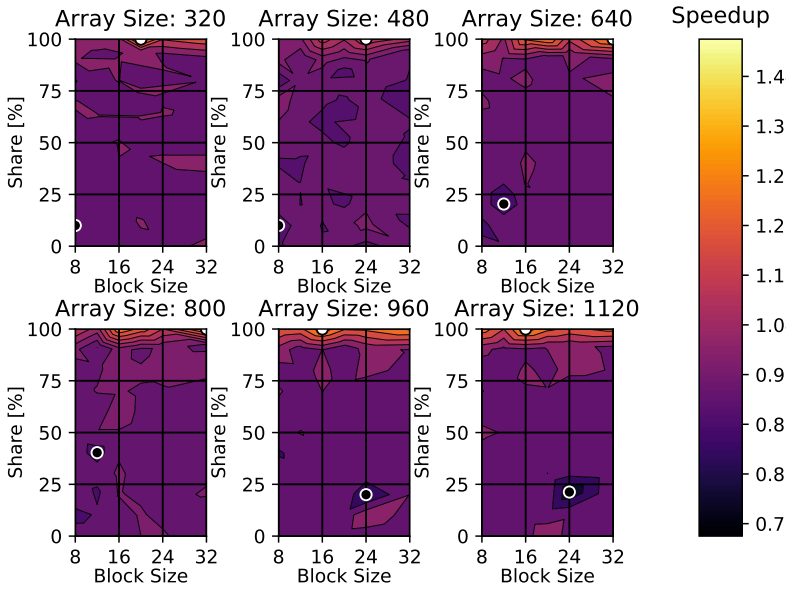

Figure 10. Hardware speedup comparison for the Euler equations.

3.3. Weak-scalability

These results raised some questions about the performance, e.g., why the swept rule performance decreases with increasing array size. We considered the weak-scalability of the algorithms to explore this. Figure 11 shows that the `Standard` solver has better weak-scalability than `Swept` for both problems. However, the slope is more shallow for the heat diffusion equation than it is for the Euler equations.

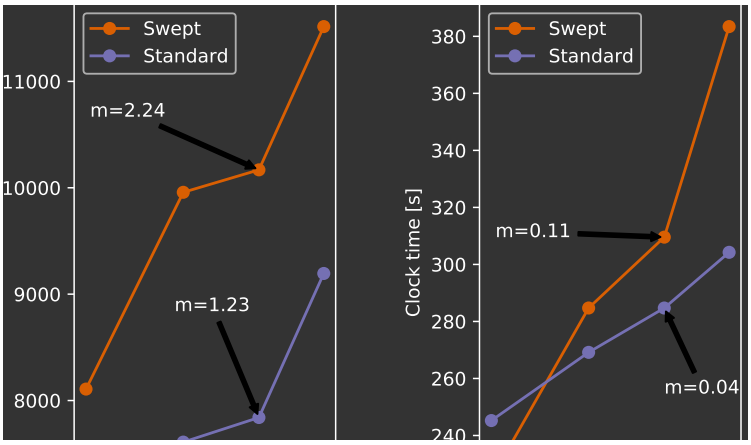

Figure 11. Weak-scalability of the `Swept` and `Standard` algorithms.

4. Discussion

In regards to the heat diffusion equation, we see an overall range of speedup from 0.22-2.71 for the first hardware set and 0.79-1.32 for the second. However the limits of speedup seem to be outliers. These speedups pale in comparison to the one-dimensional heterogeneous version which reported 1.9 to 23 times speedup and, the one-dimensional GPU version which reported 2 to 9 times speedup for similar applications for the [5,6].

For the Euler equations, we see an overall range of 0.52-1.46 for the first set of hardware and 0.36-1.42 for the second. These speedups are better aligned with values reported in other studies. The two-dimensional CPU version reported a speedup of at most 4, the one-dimensional GPU version reported 0.53-0.83, and the one-dimensional heterogeneous version reported 1.1-2.0 [4–6].

