View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Clemson University: TigerPrints

Clemson University
TigerPrints

All Theses

Theses

12-2015

A novel approach to evaluating compact finite differences and similar tridiagonal schemes on GPU-accelerated clusters

Ashwin Trikuta Srinath
Clemson University, atrikut@g.clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses

Part of the Mechanical Engineering Commons

Recommended Citation
Trikuta Srinath, Ashwin, "A novel approach to evaluating compact finite differences and similar tridiagonal schemes on GPU-accelerated clusters" (2015). *All Theses*. 2283.
https://tigerprints.clemson.edu/all_theses/2283

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

A novel approach to evaluating compact finite differences and similar tridiagonal schemes on GPU-accelerated clusters

---

A Master's Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Mechanical Engineering

---

by
Ashwin Srinath
December 2015

Accepted by:
Dr. Richard S. Miller, Committee Chair
Dr. Gang Li
Dr. Lonny L. Thompson

Abstract

Compact finite difference schemes are widely used in the direct numerical simulation of fluid flows for their ability to better resolve the small scales of turbulence. However, they can be expensive to evaluate and difficult to parallelize. In this work, we present an approach for the computation of compact finite differences and similar tridiagonal schemes on graphics processing units (GPUs). We present a variant of the *cyclic reduction* algorithm for solving the tridiagonal linear systems that arise in such numerical schemes. We study the impact of the matrix structure on the cyclic reduction algorithm and show that precomputing forward reduction coefficients can be especially effective for obtaining good performance. Our tridiagonal solver is able to outperform the NVIDIA CUSPARSE and the multithreaded Intel MKL tridiagonal solvers on GPU and CPU respectively. In addition, we present a parallelization strategy for GPU-accelerated clusters, and show scalabality of a 3-D compact finite difference application for up to 64 GPUs on Clemson's Palmetto cluster.

Dedication

I dedicate this work to my parents, and to my brother, Akhil, who pushes me always to be a better example.

Acknowledgments

I owe my deepest thanks to my advisor, Dr. Richard S. Miller, for his immense guidance and support; and the members of my committee, Dr. Gang Li and Dr. Lonny Thompson for their valuable time. I would like to acknowledge Dr. Daniel Livescu at Los Alamos National Lab for introducing us to this interesting problem, for his helpful comments, and for generous funding. I also thank Dr. Melissa Smith and Karan Sapra for their helpful guidance, and for organizing an excellent course on GPU computing. I owe a great deal of gratitude to Dr. Christopher Cox at Clemson University for his guidance on a range of topics and for many interesting conversations. I would like to thank the staff of the Cyberinfrastructure Technology Integration group at Clemson, and especially Dr. Galen Collier and Dr. Marcin Ziolkowski for their valuable comments and support. I owe thanks to Clemson University for the generous allocation of compute time on the Palmetto cluster. I acknowledge Dr. K. N. Seetharamu and Dr. V. Krishna, my advisors at PES Institute of Technology for inspiring in me a passion for scientific research. I am indebted to Dr. Anush Krishnan for introducing me to the fascinating field of scientific computing. I owe a special thanks to the Software Carpentry Foundation and its members, for helping me be a more productive programmer, a more deliberate learner, a more effective teacher, and a better person.

Finally, I thank my family and friends for everything they have done for me.

Table of Contents

Title Page . . . . **i**

Abstract . . . . **ii**

Dedication . . . . **iii**

Acknowledgments . . . . **iv**

List of Tables . . . . **vii**

List of Figures . . . . **viii**

1 Introduction and Background **1**

1.1 Motivation 1

1.2 Compact finite differences 3

1.2.1 General form of compact finite difference schemes 4

1.2.2 Wavenumber analysis of finite difference methods 4

1.2.3 Boundary conditions 6

1.2.4 Tridiagonal compact schemes 6

1.3 Graphics processing units 8

1.3.1 The CUDA Programming model 9

1.3.2 CUDA architecture and memory model 10

1.3.3 Considerations to be made while programming for GPUs . . . 15

1.3.4 Software for programming NVIDIA GPUs 16

1.4 Tridiagonal solvers for compact finite difference evaluation 16

1.4.1 Thomas algorithm 17

1.4.2 Cyclic reduction 18

1.4.3 Parallel cyclic reduction 20

1.4.4 Cyclic reduction implementation on GPUs 21

2 Proposed tridiagonal algorithm **24**

2.1 Modified cyclic reduction for near-Toeplitz systems 24

2.1.1 Forward reduction 25

2.1.2 Backward substitution 27

2.2 Implementation 27
2.2.1 Precomputing forward reduction coefficients 28
2.2.2 Global memory implementation 29
2.2.3 Shared memory implementation 31

3 Application to compact finite difference evaluation 33
3.1 Introduction 33
3.2 Compact finite difference evaluation on parallel GPU systems 34
3.2.1 Single GPU 35
3.2.2 Multiple GPUs on a single node 36
3.2.3 Distributed GPUs - restricted in one direction 38
3.2.4 Distributed GPUs in all directions 38
3.3 Distributed tridiagonal solver 41
3.3.1 General algorithm 41
3.3.2 Specialization for compact finite difference evaluations 43
3.4 GPU implementation 44

4 Results 49
4.1 Performance of GPU tridiagonal solver 49
4.1.1 NEATO: global memory v/s shared memory performance 50
4.1.2 Comparison of NEATO with Intel MKL and CUSPARSE solvers 51
4.2 Performance of compact finite difference application 53
4.2.1 Performance profiling 55
4.2.2 Strong and weak scaling 56
4.2.3 Comparison with a CPU-only approach 59

5 Conclusions and Future Work 60

Appendix 61

Bibliography 67

List of Tables

3.1 Purpose of kernels and MPI calls in compact finite difference application 46

4.1 Performance of Intel MKL, CUSPARSE and NEATO solvers. 54
4.2 Time (ms) to compute derivatives in the fastest coordinate direction - comparison with reference implementation [19] 58

List of Figures

1.1 Modified wavenumbers for different finite difference schemes. k' is the wavenumber of the approximated derivative, while k is wavenumber of the exact derivative. h represents the spacing between two grid points. The compact schemes better estimate the derivative for higher wavenumbers. 5
1.2 Scheduling of blocks to streaming microprocessors—GPUs with more SMs are able to execute more blocks concurrently. (*CUDA Programming Guide [22]*) 11
1.3 (a) Organization of shared memory as 4 byte words organized into 32 banks. Each cell (square) is a word. (b) Bank conflict free access by a warp. Each warp accesses a word from a different bank. (c) 2-way bank conflicts arising from successive threads accessing alternating words. . 13
1.4 Cyclic reduction. 19
1.5 Mapping work to blocks and threads: systems are mapped to blocks and indices are mapped to individual threads. 21
1.6 Updating $\boldsymbol{b}$ in the first forward reduction step. 22

2.1 Maximum storage required for forward reduction coefficients at all steps $m = 1, 2, ...log_2(n) - 1$ 28
2.2 Storing right hand sides and mapping to thread blocks. 29
2.3 Thread activity in shared memory (top) and global memory (bottom) implementations. 30

3.1 Computational domain in 3-D. 34
3.2 Compact finite differences - single GPU 35
3.3 Compact finite differences - multiple GPUs on same node 36
3.4 Compact finite differences - distributed GPUs and restricted in one direction 37
3.5 Compact finite difference evaluation in both coordinate directions . . 38
3.6 Compact finite differences - distributed GPUs in all directions 40
3.7 Algorithm for evaluating compact finite differences on multiple GPUs, (right: CUDA kernels and MPI calls used) 45
3.8 Construction of the reduced system and scattering of parameters. . . 47

