AN ABSTRACT OF THE THESIS OF

Daniel J. Magee for the degree of Master of Science in Mechanical Engineering presented on June 8, 2018.

Title: Swept time-space domain decomposition on GPUs and heterogeneous computing systems

Abstract approved: _____

Kyle Niemeyer

Modern scientific and engineering problems often require simulations with a level of resolution difficult to achieve in reasonable amounts of time—even in effectively parallelized programs. Therefore, applications that exploit high performance computing (HPC) systems have become invaluable in academia and industry over the past two decades. Addressing the questions that arise from continual scientific advancement requires increasing development in these large-scale computational systems; solutions from hardware and software are required to supply the necessary throughput for demand across scientific disciplines. The most important development on the hardware side has been the General Purpose Graphics Processing Unit (GPGPU), a class of massively parallel device that now composes a substantial portion of the computational power of the top 500 supercomputers. As these systems grow, barriers to increased performance arise from small costs accumulated over innumerable iterations such as latency, the fixed cost of memory accesses, which becomes significantly larger when access requires communication between two distant CPU processes. This thesis implements and analyzes swept time-space domain decomposition, a communication avoiding scheme for time-stepping stencil codes, for GPGPU and heterogeneous (CPU/GPU) architectures. The GPGPU program significantly improves the execution time of finite-difference solvers for relatively simple one-dimensional time-stepping partial differential equations (PDEs). The swept decomposition code showed speedups of 2–9 $\times$ compared with simple GPU domain decompositions and 7–300 $\times$ compared with parallel CPU versions over a range of problem sizes,

2×10^3–10^6 spatial points. However, for a more sophisticated one-dimensional system of equations discretized with a second-order finite-volume scheme, the swept rule performs 1.2–1.9 $\times$ worse than a standard implementation for all problem sizes. The program targeting heterogeneous systems with distributed memory patterns performs significantly better on both simple problems, speedup 4–18 $\times$, and more complex equation systems, speedup 1.5–3 $\times$, over the range of problem sizes, 5×10^5–10^7 spatial points.

Swept time-space domain decomposition on GPUs and heterogeneous computing systems

by

Daniel J. Magee

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 8, 2018
Commencement June 2018

Master of Science thesis of Daniel J. Magee presented on June 8, 2018.

APPROVED:

Major Professor, representing Mechanical Engineering

Head of the School of Mechanical, Industrial, and Manufacturing Engineering

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Daniel J. Magee, Author

ACKNOWLEDGEMENTS

This material is based upon work supported by NASA under award No. NNX15AU66A under the technical monitoring of Drs. Eric Nielsen and Mujeeb Malik. We also gratefully acknowledge the support of Nvidia Corporation with the donation of the Tesla K40c GPU used for this research.

I would like to thank my committee for their patience, time and insight, my advisor, Dr. Kyle Niemeyer, for introducing me to the expanding the world of computational science, my labmates for sharing the graduate school experience with me, and Oregon State University for fostering my growth and being an excellent value these last six years.

To my wife Amanda, thank you for your patience and support; thank you for sharing my dreams. To Nora, thank you for arriving right on time and helping me realize continually just how lucky I am. To my parents John and Judy, thank you for supporting me unconditionally through my many metamophoses.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

Figure Page

For Amanda and Nora

There are more things in heaven and earth, Horatio,
Than are dreamt of in your philosophy.

- *Hamlet*, 1.5.167–8

Chapter 1: Introduction

In 2009 prominent high-performance computing (HPC) researchers submitted a call to arms to the scientific programming community. Their paper launched a project based on a concept that would come to be the mission of organizations committed to developing the next generation of supercomputers: exascale [14]. The idea of exascale computing, building software for systems capable of completing 10^{18} floating-point operations per second (FLOPS). This is a somewhat arbitrary distinction, but has served its purpose by spurring action because it seemed futuristic yet imminent, a generation away but possible.

The authors who launched the International Exascale Project [14] saw that software for exascale computers would likely pose an even larger barrier to achieving this performance benchmark than the immense technical challenges involved in constructing the system itself and its components. While they envisage the integration of accelerators, such as graphics processing units (GPUs), into HPC systems at exascale, they could not have anticipated how rapidly GPU hardware and their environments have changed since the middle of this decade. This rapid progress has seen Nvidia GPUs progress from Kepler to Pascal architecture in four years and Volta architecture only one year later.

This hardware development entails increased programming capabilities in the CUDA API (the general purpose parallel computing platform and programming model developed by NVIDIA for their GPUs), such as device synchronization between disparate units and independent thread scheduling [32]. Since the GPU paradigm is a relatively young technology compared to CPUs, rapid developments are possible and change the way programmers interact with the technology. This presents a significant problem for programmers seeking to achieve the level of performance for which the hardware is designed: often legacy code is only a few years old.

The solutions presented here deal exclusively with explicit numerical methods using finite difference or volume formulations. These methods are subject to stringent stability requirements, as finer spatial grids require finer timesteps and vice versa. Despite this shortcoming, explicit methods are commonly used in solvers for fluid dynamics equations.

This thesis aims at providing direction for developing partial differential equation (PDE) solvers that meet some of the main challenges of exascale computing, particularly the increase in the latency cost of communication arising from the increasing physical size of the compute system. It investigates swept time-space domain decomposition, a method for avoiding the latency of repeated communication events, on GPUs and heterogeneous distributed memory systems. The sample programs and libraries use CUDA and Message Passing Interface (MPI), the standard programming library for large scale, multinode parallel applications to parallelize the applications across GPUs and CPUs.

1.1 Motivation

The motivation for developing exascale scientific computing libraries is self-evident within conventional scientific thought. Faster, bigger simulations make more efficient turbines, wings, engines, and HPC systems, and advance the cause of scientific understanding in theoretical topics from LIGO to LHC. This argument is true, as evidenced by the applications first in line to run on the SUMMIT supercomputer at Oak Ridge when it boots up and claims the world's fastest title this summer. These applications range from climate modeling and relativistic quantum chemistry to astro and bio physics [17].

I do not mean to propose that this line of reasoning and justification is somehow insufficient. It is quite sufficient to warrant this, and many more intensive investigations. The questions of energy, materials, climate, and particle physics are some of the fundamental questions of our time and the greater modeling resolution of exascale systems promise to illuminate corners of reality with unthinkable implications for our collective project. The specific motivation for this thesis is to develop swept time-space decomposition to accelerate large computational fluid dynamics simulations, in particular for aerospace applications; this is why I chose to use the Euler equations for gas dynamics as a representative case. However, I would like to offer another, comparatively less scientific, though no less rooted in human intellectual history, justification for the work set out in this thesis.

It has been proffered by way of explanation that a GPU is akin to the "soul" of a PC [23]. This may, and should, strike the reader as hyperbolic if taken at face value. But the metaphor is a more powerful and prescient one than we would admit, or indeed, its author likely intended.

Leaving out any supernatural implications, what is a soul? Of course, different traditions have answered this question somewhat differently, but a materialist interpretation of a common definition could fairly be parsed as: what separates humans from the rest of known biology. We don't have the biggest brains, or the most neurons, elephants do [20]. So why are elephants not writing theses?

The likely answer is that a far greater proportion of elephant neurons are in the cerebellum. Though they have 3× more neurons than humans, humans have 3× more neurons in the cerebral cortex [20]. This would suggest that, in fundamental terms, being human, building civilization, progressing to the point where we can imagine protecting ourselves from the random slings and asteroids of the cosmos, is a matter of having the right computational structures and the right software.

This may seem overly speculative and tenuous, but the third-largest supercomputer [39] is being used by a large, serious, international project aiming to simulate the human brain [29]. This HPC system contains 5320 Nvidia P100 GPGPUs. Current estimates put the computational throughput needed for the project to be between 1–10 exaFLOPS.

Maybe we are too complex to be simulated or understood or modeled. It is certainly possible that the scale of the problem exceeds computational performance. But we are currently at the point where we can dream of attempting to understand and simulate ourselves, and I can't think of any more fundamental motivation than that.

1.2 Objective

The main objective of this effort is to determine the viability of swept time-space decomposition as a suitable method for time-stepping PDEs on heterogeneous computational systems. In evaluating this objective, there are several ancillary objectives:

- Determine the best way to exploit the GPU memory hierarchy in a swept rule program.
- Determine the best way to handle high-order, multi-step methods using the swept rule.
- Determine the correct balance for assigning GPU work in heterogeneous environments.

- Determine the correct attributes of the main swept-rule concept for different architectures.
- Determine if the swept rule provides performance benefit. And, if so, under what conditions and for what problems?

Aside from the objectives concerning the swept rule, I have focused on developing an extensible piece of software to facilitate the exploration of the aforementioned objectives. The objective in this sense is to build a piece of software that remains useful after my graduation. Software meeting this objective should have several features:

- Easy interface with which to add new equations and numerical methods
- Plain style for source code
- Simple workflow to run code and view results
- Good documentation
- Easy to adapt for different platforms

1.3 Outline of Thesis

This thesis is composed of two research papers suitable for submission to a peer-reviewed journal dedicated to subjects pertaining to high performance computing, parallel computing, computational science, GPU and heterogeneous computing, or scientific computing. Chapter 2 contains the first paper published by *Journal of Computational Physics* in December 2017 [27]. This paper presents and analyzes a highly-optimized research code applying the swept rule in one dimension to a single GPU. It presents several strategies for exploiting the GPU memory hierarchy and assesses their effect on the overall performance of the swept rule compared to a naive domain decomposition strategy. This chapter explains the mechanics of the swept rule in one dimension in detail. Chapter 3 contains the second paper, submitted for publication, which builds on the first paper by extending the GPU-only case to heterogeneous, multi-node, distributed memory systems. It considers the tradeoffs between generality and optimization, and the nuances of workload balancing in one-dimensional equations while applying findings from the first paper.

Finally, Chapter 4 summarizes the conclusions of this project, addresses its shortcomings, and outlines a direction for future work.

Accelerating solutions of one-dimensional unsteady PDEs with GPU-based swept time–space decomposition

Daniel J. Magee and Kyle E. Niemeyer

Journal of Computational Physics
Vol. 357, 338–352, 2018.
https://doi.org/10.1016/j.jcp.2017.12.028

Chapter 2: Accelerating solutions of one-dimensional unsteady PDEs with GPU-based swept time–space decomposition

Abstract

The expedient design of precision components in aerospace and other high-tech industries requires simulations of physical phenomena often described by partial differential equations (PDEs) without exact solutions. Modern design problems require simulations with a level of resolution difficult to achieve in reasonable amounts of time—even in effectively parallelized solvers. Though the scale of the problem relative to available computing power is the greatest impediment to accelerating these applications, significant performance gains can be achieved through careful attention to the details of memory communication and access. The swept time-space decomposition rule reduces communication between sub-domains by exhausting the domain of influence before communicating boundary values. Here we present a GPU implementation of the swept rule, which modifies the algorithm for improved performance on this processing architecture by prioritizing use of private (shared) memory, avoiding interblock communication, and overwriting unnecessary values. It shows significant improvement in the execution time of finite-difference solvers for one-dimensional unsteady PDEs, producing speedups of 2–9 $\times$ for a range of problem sizes, respectively, compared with simple GPU versions and 7–300 $\times$ compared with parallel CPU versions. However, for a more sophisticated one-dimensional system of equations discretized with a second-order finite-volume scheme, the swept rule performs 1.2–1.9 $\times$ worse than a standard implementation for all problem sizes.

2.1 Introduction

High-fidelity computational fluid dynamics (CFD) simulations are essential for developing aerospace technologies such as rocket launch vehicles and jet engines. This project aims to accelerate such simulations to approach real-time execution—simulation at the speed of nature—in accordance with the high-performance computing development goals set

out in the CFD Vision 2030 report [36]. Classic approaches to domain decomposition for parallelized, explicit, time-stepping partial differential equation (PDE) solutions incur substantial computational performance costs from the communication between nodes required every timestep. This communication cost consists of two parts: latency and bandwidth, where latency is the fixed cost of each communication event and bandwidth is the variable cost that depends on the amount of data transferred. Latency in inter-node communication is a fundamental barrier to this goal, and advancements to network latency have historically been slower than improvements in other computing performance barriers such as bandwidth and computational power [35]. Performance may be improved by avoiding external node communication until exhausting the domain of dependence, allowing the calculation to advance multiple timesteps while requiring a smaller number of communication events. This idea is the basis of swept time-space decomposition [3, 5].

Extreme-scale computing clusters have recently been used to solve the compressible Navier–Stokes equations on over 1.97 million CPU cores [8]. The monetary cost, power consumption, and size of such a cluster impedes the realization of widespread peta- and exa-scale computing required for real-time, high-fidelity, CFD simulations. While these are significant challenges, they also provide an opportunity to develop new tools that increase the use of the available hardware resources. As the authors of CFD Vision 2030 note, "High Performance Computing (HPC) hardware is progressing rapidly and is on the cusp of a paradigm shift in technology that may require a rethinking of current CFD algorithms and software" [36]. Using graphics processing unit (GPU) hardware as the primary computation device or accelerator in a heterogeneous system is a viable, expedient option for high-performance cluster design that helps mitigate these problems. For this reason, GPUs and other emerging coprocessor architectures are increasingly used to accelerate CFD simulations [31].

GPU technology has improved rapidly in recent years; in particular, Nvidia GPUs have progressed from Kepler to Pascal architecture in four years. This development doubled and tripled peak single- and double-precision performance, respectively [33]. In addition, the presence of GPUs in clusters, such as ONRL's Titan supercomputer, has become increasingly common in the last decade. These advances have driven the development of software capable of efficiently using and unifying the disparate architectures [41]. Although the ultimate motivation of our work is accelerating the solution of PDEs—particularly relevant to fluid flow—on distributed-memory systems, in this

work we focused on a single GPU-accelerated node/workstation in the design of the algorithms and associated software. By investigating the effectiveness of the swept rule on a workstation, we provide results that can be applied to simulations on a single machine as well as an initial framework for understanding the performance of the swept rule on heterogeneous computing systems.

The swept rule operates on a simple principle: do the most work possible on the values closest to the processor before communicating. In practice, this directive results in an algorithm that advances the solution in time at all spatial points using locally accessible stencil values at the previous timestep. Because the data closest to the processor is also the least widely accessible, the strict application of this principle does not always provide the best performance, but it is a useful heuristic for implementing the procedure and analyzing its performance.

This study presents an investigation of the performance characteristics of three swept rule implementations for a single GPU in a workstation. These procedures are tested on three one-dimensional PDEs with numerical schemes of varying complexity and compared with the performance of parallel CPU algorithms and unsophisticated GPU versions. The (next) Section 2.2 describes recent work on partitioning schemes for PDEs and communication-avoiding algorithms, especially as applied to GPUs. Section 2.3 gives a brief overview of the GPU architecture, particularly the thread and memory hierarchies. Section 2.4 discusses the swept rule, and our adjustments to the original algorithm in response to the details of GPU architecture. Section 2.5 describes the swept rule implementation in detail. In Section 2.6 we present the results of the tests and, lastly, draw further conclusions in Section 2.7.

2.2 Related work

Alhubail and Wang introduced the swept rule for explicit, time-stepping, numerical schemes applied to PDEs [2, 3, 5], and our work takes their results and ideas as its starting point. The swept rule is closely related to cache optimization techniques, in particular those that use geometry to organize stencil update computation such as parallelograms [38] and diamonds [28]. The diamond tiling method presented by Malas et al. [28] is similar to the swept rule but uses the data dependency of the grid to improve cache usage rather than avoid communication. Concepts such as stencil optimization

using domain decomposition on various architectures that are fundamental to this study are explored by Datta et al. [13]. Their work explores comparisons between parallel GPU and CPU architectures and tunes the stencil algorithm with nested domain decomposition. The swept rule also has elements in common with parallel-in-time and communication-avoiding algorithms.