It is consistent between studies that the swept rule performance declines as numerical complexity increases. We suspect that the magnitude of speedups differ noticeably because performance seems to be heavily tied to the implementation. This implementation relied heavily on Python which is interpretive and generally slower than C++ or other compiled languages.

From these results, we believe that the swept rule can be helpful or harmful to simulation performance but testing is required to see which is the case. It depends on the problem solved, desired numerical scheme, and implementation used. It is also hardware dependent as is seen in Figures 6 and 10. It will not be suited for all numerical schemes especially those that are more complicated and cannot be parallelized well. We do believe, however, that optimizations can be made to obtain better performance in general.

One optimization in the right direction is that of the communication step. The deterioration of performance as array size is increased is connected to the implementation of the communication step of the `Swept` solver. We believe this issue can in part be corrected by changing the communication strategy. Currently, the dynamic programming approach considers the data and perspective of the data as if each process is looking at a certain portion. The process perspective is fixed and data is moved such that the swept solution can proceed in a simple manner. If we converted the algorithm such that the perspective is moved and not the data, we should see better performance in this regard.

The scalability results also suggest that something other than latency is more dominant in the swept rule implementation because ideally the swept rule would scale better having less latency from communication. We suspect that the implementation of the communication step of the `Swept` solver is the cause for this as well. This suspicion is supported by the difference in scalability between the two cases. During the communication step, some of the data transferred is not necessary for the calculation steps which leads us to believe that bandwidth is dominating the communication. The Euler equations have quite a bit more data to communicate and thus would perform worse in this case. Larger amounts of data when solving these equations also contributes to deterioration of performance as the array size increases. The communication of unnecessary data could be corrected but it would involve some more complicated logic which may or may not yield performance benefits.

Other causes of poor performance could be steps that the `Swept` solver takes but the `Standard` does not. `PySweep` determines and stores the appropriate blocks for communication prior to the simulation to avoid conditionals with the intention of boosting GPU performance. This process was parallelized as much as possible but there are some serial parts which undoubtedly reduce the performance of the algorithm.

Scalability and array size aside, performance is also tied to the block size. It is interesting that the best case frequently occurs between lower block sizes because the block size directly limits the number of steps that can be taken prior to a communication. So, we originally expected better performance with larger block sizes but this is not necessarily the case in all the tests. We believe this could be the result of the bandwidth expense dominating the simulation. If that is the case, this would also improve with the aforementioned optimizations.

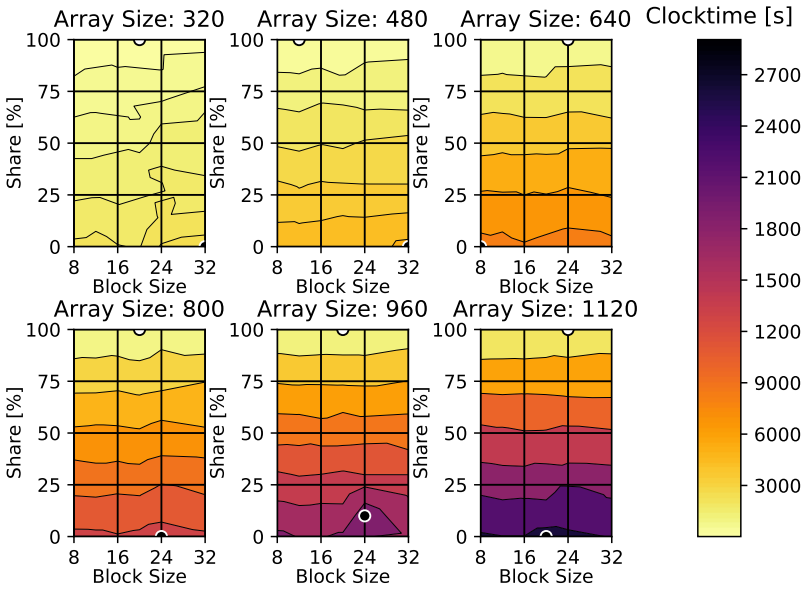

Figure 12. Clock time results for the compressible Euler equations with Nvidia GeForce GTX 1080 Ti GPUs and Intel Skylake Silver 4114 CPUs.