4.1 Comparison of global memory and shared memory implementations of NEATO (2D problems). 50
4.2 Comparison of global memory and shared memory implementations of NEATO (3D problems). 51
4.3 Relative solver performance for 2-D problems. Relative time defined as: 52
4.4 Relative solver performance for 3-D problems. Relative time defined as: 52
4.5 Solving problem sized 1024^3 on 64 GPUs 55
4.6 Solving problem sized 2048^3 on 64 GPUs 55
4.7 Strong scaling for multi-GPU compact finite difference, problem size: 256^3. 56
4.8 Strong scaling for multi-GPU compact finite difference, problem size: 512^3. 57
4.9 Weak scaling for multi-GPU compact finite difference, problem size: 128^3 per process. 57
4.10 Weak scaling for multi-GPU compact finite difference, problem size: 256^3 per process. 58
4.11 Speedups over reference implementation for computing derivative in the fastest coordinate direction 59

Chapter 1

Introduction and Background

1.1 Motivation

In the direct numerical simulation (DNS) of fluid systems, the aim is to resolve the smallest features of the flow without resorting to any modeling of the turbulence. This involves the use of extremely fine computational grids to discretize the flow domain, and some numerical method for solving the flow equations on this grid. Among the most popular numerical approaches for solving the flow equations is the *finite difference method*. The finite difference method approximates the spatial and temporal derivatives appearing in the partial differential equations that describe the flow using finite difference schemes. *Compact finite difference schemes* are a class of finite difference schemes that have found widespread adoption in DNS codes for their high order of accuracy and small stencil widths. The evaluation of compact finite differences requires the repeated solution of *banded* linear systems, making them fairly complex and expensive computationally.

When solving for flows numerically, at each of the grid points in the computational domain, various data about the flow must be stored—for example, the geo-

metric coordinates (x, y and z), pressure, temperature and velocities at that point. For even small problem sizes, the amount of memory required to store this data can quickly exceed the capacity of modern workstations/PCs. Thus, a *distributed memory* parallel system is generally required for performing DNS. Here, the traditional approach has been to distribute parallel tasks among individual CPU cores, or groups of CPU cores that share common memory spaces. In the latter case, each group of cores constitutes a *shared memory* system, and the overall system is referred to as a *hybrid system.*

The workhorse for computation in the above described parallel systems is the CPU core, however, more recently, Graphics Processing Units (GPUs) are being used for performing intensive calculations. GPUs, while themselves being highly parallel processors, can also function as accelerators in distributed memory systems (GPU clusters). However, applications that exploit such systems need careful redesign of algorithms—and sometimes, substantial changes to code—to see significant performance improvements. This is because the CPU and GPU have very different architectures, and any naïve "parallelization" of algorithms designed for the CPU is likely not take full advantage of the GPU's memory hierarchy and compute ability.

The objective of this work is to develop an approach for evaluating compact finite differences—and other numerical schemes leading to tridiagonal systems—on multiple GPUs in a distributed system. This is of interest because the evaluation of spatial derivatives using compact finite difference schemes is one of the most expensive tasks in DNS. But perhaps more significantly, it encompasses several computational patterns such as pointwise updates, stencil evaluations, and solutions of distributed tridiagonal systems. Efficient implementation of these computational patterns on GPUs is of interest in other areas of CFD, and scientific computation in general.

1.2 Compact finite differences

Numerical evaluation of derivatives is a central component in scientific computing. The simplest and most widely-used approach for numerical differentiation is the *finite difference* approximation, wherein the numerical approximation of the derivative is expressed as a *difference equation*. A key application of the finite difference approximation is in *finite difference methods*, a family of numerical methods for solving differential equations in which the derivatives are approximated using finite difference approximations. For example, we may consider a uniformly sampled function $f(x)$, sampled at points $x_1, x_2, x_3, \ldots x_n$. At each sample point i, we may approximate the derivative as a combination of the function values at i, and its neighbouring points:

1. When the derivative at i is expressed as some combination of the function values at i, $i+1$, $i+2$, $\ldots$, we refer to the approximation as a *forward* difference.

2. When the derivative is expressed as some combination of the function values at i, $i-1$, $i-2$, $\ldots$, we refer to the approximation as a *backward* difference.

3. When the derivative is expressed as some combination of the function values on *both* sides of i, i.e, at $\ldots$, $i-2$, $i-1$, i, $i-1$, $i-2$, $\ldots$, we refer to the approximation as a *central* difference.

The approximation of higher order derivatives generally requires inclusion of a larger number of points in the finite difference approximation (referred to as the finite difference *stencil*). For a given order of derivative, finite difference approximation of arbitrary stencil widths may be derived, with larger stencils associated with higher accuracy [10]. In the evaluation of spatial derivatives, the above schemes are referred

to as *explicit* schemes, as the derivative at each point can be expressed as some explicit combination of function values.

1.2.1 General form of compact finite difference schemes

Compact schemes express the derivative at a point i in terms of function values *and derivatives* at neighbouring points, i.e., the derivative is expressed *implicitly.* For example, if f_i represents the value of a uniformly sampled function evaluated at the ith sample point, the first derivative f'_i can be approximated from a relation of the form:

$$\begin{aligned} f'_i + \alpha(f'_{i-1} + f'_{i+1}) + \beta(f'_{i-2} + f'_{i+2}) + \ldots = a\frac{f_{i+1} - f_{i-1}}{h} + b\frac{f_{i+2} - f_{i-2}}{h} + \\ c\frac{f_{i+3} - f_{i-3}}{h} + \ldots \end{aligned} \tag{1.1}$$

where α, β, a, b, c, etc., are parameters that must satisfy certain constraints [12, 16], and h is the space between two grid points. We note that for equations of the form in Eq. (1.1), the derivative at any points i cannot be computed explicitly. Instead, we must write similar equations for *all* points i in the range. This results in a system of linear equations with unknowns $\{f'_1, f'_2, \ldots, f'_n\}$, which may be represented by the matrix system $A\boldsymbol{x} = \boldsymbol{d}$, where $\boldsymbol{x}$ is the vector $\{f'_1, f'_2, \ldots, f'_n\}$, $\boldsymbol{d}$ is a vector of right hand sides [Eq. (1.1)], and A is, in general, a *banded* matrix.

1.2.2 Wavenumber analysis of finite difference methods

For DNS applications, the relevant measure of accuracy of a finite difference scheme is obtained from the so-called *modified wavenumber* approach. Here, we test the scheme's ability to accurately estimate the derivative of sinusoidal functions with increasing wavenumbers (frequencies), given a specified grid size. We expect that

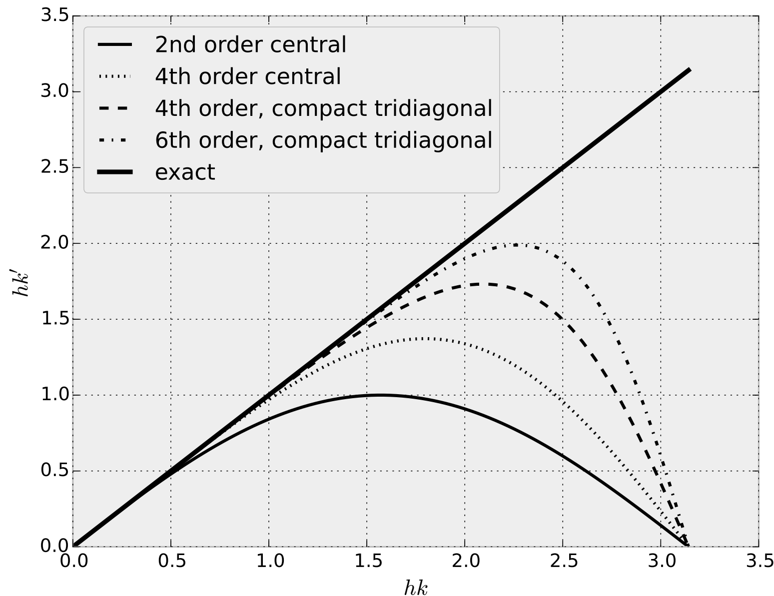

Figure 1.1: Modified wavenumbers for different finite difference schemes. k' is the wavenumber of the approximated derivative, while k is wavenumber of the exact derivative. h represents the spacing between two grid points. The compact schemes better estimate the derivative for higher wavenumbers.

the approximation of the derivative becomes more difficult with increasing wavenumbers, as the function value varies more rapidly. The wavenumber of the approximate derivative as given by the finite difference scheme is compared with the wavenumber of the exact derivative. The result for different schemes is shown in Fig. 1.1.