Parallel-in-time methods [18], such as multigrid-reduction-in-time (MGRIT) algorithms [16], accelerate PDE solutions with time integrators that overcome the interdependence of solutions in the time domain, allowing parallelization of the entire space-time grid. These methods calculate the solution over the space-time domain using a coarse grid and iterate over successively finer grids to achieve the desired accuracy. The use of coarse grids in parallel-in-time methods reduces efficiency and accuracy when applied to nonlinear systems [3]. This shortcoming is intuitive: since chaotic, nonlinear systems may suddenly change in time, and coarse grids are prone to aliasing, the required grid granularity diminishes gains in performance. The swept rule arises from the same motivation, but does not seek to parallelize the computation in time or vary dimensions during the process.

The swept rule does not alter the numerical scheme; it decomposes the domain and organizes computation. That is, compared to a classic domain decomposition, the swept rule performs the same operations in a different order and location. In this way communication-avoiding algorithms share many implementation details with the swept rule. Recent developments in communication-avoiding algorithms for GPUs have generally focused on applications involving matrices such as QR and LU factorization. The LU factorization algorithm presented by Baboulin et al. [7] is motivated by the increasing use of GPU accelerators in large-scale, heterogeneous clusters. This method splits tasks between the GPU and CPU, minimizing communication between devices. This allows the communication and computation performed on each device to overlap, so all data transfer occurs asynchronously with computation. We explore this approach—overlapping data transfer with hybrid computation—in this article. The motivation and structure of our study is comparable to the work of Anderson et al [6]. They developed methods for arranging and tuning computation for single general-purpose GPU (GPGPU) in a desktop workstation, without altering the basic QR factorization algorithm. Similarly, this study focuses on adapting the swept rule to a single GPU. The swept rule is a strategy for arranging the computational path of explicit numerical methods, and this work

seeks to design the data structures and operations used in that path to achieve the best performance on GPUs.

2.3 GPU architecture and memory

The impressive parallel processing capabilities of modern GPUs resulted from architecture originally designed to improve visual output from a computer. GPU manufacturers and third-party authors [34, 32, 9, 37] have described the features of this architecture in great detail, while others discussed general best practices for developing efficient GPU-based algorithms [11, 31]. Particular aspects of this architecture, such as the unique and accessible memory hierarchy, are at the core of this work, so some explanation of its relevant elements is necessary before describing the details of the implementation.

Programs that run on the GPU can be implemented using several software packages, the most common of which are the OpenCL and OpenACC frameworks, and the CUDA parallel computing platform. These packages use different nomenclatures and are compatible with different hardware types. In this project all programs use CUDA, which is exclusively compatible with Nvidia GPUs; therefore, all descriptions of GPU hardware presented here use the CUDA nomenclature. CUDA programs consist of functions, referred to as kernels, launched from a C/C++ host program. The CPU executes the host code and specifies the size and number of blocks when calling a kernel, the stream (queue) in which the kernel will be launched, and the amount of shared memory to be allocated per block at runtime. All threads in a warp—a group of 32 threads that execute as a single-instruction multiple thread (SIMT) unit—must be in the same block, so for good practice blocks should launch with some multiple of 32 threads.

The information presented here is valid for all Nvidia GPUs with compute capability 3.0 or higher (i.e., Kepler architecture or later). The device used in this study is a Tesla K40c GPGPU, compute capability 3.5. This device contains 15 streaming multiprocessors, each capable of processing 64 warps of 32 threads, or 2048 total threads, at once [33]. A maximum of 16 blocks may concurrently reside on a streaming multiprocessor; blocks may not be split between streaming multiprocessors. While each streaming multiprocessor can support 2048 resident threads, in practice their capacity is often lower because each thread or block makes demands on limited memory resources—most notably the shared memory and registers. Each streaming multiprocessor on the Tesla K40c has

48 kB of shared memory and 65536 registers available. Registers offer the fastest access, but are the most limited memory type and are private to each thread, but can be accessed by other threads in the same warp using shuffle operations available on devices with compute capability 3.5 or higher. Shared memory is a controllable portion of the L1 cache accessible only to threads within a block. As a result, for a thread to read a value stored in shared memory in a different block, a thread with access to that value must write the value to global memory where the reader thread has access. Global memory is the slowest and most plentiful memory type, and where data copied from the host program resides. Global memory stores all variables passed to a kernel and large arrays declared therein.

Other memory types in the CUDA memory hierarchy include constant, texture, and surface; of these, the work presented here only uses constant memory. Constant memory is read-only, available to all kernels for the lifetime of an application, and quick to access when all threads access the same location. This makes it a convenient and performance conscious choice for storing constant values of the governing equations calculated at runtime [32].

2.4 Methodology

2.4.1 Experimental method

The primary goal of this study is to compare the performance of the swept rule to a simple domain decomposition scheme, referred to as `Classic`, on a GPU. A domain decomposition scheme is a way of splitting up a large problem so tasks can be shared by discrete workers working on distinct parts of the problem. In this case the decomposition scheme divides the space-time domain of the PDE solution. We will compare the performance of these methods by the time cost of each timestep, including the time required to transfer data to/from the GPU. While encoding the `Classic` is relatively straightforward, finding the best approach for the swept rule on the GPU presents a more subtle problem.

In the original swept rule approach [3], the spatial domain is partitioned into independent pieces called "nodes" that correspond to compute nodes on a distributed system with private memory spaces. A major concern in adapting this nodal analogy to a single GPU is the type of memory allocated for the working array, the container for the values

that are the solution to and the basis for each timestep. Several available approaches exist to map the original analogy for a node to a single GPU; here, we will explore three of them: `Shared`, `Hybrid`, and `Register` (which we will describe in detail in Section 2.5).

In all approaches we map one thread to one spatial point. Handling more than one spatial point per thread would allow for larger nodes but would require more resources and complicate the procedure without reducing thread idleness. Additional swept rule properties could be adjusted for potential implementation variants such as the data structure to hold the working values, and the method to globally synchronize threads. However, for the purposes of this study, we did not vary these attributes. In all cases in this study the working array is a standard, one-dimensional C array with two flattened rows; and kernel calls are implicitly synchronized by returning control to the queue in the host program.

2.4.2 First-order domain of dependence

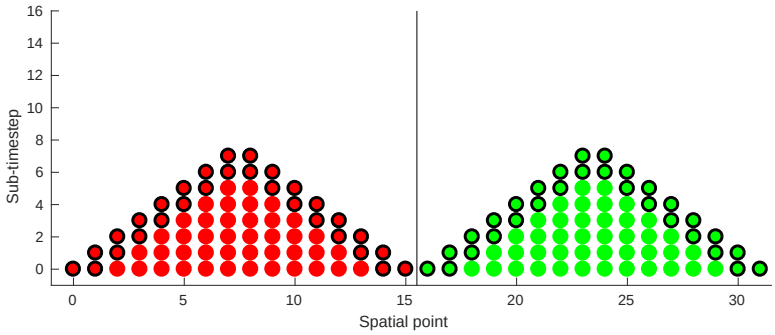
[1]

A domain of dependence is a region on the space-time grid that can be completed by a numerical scheme with some set of initial values. For the purposes of this project, numerical schemes consist of stencil operations, where a value at a grid point is updated based on the weighted contribution of values at grid points in the vicinity. A three-point stencil uses values at neighboring grid points only, and the timestep of any numerical scheme can be decomposed into a series of sub-timesteps that only require a three-point stencil [40]. The numerical scheme defines the order[2]of the domain of dependence: it increases by one for every two sub-timesteps required per timestep. Therefore, a domain of dependence is *first-order* if all sub-timesteps in the numerical scheme use a three-point stencil, and intermediate values are required no more than two steps after they are calculated. The initial incarnation of the swept rule presented by Alhubail and Wang [3, 2] decomposes multi-step timesteps and large stencils into sub-timesteps with a three-point stencil. This regularizes the procedure and ensures that all equations and schemes can be evaluated using a swept decomposition with a first-order domain of dependence, but requires more memory to store the intermediate values that result from each sub-timestep. In order to conserve limited private memory, for the GPU-based swept rule a first-order domain of dependence is applicable to schemes that require two or fewer three-point stencil sub-timesteps per timestep. Figure 2.1 shows the first two stages of the swept rule

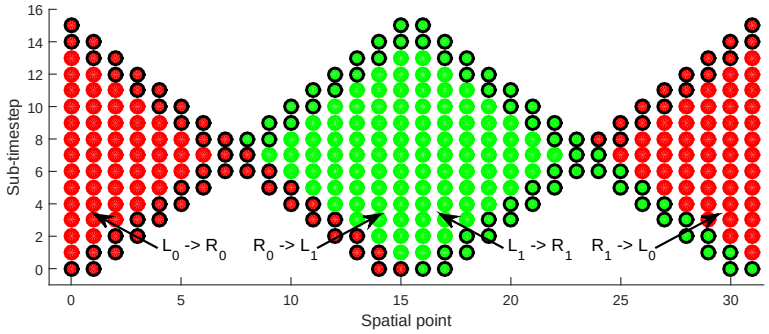
using $k = 2$ nodes with $n = 16$ spatial points.

At the start of the first step, shown in Figure 2.1a, the initial conditions are passed to the kernel and each node evaluates the solution at as many points as possible in the space-time grid. The initial domain of dependence forms a triangle. The working array

[2]Order in this sense does not refer to accuracy or stability, but to the properties of the domain of dependence resuling from the equation and numerical scheme applied to it.



(a) The first step of the swept rule. Values at $t = 0$ are split between nodes 0 and 1, which compute solutions in their domain of dependence, a triangle in the space-time plane. The edge values are collected in global arrays L_0/R_0 and L_1/R_1.



(b) The second step in the swept rule. The nodes pass their right edge to the neighboring node. The passed values become the initial left edge, and the left edge from the previous stage becomes the right edge. Each node advances through their domain of dependence, a diamond in space-time.

Figure 2.1: The first two steps of the swept rule for a numerical scheme with a first-order domain of dependence. L_i/R_i refer to left/right arrays of node i, which collect edge values shown with thick bordered dots [25].

stores the solutions as each node steps through time and is maintained in a fast memory space private to node member threads. When each node cannot advance any further it's necessary for each to pass one edge to a neighboring node and retain the values of the other edge. Figure 2.1b shows how the second step proceeds from the first using the edge values.

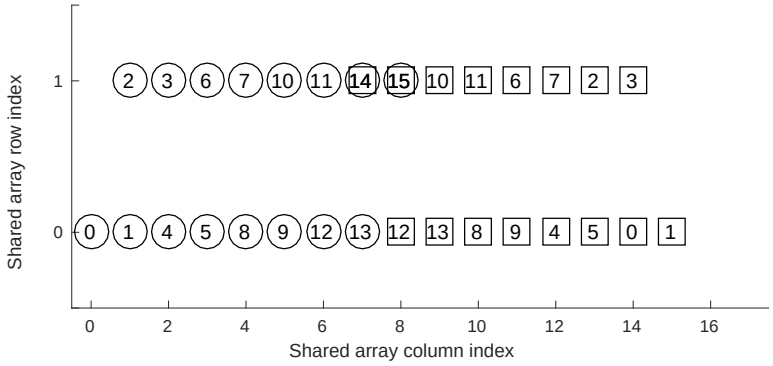
For the nodes to communicate as Figure 2.1 illustrates, the values on the edges must be passed to global arrays available to all nodes since any sufficiently fast memory location is private to a subset of threads terminated on kernel exit. Consequently, any data necessary to continue the computation must be shared between nodes using an intermediary container. The information needed to begin a new nodal cycle, including the values retained by each node, must be read in from, and out to, global memory at the beginning and end of each kernel, respectively. Figure 2.2a shows how the working array values are stored and passed to the global arrays.

Figure 2.1b illustrates the reason the two edges are stored individually as L_i and R_i: only one of the edges is passed between nodes. After the first step, the right edge is passed to its right neighbor node; the left edge is stationary. At the beginning of the subsequent kernel, L_i and R_i are swapped and reinserted into the working array to seed the next progression with one data transfer event from global to private memory as shown in Figure 2.2b. When the right edge is passed between nodes, node 0 is split across the spatial boundary and must apply the boundary conditions at its center.

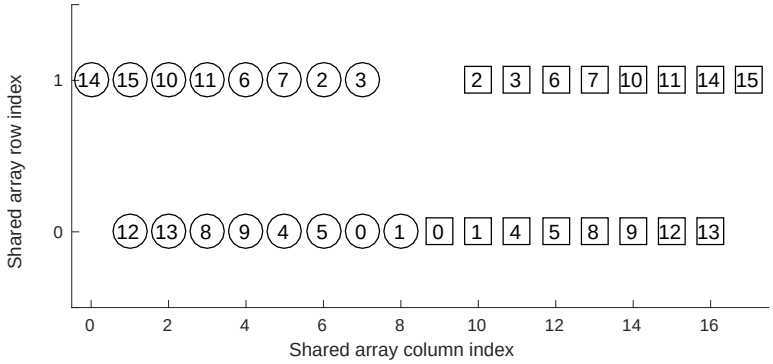
The swept rule allows the computation to advance by n timesteps with two, rather than n, global memory accesses, where n is the number of threads per block (or the number of spatial points per node). The procedure advances by passing values in the alternating directions and zig-zagging the location of the nodes in this fashion until the simulation is complete. Since the diamonds shown in Figure 2.1b do not store all the values at a single timestep, the simulation can only output values when a complementary triangle is computed and the final n-length local tier is returned. This kernel can only be called after the values are passed to the left, so the results can only be read out every nth timestep.

Although we already described the working array and showed in Figure 2.2 how the relevant values are communicated between the nodes, it is instructive to outline the performance concerns that motivate this arrangement. As Section 2.3 describes, the number of resident threads on a streaming multiprocessor depends on the GPU

architecture and resources requested at kernel launch. For instance, storing every double-precision value in the triangle shown in Figure 2.1a in shared memory in a kernel with 512 threads per block would require $8 \times 65792 = 514\,\text{kB}$ of shared memory—this is over



(a) At the end of a kernel (completion of an inverted triangle), the working array is stored and passed from private memory to global memory. The location shows the index of the working array in (private) shared memory. The number in the shape refers to the offset global memory index where the working value is passed.



(b) Edges are reinserted to seed the inverted triangle of a new swept cycle. The location shows the index of the working array in (private) shared memory receiving the global memory value denoted by the number.

Figure 2.2: The procedure for edge passing shown in Figure 2.1b. The global arrays L_i and R_i are represented by circles and squares, respectively, and the numbers in those shapes represent the position of the value in the global array. The axes describe the location of the value in the working array [25].

100 times greater than the 48 kB limit for Nvidia GPUs with Kepler architecture. The maximum number of threads per streaming multiprocessor would be limited to 128, which would negatively impact program performance.

Figure 2.1a shows that the interior of the triangle is only needed to progress to the next timestep, and that edges on even and odd tiers do not overlap in the spatial domain. Thus, the triangle may be stored as a matrix with two rows, where the first and second rows contain the even and odd sub-timesteps results, respectively. The interior values are overwritten once they are used, and only the edge values remain. Figure 2.2a shows the result of a local computation with this method. The last two values, the tips of each triangle, are copied into both arrays.