The fastest option is typically the most desired outcome when it comes to high performance computing applications. While in some cases a GPU share of less than 100% yield faster results, this is not one of those cases. Figure 12 demonstrates that the clock time only increases as share decreases. Note, figures for clock time in other cases were not presented because they show the same result—100% has the lowest clock time. However, the optimal configuration is useful if simulation requires data greater than the limit of the GPU(s). We expected that higher values of share would typically yield the best results but that was not necessarily the case. The first set of hardware followed this trend but the second set did the opposite.

5. Conclusions

`PySweep` is a two-dimensional implementation of the swept rule on heterogeneous architecture. It was tested over an array of GPU shares, block sizes, and array sizes with two hardware configurations each containing two nodes. The purpose of this study was to understand the objectives as stated in section 2.

Overall, we were able to gain insight to the performance benefits of the swept rule on distributed heterogeneous architecture. We showed that the swept rule can be both beneficial and detrimental. We believe that these benefits can be furthered by optimization but even then the performance is seemingly limited and will only be further limited with higher dimensionality and more practical numerical schemes. This is evident by the differing performance between simple schemes in not only this study but its precursors.

Next, we wanted to consider the impact of different hardware and determined that there are some substantial differences. The first set of hardware, which is newer technology, produced more speedup between the `Swept` and `Standard` solvers but performed worse when compared to the same solver with the other hardware set. We suspect this is because the first hardware set has nearly double the bandwidth capabilities of the second set which makes the bandwidth expense less dominant in swept communications. Better performance when comparing `Swept` and `Standard` for the first hardware set also supports the conclusion in section 3 that bandwidth is one of the dominating expenses in the current implementation.

Finally, we wanted to consider the impact of the swept performance on parameters of interest, e.g., share, block size, and array size. We considered the mode of the optimal and worst cases to further draw conclusions here. `Pysweep's` performance decreases as array size increases. This decrease in performance is a result of the implementation which can be somewhat corrected to mitigate this. The share does have an impact on shortest simulation time but does not impact the optimal configuration as much. In fact, the number of occurrences of best and worst cases were very similar for each value of share. We believe the number of occurrences are similar and share does not greatly impact the optimal configuration because of the shared memory paradigm which mitigates cost of intra-node communication. Configuration aside, it was always fastest to use a share of 1.

The block size greatly impacted the performance of the swept rule. We determined that the optimal cases occur most in a range of 12-24. The worst cases are an inverse of this where the greatest number of occurrences are on the boundary block sizes 8 and 32. It was expected that the largest block size would have the greatest performance and the smallest would have the worst because it limits the swept steps. However, this was not exactly the case. We believe this is a result of balancing thread occupancy and latency improvements. Smaller blocks have more communications and larger blocks have greater thread vacancy towards the end of a simulation phase. So, it makes sense that the best cases are balancing the latency savings while not completely sacrificing thread occupancy.

We can conclude from this study that the swept rule is highly implementation dependent and it does have potential for benefit, albeit this is limited for complicated numerical schemes. As a rough heuristic, we would suggest starting with a block size of 16, a share above 80%, and tuning to optimal performance from there. However, there are multiple cases in which the optimal performance is far from these parameters. Unfortunately, a sweep of the parameters is the best way to determine the best configuration. Finally, we believe that to optimize `PySweep` we would need to further explore strategies for the communication step and utilize a compiled language for the core of the code to better realize the effects of the swept rule. However, this version was a good start and provided insight into the potential of the method.

Funding: This material is based upon work supported by NASA under award No. NNX15AU66A under the technical monitoring of Drs. Eric Nielsen and Mujeeb Malik.

Data Availability Statement: All code to reproduce the data can be found at https://github.com/anthony-walker/pysweep-git

.

Acknowledgments: We gratefully acknowledge the support of NVIDIA Corporation, who donated a Tesla K40c GPU used in developing this research.