The ability of a finite difference scheme to accommodate large wavenumbers is extremely relevant in computational fluid dynamics applications [15]. In DNS, the scheme *must* capture the rapidly varying characteristics of the flow associated with the turbulence. For this purpose, higher order explicit schemes may be considered. While they are straightforward to compute, they are associated with large stencil widths. In most applications, large stencil widths are undesirable. This is because the arithmetic intensity increases with stencil size, i.e., a larger number of computations must be

performed per grid point. Further, the amount of boundary information that must be exchanged between parallel processes increases, which can seriously affect overall performance.

From Fig. 1.1, it is clear that the compact schemes are better able to compute derivatives for larger wavenumbers for a given stencil width. This makes them the superior choice for DNS, in which the flow quantities exhibit spatial variation associated with high wavenumbers.

1.2.3 Boundary conditions

One of the advantages of the compact finite difference approach is that it accommodates *non-periodic* boundary conditions. This is in contrast to other methods used in DNS, such as *spectral* methods. We note that the Eq. (1.1) cannot be applied near the boundary points. At the boundaries, *non-centered* or *one-sided* finite difference approximations are required. Some considerations are made in choosing these approximations: firstly, the bandwidth of the resulting banded matrix must be preserved. Secondly, the width of the boundary stencils must be lower than the interior stencils, as higher order boundary stencils are unstable [12]. In general, boundary equations for the first derivative are of the following form [16]:

$$f'_1 + \alpha' f'_2 = \frac{1}{h}(a' f_1 + b' f_2 + c' f_3 + d' f_4) \tag{1.2}$$

1.2.4 Tridiagonal compact schemes

As mentioned in Sec. 1.2.1, compact finite difference schemes lead to *banded* linear systems. The simplest classes of banded matrix include diagonal, tridiagonal and pentadiagonal matrices. When $\alpha = \beta = \ldots = 0$, A is a *diagonal* matrix. In this

case, the scheme is *explicit* and the derivatives are straightforward to evaluate: this simply involves the application of the right-hand side stencil at each point. When $\alpha \neq 0, \beta = \ldots = 0$, A is tridiagonal. The evaluation of the derivatives requires the solution of the resulting tridiagonal system. When $\alpha \neq 0$ and $\beta \neq 0$, pentadiagonal systems arise. These can be more expensive and difficult to evaluate than tridiagonal systems using a direct method.

This work will focus on compact finite difference schemes that lead to tridiagonal systems. An example of such a scheme is obtained by substituting $\beta = 0$, $\alpha = \frac{1}{4}$ and $a = \frac{3}{4}$ in Eq. (1.1) (all other coefficients are set to zero). This leads to a fourth-order accurate tridiagonal compact finite difference scheme, known also as the Padé scheme. At the boundaries, the following *third-order* accurate approximations are used:

$$f_1' + 2f_2' = \frac{-5f_1 + 4f_2 + f_3}{dx} \tag{1.3}$$

$$f_n' + 2f_{n-1}' = \frac{5f_n - 4f_{n-2} - f_{n-1}}{dx} \tag{1.4}$$

The resulting tridiagonal system is then:

$$\begin{bmatrix} 1 & 2 & & & & & & & \\ 1/4 & 1 & 1/4 & & & & & & \\ & 1/4 & 1 & 1/4 & & & & & \\ & & 1/4 & 1 & 1/4 & & & & \\ & & & 1/4 & 1 & 1/4 & & & \\ & & & & & \ddots & & & \\ & & & & & & \ddots & & \\ & & & & & & & \ddots & \\ & & & & & & & 2 & 1 \end{bmatrix} \begin{bmatrix} f'_1 \\ f'_2 \\ f'_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ f'_{n-1} \\ f'_n \end{bmatrix} = \begin{bmatrix} \frac{-5f_1+4f_2+f_3}{2dx} \\ \frac{3(f_3-f_1)}{4dx} \\ \frac{3(f_4-f_2)}{4dx} \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \frac{3(f_n-f_{n-2})}{4dx} \\ \frac{5f_n-4f_{n-1}-f_{n-2}}{2dx} \end{bmatrix} \tag{1.5}$$

Similar tridiagonal schemes are available to evaluate higher order derivatives. Compact schemes for second, third and fourth derivatives are derived in [16].

1.3 Graphics processing units

The evolution of graphics processing units (GPUs) has traditionally been guided by the requirements of graphics applications such as video gaming. In general, these graphics computations involve manipulations and calculations with pixel data. Because these calculations are largely independent, the GPU hardware is specialized to perform several of them *in parallel*. Therefore, the GPU may be regarded as a highly multithreaded processor. Initially, programming GPUs for scientific computation tasks was a cumbersome process, as the programming interfaces were designed for graphics computations, and attempts to use them for other purposes were essentially "hacks". Moreover, there was little support for double precision arithmetic, which is

critical for many applications. But with the introduction of the NVIDIA Compute Unified Device Architecture (CUDA) parallel computing platform and programming model, CUDA-enabled GPUs have come into the mainstream in high-performance computing. A history of the development of GPUs for general purpose computation, and of CUDA specifically is available in [13]. We provide the essential and relevant details about CUDA here, and refer to the CUDA Programming Guide [22] for a detailed outlook.

1.3.1 The CUDA Programming model

CUDA is the name for the parallel computing platform developed by NVIDIA, as well as the application programming interface for programming their GPUs. CUDA allows general-purpose applications to be more easily programmed for the GPU. Two essential features of the CUDA programming model are *kernels* and *thread organization*.

Kernels

The CUDA application programming interface (API) is available as an extension to a programming languages (C, C++, Fortran, etc.), allowing certain portions of code to execute on the GPU. The rest of the code is executed as usual on the CPU. In CUDA terminology, the CPU is referred to as the *host*, and the GPU is referred to as the *device*. The special pieces of code that execute on the GPU are known as *kernels*. Kernels have similar syntax as "host code", but in general, are more restricted in the features of the underlying language they can use. In C, for example, kernels are written as *functions*, and are called by the host code using (almost) the same conventions. Thus, a C program that uses the GPU looks and behaves very much like a

normal C program, but includes special calls to these kernels. When the application is launched, a CPU thread executes the host-code as usual, but upon encountering a call to the kernel, it passes program control over to the GPU. After the kernel is finished executing, control is passed back to the CPU, and this process may repeat when another kernel call is encountered. The CPU and GPU may also operate *asynchronously*, i.e., control may be passed back to the CPU *before* kernel completion, in which case explicit synchronization between the host and device may be necessary. Apart from kernel calls, the host code can also call functions to allocate and deallocate device memory, query device information, perform device synchronization, etc.

Thread organization

A kernel is executed in parallel by several lightweight *threads*. Threads are organized into groups called *thread blocks*, or just *blocks*, and the different blocks constitute a *grid* of blocks. In many cases, the GPU is used to process array data, and each thread is mapped to a single array element. In most applications, the array is logically 1-, 2- or 3-dimensional. Thus, for convenience, the grid and block sizes can be 1-, 2- or 3-dimensional (hence the terminology *grid* and *blocks*). This kind of thread organization is optional, and n-dimensional arrays can be processed using just 1-dimensional grids and blocks.