At the start of the next kernel, the left and right arrays are inserted into the working array to seed successive calculations as Figure 2.2b shows. The edges of the previous cycle's first row are moved to the center, and the center of the current top row is now left open for the first tier of the inverted triangle. Each row requires space for $n + 2$ values because two edge values are required on either side to complete the stencil for the longest row where n values are computed. The computation proceeds by filling the empty two indices on the top row, overwriting the bottom row's middle four indices, and so on. In contrast to the memory demands of storing the entire nodal computation, this method uses only $2 \times (n + 2)$ values. For a block with $n = 512$ threads, this requires $8 \times 1028 \approx 8$ kB. In general, a stencil with width k would require storing $2 \times (n + k - 1)$ values in the working array. By reducing the amount of shared memory to only 8 kB, the kernel is less limited and achieves higher occupancy, the number of threads each streaming multiprocessor is capable of handling simultaneously.

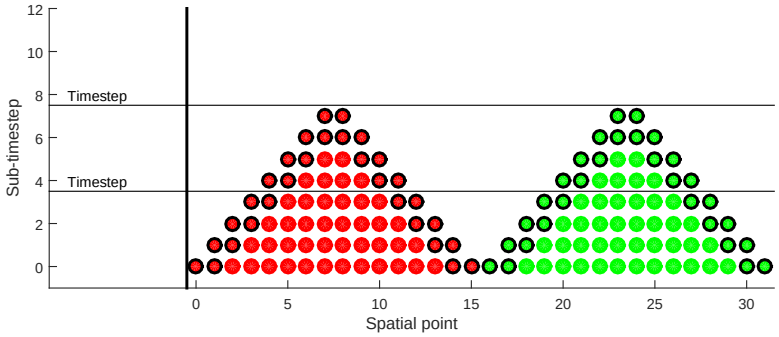
2.4.3 Higher-order domain of dependence

The first-order domain of dependence suffices for relatively simple problems. However, more complicated problems with elements such as nonlinear equations, discontinuities, or higher-order derivatives require more sophisticated procedures. The original swept rule program calculates and stores intermediate values at sub-timesteps for higher-order schemes to avoid using larger stencils [3]. Breaking a timestep into a series of sub-timesteps allows any numerical scheme for any equation of the same dimension to be decomposed in the same way since all stages in the computation depend on the minimum

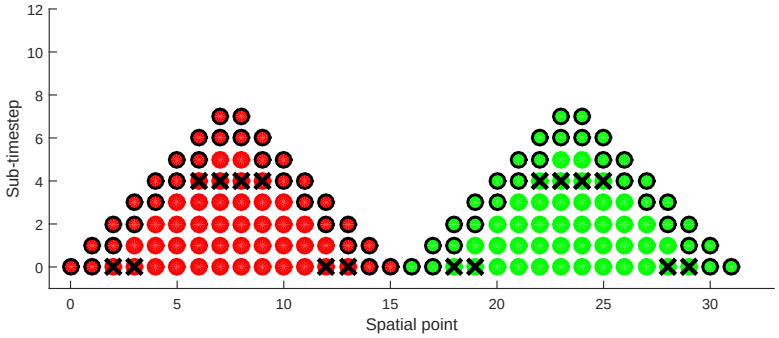
stencil, that is, only the neighbors of the current spatial point [40]. For example, a second-order in time midpoint method applied to a fourth-order differential equation would require four sub-timesteps per timestep. The first sub-timestep would find the second derivative of the dependent variable so that the second sub-timestep, the midpoint solution, would only require a three point stencil: the second derivative and initial values at the neighboring spatial points. The third sub-timestep would find the second derivative using the midpoint solution, and the final sub-timestep would complete the timestep using the results of the second and third sub-timesteps on the three-point stencil and the previous timestep solution at the current spatial point. This approach presents a storage and data transfer problem on the GPU because values in the interior of the working array are overwritten two sub-timesteps after they are calculated. These forgotten but required values are marked with an "$\times$" in Figure 2.3b.

Saving four values per tier would fix the problem, but requires a larger matrix in shared memory, which would diminish occupancy and require more unnecessary values to be passed between nodes. Figure 2.4 shows our solution to this problem: a five-point stencil that requires two sub-timesteps per timestep—a predictor and a final step. This flattens the triangle or diamond in the time domain and requires four values per sub-timestep, but the two-row matrix may be used as described in Section 2.4.2 and illustrated by Figure 2.2 with minor adjustments. The same number of values are transferred between nodes in each communication, but inevitably more communication events are required to advance the solution. Conveniently, the predictor-corrector method ensures that all odd tiers, the second matrix row, will contain predictor values, and the bottom row will hold final values.

The problem that motivates our adjustment to the swept rule is the result of both the midpoint method and the five-point stencil discretization. Either of these circumstances alone would accommodate the method described in the previous section. Generally, the decomposition must be adjusted in this fashion when there are more than two sub-timesteps per timestep. It would need to be adjusted further for problems with more than four sub-timesteps per timestep.



(a) First step of swept rule for discretization with four, three-point stencil sub-timesteps per timestep.



(b) Using four sub-timesteps per timestep will overwrite values marked with an "×" before they are needed.

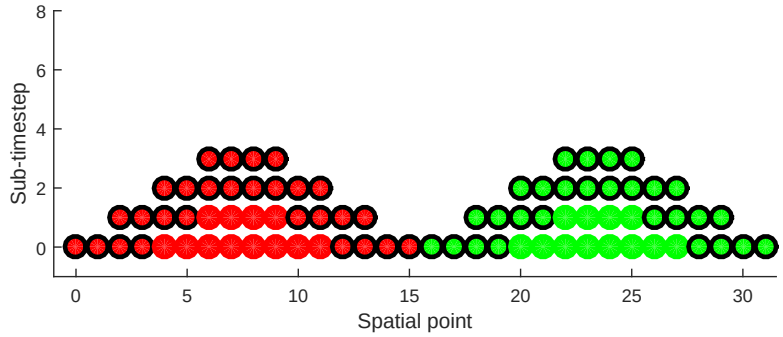
Figure 2.3: Conflicts in the domain of dependence for discretizations requiring more than two sub-timesteps per timestep [25].

2.5 Implementation

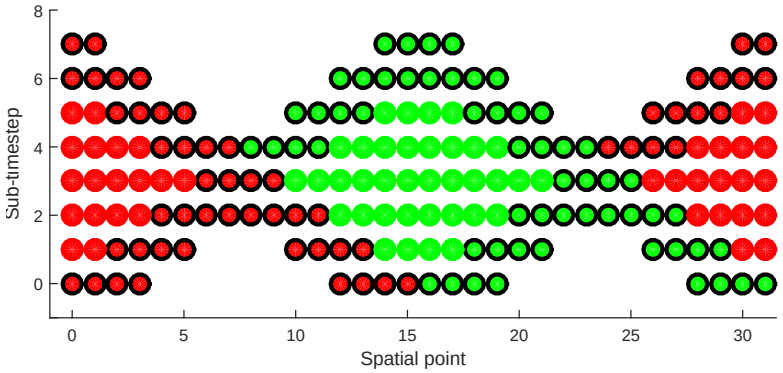
2.5.1 Swept rule variants

While the order of the swept domain of dependence is a natural consequence of the equations and numerical scheme, the computational hierarchy available on GPUs allows this structure to be implemented in different ways. To more thoroughly investigate the effects of the swept rule, we consider three versions: `Shared`, `Hybrid`, and `Register`, and compare them with a basic decomposition algorithm, `Classic`. In all of these programs, one GPU thread is assigned to one spatial point.

The `Classic` algorithm is a naive GPU implementation of the numerical solution and



(a) First step in swept rule for second-order domain of dependence. The working array is able to be folded and passed as shown in Figure 2.2



(b) Second step in the swept rule second-order domain of dependence.

Figure 2.4: The first two steps of the swept rule for a numerical scheme with a second-order domain of dependence [25].

the baseline against which the efficacy of the swept rule is measured. It advances one sub-timestep per kernel call and uses global memory to store the working array. Figure 2.6 shows the structure of the procedure, and it is similar for all problems and discretizations.

We consider the `Shared` strategy for implementing the swept rule the most natural way to map the analogy of CPU nodes to GPU architecture. It is applied to every test case and is considered the "default" swept rule GPU version for comparing the performance of the GPU programs with their MPI-based CPU counterparts [2]. The `Shared` version treats a block as a node and uses shared memory for the working array. Each block has exclusive access to a shared memory space and may contain up to 1024 threads, so it has

```
__global__ void
upTriangle(const REAL *IC, REAL *right, REAL *left) {
//tid = threadIdx.x (bottom row idx),
//tidT = tid+blockDim.x (top row idx);
//tids and tidTs: stencil for each tidT and tid respectively.
//shareT = working array in shared memory
//Read initial data in from IC to shareT.

if (tid > 1 && tid <(blockDim.x-2)) {
  shareT[tidT]=predictorStep(shareT, tids);
  }
__syncthreads();
//4 is stencil length - 1
for (int k = 4; k<(blockDim.x/2); k+=4)
  {

  if (tid < (blockDim.x-k) && tid >= k) {
    shareT[tid]+=finalStep(shareT, tidTs);
    }
  k += 2;
  __syncthreads();
  if(tid < (blockDim.x-k) && tid >= k) {
    shareT[tidT]=predictorStep(shareT,tids);
    }
  __syncthreads();
  }
//Read out the edges to left and right arrays.
}
```

Figure 2.5: Main loop of the starting kernel for the swept rule as illustrated by Figure 2.4a.

```
__global__ void
classicKS(const REAL *ks_in, REAL *ks_out, bool final)
{
//Global Thread ID
int gid = blockDim.x * blockIdx.x + threadIdx.x;
//number of spatial points - 1
int lastidx = ((blockDim.x*gridDim.x)-1);
//Stencil indices.
int gids[5];

//True for all spatial points from periodic BCs.
#pragma unroll
for (int k = -2; k<3; k++) {
  gids[k+2] = (gid + k) & lastidx;
  }
//Final is false for predictor step, true otherwise.
if (final) {
  ks_out[gid] += finalStep(ks_in, gids);
  }
else {
  ks_out[gid] = predictorStep(ks_in, gids);
  }
}
```

Figure 2.6: `Classic` kernel for Kuramoto-Sivashinsky solver. Final and predictor step functions can be written in C with `__device__` keyword.

access to fast memory and the capacity for various node sizes. There are some drawbacks to this version: a high number of idle threads for sub-timesteps where the domain of dependence contains relatively few spatial points, poor utilization of CPU resources, and the fact that shared memory is not the fastest memory type [19].

The `Hybrid` strategy uses the same GPU procedure as `Shared`, uses the CPU to compute the node that is split across the boundary, as seen in Figure 2.1b, Transfers between the host and device are costly operations, but the devices can execute instructions concurrently—so if the CPU can complete the boundary node before the GPU finishes the other nodes, no penalty arises. In this study, we apply this strategy to problems with non-periodic boundary conditions as a way to mitigate the underutilization of the CPU and the thread divergence that results from applying boundary conditions in a GPU kernel.

The `Register` approach is applied to problems with periodic boundary conditions because of the difficulty involved in applying boundary conditions using warp shuffle functions. This implementation limits the number of points in a node to the size of a warp, 32 threads (which has been constant over several iterations of Nvidia GPUs). In

this version the values are initially read from global memory to shared memory to the registers. Passing the values to the intermediate shared memory is necessary because the shuffle operations that trade registers between threads only operate on active threads in the warp being called; if some threads are masked, they will be unable to supply the necessary stencil values. Thus, data must be moved between memory levels at each tier rather than once at the start and end of the kernel. This seriously limits the `Register` approach, but it still warrants exploration since registers are the fastest memory type.

2.5.2 Test cases

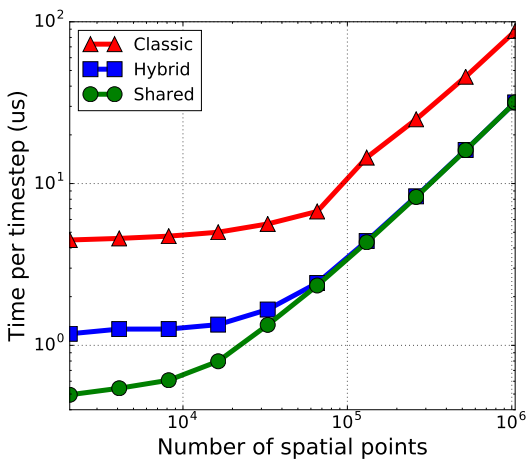
We present three test cases to demonstrate the performance and functionality of the GPU-based swept rule in one spatial dimension: the heat equation, Kuramoto–Sivashinsky (KS) equation, and Euler equations for compressible flow. Appendices A.2–A.4 contain the full derivations of the procedures for the numerical solutions. First, we chose the heat equation for its simplicity and familiarity. Here it is discretized with a first-order scheme using forward differencing in time and central in space. Next, we selected the KS equation to demonstrate the swept rule for higher-order, nonlinear PDEs. We discretized the KS equation with second-order, central differencing in space, which requires a five-point stencil, and a second-order Runge–Kutta method in time. Lastly we chose to solve the Euler equations, a system of quasilinear, hyperbolic equations for describing compressible, inviscid flow. The conservative form of these equations are applied to the Sod shock tube problem to demonstrate the application of the swept rule to a canonical CFD problem involving discontinuities and several dependent variables. These equations are discretized with a second-order, finite-volume scheme in space and a second-order method in time. The heat equation requires a first-order domain of dependence, while the KS and Euler equations require second-order domains of dependence. These problems also provide examples of various types of boundary conditions. The heat and Euler equations are solved with reflective and Dirichlet boundary conditions, respectively; therefore, the boundary conditions must be imposed with control flow. The KS equation uses periodic boundary conditions, which is a convenient formulation for the swept rule that splits a node across the boundary.

2.6 Results and discussion

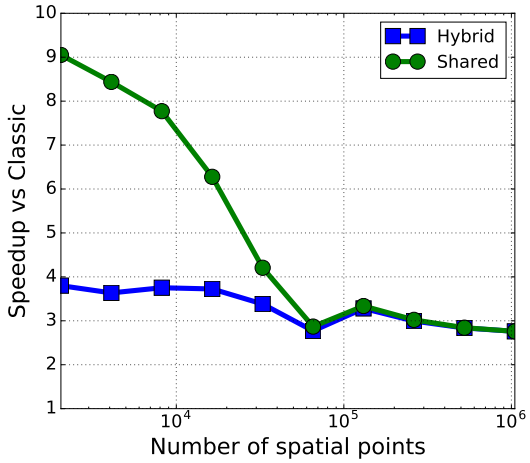
All tests presented here were performed on a single workstation with a Tesla K40c GPU and an Intel Xeon 2630-E5 CPU with eight cores and 16 potential threads. The GPU-based swept rule algorithms and test cases were implemented in `1DSweptCUDA v2` [26]. For the results we present here, each program was executed in double precision with $\{2^x \mid x \in \mathbb{N} \mid 4 < x < 11\}$ threads per block and $\{2^x \mid x \in \mathbb{N} \mid 10 < x < 21\}$ spatial points. Each run advanced 50,000 timesteps and recorded the average time per timestep; the initial GPU memory allocations and data transfer between the host and device are included in the overall time measurement. Then, we collected the best time per timestep for each number of spatial points. We repeated this procedure five times and took the average to obtain the results presented here. There were no significant differences in the results between the tests for the same configuration.