Appendix A. Heat Diffusion Equation

The heat diffusion problem is as follows:

$$\begin{aligned}
&\rho c_p \frac{\partial T}{\partial t} = k\frac{\partial^2 T}{\partial x^2} + k\frac{\partial^2 T}{\partial y^2}, \quad 0 < x < 1, \quad 0 < y < 1, \quad t > 0 \\
&T(0,y,t) = T(1,y,t), \quad 0 < x < 1, \quad t > 0 \\
&T(x,0,t) = T(x,1,t), \quad 0 < y < 1, \quad t > 0 \\
&T(x,y,0) = \sin(2\pi x)\sin(2\pi y)\,, \quad 0 \leq x \leq 1, \quad 0 \leq y \leq 1
\end{aligned}$$

with an analytical solution of

$$T(x,y,t) = \sin(2\pi x)\sin(2\pi y)e^{-8\pi^2 \alpha t}$$

and an update equation of

$$T_{i,j}^{k+1} = T_{i,j}^{k} + \frac{\alpha\Delta t}{\Delta x^2}\left(T_{i+1,j}^{k} - 2T_{i,j}^{k} + T_{i-1,j}^{k}\right) + \frac{\alpha\Delta t}{\Delta y^2}\left(T_{i,j+1}^{k} - 2T_{i,j}^{k} + T_{i,j-1}^{k}\right).$$

Appendix B. Compressible Euler Vortex

The compressible Euler equations and the equation of state used are as follows where subscripts represent derivatives with respect to the spatial dimensions (x and y) or time (t).

$$\begin{aligned}
\rho_t &= (\rho u)_x + (\rho v)_y \\
(\rho u)_t &= (\rho u + P)_x + (\rho uv)_y \\
(\rho v)_t &= (\rho v + P)_y + (\rho uv)_x \\
E_t &= ((E+P)u)_x + ((E+P)v)_y \\
E &= \frac{P}{\gamma - 1} + \frac{1}{2}\rho(u^2 + v^2)
\end{aligned}$$

The analytical solution to the isentropic Euler vortex was used as the initial conditions and in validation. The analytical solution was developed from [24]. The solution is simple in the sense that it is just translation of the initial conditions. It involves superimposing perturbations in the form:

$$\begin{aligned}
\delta u &= -\frac{y}{R}\Omega \\
\delta v &= -\frac{x}{R}\Omega \\
\delta T &= -\frac{\gamma - 1}{2}\Omega^2 \\
\Omega &= \beta e^{f}, \text{ where } f(x,y) = -\frac{1}{2\sigma^2}\left[\left(\frac{x}{R}^2\right) + \frac{y}{R}^2\right].
\end{aligned}$$

The initial conditions are then

$$\begin{aligned}
\rho_0 &= (1 + \delta T)^{\frac{1}{\gamma - 1}}, \\
u_0 &= M_\infty \cos(\alpha) + \delta u, \\
v_0 &= M_\infty \sin(\alpha) + \delta v, \\
p_0 &= \frac{1}{\gamma}(1 + \delta T)^{\frac{\gamma}{\gamma - 1}}.
\end{aligned}$$

The specific values used are shown in Table 1

α	M_∞	ρ_∞	p_∞	T_∞	R	σ	β	L
45°	$\sqrt{\frac{2}{\gamma}}$	1	1	1	1	1	$M_\infty \frac{5\sqrt{2}}{4\pi} e^{1/2}$	5

Table 1: Conditions used in analytical solution [32]

Similar to the heat diffusion equation, periodic boundary conditions were implemented. The numerical scheme implemented was nearly a carbon copy of what was done in [5] except it was extended into two dimensions. A five point finite volume method was used in space with a minmod flux limiter and second order Runge-Kutta was used in time. Consider equations of the form

$$\frac{\partial Q}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} = 0,$$

where

$$Q = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}, F = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E+p)u \end{bmatrix} \text{ and, } G = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ (E+p)v \end{bmatrix}$$

The pressure ratio was used in the minmod limiter to compute reconstructed values on the cell boundaries.