1.3.2 CUDA architecture and memory model

From the hardware perspective, an NVIDIA GPU may be viewed primarily as a collection of so-called Streaming Microprocessors (SMs). When a kernel is launched with specified grid size (number of blocks) and block size (threads per block), each block is assigned to an SM. Threads within a thread block can execute concurrently

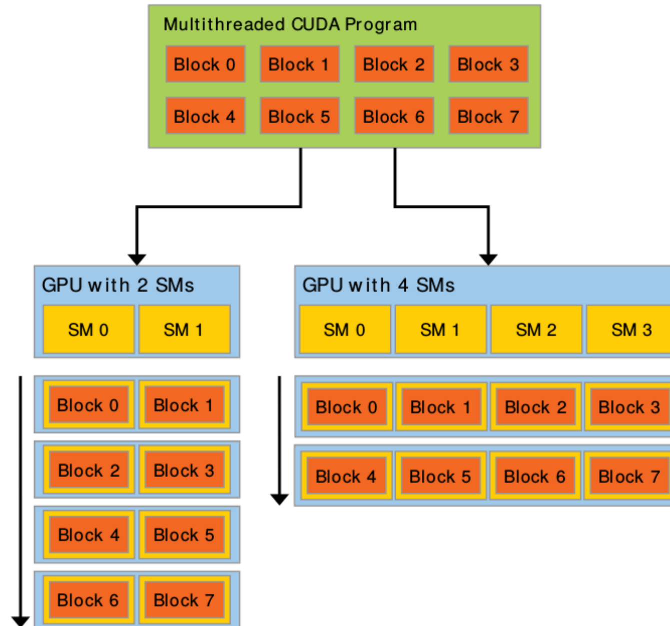

Figure 1.2: Scheduling of blocks to streaming microprocessors—GPUs with more SMs are able to execute more blocks concurrently. (*CUDA Programming Guide [22]*)

on the SM, and an SM can execute several thread blocks concurrently. It is imperative that blocks are able to run independently, in series or in parallel. This feature gives CUDA programs their scalability (Fig. 1.2). The same program runs faster on a GPU with a larger number of SMs, simply because more blocks may run concurrently. The number of thread blocks that an SM can execute concurrently is limited by a number of factors, and maximizing this number is often key to obtaining good performance. Another key consideration in the implementation of algorithms on GPUs is the available *memory hierarchy*. Threads executing a kernel can read and write to several different memory spaces, each with different capacity, latency and bandwidth.

Global memory

Data from the host is first read into the device's *global memory*. Similarly, data from the device is read back into the host from global memory. This is done in the host-code using special functions provided by the CUDA API. Data transfer between the host and the device is extremely slow relative to data access within the device. This data transfer rate is limited by the bandwidth of the PCI-e bus between the host and device. All threads executing a kernel can read and write to locations in global memory, and it is the largest memory space that is writable by the device. For example, the current NVIDIA Tesla K20 accelerator has about 4 Gigabytes of usable global memory. Thread access to global memory has a long latency and it is especially inefficient when successive threads in a block access memory locations that are far apart. Data in global memory remains persistent throughout the execution of the program.

Shared memory

Threads within a block have access to a common, fast *shared memory*. All threads within a block can read and write to the block's shared memory, but they may *not* access shared memory of another block. Thread access to shared memory can be expected to be much faster than access to global memory. Data that may be repeatedly used in a kernel are good candidates for placement in shared memory The contents of shared memory are managed by the kernel code, and for this reason, shared memory is often viewed as explicitly managed cache. Data in shared memory does *not* persist after kernel execution.

Shared memory is organized as a collection of 4 byte "words". Each consecutive word in shared memory belongs to a different bank - modern GPUs have 32

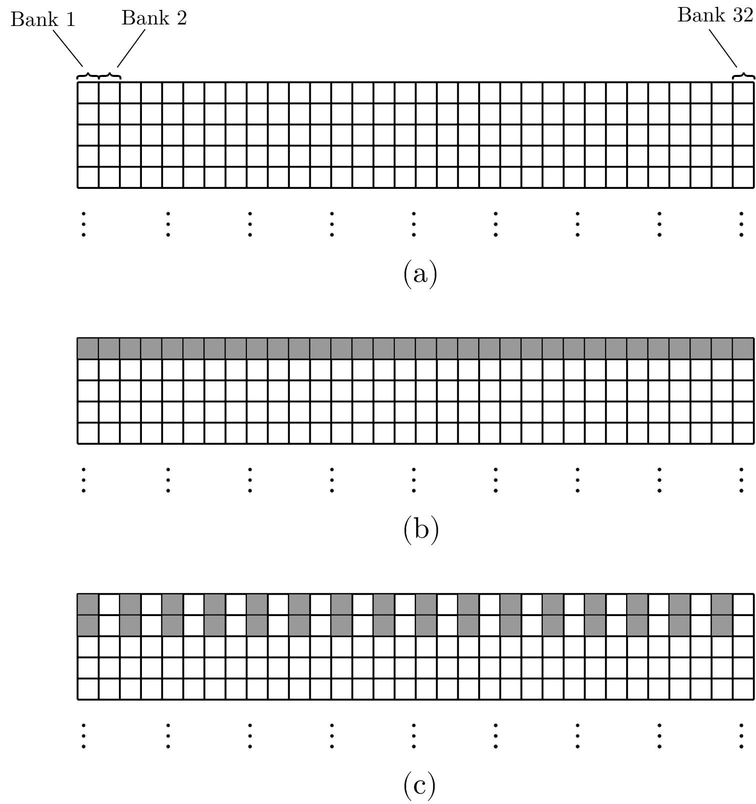

Figure 1.3: (a) Organization of shared memory as 4 byte words organized into 32 banks. Each cell (square) is a word. (b) Bank conflict free access by a warp. Each warp accesses a word from a different bank. (c) 2-way bank conflicts arising from successive threads accessing alternating words.

banks. The GPU schedules thread execution within a block in groups of 32 (termed as thread *warps*). Ideally, these 32 threads can access shared memory locations concurrently. However, when two or more threads in a warp access words from the same bank, a *bank conflict* is said to occur, and the access is serialized. Thus, a warp of 32 threads reading 32 successive `float`s (4 bytes) in a shared memory array is perfectly parallelized. But a warp of 32 threads reading 32 alternating `float`s in a shared memory array leads to bank conflicts (Fig. 1.3). Because the floats are placed in consecutive banks, the 1st and 17th threads will access `float`s from the same bank, as will the 2nd and 18th threads, and so on. This specific kind of bank conflict is termed a *two-way* bank conflict - threads access two words from the same bank. It is observed that higher-order bank conflicts may occur, if threads in a warp access `float`s in strides of 4, a *four-way* bank conflict can occur. In the worst case, successive threads may access `float`s in strides of 32, in which case all threads in the warp request memory from the same bank: a *32-way* bank conflict.

Registers and local memory

Each thread also has private *local memory*, and access to extremely fast registers. Unlike CPUs, the GPU has a large number of registers—a thread executing a kernel will typically attempt to store non-array variables defined in the kernel in registers. Thread access to registers has the lowest latency compared to all other memory spaces. The number of registers available to each thread is limited, and if exceeded, data is instead stored in the thread's local memory. Access to local memory is much slower than registers, so this is typically avoided.

Limiting shared memory and register usage

While shared memory and registers can service memory requests much faster than global memory, their overuse can lead to performance degradation. The amount of shared memory per SM and the number of registers per SM is limited. For the current NVIDIA Tesla K20 accelerator, the amount of shared memory per SM is limited to 48 KiB, and the number of registers per SM is limited to 65536. These are also the limits on the resources that can be allocated for each *block*. However, allocating 48 KiB shared memory or using 65536 registers for each block is ill-advised, as this effectively restricts the number of blocks that each SM can run at any given time to 1. If each block allocates 24 KiB of shared memory, then an SM can run only 2 blocks concurrently. Thus, the resources allocated per-block affects the overall parallelism that can be exploited from the GPU.

1.3.3 Considerations to be made while programming for GPUs

As seen in the previous sections, several factors must be considered while designing and implementing algorithms for GPUs. Failure to include these considerations can easily lead to poor performance, and no significant speedup may be noted over CPUs. In fact, one may even note performance degradation. We list the primary considerations here:

- In any application, the data transfers between the host and device must be minimized. Ideally, data is read into the device from the host *once* and from the device to the host *once*.
- Thread access to global memory must be *coalesced*, i.e., successive threads must read successive locations in global memory.

- Shared memory access must be free of bank conflicts as much as possible—in general, this means avoiding strided shared memory access.
- The amount of shared memory and registers allocated for each block is kept minimum.