Figures 2.7a and 2.8a show the execution time per timestep for all algorithms applied to the heat and KS equations. When applied to these problems, the swept rule algorithm outperforms the `Classic` procedure, and the GPU-only shared memory version, `Shared`, is faster than the alternate swept rule versions. Figures 2.7b and 2.8b show the speedup of the swept rule programs, or the ratio of time costs with the `Classic` version. Both cases exhibit similar performance patterns: `Shared` generally provides a larger speedup for small spatial domains ($< 10^4$ spatial points), but only a $2\times$ speedup for large ones ($> 10^5$ spatial points). Figures 2.7a and 2.8a show the performance trends of the algorithms with respect to the spatial domain size. Their time costs are insensitive to the spatial domain at smaller domain sizes and grow linearly with increasing domain size after about 3×10^5 spatial points. The similarity in the performance trends of the heat and KS programs is intuitive; although the KS equation is nonlinear, fourth-order, and discretized with a higher-order scheme, both are continuous, scalar equations and use finite difference schemes.

The speedup of the swept rule declines as the number of spatial points in the domain increases, caused primarily by occupancy and the fixed cost of kernel launch. The occupancy of a kernel is the number of threads that can reside concurrently on a streaming multiprocessor at launch. This quantity is determined by the block size, since blocks of threads are indivisible, and the resources requested by the kernel. The swept rule requests more resources, specifically shared memory and registers, than `Classic` to store

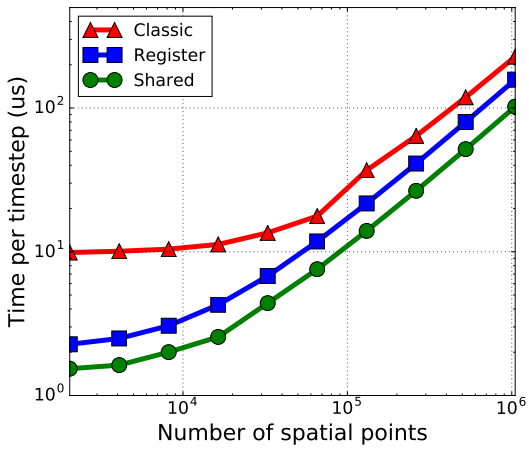


(a) Time cost of GPU classic and swept domain decomposition algorithms.

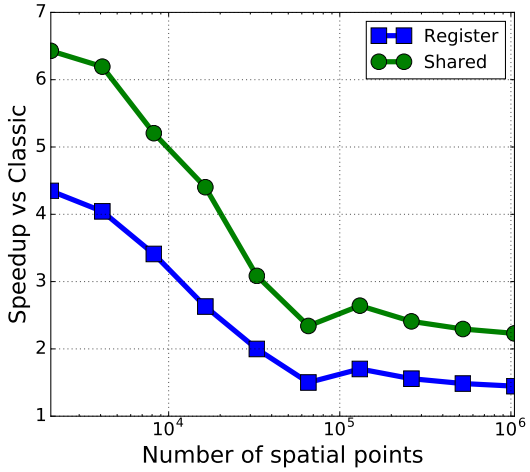


(b) Speedup of swept rule programs with respect to the `Classic` version.

Figure 2.7: Performance comparison of the GPU heat equation programs [25].



(a) Time cost of GPU classic and swept domain decomposition algorithms.



(b) Speedup of swept rule programs with respect to the `Classic` version.

Figure 2.8: Performance comparison of the GPU Kuramoto–Sivashinsky equation programs [25].

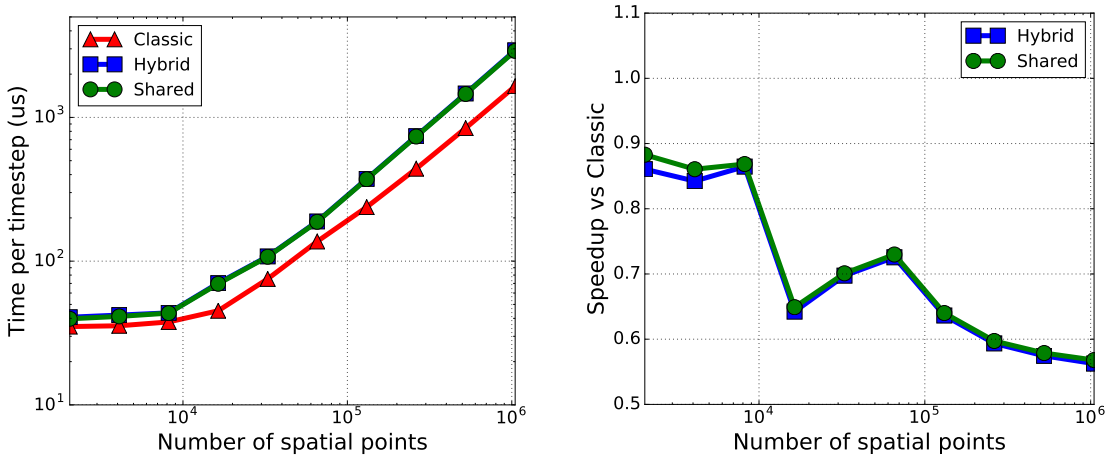
the working array and carry out its procedure, which involves more steps. If more resources are requested than are available to the streaming multiprocessors on the device,

the GPU launches waves of blocks. If the occupancy is limited by resource allocation, the kernel must begin launching waves of blocks on domains with fewer spatial points. Each wave must wait until the previous one completes before beginning; theoretically this implies two waves will cost about twice as much as one. As the number of spatial points in the domain grows, more blocks are launched, and the difference in the number of waves per kernel call increases more quickly for the swept rule program. This conclusion arises from the observation that the time cost of the swept rule versions begins growing linearly at a smaller spatial domain size, about 2×10^4 points, than `Classic`, about 6×10^4 points, as shown in Figures 2.7a and 2.8a.

By timing the launch of many empty kernels and taking their average, we measured the fixed cost of kernel launch on the testing workstation to be about 4 µs. We conclude that this cost dominates the performance of `Classic` at small problem sizes because it is quite close to the cost of each timestep for spatial domains with less than 3×10^4 spatial points for both the heat and KS equations. This fixed cost accounts for less time cost at larger spatial domain sizes where all kernels must launch several waves of blocks, so the portion of the swept rule speedup from avoiding kernel launches becomes negligible, and the overall speedup of the swept rule is reduced.

Each of the swept rule variants experiences these trends, but in all cases the `Shared` version performs better. Figure 2.7 shows the heat equation test, where the `Hybrid` version performs as well as `Shared` for large spatial domain sizes and 2–3 times worse for small ones. The `Hybrid` version uses the same GPU kernels as `Shared`, but uses the CPU to calculate the first node when it is split across the boundary. At smaller domain sizes, the data transfer between the host and device dominates the cost of the `Shared` scheme. At larger domain sizes, this cost drops in comparison with the overall costs, but at the same time the thread divergence caused by handling boundary conditions also drops in importance. Thus, the `Shared` and `Hybrid` methods perform similarly at larger domain sizes (i.e., above 6×10^4 spatial points).

Figure 2.8 shows the KS equation test, where, similar to the heat equation, the `Shared` version performs better in all cases. While registers are the fastest memory type, because of memory access rules the `Register` version must use a warp as a node, which limits the domain of dependence to 32 spatial points. A `Register` program with $n = 32$ must launch four times as many kernels to advance the same number of timesteps as `Shared` with $n = 128$ (most often the best block size). Despite these differences, the `Register`



(a) Time cost of GPU classic and swept domain decomposition algorithms. `Hybrid` and `Shared` lines overlap.

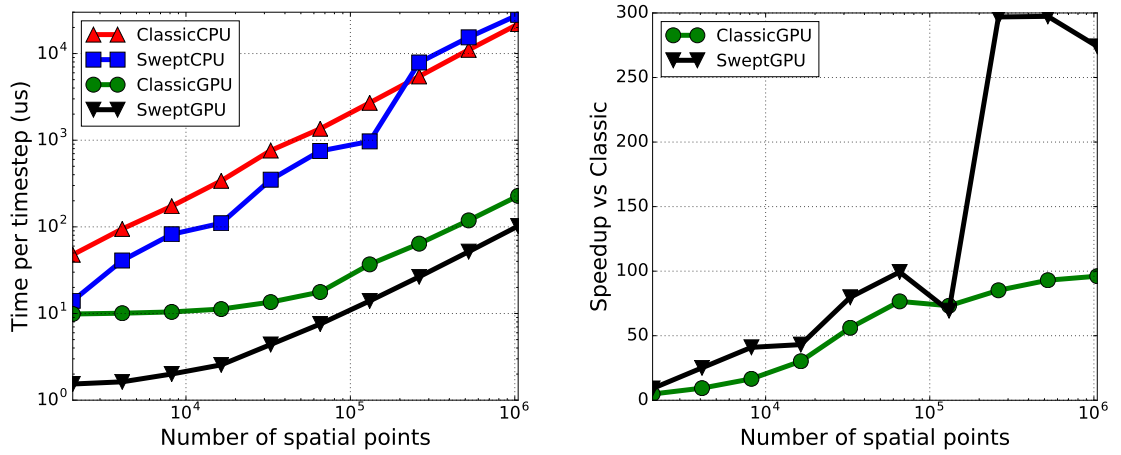
(b) Speedup of swept rule programs with respect to the `Classic` version.

Figure 2.9: Performance comparison of the GPU Euler equation programs [25].

version exhibits a similar performance trend to `Shared`, which shows about 1.5× the speedup of `Register` at all spatial domain sizes. The limit on node size results in a constant four timesteps per cycle for the KS equation, which only reduces the number of communication events by 1/8 compared with `Classic`.

In contrast to the previous test cases, Figure 2.9 shows that the classic decomposition outperforms the swept rule at all problem sizes when applied to the Euler equations. This result differs from the findings of Alhubail and Wang [3] for the swept rule implemented only with MPI on CPUs, which produces roughly equivalent speedups for both the KS and Euler equations. The GPU implementation of the swept rule for the Euler problem involves much greater arithmetic intensity than the other problems, causing greater low-level memory usage. This limits the performance of the swept scheme on a GPU more than a CPU, reducing the benefits of the scheme over the classic approach within a single GPU compared with the other problems. Those cases involve higher ratios of floating-point operations to memory accesses (both read and write). This will particularly degrade performance as the intra-domain timesteps proceed and nodes become inactive.

With regard to the potential performance of the swept rule, this result is not encouraging for more complex problems in more dimensions. However, on a larger scale in a



(a) Time cost of CPU and GPU programs for KS equation.

(b) Performance improvement of GPU program compared to the same algorithm on the CPU.

Figure 2.10: Performance comparison of CPU (MPI) and GPU (CUDA) programs for the KS equation [25].

cluster, where global memory is used for the working array and the communication to be avoided is through a physical network between processors, we expect that the swept rule will provide a benefit for these types of problems.

Figure 2.10 compares the CPU-based KS equation programs with the GPU algorithms, `Classic` and `Shared`. This CPU version is parallelized with MPI, similar to that originally presented by Alhubail and Wang [2]. Figure 2.10 refers to the `Shared` program as `SweptGPU` because it serves as the "default" swept rule version on the GPU for comparison with the CPU-based swept rule. The original CPU program was designed for a distributed-memory cluster, but the Alhubail and Wang's original performance study used only two processes on two CPUs [3]. The parallel CPU program was tested similarly to the GPU programs: it was run on the same workstation with the same spatial domain sizes and 2–16 threads. Here we compare the best results for each number of spatial points, which usually occurred with 16 threads. This did not degrade the performance compared with the original study; in fact, both CPU versions of the program performed significantly better, and for most spatial domain sizes the CPU-based swept rule improved by about 6 $\times$. Since the test used up to eight times the number of threads,

this result supports the validity of repurposing this code and comparing the result with the GPU versions.

Figure 2.10 illustrates the performance benefits of the GPU architecture more than the swept rule itself. On small spatial domains (e.g., less than 1×10^4 points) the GPU can assign one thread to each spatial point and process all in a single wave. There it improves performance over the CPU version by less than $50\times$, because the work does not fully utilize the GPU. On larger spatial domains (e.g., more than 1×10^4 points), where the work completely utilizes the resources of the GPU, the performance increases continue growing with problem size. The increase of the GPU programs' improvement with respect to the number of points in the spatial domain inverts the trend of the swept rule's improvement with respect to the `Classic` kernel. Both GPU algorithm types show a speedup of about $5\times$ for the smallest spatial domain. At the other end, the speedup grows to about $300\times$ for the swept and $100\times$ for the classic domain decomposition.

2.7 Conclusions

In this study we compared the time per timestep of three swept rule time-space decomposition implementations to a simple domain decomposition scheme, `Classic`, for GPU accelerators. These programs were evaluated on a range of spatial domain sizes. The `Classic` and `Shared` programs were also compared with their CPU-based counterparts parallelized with MPI.

Generally, the swept rule performs better relative to `Classic` when the kernel launch cost is a substantial element of the program cost and the spatial grid has fewer points than concurrently launchable threads on the GPU. Once this is exceeded, the launch cost penalty becomes negligible and greater resource usage penalizes the swept rule in the form of reduced occupancy and reduced availability of the L1 cache, which is reserved for shared memory. But, like the initial performance penalty for the `Classic` program, the resource usage penalty does not scale and all programs see their time cost rise nearly linearly with respect to spatial domain size after about 10^5 spatial points.

Both alternate swept-rule procedures, `Hybrid` and `Register`, performed slightly worse than the `Shared` version. We attribute the failure of the `Hybrid` routine to improve performance to improvements made in the handling of the boundary conditions in `Shared` rather than the cost of host-to-device communication, which is handled with asynchronous

streams. Though the `Hybrid` version does not improve performance, it does not substantially degrade it either, especially with large spatial domain sizes. Ultimately, `Hybrid` computation seeks to solve a problem more efficient to solve within the GPU paradigm. The `Register` computation shows more promise but falls short because of the limited node size.

This study also shows that implementing a `Classic` decomposition on the GPU for an explicit numerical scheme is simple and may result in noticeable performance improvements. This may be enough for many applications, but in performance-critical cases the swept rule may further reduce execution time. The performance of the `Classic` decomposition programs may be improved by altering the synchronization method. Implicit synchronization incurs the cost of kernel launch at each sub-timestep, and this kernel executes so quickly at small problem sizes that synchronization cost dominates the performance of the program. Using off-the-shelf, GPU-based synchronization [43] could also provide performance benefits for the swept rule, and in particular `Register`, which is also limited by kernel launch cost for small spatial domains.

The specific results presented here depend on the hardware used. A commercial GeForce GPU with Kepler architecture would perform worse than the Tesla K40c, which is designed for computation as opposed to actual graphics. Despite the significant performance increase shown here, the Tesla K40c is three-year-old technology. The current state-of-the-art GPGPU with Pascal architecture, the Tesla P100, offers twice the K40c base clock speed, nearly four times the number of streaming multiprocessors, and an extra 16 kB of shared memory independent of the L1 cache. Additional dedicated shared memory could dramatically impact the swept rule's performance for the Euler equations. We predict that this device could at least halve the execution times shown here and maintain insensitivity to problem size up to over 10^5 spatial points, potentially resulting in speedups on the order of 10^3 over CPU parallel versions. Future GPU accelerators could further improve performance, as well as devices with similar SIMT architectures like Intel MIC/Xeon Phi.