$$P_{r,i} = \frac{P_{i+1} - P_i}{P_i - P_{i-1}}$$

$$Q_n^i = \begin{cases} Q_o^i + \frac{\min(P_{r,i}^1, 1)}{2}(Q_o^{i+1}) - Q_o^i) & 0 < P_{r,i} < \infty \\ Q_0^i \end{cases}$$

$$Q_n^{i+1} = \begin{cases} Q_o^{i+1} + \frac{\min(P_{r,i+1}^{-1}, 1)}{2}(Q_o^i - Q_o^{i+1}) & 0 < P_{r,i}^{-1} < \infty \\ Q_0^{i+1} \end{cases}$$

The flux is then calculated with the reconstructed values for i and i+1 accordingly and used to step in time.

$$F_{i+1} = \frac{1}{2}\left(F(Q^{i+1}) + F(Q^i) + r_{x,sp}(Q^i - Q^{i+1})\right)$$
$$G_{i+1} = \frac{1}{2}\left(G(Q^{i+1}) + G(Q^i) + r_{y,sp}(Q^i - Q^{i+1})\right)$$

where $r_{i,sp}$ is the spectral radius or largest eigenvalue for the appropriate Jacobian matrix. These flux calculations are then used to apply the second order Runge-Kutta in time—the process looks like

$$Q_i^* = Q_i^n + \frac{\Delta t}{2\Delta x}(F_{i+1/2}(Q^n) + F_{i-1/2}(Q^n)) + \frac{\Delta t}{2\Delta y}(G_{i+1/2}(Q^n) + G_{i-1/2}(Q^n)),$$
$$Q_i^{n+1} = Q_i^n + \frac{\Delta t}{\Delta x}(F_{i+1/2}(Q^*) + F_{i-1/2}(Q^*)) + \frac{\Delta t}{\Delta y}(G_{i+1/2}(Q^*) + G_{i-1/2}(Q^*)).$$

Appendix C. Hardware

Hardware Specifications	Set 1	Set 2
GPU	Nvidia Tesla V100-DGXS-32GB	Nvidia GeForce GTX 1080
Architecture	Volta	Pascal
NVIDIA CUDA® Cores	5120	3584
Memory Type	32 GB HBM2	11 GB GDDR5X
Bus Width	4096 bit	352-bit
Memory Bandwidth (GB/sec)	900	484
CPU	Intel E5-2698v4	Intel Skylake Silver 4114
Cores	20	10
Processor Base Frequency	2.20 GHz	2.20 GHz
Cache	50 MB Intel® Smart Cache	13.75 MB L3 Cache

Table 2: Common specifications found between hardware sets from NVIDIA's and Intel's websites [26–29].