1.3.4 Software for programming NVIDIA GPUs

As described, the CUDA programming interface allows developing general-purpose GPU applications—however, there are other options. The OpenCL framework [26] is used to write applications that can be ported across a variety architectures including NVIDIA GPUs. and introduces almost no extra code. Several GPU accelerated libraries [1] allow existing applications to be ported to GPUs by offering compatibility with industry standard CPU libraries. The OpenACC [2] toolkit allows portions of code to be offloaded to GPU accelerators simply by use of compiler directives, similar to OpenMP. This makes porting existing applications extremely easy. Many frameworks for scientific computing such at PETSc [3] also allow the use of GPUs for most of their functionality.

We use the CUDA interface because it fully exposes the features of the underlying architecture that can be exploited. This is especially important due to the complex computational and memory access patterns involved in our algorithms.

1.4 Tridiagonal solvers for compact finite difference evaluation

As mentioned in 1.2.4, this work is concerned with *tridiagonal* compact finite differences for their relative ease in evaluation. The resulting tridiagonal system must

be solved repeatedly for all lines of grid points in the computational grid. For a regular Cartesian grid (generally employed in DNS), the coefficients of the tridiagonal systems are the same for each grid line, and only the right hand sides are different. Thus, the compact finite difference evaluation effectively requires the solution of a tridiagonal system for several right hand sides. In this section, we describe the applicability of some algorithms for solving this problem on the GPU. We refer to [6] for a more comprehensive list of algorithms, along with details of their GPU implementation.

1.4.1 Thomas algorithm

$$\begin{pmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & c_3 & & & \\ & & a_4 & b_4 & c_4 & & \\ & & & & \ddots & & \\ & & & & & \ddots & \\ & & & & & a_n & b_n \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ \vdots \\ \vdots \\ d_n \end{bmatrix} \tag{1.6}$$

Traditionally, the Thomas algorithm is employed for solving general linear systems of the form $A\boldsymbol{x} = \boldsymbol{d}$, where A is a tridiagonal matrix with diagonals $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ [Eq. (1.6)]. The algorithm is derived from the more general Gaussian (LU) algorithm applied to tridiagonal matrices. Implementations of this algorithm are relatively straightforward and compact [24]. Only the non-zero diagonals of the tridiagonal matrix are stored, and the algorithm performs $2N$ steps to solve the system. It is also stable for diagonally dominant matrices. The Thomas algorithm has an algorithmic complexity of $O(N)$, and is the fastest serial solver for tridiagonal systems. It is

thus well suited to solution by a single CPU thread. While the algorithm itself is sequential, in a multithreaded environment, the different CPU threads may be used to solve the different independent tridiagonal systems simultaneously.

A similar parallelization strategy may be extended to GPUs, where each GPU thread solves an independent tridiagonal system. The coefficient right hand sides are assumed to be stored contiguously as a single array in memory. If each thread works entirely with global memory, *uncoalesced* memory access is observed. This cost *may* be amortized for a large enough number of systems, as is the case for 3-D problems [25]. If shared memory is used, then low parallelism is exhibited, as the amount shared memory allocated by each thread is relatively high ($4N$). An approach that uses the parallel Thomas algorithm effectively is described by Chang et al. [5]. Here, a *parallel cyclic reduction* algorithm is first used to reduce a tridiagonal system into several smaller tridiagonal system, and the parallel Thomas algorithm is then used to solve the smaller systems in parallel.

1.4.2 Cyclic reduction

Two other popular algorithms for solving tridiagonal systems are the cyclic reduction (CR) and the related parallel cyclic reduction (PCR) algorithms.

The cyclic reduction algorithm consists of two phases: *forward reduction* and *backward substitution* (Fig. 1.4). In the forward reduction phase, every even-indexed equation i is expressed as a linear combination of equations i, $i-1$ and $i+1$, yielding a new tridiagonal system of $n/2$ equations in $n/2$ unknowns [Eqs. (1.7) - (1.10)]. The process is repeated until a system of 2 equations in 2 unknowns is left.

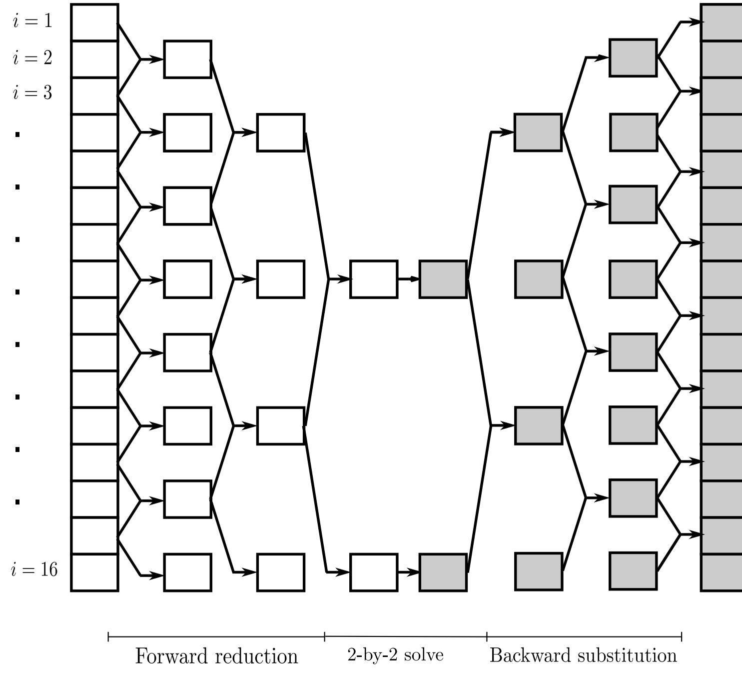

Figure 1.4: Cyclic reduction.

$$a_i' = -a_{i-1}k_1 \tag{1.7}$$

$$b_i' = b_i - c_{i-1}k_1 - a_{i+1}k_2 \tag{1.8}$$

$$c_i' = -c_{i+1}k_2 \tag{1.9}$$

$$d_i' = d_i - d_{i-1}k_1 - d_{i+1}k_2 \tag{1.10}$$

where,

$$k_1 = \frac{a_i}{b_{i-1}} \tag{1.11}$$

$$k_2 = \frac{c_i}{b_{i+1}} \tag{1.12}$$

The 2-by-2 system of equations is solved trivially, yielding x_n and $x_{n/2}$. In the backward substitution phase, every odd-indexed unknown x_i is solved for by substituting the known values of x_{i-1} and x_{i+1} [Eq. (1.13)].

$$x_i = \frac{d'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i} \tag{1.13}$$

For the last index $i = n$, the forward reduction step is instead:

$$a'_n = -a_{n-1} k_1 \tag{1.14}$$

$$b'_n = b_n - c_{n-1} k_1 \tag{1.15}$$

$$d'_n = d_n - d_{n-1} k_1 \tag{1.16}$$

And for $i = 1$, the backward substitution step is instead:

$$x_1 = \frac{d'_1 - c'_1 x_2}{b'_1} \tag{1.17}$$

In practice, the right-hand side vector can be safely overwritten with the solution values in backward substitution.

Thus, in the best case (n parallel processors), cyclic reduction requires $2log_2(n) - 1$ steps. For even moderately large n, this is significantly smaller than the $2n$ steps required by the Thomas algorithm. This makes cyclic reduction a good fit for massively parallel architectures like GPUs.