Our future work will focus on further developing the swept rule for use on distributed memory systems with heterogeneous node architectures and implementing the swept rule in higher dimensions on the GPU. We expect the performance benefits to increase from the swept rule when applied to a distributed memory system comprised of nodes with CPUs and GPUs, where network communications incur more latency than intra-GPU

memory access. However, we anticipate the performance benefits of the swept rule will diminish for higher dimensions; in two dimensions the swept rule requires three stages of inter-node communication to advance one cycle, analogous to a single diamond in one dimension. This, along with the associated increase in required memory allocation, arithmetic intensity, and kernel launch events could limit the performance of the swept rule as observed by Alhubail et al. [5]. In spite of these challenges, recent developments such as grid-wide synchronization and increased shared memory capacity could aid the performance of the swept rule and permit the exploration of additional design options.

Applying the swept rule for explicit PDE solutions to heterogeneous computing systems

Daniel J. Magee and Kyle E. Niemeyer

Under review.

Chapter 3: Applying the swept rule for explicit PDE solutions to heterogeneous computing systems

Abstract

Applications that exploit high performance computing (HPC) systems have become invaluable in academia and industry over the past two decades. The most important development of the last decade in HPC from the hardware perspective has been the General Purpose Graphics Processing Unit (GPGPU), a class of massively parallel devices that now contributes a substantial balance of the computational power in the top 500 supercomputers. As these systems grow, barriers to increased performance arise from small costs accumulated over innumerable iterations such as latency, the fixed cost of memory accesses, which becomes significantly larger when access requires communication between two distant CPU processes. The time-space decomposition rule is a communication-avoiding technique for time-stepping stencil update formulas that attempts to sidestep a significant amount of latency costs. This work extends the swept rule by targeting heterogeneous, CPU/GPU architectures representative of current and future HPC systems. We compare our approach to a naive decomposition scheme with two test equations using an MPI+CUDA pattern on 40 processes over two nodes containing one GPU. We show that the swept rule produces a 4–18 $\times$ speedup with the heat equation and a 1.5–3 $\times$ speedup on the Euler equations using the same processors and work distribution. These results demonstrate the effectiveness of the swept rule for different levels of problem complexity on compute systems that incur a substantial portion of their overall cost from latency.

3.1 Introduction

Computational fluid dynamics simulations are at the heart of technological development in industries vital to high and rising standards of living around the world. Performing simulations at a level of fidelity necessary for insight consumes more resources than individual workstations can reasonably accommodate. As a result, they are generally performed on

clusters of many nodes comprising several multi-core CPUs—and increasingly with other specialized "accelerator" co-processors. These heterogeneous computing systems have become ubiquitous in areas of research dependent on large amounts of data, complex numerical transformations, or densely connected systems of constraints. Addressing these problems reveals a further horizon where new, more complex questions emerge. Steady progress requires increasing development in large-scale computational systems; solutions from hardware and software are required to supply the necessary throughput for demand in numerous fields, including several not known for their reliance on computational science such as medical diagnostics, genetics, marketing and biology.

In many ways, progress has been sustained through the development of accelerators or co-processors which augment the computational capabilities of the CPU, including general purpose graphics processing units (GPGPU). These devices have grown in power and complexity over the last decade, leading to an increasing reliance on them for energy-efficient floating-point processing power in clusters [1]. However, as clusters grow in complexity, computational power, and physical size, latency and bandwidth costs limit the performance of applications that require regular inter-node communication. Bandwidth is the amount of memory that can be communicated per unit of time, and latency is the fixed cost of a communication event—the travel time of the leading bit over the connection. Traditionally, solving partial differential equations (PDEs) using explicit schemes on clusters involves domain decomposition, with required internode communication of small data packets for boundary information at every time step. The frequency of these communication events renders their fixed cost (i.e., latency) a significant barrier to the scalability and robustness of stencil update algorithms on large multi-node computing systems.

This project seeks to address, however nascently, two of the challenges on the route to exascale computing systems identified by Alexandrov in his editorial in a recent special issue of the *Journal of Computational Science*: the need for "novel mathematical methods... to hide network and memory latency, have very high computation/communication overlap, have minimal communication, have fewer synchronization points", and "mathematical methods developed and corresponding scientific algorithms need to match these architectures [standard processors and GPGPUs] to extract the most performance. This includes different system-specific levels of parallelism as well as co-scheduling of computation" [1].

In this paper, we describe the development and performance analysis of a PDE solver targeting heterogeneous computing systems (i.e., CPU/GPU) using the swept rule, a communication-avoiding, latency-hiding domain decomposition scheme [3, 5]. Section 3.2 describes recent work on domain decomposition schemes with particular attention to applications involving PDEs and heterogeneous systems. Section 3.3 describes the questions this study seeks to answer. Section 3.4 introduces swept time-space decomposition and discusses the experimental hardware, procedure, and factors used to evaluate the program performance. The section also analyzes and justifies the design decisions we made concerning elements of the program which we held constant but could potentially have investigated. In Section 3.5 we present the results of the tests and describe the hardware and the testing procedures used; lastly in Section 3.6 we draw further conclusions, describe future challenges in this project, and outline plans for prioritizing and overcoming them.

3.2 Related Work

In our previous work [27] we investigated methods for exploiting the exposed GPU memory hierarchy to effectively execute swept time-space decomposition for stencil computation on a single GPU. Alhubail et al. [2, 3, 5] first developed the swept rule for CPU-based operation one and two dimensions; Alhubail and Wang also demonstrated how complex schemes can be decomposed into update formulas suitable for the swept rule [40]. We use this technique, or "`lengthening`", in the implementation of the swept rule discussed in this work, and contrast it with another method for dealing with complex schemes: "`flattening`", which our previous GPU-only swept rule implementation used [27]. Section 3.4.2 quantitatively compares the two techniques. In addition, Alhubail and Wang applied this procedure to automatically generate C source code for solving the heat and Kuramoto–Sivashinsky equations using the swept rule on CPU-based systems [4]. These articles—those written by Alhubail and Wang and our previous study—comprise the body of work on the swept rule to date, which this work expands upon.

Memory hierarchies are defined by a series of locations where memory is scarce and quickly accessible to plentiful and inefficient. By storing the working data in memory more accessible to the processor as long as possible, communication-avoiding algorithms speed up computation by reducing inter-process communication or global memory accesses in

parallel programs. Swept time-space decomposition is a type communication-avoiding algorithm because it seeks to reduce the number of communication events between processor and less accessible memory resources, unlike most communication-avoiding algorithms though, it does not perform redundant communications. The heterogeneous communication-avoiding LU factorization algorithm presented by Baboulin et al. [7] investigates the task splitting between the GPU and CPU and minimizes inter-device communication. Their results show an appreciable benefit by splitting the types of tasks performed on the CPU and GPU, reducing overall communication, and effectively overlapping computation and communication.

Studies of stencil optimization techniques over the last decade often address concerns closely related to the work presented here. Datta et al. [13] explored domain decomposition with various launch parameters on various heterogeneous architectures and nested domain decomposition within levels of the memory hierarchy. Malas et al. [28] previously explored similar diamond tiling methods, which use the data dependency of the grid to improve cache usage.

Swept time-space decomposition is also conceptually related to parallel-in-time methods [18], such as multigrid-reduction-in-time [16]. These algorithms overcome the interdependence of solutions in the time domain, and parallelize the time dimension as if spatial. The technique iterates over a series of fine and course grids using an initial guess for the entire solution domain, and effectively smoothes out the errors in the solution. Historically, parallel-in-time methods were considered unsuitable for nonlinear problems since the use of coarse grids substantially degraded efficiency and accuracy [3]. However, recent developments applying optimization and auto-tuning techniques have matched the scaling of linear solvers [15]. Parareal is a parallel-in-time method that solves multiple time steps in parallel on a fine grid and corrects the results on a coarse grid until the solution converges, resulting in a solution with the accuracy of the fine grid. Wu and Zhou proposed a new local time-integrator for this method that shows considerable promise for accelerating convergence rates in fractional differential equations [42].

Distributed, remote, multi-node systems have become the centers of scientific computing over the last decade, and have become increasingly heterogeneous in recent years. Therefore, domain decomposition on these systems has received a good deal of recent attention. In particular, Huerta et al. used methods from process engineering, including experimental design and non-continuous linear models in an experimental parameter

space paradigm, to investigate the performance of a well known benchmark used to rank HPC clusters, HPL, with respect to workload division on a heterogeneous system [21]. From our perspective, this is an underused technique in the field of HPC, which we could apply to great effect in future studies with a more mature code base. However, at our current stage, such a thoroughgoing analysis would not provide actionable insights beyond what we have already gleaned from our comparatively simpler methods.

3.3 Objectives

This study concerns the construction and analysis of a program that applies swept time-space decomposition to explicit stencil computations intended for distributed memory systems with heterogeneous architecture, that is, computational operations are performed by several CPUs and co-processors, in particular Nvidia GPUs. The software is written in C++ and CUDA and uses the Message Passing Interface (MPI) library [10] to communicate between CPU processes and the CUDA API to communicate between the GPU and the CPU.

While stencil computation is a relatively simple procedure, applying linear operations to individual spatial points and their neighbors and the complexities introduced both by a heterogeneous architecture and swept time-space decomposition require a significant number of design decisions. In this work we investigated the performance impact of the most immediately salient and configurable decisions, and constrained other potential variations with reasonable or previously investigated values. Our investigations focus on answering the following questions:

1. Does the swept rule reduce time-cost under optimal launch bounds over the domain of grid sizes?

2. How much work should we give to the GPU in a heterogeneous system?

3. How should we decompose the stencils in multi-step methods? (Further discussion in Section 3.4.2)

4. Does the size of the domain of dependence substantially affect performance?

3.4 Methodology

3.4.1 Swept time-space decomposition

The swept rule exhausts the domain of dependence—the portion of the space-time grid that can be solved given a set of initial values, referred to here as a "block"—before passing the grid points on the borders of each process. We refer to the program that implements the swept rule as `Swept`, and the program that uses naive domain decomposition, that is passing between processes at each timestep, is referred to as `Classic`. This way the simulation may continue until no spatial points have available stencils; the required values may then be passed to the neighboring process (i.e., neighboring subdomain) in a single communication event. Both Alhubail and Wang, and Magee and Niemeyer, provide detailed explanations and graphical depictions of the swept rule in one dimension, for various architectures [3, 27].

The heterogeneous one-dimensional swept rule begins by partitioning the computational grid and allocating space for the working array in each process. In this case, the working array is of type `states`, a plain old data C struct that contains the dependent and intermediate variables needed to continue the procedure from any time step. Working array size is determined by the number of domains of dependence controlled by the process, $nBlocks$, and the number of spatial points covered by a domain of dependence, tpb (threads per block). Here we use "block" to represent a domain of dependence; it comes from the GPU/CUDA construct representing a collection of threads. The program allocates space for $nBlocks \times tpb + (tpb + 2)/2$ spatial points and initializes the first $nBlocks \times tpb + 2$ points. The initialized points require two extra slots so the edge domains can build a full domain width on their first step. Interior domains in the process share their edges with their neighbors; there is no risk of race conditions since even the simplest numerical scheme requires at least two values in the `state` struct, which allows the procedure to alternate reading and writing those values. Therefore, even as a domain writes on an edge data point that its neighbor must read, the value the neighbor requires is not modified.

The first cycle completes when each domain has progressed to the sub-time step $tpb/2$ where it has computed two values at the center of the spatial domain. At this point each process passes the first $tpb/2 + 1$ values in its array to the left neighboring process. Each

process receives the neighbor's buffer and places it in the last $tpb/2 + 1$ slots; that is, starting at the $nBlocks \times tpb$ index. It proceeds by performing the same computation on the centerpoints, starting at global index $tpb - 1$ (adjusted index $tpb/2 - 1$), of the new array and filling in the uncomputed grid points at successive sub-time steps with a widening spatial window until it reaches a sub-time step that has not been explored at any spatial point and proceeds with a contracting window. Geometrically, the first cycle completes a triangle, the second completes a diamond. When the diamond is complete, it passes the last $tpb/2 + 1$ time steps in the array and inputs the received buffer starting at position 0. Now it performs the diamond procedure again, this time the global and adjusted index are identical and it starts at index $tpb/2 - 1$.

The procedure continues in this fashion until the final time step is reached, at which point it stops after the expanding window reaches the domain width and outputs the solution which is now current at the same time step within and across all domains and processes. Therefore, the triangle functions are only used twice if no intermediate time step results are output, the rest of the cycles are completed in a diamond shape.

3.4.2 Primary data structure

Implementing the swept rule for problems amenable to single-step PDE schemes is straightforward, but dealing with more realistic problems often requires more complex, often multi-step numerical schemes. Managing the working array and developing a common interface for these schemes, requires making design decisions that have substantial impacts on performance.

One strategy for dealing with this complexity we term `flattening` since it flattens the domain of dependence in the time dimension by combining several potential atomic stages into single steps with wider stencils. This strategy is more memory efficient for the working array which contains instances of the primary data structure at each spatial point, but it cannot easily accommodate different methods and equations. It also introduces additional complexity from parsing the arrays and requires additional register memory for function and kernel arguments and ancillary variables.

In the new implementation shown here we use the `lengthening` strategy, also referred to as "atomic decomposition", which is instantiated as a struct to generalize the stages into a user-defined data type. It requires more memory to be used in the primary data

```
// Q = {rho, rho*u, rho*E}
struct states {
    double3 Q[2]; // State Variables
    double Pr; // Pressure ratio
};

__device__ __host__
void stepUpdate(states *state, const int idx, const int
      tstep)
{
    int ts = tstep % 4; // 4 is number of steps in cycle
    if (tstep & 1) pressureRatio(state, idx, ts);
    else           eulerStep(state, idx, ts);
}

__global__ void classicStep(states *state, const int
     tstep)
{
    int gid = blockDim.x * blockIdx.x + threadIdx.x + 1;
    stepUpdate(state, gid, tstep);
}
```

Figure 3.1: Skeleton for the `lengthening` method in the `Classic` program. The `states` structure contains all the information to step forward at any point. The user is only responsible for writing the `eulerStep` and `pressureRatio` functions and accessing the correct members based on the timestep count

```
__global__ void classicStep(const double *s_in, double
     *s_out, bool final)
{
int gid = blockDim.x * blockIdx.x + threadIdx.x;
//number of spatial points - 1
int lastidx = ((blockDim.x*gridDim.x));
int gids[5];

for (int k = -2; k<3; k++) gids[k+2] = (gid + k) %
     lastidx;

//Final is false for predictor step, true otherwise.
if (final) s_out[gid] += finalStep(s_in, gids);
else      s_out[gid] = predictorStep(s_in, gids);
}
```

Figure 3.2: Skeleton for the `flattening` method in the `Classic` program. The subtimesteps are compressed to a step with a wider stencil. The two arrays which alternate reading and writing are explicitly passed and traded in the calling function.

structure; for instance, our `flattening` version of the Euler equations carried six doubles per spatial point since the pressure ratio used by the limiter was rolled into the flattened step. These strategies are described in Figures 3.1 and 3.2. By restricting the stencil to three points, the `lengthening` method requires the pressure ratio to be stored and passed through the memory hierarchy meaning the data structure carries seven doubles per spatial point for the Euler equation.