References

1. Alhubail, M.; Wang, Q. The swept rule for breaking the latency barrier in time advancing PDEs. *Journal of Computational Physics* **2016**. doi:10.1016/j.jcp.2015.11.026.
2. Oancea, B.; Andrei, T.; Mariana Dragoescu, R. GPGPU COMPUTING. Technical report.
3. Alexandrov, V. Route to exascale: Novel mathematical methods, scalable algorithms and Computational Science skills, 2016. doi:10.1016/j.jocs.2016.04.014.
4. Alhubail, M.; Wang, Q.; Williams, J. The swept rule for breaking the latency barrier in time advancing two-dimensional PDEs. Technical report, 2018.
5. Magee, D.J.; Niemeyer, K.E. Accelerating solutions of one-dimensional unsteady PDEs with GPU-based swept time–space decomposition. *Journal of Computational Physics* **2018**, *357*, 338–352. doi:10.1016/j.jcp.2017.12.028.
6. Magee, D.J.; Walker, A.S.; Niemeyer, K.E. Applying the swept rule for solving explicit partial differential equations on heterogeneous computing systems. *Journal of Supercomputing* **2020**. doi:10.1007/s11227-020-03340-9.
7. Gander, M.J. 50 Years of Time Parallel Time Integration; 2015; pp. 69–113. doi:10.1007/978-3-319-23321-5{_}3.
8. Falgout, R.D.; Friedhoff, S.; Kolev, T.V.; MacLachlan, S.P.; Schroder, J.B. Parallel time integration with multigrid. *SIAM Journal on Scientific Computing* **2014**, *36*, C635–C661. doi:10.1137/130944230.
9. Lions, J.L.; Maday, Y.; Turinici, G. Résolution d'EDP par un schéma en temps < pararéel >. Technical report, 2013.
10. Maday, Y.; Mula, O. An adaptive parareal algorithm. *Journal of Computational and Applied Mathematics* **2020**, *377*. doi:10.1016/j.cam.2020.112915.
11. Wu, S.L.; Zhou, T. Parareal algorithms with local time-integrators for time fractional differential equations Fast Computation Algorithms for Differential Equations View project Numerical methods for time and space fractional partial differential equations View project Parareal algorithms with local time-integrators for time fractional differential equations. *Article in Journal of Computational Physics* **2018**, *358*, 135–149. doi:10.1016/j.jcp.2017.12.029.
12. Emmett, M.; Science, M.M.i.A.M.; Computational., 2012, u. Toward an efficient parallel in time method for partial differential equations. *msp.org*.
13. Minion, M.L.; Speck, R.; Bolten, M.; Emmett, M.; Ruprecht, D. INTERWEAVING PFASST AND PARALLEL MULTIGRID. Technical report.
14. Hahne, J.; Friedhoff, S.; Bolten, M. PyMGRIT: A Python Package for the parallel-in-time method MGRIT **2020**.
15. Kowarschik, M.; Weiß, C. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms; 2003; pp. 213–232. doi:10.1007/3-540-36574-5{_}10.
16. Demmel, J.; Hoemmen, M.; Mohiyuddin, M.; Yelick, K. *Avoiding Communication in Sparse Matrix Computations*.
17. Ballard, G.; Demmel, J.; Holtz, O.; Schwartz, O. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications* **2011**, *32*, 866–901. doi:10.1137/090769156.
18. Baboulin, M.; Donfack, S.; Dongarra, J.; . . . , L.G.P.C.; 2012, u. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. *Elsevier*.
19. Khabou, A.; Demmel, J.W.; Gu, M. LU factorization with panel rank revealing pivoting and its communication avoiding version. Technical report, 2012.
20. Solomonik, E.; Zurich, E.; Ballard, G.; Demmel, J. A communication-avoiding parallel algorithm for the symmetric eigenvalue problem Torsten Hoefler. Technical report.
21. Amazon EC2 Pricing - Amazon Web Services.
22. Dalcín, L.; Paz, R.; Storti, M. MPI for Python. Technical report.
23. Klöckner, A.; Pinto, N.; Lee, Y.; Catanzaro, B.; Computing, P.I.P.; 2012, u. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Elsevier*.
24. Spiegel, S.C.; Huynh, H.T.; Debonis, J.R. A Survey of the Isentropic Euler Vortex Problem using High-Order Methods. Technical report.
25. Leveque, R.J. Finite Volume Methods for Hyperbolic Problems. Technical report, 2002.
26. Intel® Xeon® Silver 4114 Processor 123550.
27. Intel® Xeon® Processor E5-2698 v4 91753.
28. GeForce GTX 1080 Ti Graphics Cards | NVIDIA GeForce.
29. NVIDIA V100 | NVIDIA.
30. Hoefler, T.; Dinan, J.; Buntinas, D.; Balaji, P.; Barrett, B.; Brightwell, R.; Gropp, W.; Kale, V.; Thakur, R.; Hoefler ETH Zurich, T.; Dinan, J.; Buntinas, D.; Balaji, P.; Thakur, R.; Barrett, B.; Brightwell, R.; Gropp, W.; Kale, V. MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory **2013**. *95*, 1121–1136. doi:10.1007/s00607-013-0324-2.
31. Collette, A. HDF5 for Python **2008**.
32. Shu, C.W. Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. In *Advanced numerical approximation of nonlinear hyperbolic equations*; Springer, 1998; pp. 325–432.