1.4.3 Parallel cyclic reduction

The parallel cyclic reduction (PCR) algorithm has only the forward reduction phase. The first forward reduction step is applied to the odd and even indexed equations seperately, yielding two reduced systems of size $n/2$. Forward reduction

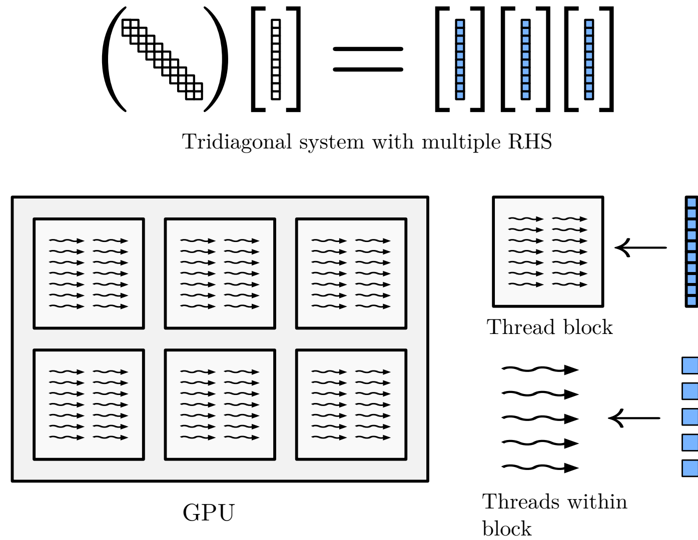

Figure 1.5: Mapping work to blocks and threads: systems are mapped to blocks and indices are mapped to individual threads.

applied to both of these systems then yields *four* reduced systems of size $n/4$. The process is repeated till $n/2$ 2-by-2 systems are left, all of which can be solved trivially. The PCR algorithm requires half the number of steps required by CR ($log_2(n)$), but does significantly more computation per-step. Further, unlike CR, PCR can be implemented free of bank conflicts [30].

1.4.4 Cyclic reduction implementation on GPUs

The algorithm proposed in this work is based on cyclic reduction, so it is pertinent to discuss the implementation of cyclic reduction on GPUs, the associated issues, and the relevant literature.

In the GPU implementation of cyclic reduction, blocks are assigned to tridiagonal systems (when solving multiple systems), and threads within a block are assigned to equations, or *indices* (Fig. 1.5). In this way, several grid lines are solved concurrently by the GPU. During the forward reduction phase, the threads assigned to each

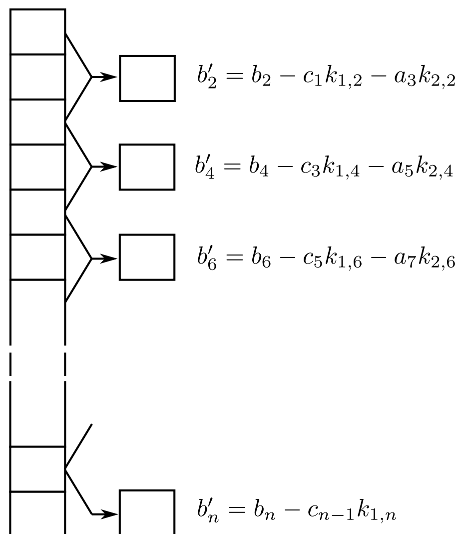

Figure 1.6: Updating $\boldsymbol{b}$ in the first forward reduction step.

even index i compute the coefficients and right hand side for the reduced tridiagonal system a'_i, b'_i and c'_i and d'_i. In practice, the coefficients and right hand side are updated *in-place*. Figure 1.6 shows the updates to the coefficient array $\boldsymbol{b}$ in the first forward reduction step, a similar pattern is seen for the arrays $\boldsymbol{a}$, $\boldsymbol{c}$ and $\boldsymbol{d}$. In each step of forward reduction, a thread accesses values from the coefficient arrays and right hand side in a *strided* fashion. At every subsequent step, this stride is doubled, while the number of active threads is halved 1.4. In the backward substitution phase, the strides are halved at each step, while the number of active threads is doubled.

Several issues are encountered in the GPU implementation:

1. GPU utilization is low towards the end of forward reduction, and in the beginning of backward substitution.

2. Because coefficients and right hand sides are updated *in-place*, synchronization between the blocks is required at the end of each step.

3. If the threads work entirely with global memory, memory accesses are increas-

ingly *uncoalesced* in the forward reduction phase (and become increasingly coalesced in the backward substitution phase).

4. The use of shared memory prevents uncoalesced global memory access. Unfortunately, the power-of-two strides at each successive leads to bank conflicts, as described in Sec. 1.3.2. The bank conflicts become increasingly severe at each forward reduction phase, and decreasingly so during the back substitution phase.

5. The limited amount of shared memory places restrictions on the size of tridiagonal systems that can be solved, and also on the number of systems that can be solved concurrently.

Despite these issues, cyclic reduction remains an attractive algorithm for GPUs, for its low algorithmic complexity, and lower work per step compared to PCR. Much work has been done on addressing these problems and optimizing cyclic reduction performance on GPUs. Zhang et al. [30] propose a *hybrid* solver that uses both cyclic reduction and parallel cyclic reduction to reduce the number and severity of bank conflicts, and also to have better thread activity overall. Göddeke et al. [11] use a method of separately storing the even and odd indexed equations to arrive at a bank-conflict free solver at the cost of additional shared memory usage. Davidson et al. [8] describe the method of *register packing*—performing more computations on *registers*, rather than shared memory—as a means to reduce shared memory usage in cyclic reduction. Esfahanian et al. [9] avoid shared memory (and associated bank conflicts entirely) using a data rearrangement scheme to improve global memory access. Our approach takes a different route, and is focused on exploiting the specific matrix structure to reduce the number of computations and memory accesses performed by each thread at every cyclic reduction step.

Chapter 2

Proposed tridiagonal algorithm

2.1 Modified cyclic reduction for near-Toeplitz systems

Here, we explore the idea of exploiting the relatively simple matrix structure of the coefficient matrix appearing in tridiagonal compact finite difference schemes. The coefficient matrix A for tridiagonal schemes has the following general form:

$$A = \begin{pmatrix} b_1 & c_1 & & & & & \\ a_0 & b_0 & c_0 & & & & \\ & a_0 & b_0 & c_0 & & & \\ & & a_0 & b_0 & c & & \\ & & & & \ddots & & \\ & & & & & \ddots & \\ & & & & & a_n & b_n \end{pmatrix} \tag{2.1}$$

where, in general

$$a_0 \neq a_n$$

$$b_1 \neq b_0 \neq b_n$$

$$c_1 \neq c_0$$

We refer to matrices with this specific structure as *near-Toeplitz tridiagonal matrices*, and the corresponding linear systems as *near-Toeplitz tridiagonal systems*. These matrices appear in a wide range of applications [27] such as alternating direction implicit methods, line relaxation methods, and numerical solutions to one-dimensional differential equations. We describe an approach for solving such *near-Toeplitz* tridiagonal systems efficiently on GPUs.

Below, we present the effect of the matrix structure [Eq. (2.1)] on the forward reduction and backward substitution phases of the cyclic reduction algorithm.

2.1.1 Forward reduction

The forward reduction phase reduces a $n - by - n$ system of equations to a $2 - by - 2$ system of equations in $log_2(n) - 1$ steps, by applying Eqs. (1.7) - (1.10) to every even-indexed equation at each step. When solving the tridiagonal systems with the same coefficient matrix, but repeatedly for different right hand sides, we note that the results of Eqs. (1.7) - (1.9) remain unchanged for the different right hand sides. These results correspond to the coefficients of the tridiagonal system produced at each forward reduction step. Thus, given a tridiagonal system, we may precompute the coefficients of all the reduced systems appearing in the forward reduction steps, and reuse them for each right-hand side. We note that for a general tridiagonal system,

this requires storage for $3.(2n-2)$ coefficients, in addition to the $3n$ coefficients for the original tridiagonal system.

Let us consider the case when the tridiagonal system is *near-Toeplitz*, i.e., when the coefficient matrix is of the form A in Eq. (2.1). We examine the effect of the first forward reduction step by making the following substitutions in Eqs. (1.7) - (1.9):

$$
\begin{aligned}
a_2 = a_3 = a_4 = \dots \qquad & \equiv a_0 \\
b_2 = b_3 = b_4 = \dots \qquad & \equiv b_0 \\
c_2 = c_3 = c_4 = \dots \qquad & \equiv c_0
\end{aligned}
$$

We observe that the resulting coefficients a'_i, b'_i and c'_i correspond to the coefficients of a tridiagonal matrix with exactly the *near-Toeplitz* structure of A. This form-preserving property of cyclic reduction has been reported for block Toeplitz tridiagonal systems by Bini et al. [4].