To gauge the influence of this change in primary data structure, we compared the performance of the Kuramoto–Sivashinksy (KS) equation, using a second-order Runge–Kutta finite-difference method in time and central differencing in space with periodic boundary and initial conditions. A complete explanation of this method applied to the KS equation can be found in the appendix of our previous paper [27]. We implemented each combination of the classic and swept decomposition techniques and the `flattening` and `lengthening` data structures. We used the KS equation because it was easy to adapt to both styles due to its periodic boundary conditions, and since it requires four atomic

stages in the lengthened structure such that the versions are suitably different.

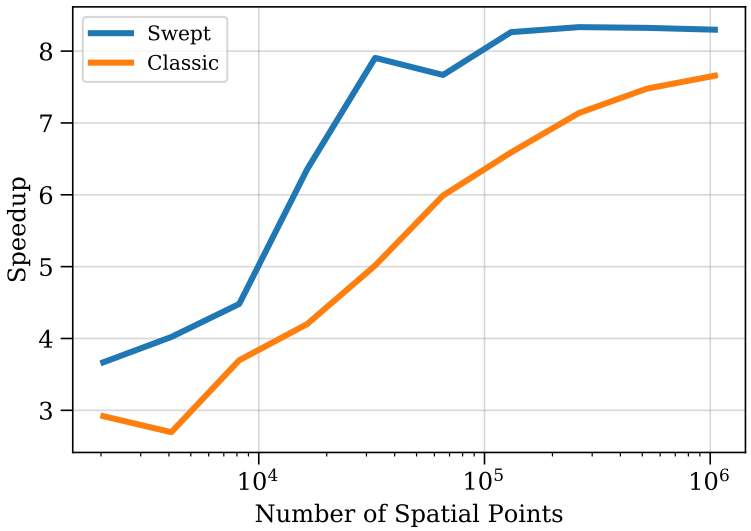


Figure 3.3: Speedup of `flattening` compared to the same scheme using `lengthening` applied to the KS equation on the GPU only.

Figure 3.3 compares the performance of the memory storage techniques in an experiment executed on a workstation with an Intel Xeon 2630-E5 v3 and an Nvidia Tesla K40c. The `Classic` program using the `flattening` method using is compared to the same program using the `lengthening` method and likewise for `Swept`. As Figure 3.3 shows, the `flattening` method is faster than `lengthening` for both decomposition methods and gets faster as the number of spatial points increases. There is not much difference in the trends between swept and classic decomposition methods, but it does appear that the less performative data structure affects the `Swept` program more. Much of the reason that `lengthening` is substantially less performative is an artifact of GPU architecture, so it is difficult to generalize the results of this study to heterogeneous systems. GPU memory accesses, from global and shared memory, are more sensitive to irregularities because of the parallel nature of the device as a whole; it is not designed to be used serially. The extra memory requirements of the `lengthening` method also consume limited shared memory resources on the GPU, which diminishes the L1 cache capacity used to accelerate global memory accesses on Kepler-generation GPUs. While locality is a significant issue

for effective CPU memory accesses, it has a larger impact on GPU performance.

We feel that it is important to present these findings so that others who implement the swept rule will have a more thorough understanding of the tradeoffs inherent in the program design choices across architectures. But, since the `Swept` and `Classic` algorithms perform similarly compared to the `flattening` method, the conclusions we derive from our experiments using the `lengthening` method remain valid. And we believe that, in this case, the values that the `lengthening` method provides: extensibility and regularity are of greater value than the absolute best performance.

3.4.3 Program design features

Our earlier implementation of the swept rule for GPUs [27] required creating separate single-file C++/CUDA programs for each problem. We have determined that this approach is insufficient for enabling the support for further exploration that is essential to the accurate analysis of highly architecture and problem dependent algorithms. Our previous analysis showed that the performance characteristics of the GPU-based swept rule depend on the boundary conditions, numerical method, and governing equation(s). In this work we endeavor to create a convenient and reusable interface through which other can reproduce our experiments, explore different equations and numerical methods to examine performance characteristics the performance characteristics of the swept rule.

We have created this interface by separating the domain decomposition and grid generation, and passing a map between the equation-specific part of the code, defined by the user, and the generic part of the code. In practice, the user defines an initialization procedure that defines any constant terms in the governing equation(s), e.g., the Fourier number in the heat equation. The user is guaranteed to receive the standard grid variables, e.g., Δx, Δt, in the map along with any other values needed to define the constant terms, i.e., the thermal diffusivity for the heat equation. The user defines these non-standard constant terms in a JSON file passed to the program on the command line or as command line key/value pair arguments. By defining these fundamental concepts in a separate JSON file, variables that define the grid or material constants in the equations can be redefined without the need to recompile.

This structure requires a solver interface that operates on every equation and numerical scheme the same way, which in turn requires the use of a different strategy for

decomposing more complex equations and domains. The clearest way to create a general form for a user-defined equation is to oblige the user to decompose the numerical formula into atomic stages, a series of steps requiring only three-point stencils as described by Wang [40], and provide a step counter in the root function to define the sequence of stages. The user must also define the data structure where results of each stage are collected, this structure takes the form of a plain old data struct named `states`.

Our first study showed that shared memory is the most effective means of exploiting the memory hierarchy for this application on this generation of GPUs [27]. We rely on this conclusion here to again use the GPU characteristics to determine the size of the domains of dependence. Each GPU thread is mapped to a single spatial point and each CPU process, having only one available thread, traverses a number of domains in serial. This limits the size of the domain of dependence to the allowable number of threads in each block launched in the GPU kernel, which depends on the occupancy of the most-restrictive kernel as determined by the shared memory and register resource requirements.

Our approach organizes the available processors by assigning each GPU on a node to one MPI process (i.e., CPU core) on the same node that has exclusive control over it; that process also manages one domain on either side of the GPU subdomain. Thus, to facilitate this, each MPI process comprises a positive, even number of subdomains. All subdomains on a node must contain the same number of points, and all MPI processes evaluate the same number of subdomains. Processes that control a GPU simply "contain" additional subdomains equal to the GPU affinity times the number of domains assigned to an MPI process. For example, a domain of 160 points could be decomposed into subdomains of 16 points on a node with four CPU cores and one GPU; each processor would compute the time-stepping for two subdomains, while the MPI process controlling the GPU comprises four subdomains in total (two on the CPU, and two on the GPU). This corresponds to a GPU affinity of one, since the GPU subdomain equals the CPU subdomain sizes.

Assigning a GPU to a single process reduces complexity and avoids using the GPU at the spatial boundaries where imposing boundary conditions causes thread divergence. This is useful from a conceptual standpoint, even though we previously found that boundary conditions only affect GPU performance in a minor way [27].

Our program uses the MPI+CUDA paradigm, and assigns one MPI process to each

core for the life of a program. We considered using an MPI+OMP+CUDA paradigm by assigning an MPI process to each socket, and launching threads from each process to occupy the individual cores, but recent work has shown that this approach rarely improves performance on clusters of limited size for finite volume or finite difference solvers [22, 24]. This conclusion has led widely used libraries, such as PETSc, to opt against a paradigm of threading within processes [30].

3.4.4 Experimental method

We endeavor to address the questions presented in Section 3.3 by varying three primary attributes of the decomposition: threads per block, GPU affinity, and grid size. We repeatedly executed our two test equations, the heat and Euler equations, over the experimental domain of these variables using `Swept` and `Classic`, exchanging borders every sub-time step, decomposition methods. In our program implementing the swept rule in one-dimension on heterogeneous systems, hSweep, threads per block is synonymous with the size of the domain-of-dependence, but we refer to it using GPU terminology because each domain is launched as a block of threads on the GPU. A block is an abstract grouping of threads that share an execution setting, a streaming multiprocessor, and access to a shared memory space, a portion of the GPU L1 cache. hSweep uses the swept rule to avoid communication between devices and processes and exploits the GPU memory hierarchy to operate on shared memory quantities closer to the processor. Since this multi-level memory scheme influences the swept-rule performance and GPU execution, the resultant effects are difficult to predict. The independent variables GPU affinity and grid size are more straightforward. The grid size is the total number of spatial points in the simulation domain, and is provided by the user; however, the program revises this number to provide a grid that fits the other program settings that the grid must accommodate: the threads per block, GPU affinity, and number of processes. The GPU affinity is the portion of the computational grid that is processed by the GPU, expressed as a ratio of the number of domains-of-dependence assigned to the GPU to those assigned to a single MPI process (on a CPU core). GPU affinity, like the other experimental variables, should be given as an integer, since we have determined that it is beneficial for the GPU to handle a larger portion of the overall grid than a single MPI process.

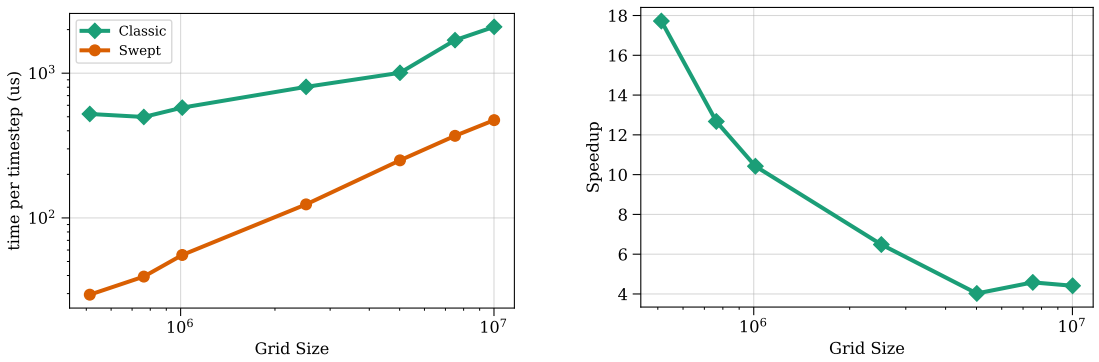
In our previous study of the swept rule [27], the experimental domain was clearly

defined by the particular properties of GPU architecture. Because a warp contains 32 threads and a block cannot exceed 1024 threads, here we constrained the number of threads per block, which is also the width of the domain of dependence, in our experiments to be a power of 2 from 32–1024. To enforce regularity, we constrained our experimental problem size—the number of spatial points in the grid—to be a power of 2 larger between 1024 and 2^{21}.

Using CPU parallelism across 40 processes and GPU affinity as a variable of interest in this study, eliminates the potential for regularity in the experimental grid. To remedy this, we relaxed the constraints on the experimental launch conditions so that the number of threads per block is required to be a multiple of 32 from 32–1024 rather than a power of two. In addition, at runtime the program uses the number of processes, threads per block, GPU affinity, and desired grid size to determine the closest grid size to the requested value that accommodates the constraints. This results in different grid sizes for the same experimental settings. To assess the performance at various settings, we interpolated each result to the requested grid size from the actual grid size.

The addition of GPU affinity as an independent variable introduces further complication to the experimental domain. While our experiments are constrained by GPU architecture in threads per block and by the number of processes and blocks in problem size, we initially have no clear indication of what the experimental limits of GPU affinity should be—so we took an iterative approach. First, we ran a screening study and executed the programs over a broad range of conditions: eight block sizes from 64–768, 11 GPU affinities from 0–80, and four grid sizes from 5×10^5–10^7. This showed us that the best affinity for all the programs would likely fall between 20–60 and that all threads per block values could provide the best performance. This was somewhat disappointing, since we had hoped to narrow the range for both GPU affinity and threads per block further in order to experiment on a finer increment of grid size in a reasonable amount of time. For the final experiment, we used the same block sizes, GPU affinity values from 20–60 in increments of 4, and seven grid sizes over the same range.

In this study, we solve the one-dimensional heat equation using a first-order forward in time, central in space method and Euler equations for a shock wave using a second-order finite-volume scheme with minmod limiter. Explanations of these methods can be found in the appendix of our previous paper [27].



(a) Time cost of heat equation program at best launch condition.

(b) Speedup of swept version at best launch condition.

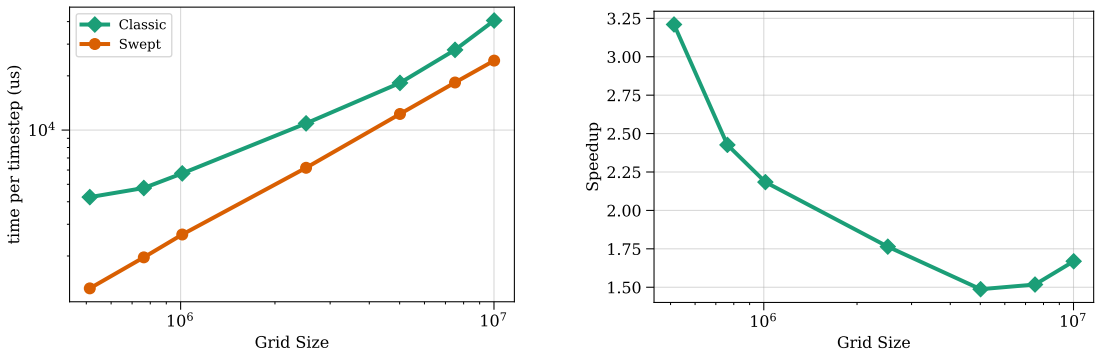
Figure 3.4: Performance comparison of the hSweep heat equation programs.

3.5 Results

All programs executed at least 4000 timesteps; they were compiled with CUDA v8 and `gcc v4.8.5`. All programs were launched with `mpirun v3.1.4`. Host-side memory management, grid generation, and initial conditions are not included in the timing measurement, but the measurement includes device-side global memory allocation and initial and final memory transfer between the host and device. The heterogeneous swept rule algorithms and test cases were implemented in `hSweep v2.0` [12].

We obtained the results presented here by running the `Classic` and `Swept` programs on two nodes of the Oregon State University College of Engineering cluster. Each node contains two sockets with Intel Xeon 2660-E5 v3 processors containing ten cores each operating at 2.60 GHz. An Nvidia Tesla K40m GPGPU is available to one of these nodes through a PCI connection.

Figure 3.4a compares the computational time cost per time step of the `Classic` and `Swept` algorithms applied to the heat equation. These results are generally consistent with our expectations for several reasons. First, we observe that in all previous experiments [27, 3], `Swept` achieves a period of strong speedup but reaches its asymptotic performance at smaller grid sizes than `Classic`. For `Classic` this occurs relatively soon after `Swept`, and the time cost of both algorithms with respect to grid size grows similarly so the relative speedup of `Swept` declines as well although the absolute speedup remains



(a) Time cost of Euler equation program at best launch condition.

(b) Speedup of swept version at best launch condition.

Figure 3.5: Performance comparison of the hSweep Euler equations programs.

constant.

Second, although the minimum grid size in this study is about 100× times larger than our previous study, and the maximum grid size is about 10× times larger, Figure 3.4b shows a remarkably similar trend in the speedup of swept decomposition. In general the speedup exhibited by the heterogeneous case is 2× the speedup at the analogous grid size in the GPU-only case over the experimental range. The relative speedup of the swept rule is expected since the latency that our program avoids is a much larger cost in internode communication.

Figure 3.5 compares the time cost per timestep of the `Classic` and `Swept` algorithms with the Euler equations and shows the same performance trends as the heat equation. In this case, the communication costs that the program avoids are significant enough that swept decomposition provides a tangible benefit despite the extra complexity, management, and memory resources that it requires. This shows that swept time-space domain decomposition is a viable method for complex equations in one dimension on systems with substantial communication costs and various architectures.

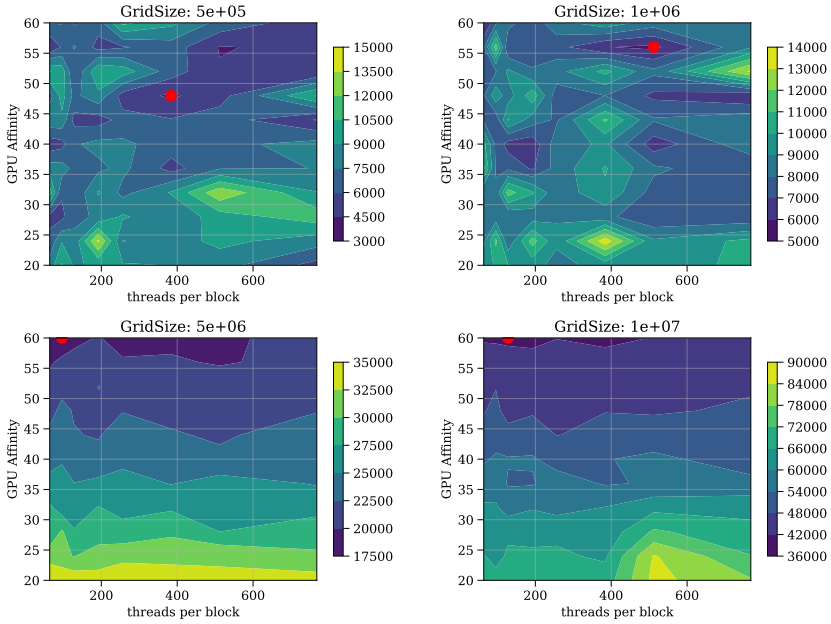
Figures 3.6 and 3.7 show contour maps of the results of the complete experiment, as described at the end of Section 3.4.4, on the equations. By creating similar maps in the first stage of the experiment, we were able to narrow the range of possible best affinity values for all experimental settings and run an experiment with greater granularity in the affinity dimension. In the interest of most accurately describing the performance of

our program, we present the results of the final rather than the screening experiment.

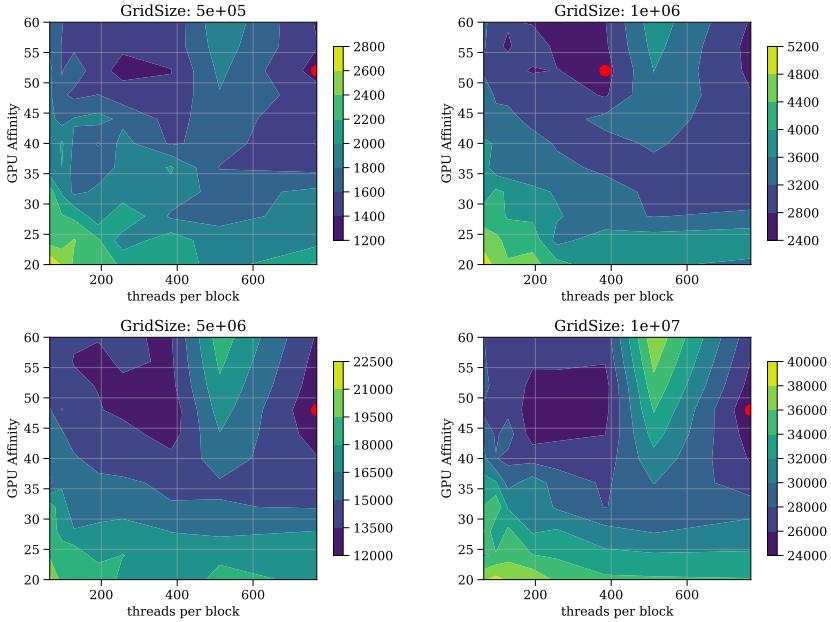
Figure 3.6 shows interesting characteristics of the programs that vary by runtime configuration and decomposition method. Notably, these results show that for the Euler equations, the `Swept` program often achieves best performance at GPU affinities between 45–55 and often does best with 768 grid points in the domain of dependence, but performs particularly poorly with 512 points. From other perspectives the performance profile of the `Swept` Euler equations program appears quite regular, so a sudden drop in performance followed by a substantial increase is unexpected. The consistency of the influence of GPU affinity allows further studies to explore more granular performance characteristics. While the saddle along the threads per block axis is problematic for our general recommendations, we also observe that the program consistently exhibits similar performance from 192–384 threads per block.

Additionally, we observe from these maps that `Swept` produces a more orderly performance profile than `Classic`. For grid sizes under 10^6 spatial points, the classic decomposition technique produce peaks and valleys over a wide range of threads per block and GPU affinities. On larger grids, that chaos regularizes and the best performance occurs with smaller grids and larger GPU affinities. This suggests that we may have truncated the GPU affinity dimension prematurely in our experiments; however, from our observations we conclude that the best performance is well approximated by the experimental grid and that the most performative configuration overall will not lie far outside of the grid if it does. The regularity we observe in the experimental grid for `Swept` programs also extends in the grid size dimension as described by Figures 3.4 and 3.5 where swept decomposition produces a nearly perfect power law curve for each problem. These figures show a significant improvement for `Swept` programs over `Classic` ones. This relative advantage diminishes with greater problem sizes, but for the heat equation, Figure 3.4b shows an $18\times$ speedup at 5×10^5 spatial points falling to about a $4\times$ speedup at 10^7.

Though we would emphasize that these results are representative of narrow experimental conditions, architectures and design settings, the regularity of the `Swept` program performance allows us to present fitted results in Table 3.1, corresponding to the data points presented in Figures 3.4a and 3.5a, that may illuminate and guide future work on this and similar topics.

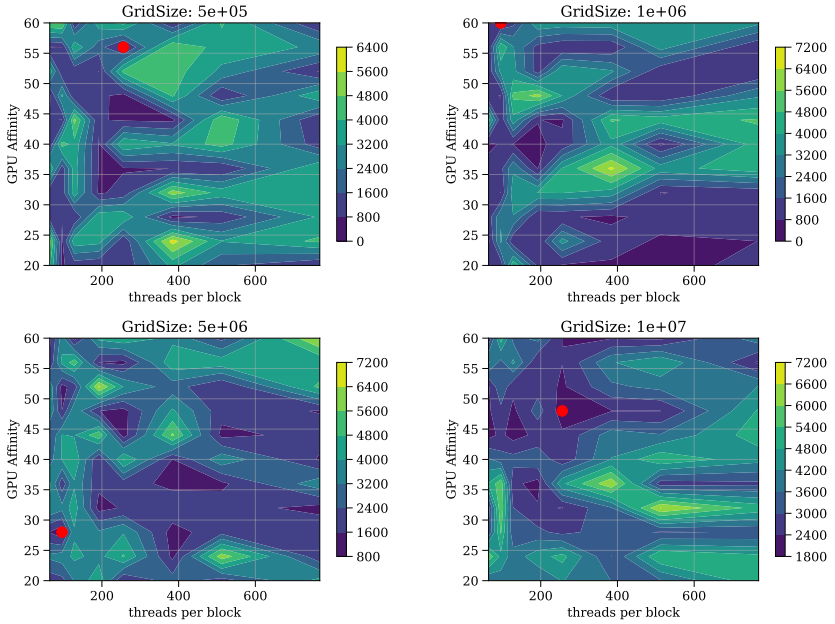


(a) Classic decomposition

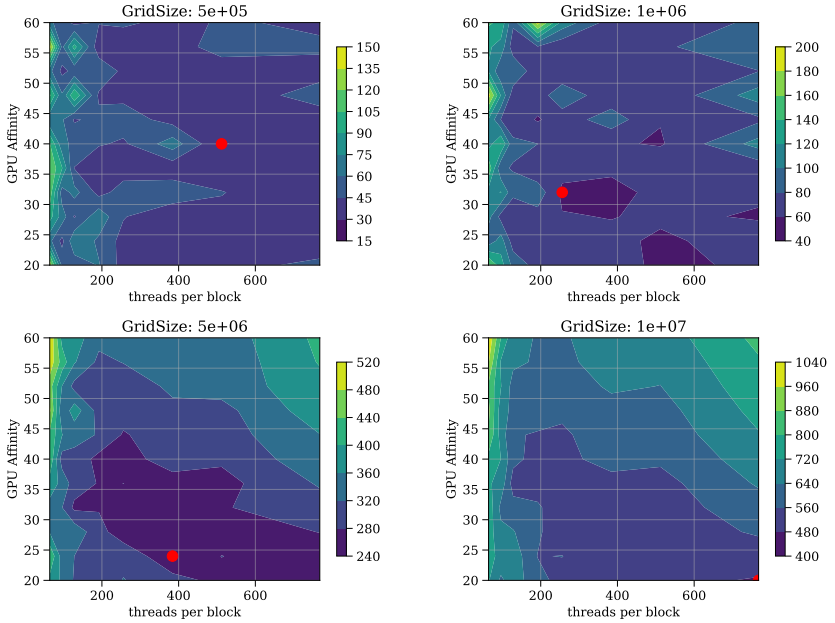


(b) Swept decomposition

Figure 3.6: A map of the time cost per timestep of the Euler equations at 4 grid sizes. The red dot signifies the best performance.



(a) Classic decomposition



(b) Swept decomposition

Figure 3.7: A map of the time cost per time step of the heat equation at 4 grid sizes. The red dot signifies the best performance.

Table 3.1: Coefficients for power-law fit of grid size vs time per timestep ($y = Ax^b$) of `Swept` performance at best runtime configuration.

Equation	A	b	R^2
Euler	3.55×10^{-3}	0.976	0.999
Heat	1.08×10^{-4}	0.949	0.999

3.6 Conclusions

We examined the performance characteristics of design choices that must be made when applying the swept rule to partial differential equations on heterogeneous computational architectures using swept-time space decomposition. These design choices are: how many threads per block—i.e., points per domain—to assign, what proportion of the total domain to assign to a GPU, and how to efficiently and generally store the working array throughout the simulation.

Our study aimed to answer the primary questions concerning these design choices laid out in Section 3.3. First, we found that the best number of points to assign to each domain varies with the algorithm, governing equation(s), and grid size. To achieve the best performance on repeated similar runs, any program should be tested over a limited number of time steps and tuned to the best result; however, in general we recommend choosing a per-process domain size value from 96–384 that is a multiple of 32. This is consistent with our previous results for the GPU-only implementation of the swept rule. Next, we concluded that while a GPU affinity is best chosen after a similar tuning experiment, for more complex equations an affinity from 40–60 performs well and an affinity from 20–40 does best for simpler problems. Next, there is a significant tradeoff between extensibility and performance associated with the primary data structure and compression scheme applied to the working quantity in the simulation. We choose to continue working with the `lengthening` method despite its performance drawbacks because it simplifies development substantially and has facilitated the development of this framework with which we can continue to develop codes and tests based on the swept rule. Finally, although any conclusions drawn from an experiment on only two nodes are limited, we showed a significant relative improvement over our previous results for the

Euler equations using a fine-tuned GPU-only program [27].

Future work in this project will continue adapting the swept rule to higher dimensions, architecture types, and grid formations. For example, while Alhubail and Wang demonstrated the two-dimensional swept rule for CPU-based clusters [5], we have not yet extended this to heterogeneous systems.

In addition to adapting the algorithm to higher dimensions, we recognize the need to develop new experiments that examine the scaling characteristics of the program as additional computational resources are added. We plan on and executing those experiments on cloud systems like Amazon Web Services, Microsoft Azure, or Nvidia GPU Cloud. As we conduct these experiments, we hope to gain greater insight into the factors affecting performance and develop a more robust performance model for the swept rule.

Chapter 4: Summary and Conclusions

4.1 Summary

In this thesis, I have explored methods for constructing an efficient swept time-space decomposition implementation for solving partial differential equations with explicit numerical methods. Two implementation stages have been presented: the first stage targeted individual GPUs as the primary computational platform, and the second targets heterogeneous computing systems, which use CPU and GPU processors together in a system spanning several nodes. The results of the first stage informed the GPU memory hierarchy used in the second stage where the GPU procedure is nested within the inter-process, inter-device, and inter-node communication scheme of the broader heterogeneous program. In the first stage, we created rigid, highly tuned programs for four implementation strategies the heat, Kuramoto–Sivashinsky, and Euler equations and tested these by evaluating them at different threads per block over the domain of experimental problem sizes. In the heterogeneous experiments, we created more flexible code, used the best implementation strategy for the GPU, and investigated the amount of work given to the GPU as a multiple of the quantity of work given to a CPU process (i.e., the GPU affinity).

In the two papers presented in this thesis, Chapters 2 and 3, I pointed out several conclusions that seem particularly relevant to future efforts in this area:

- The swept rule does provide a significant performance benefit but the benefit depends on the hardware and problem characteristics. More complex problems will perform worse with the swept rule; larger, more latency-bound systems will perform better; and smaller grid sizes perform better up to a point. Comparing the performance of the Euler equations in Chapter 2 and 3, there is a point where a complex problem that does not see benefit on one system will begin to see a benefit.

- Run stencil and other easily parallelizable codes on GPUs. The performance benefits for most long-running physics codes are substantial, to use a conservative qualifier. Chapter 2 shows speedups on the order of 100× for large problems over

an MPI implementation occupying the entire current generation CPU. Part of this speedup is due to the implementation, but the point stands that both the swept and naive decompositions show large improvements on the GPU. The code example in Section 2.5 shows how simply efficient, naive decompositions can be realized on the GPU.

- Shared memory is the most convenient and effective way to exploit the memory hierarchy on the GPU. This point has only become more relevant since future generations of Nvidia GPUs have expanded shared-memory capacity and capabilities, whereas the lower level, faster registers remain the same.

- The best runtime configuration settings vary substantially with the details of each governing equation and algorithm; it is a good idea to perform a manual tuning test before launching many iterations of a program for production. But, as a general rule, choosing a domain of dependence size between 96–384 tend to perform well across different algorithms. GPU affinities from 40–60 work better on more complicated problems while 20–40 work better on simpler ones. This is to say, running a program with 256 threads per block and a GPU affinity of 40 will always do reasonably well.

- There is a significant tradeoff between extensibility and performance associated with the primary data structure and compression scheme applied to the working quantity in the simulation. If performance is a top-tier concern, a developer is probably better off writing their own solver. Even a naive decomposition on the GPU, which is extremely simple, will outperform the swept rule with the right data storage and time-stepping mechanism.

4.2 Future Work

Recommendations for future work on the swept rule implemented on heterogeneous systems are thoroughly described in Section 3.6. In addition, there are several areas where the hSweep package can be improved or developed. In particular, implementing reflective boundary conditions, simplifying user input, and expression parsing for initial conditions would substantially improve the usability of the package for both researchers and users.

In addition, a GPU-only swept rule package for two-dimensional problems would be useful for practical use, since current GPUs are actually powerful enough to tackle problems with the power of several cluster nodes, as we have seen in this project with a GPGPU three generations behind the most recent release. This package would also be useful for experimentation. The cooperative group feature introduced in CUDA 9 offers synchronization across blocks for all active threads on a GPU. This would allow a swept computation to occur entirely on the GPU without passing control back to the CPU to implicitly synchronize the procedure. The effect of this strategy on this algorithm remains to be seen, but, from experience, I would expect it to have a significant positive impact on performance especially for classic decomposition.

Bibliography

[1] Vassil Alexandrov. Route to exascale: Novel mathematical methods, scalable algorithms and computational science skills. *Journal of Computational Science*, 14:1–4, 2016. The Route to Exascale: Novel Mathematical Methods, Scalable Algorithms and Computational Science Skills.