The fact that the reduced system at each step is *near-Toeplitz* can be exploited to reduce the cost of precomputing and storing the coefficients. Each *near-Toeplitz* matrix is completely defined by only a handful of coefficients: $\{b_1, c_1, a_0, b_0, c_0, a_n, b_n\}$, making its storage extremely compact compared to the case of general tridiagonal systems. In addition, it is advantageous to precompute and store the auxiliary variables k_1 and k_2, which can similarly be stored compactly. With all the forward reduction coefficient matrices and auxiliary variables precomputed and stored, the mth forward reduction step for equation i is reduced only to the right hand side update:

$$d'_i = d_i - d_{i-1}k_1^m - d_{i+1}k_2^m \tag{2.2}$$

where k_1^m and k_2^m are precomputed values of k_1 and k_2 for all "inner" equations at the mth step. For the "outer" equations $i = 2$ and $i = n$, we have instead:

$$d'_2 = d_2 - d_1 k_{1,1}^m - d_3 k_2^m \tag{2.3}$$

$$d'_n = d_n - d_{n-1} k_{1,n}^m \tag{2.4}$$

here $k_{1,2}^m$ and $k_{1,n}^m$ are precomputed values of k_1 for the "outer" equations at the mth step.

2.1.2 Backward substitution

The backward substitution step for all equations $i > 1$ is

$$x_i = \frac{d'_i - a^m x_{i-1} - \ c^m x_{i+1}}{b^m} \tag{2.5}$$

where a^m, b^m and c^m are the precomputed coefficients for $i > 1$ at the step m. For the first equation, we have instead:

$$x_1 = \frac{d'_1 - c^m x_2}{b_1^m} \tag{2.6}$$

where b_1^m is the coefficient computed for $i = 1$ at step m.

2.2 Implementation

In this section, we describe the implementation of a GPU solver for solving a given *near-Toeplitz* tridiagonal system for multiple right hand sides. Our implementation uses the NVIDIA CUDA platform for programming the GPU, but is easily translated to OpenCL. The Python programming language is used to interface with

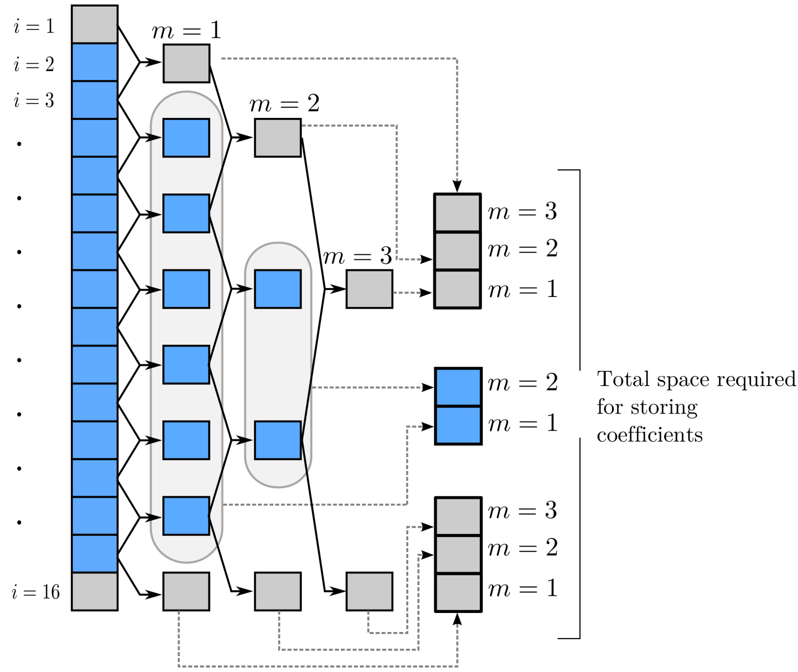

Figure 2.1: Maximum storage required for forward reduction coefficients at all steps $m = 1, 2, ...log_2(n) - 1$

CUDA, by use of the PyCUDA [14] library. We develop two approaches based on the common idea of precomputed forward reduction coefficients—one that leverages the GPU's *shared memory*, and the other working directly with global memory. The GPU kernels for both implementations are relatively straightforward and compact, spanning no more than 100 lines of code (see Appendix).

2.2.1 Precomputing forward reduction coefficients

Figure 2.1 shows the *maximum* amount of storage required for storing each of the coefficients $\{b_1, c_1, a_0, b_0, c_0, a_n, b_n\}$, or auxiliary variables k_1 and k_2. A careful analysis of the forward reduction and backward substitution equations reveals that the actual amount of storage is somewhat less. For instance, the values b_n^m are unused in Eqs (2.2) - (2.6). The values a_n^m and b_n^m are required only for solving the $2 - by - 2$

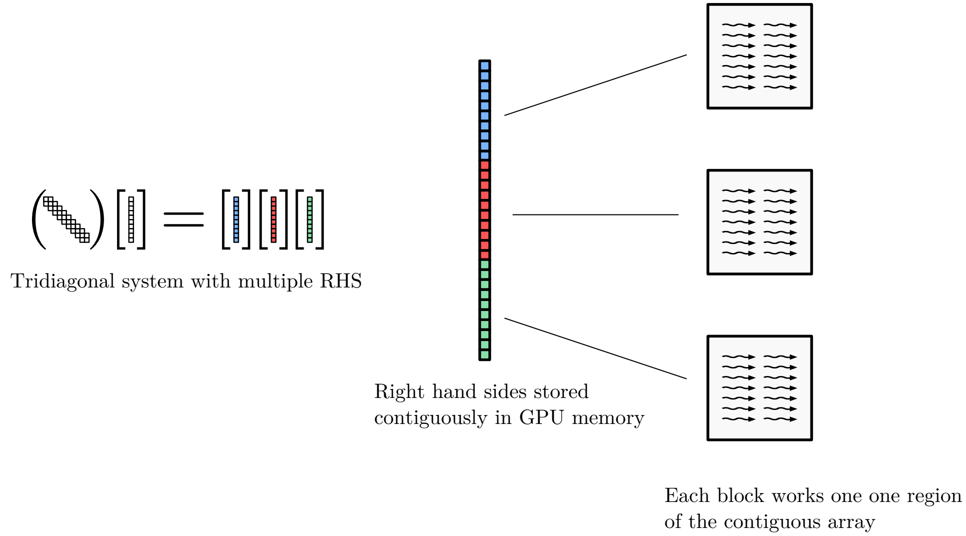

Figure 2.2: Storing right hand sides and mapping to thread blocks.

system at the end of the forward reduction phase, i.e., for $m = log_2(n) - 1$, and they are not stored for the previous steps. Similarly, the values c_1^m are equal to c_0^m, and do not require separate storage. The set of precomputed coefficients required to be stored is then $\{a_0^m, b_0^m, c_0^m, k_1^m, k_2^m, b_1^m, k_{1,1}^m, k_{1,n}^m\}$. Each of these "coefficient arrays" are computed on the CPU and transferred to the GPU. Additionally, the two scalars $a_n^{log_2(n)-1}$ and $b_n^{log_2(n)-1}$ are required for the 2-by-2 solve. The right hand sides that the system must be solved for are stored in a single contiguous array in GPU memory 2.2. The precomputed coefficient arrays and the right hand side array are passed as inputs to the compute kernels that implement the modified cyclic reduction. We describe two implementations of the kernels, the first works entirely with global memory, and the second leverages the GPU's *shared* memory.