[2] M. Alhubail and Q. Wang. KSIDSwept, git commit `e575d73`. https://github.com/hubailmm/K-S_1D_Swept, 2015.

[3] Maitham Alhubail and Qiqi Wang. The swept rule for breaking the latency barrier in time advancing PDEs. *Journal of Computational Physics*, 307:110–121, 2016.

[4] Maitham Alhubail and Qiqi Wang. Improving the strong parallel scalability of cfd schemes via the swept domain decomposition rule. Grapevine, Texas, January 2017. American Institute of Aeronautics and Astronautics.

[5] Maitham Makki Alhubail, Qiqi Wang, and John Williams. The swept rule for breaking the latency barrier in time advancing two-dimensional PDEs. arXiv:1602.07558 [cs.NA], 2016.

[6] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-avoiding QR decomposition for GPUs. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 48–58, May 2011.

[7] Marc Baboulin, Simplice Donfack, Jack Dongarra, Laura Grigori, Adrien Rémy, and Stanimire Tomov. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. *Procedia Computer Science*, 9:17–26, 2012.

[8] Iván Bermejo-Moreno, Julien Bodart, Johan Larsson, Blaise M. Barney, Joseph W. Nichols, and Steve Jones. Solving the compressible Navier–Stokes equations on up to 1.97 million cores and 4.1 trillion grid points. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 62:1–62:10, New York, NY, USA, 2013. ACM.

[9] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *J. Parallel Distrib. Comput.*, 73(1):4–13, 2013.

[10] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. In *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218. Birkhäuser Basel, 1994.

[11] Felipe A Cruz, Simon K Layton, and L A Barba. How to obtain efficient GPU kernels: An illustration using FMM & FGT algorithms. *Comput. Phys. Comm.*, 182(10):2084–2098, October 2011.

[12] Kyle E. Niemeyer Daniel J. Magee. Niemeyer-Research-Group/hSweep: MS Thesis Official Niemeyer-Research-Group/hSweep: MS Thesis Official, June 2018.

[13] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[14] Jack Dongarra, Pete Beckman, Patrick Aerts, Frank Cappello, Thomas Lippert, Satoshi Matsuoka, Paul Messina, Terry Moore, Rick Stevens, Anne Trefethen, and Mateo Valero. The international exascale software project: a call to cooperative action by the global high-performance community. *The International Journal of High Performance Computing Applications*, 23(4):309–322, 2009.

[15] R. D. Falgout, T. A. Manteuffel, B. O'Neill, and J. B. Schroder. Multigrid reduction in time for nonlinear parabolic problems: A case study. *SIAM Journal on Scientific Computing*, 39(5):S298–S322, 2017.

[16] Robert Falgout, Stephanie Friedhoff, Tzanio Kolev, Scott MacLachlan, and Jacob B. Schroder. Parallel time integration with multigrid. *PAMM*, 14:951–952, 12 2014.

[17] Michael Feldman. Oak ridge readies summit supercomputer for 2018 debut. https://top500.org/news/oak-ridge-readies-summit-supercomputer-for-2018-debut/, November 2017. Accessed: 27 May 2018.

[18] Martin J. Gander. 50 years of time parallel time integration. In T Carraro, M Geiger, S Körkel, and R Rannacher, editors, *Multiple Shooting and Time Domain Decomposition Methods*, volume 9 of *Contributions in Mathematical and Computational Sciences*, pages 69–113. Springer, Cham, 2015.

[19] Mark Harris. CUDA pro tip: Do the Kepler shuffle. https://devblogs.Nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle/, February 2014. Accessed: 3 June 2016.

[20] Suzana Herculano-Houzel, Kamilla Avelino-de Souza, Kleber Neves, Jairo Porfirio, Débora Messeder, Larissa Mattos Feijó, José Maldonado, and Paul R Manger. The elephant brain in numbers. *Frontiers in Neuroanatomy*, 8:46, 2014.

[21] Y. A. Huerta, B. Swartz, and D. J. Lilja. Determining work partitioning on closely coupled heterogeneous computing systems using statistical design of experiments. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 118–119, Oct 2017.

[22] Dana A. Jacobsen and Inanc Senocak. Multi-level parallelism for incompressible flow computations on gpu clusters. *Parallel Computing*, 39(1):1–20, 2013.

[23] Kevin Krewell. What's the difference between a CPU and a GPU? https://blogs.Nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/, December 2009. Accessed: 23 May 2018.

[24] Fengshun Lu, Junqiang Song, Fukang Yin, and Xiaoqian Zhu. Performance evaluation of hybrid programming patterns for large cpu/gpu heterogeneous clusters. *Computer Physics Communications*, 183(6):1172–1181, 2012.

[25] Daniel J. Magee and Kyle E. Niemeyer. Data, plotting scripts, and figures for "Accelerating solutions of PDEs with GPU-based swept time-space decomposition", May 2017.

[26] Daniel J. Magee and Kyle E. Niemeyer. Niemeyer-Research-Group/1DSweptCUDA v2, May 2017.

[27] Daniel J. Magee and Kyle E. Niemeyer. Accelerating solutions of one-dimensional unsteady PDEs with GPU-based swept time–space decomposition. *Journal of Computational Physics*, 357:338–352, 2018.

[28] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM Journal on Scientific Computing*, 37(4):C439–C464, 2015.

[29] Henry Markram. The human brain project. *Scientific American*, 306(6):50–55, 2012.

[30] Richard Tran Mills, Karl Rupp, Mark Adams, Jed Brown, Tobin Isaac, Matt Knepley, Barry Smith, and Hong Zhang. Software strategy and experiences with manycore processor support in petsc. SIAM Pacific Northwest Regional Conference, October 2017.

[31] Kyle E Niemeyer and Chih Jen Sung. Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *Journal of Supercomputing*, 67(2):528–564, February 2014.

[32] Nvidia Corporation. CUDA C programming guide, 2016. Version 8.0.

[33] Nvidia Corporation. Whitepaper Nvidia Tesla P100. http://www.Nvidia.com/object/pascal-architecture-whitepaper.html, 2016.

[34] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proc. IEEE*, 96(5):879–899, January 2008.

[35] David A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, October 2004.

[36] J P Slotnick, Abdollah Khodadoust, J J Alonso, D L Darmofal, W D Gropp, E A Lurie, and D J Mavriplis. CFD vision 2030 study: A path to revolutionary computational aerosciences. NASA Technical Report, NASA/CR-2014-218178, NF1676L-18332, March 2014.

[37] Duane Storti and Mete Yurtoglu. *CUDA for Engineers: An introduction to High-Performance Parallel Computing*. Addison-Wesley, 2015.

[38] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache oblivious parallelograms in iterative stencil computations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 49–59, New York, NY, USA, 2010. ACM.

[39] Swiss National Supercomputing Centre. "piz daint", one of the most powerful supercomputers in the world. https://www.cscs.ch/fileadmin/user_upload/contents_publications/factsheets/piz_daint/FSPizDaint_2017_EN.pdf, 2017.

[40] Qiqi Wang. Decomposition of stencil update formula into atomic stages, 2017.

[41] F.D. Witherden, A.M. Farrington, and P.E. Vincent. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185(11):3028–3040, 2014.

[42] Shu-Lin Wu and Tao Zhou. Parareal algorithms with local time-integrators for time fractional differential equations. *Journal of Computational Physics*, 358:135–149, 2018.

[43] S. Xiao and W. C. Feng. Inter-block GPU communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

APPENDICES

Appendix A: GPU-based swept time–space decomposition

A.1 Availability of materials

The results for this article were obtained using `1DSweptCUDA` v2 [26]. All figures, and the data and plotting scripts necessary to reproduce them, are available openly under the CC-BY license [25].

A.2 Heat equation

The unsteady heat conduction equation without volumetric heat flux is

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T \, . \tag{A.1}$$

where T is temperature, t is time, and α is thermal diffusivity. In one dimension, this is reduced to

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} \, . \tag{A.2}$$

Discretizing Eq. (A.2) with forward differencing in time and central in space yields

$$\frac{T_i^{m+1} - T_i^m}{\Delta t} = \alpha \frac{T_{i+1}^m + T_{i-1}^m + 2T_i^m}{\Delta x^2} \, , \tag{A.3}$$

where i is the spatial node index and m is the time index corresponding to time t^m. This is a first-order, explicit, finite-difference approximation. To step forward in time, define the Fourier number, $\mathrm{Fo} = \frac{\alpha \Delta t}{\Delta x^2}$, where Δt is the timestep size and Δx is the spatial grid size, and solve for temperature at the next timestep:

$$T_i^{m+1} = \mathrm{Fo}(T_{i+1}^m + T_{i-1}^m) + (1 - 2\mathrm{Fo})T_i^m \, . \tag{A.4}$$

In this study the approximation is evaluated with insulated boundary conditions at both

ends and n spatial points:

$$T_{-1} = T_1 \quad \text{and} \quad T_{n+1} = T_{n-1} \, . \tag{A.5}$$

A.3 Kuramoto–Sivashinsky equation

The Kuramoto–Sivashinsky equation is a nonlinear, fourth-order, one-dimensional unsteady PDE:

$$u_t = -(uu_x + u_{xx} + u_{xxxx}) = -\left(\frac{1}{2}u_x^2 + u_{xx} + u_{xxxx}\right) \, , \tag{A.6}$$

where u is the dependent chaotic variable (e.g., fluid velocity). It is discretized similarly to the heat equation, as shown in Eq. (A.7), with central differencing in space. In this case, decomposing the fourth spatial derivative requires a five-point stencil:

$$\frac{u_i^{m+1} - u_i^m}{\Delta t} = -\left(\frac{(u_{i+1}^m)^2 - (u_{i-1}^m)^2}{4\Delta x} + \frac{u_{i+1}^m + u_{i-1}^m + 2u_i^m}{\Delta x^2} + \frac{u_{i+2}^m - 4u_{i+1}^m + 6u_i^m - 4u_{i-1}^m + u_{i-2}^m}{\Delta x^4}\right) \, . \tag{A.7}$$

The chaotic nature of the problem necessitates a higher-order scheme overall; therefore, an explicit, second-order Runge–Kutta scheme, also known as the midpoint method, is applied to the time domain.

Let the right-hand side of Eq (A.7) be $f(u(x,t))$, then the predictor solution is found at $u_i^{m+1/2}$:

$$u_i^{m+1/2} = u_i^m + \frac{\Delta t}{2} f(u_i^m) \, . \tag{A.8}$$

Then $f(u_i^{m+1/2})$ may be evaluated and added to u_i^m to obtain u_i^{m+1}:

$$u_i^{m+1} = u_i^m + \Delta t f(u_i^{m+1/2}) \, . \tag{A.9}$$

The problem demonstrated here uses periodic initial and boundary conditions. That is, the stencil at point 0 includes points n and $n-1$ and the initial condition is periodic and continuous at the spatial boundaries.

A.4 Euler equations (Sod shock tube)

The Sod shock tube problem is a one-dimensional unsteady compressible flow problem based on the nonlinear, quasi-hyperbolic Euler equations:

$$\frac{\partial Q}{\partial t} + \frac{\partial F}{\partial x} = \frac{\partial Q}{\partial t} + J\frac{\partial Q}{\partial x} = 0 \tag{A.10}$$

where J is the Jacobian matrix,

$$Q = \begin{Bmatrix} \rho \\ \rho u \\ \rho e \end{Bmatrix} , \tag{A.11}$$

$$F = \begin{Bmatrix} \rho u \\ \rho u^2 + P \\ u(\rho e + P) \end{Bmatrix} , \tag{A.12}$$

ρ is density, e is internal energy, u is velocity, P is pressure given by

$$P = (\gamma - 1)(e - \frac{\rho u^2}{2}) , \tag{A.13}$$

and $\gamma = 1.4$ is the heat capacity ratio of air.

The initial boundary conditions, given in Eq. (A.14), are constant values for the state variables on either side of a diaphragm separating two parcels of the same fluid. The spatial boundary conditions are these values at their respective ends of the tube:

$$\begin{Bmatrix} \rho_L \\ u_L \\ P_L \end{Bmatrix} = \begin{Bmatrix} 1.0 \\ 0.0 \\ 1.0 \end{Bmatrix} \text{ and } \begin{Bmatrix} \rho_R \\ u_R \\ P_R \end{Bmatrix} = \begin{Bmatrix} 0.125 \\ 0.0 \\ 0.1 \end{Bmatrix} . \tag{A.14}$$

The equation is discretized using a second-order, finite-volume scheme with cell-centered values. The first step in the solution is evaluating the pressure ratio at the current timestep over a five-point stencil

$$P_{r,i} = \frac{P_{i+1} - P_i}{P_i - P_{i-1}} \text{ at } i-1,\ i,\ i+1 \tag{A.15}$$

This value is used with a minmod limiter to compute reconstructed values on both sides, L and R, of the current cell boundaries at $i \pm 1/2$:

$$Q_n^L = \begin{cases} Q_o^L + \frac{\min(P_r^L, 1)}{2} * (Q_o^R - Q_o^L), & \text{if } 0 < P_r^L < \infty \\ Q_o^L, & \text{otherwise.} \end{cases} \tag{A.16}$$

$$Q_n^R = \begin{cases} Q_o^R + \frac{\min((P_r^R)^{-1}, 1)}{2} * (Q_o^L - Q_o^R), & \text{if } 0 < \frac{1}{P_r^R} < \infty \\ Q_o^R, & \text{otherwise.} \end{cases} \tag{A.17}$$

where subscript n refers to the reconstructed, or new, values on the edge of the interface and o refers to the original values. For example, at $i - 1/2$ the original values on the left side of the interface are at $i - 1$. These reconstructed boundaries do not represent solutions for any grid cell; they are temporary values that interpolate the solutions.

Once we have the reconstructed values on either side of the interface, we can calculate the flux at that cell boundary with

$$\text{Flux} = \frac{1}{2} * (F(Q^R) + F(Q^L) + r_{sp} * (Q^L - Q^R)) \tag{A.18}$$

where $F(Q)$ is given by Eq. (A.12) and r_{sp} is the spectral radius, the largest eigenvalue of the Jacobian matrix J.

The spectral radius can be found with the Roe average Q at the interface

$$Q_{sp} = \begin{Bmatrix} \rho_{sp} \\ u_{sp} \\ e_{sp} \end{Bmatrix} = \begin{Bmatrix} \sqrt{\rho_L * \rho_R} \\ \frac{\sqrt{\rho_L} * u_L + \sqrt{\rho_R} * u_R}{\sqrt{\rho_L} + \sqrt{\rho_R}} \\ \frac{\sqrt{\rho_L} * e_L + \sqrt{\rho_R} * e_R}{\sqrt{\rho_L} + \sqrt{\rho_R}} \end{Bmatrix} \tag{A.19}$$

and P_{sp} with Q_{sp} using Eq. (A.13).

The spectral radius is given by

$$r_{sp} = \sqrt{\frac{\gamma * P_{sp}}{\rho_{sp}}} + |u_{sp}| \tag{A.20}$$

Repeating this process at both interfaces yields all required values to solve for a

timestep

$$Q_i^{n+1} = Q_i^n + \frac{\Delta t}{\Delta x}(\text{Flux}_{i+1/2}^{n+1/2} - \text{Flux}_{i-1/2}^{n+1/2}) \tag{A.21}$$

The results presented here for the Euler equations use a second-order Runge–Kutta scheme in time, which can be obtained with the same procedure shown in Eqs. (A.9) and (A.8).