2.2.2 Global memory implementation

This implementation works entirely on the GPU's global memory, Here, we define two kernels - one for the forward reduction step, and the other for the backward

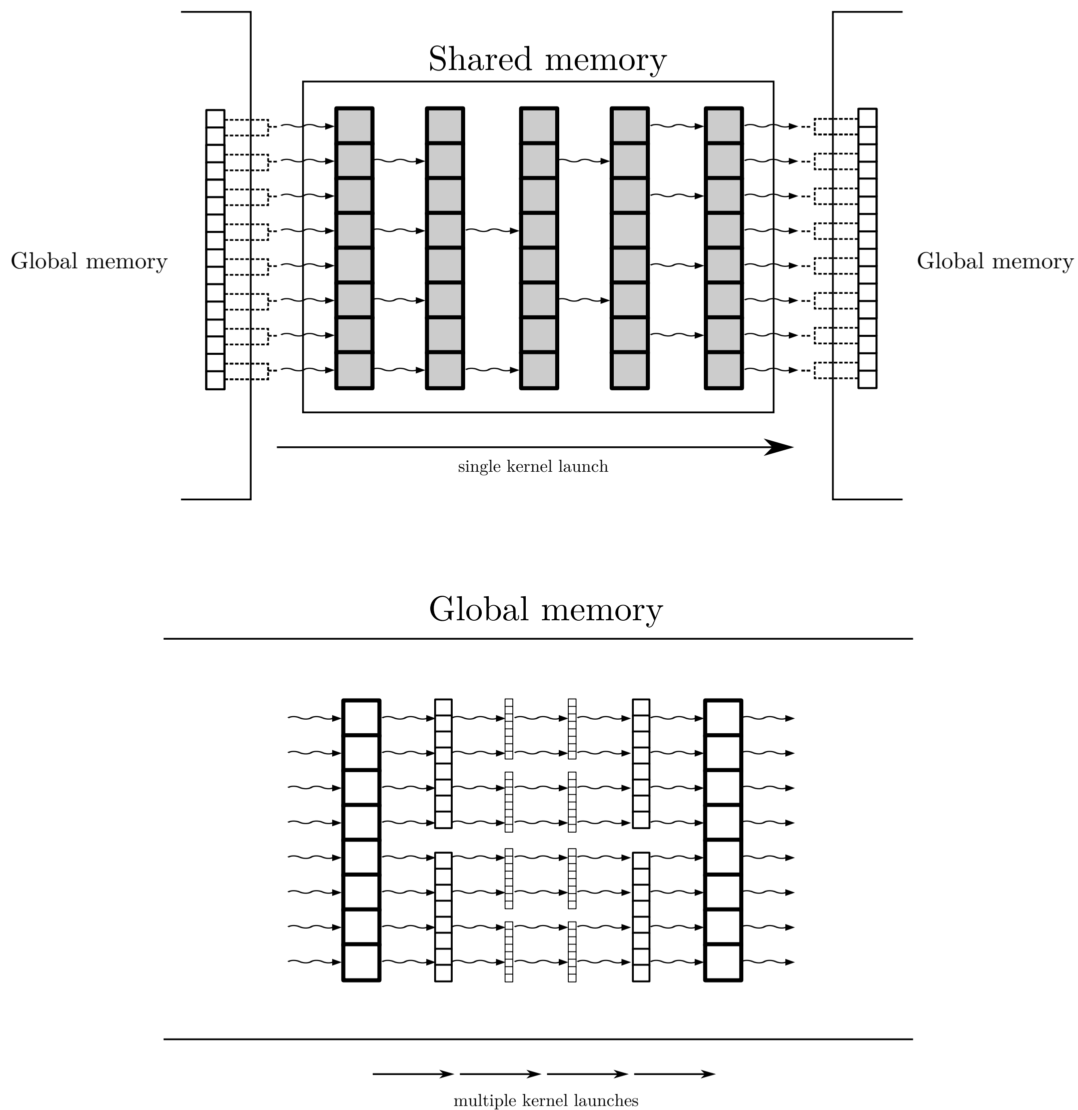

Figure 2.3: Thread activity in shared memory (top) and global memory (bottom) implementations.

substitution step. Each kernel is called $log_2(n) - 1$ times. An extra call to the forward reduction kernel performs the two-by-two solve. At each step, the size of the thread blocks is determined by the *stride* between elements accessed at that step. For the forward reduction phase, we use $n/2$ threads per block for the first step, $n/4$ threads for the second step, and so on. The pattern is reversed for the backward substitution phase, beginning with 2 threads per block for the first step. Although this ensures that there are no inactive threads at any stage, the occupancy of the GPU is still very low during the end of forward reduction and the end of backward substitution. The precomputed coefficient arrays and right hand are accessed by the kernels from global memory. The kernel suffers from strided memory access for the right-hand side, but the precomputed coefficient values are accessed without major coalescing problems. Further, by precomputing the forward reduction coefficients, we greatly reduce the number of computations (and thus the number of uncoalesced memory accesses).

2.2.3 Shared memory implementation

In the shared memory approach (Fig. 2.3), we launch a single kernel to perform the entire cyclic reduction solve. The kernel is launched with $n/2$ threads per block. Each thread block is allocated a block of shared memory of size $n/2$. Each thread of a block performs the first reduction step [Eq. (2.2)] by accessing the required values d_i, d_{i-1} and d_{i+1} from global memory, storing the result in shared memory. In subsequent reduction steps, d_i, d_{i-1} and d_{i+1} are accessed from shared memory, avoiding the uncoalesced global memory accesses seen in the global memory implementation. In each back substitution step, threads overwrite the existing values in shared memory with the values of the solution. In the final step, shared memory is filled completely with the even-indexed solution values. Each thread then computes an odd-indexed solution

value, storing it directly in global memory and copies the even-indexed solution value from shared memory to global memory. The entire solution is done in a single kernel launch to hide the latency of memory transfers between global and shared memory. Explicit synchronization between the threads of a block is required within the kernel at the end of each step.

The shared memory implementation suffers from two major issues: first, the number of active threads is halved at each forward reduction step, (and subsequently doubled at each backward reduction step). Synchronization between threads of a block is necessary at each step. Thus, a significant portion of the kernel execution time is spent by idle threads waiting for active threads to complete execution. Secondly, the strided access to the right-hand side values leads to bank-conflicts. However, the number of bank conflicts is significantly smaller than in cyclic reduction implementations for general tridiagonal systems, due to the reduced number of computations performed.

Chapter 3

Application to compact finite difference evaluation

3.1 Introduction

Here, we discuss the application of the tridiagonal solver developed in the previous section to the evaluation of compact finite differences—which are used widely in the direct numerical simulation of fluid flows. In typical DNS applications, the computational domain considered is a 3-dimensional, regular, Cartesian grid with nx, ny and nz grid points and grid spacing of dx, dy and dz in the respective coordinate directions (Fig. 3.1). Structured grids such as these are widely used because they lend themselves naturally to finite-difference methods. To resolve all the spatial features of the flow, the grid spacing is kept very small. Consequently, to simulate flows of practical sizes, the number of grid points must be very large. Because of the high computational cost and memory requirements associated with the large number of grid points parallelism becomes mandatory. We have spoken so far of GPUs as massively parallel systems. However, each GPU has a very limited amount of global

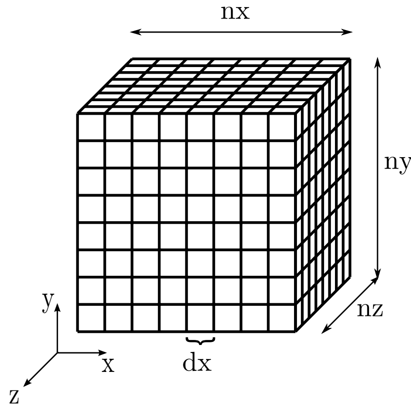

Figure 3.1: Computational domain in 3-D.

memory: current NVIDIA Tesla K20 GPUs have about 4 Gigabytes. Thus, to accommodate the larger problem sizes in DNS, the use of *multiple* GPUs is necessary, which introduces a second level of parallelism. In the next section, we describe strategies to distribute the problem domain among the GPUs in such dual-level parallel systems.

3.2 Compact finite difference evaluation on parallel GPU systems

The computation of derivatives using compact finite differences involves two primary steps:

1. The evaluation of the right hand sides of Eq. (1.1) at each grid point.

2. The solution of the tridiagonal system [Eq. (1.1)] for each line of grid points.

Both of these operations are amenable to parallel operations. The right hand side calculation is a pointwise *stencil* operation, i.e., at every point in the domain, the value of the right hand side is computed as a combination of function values at that point and its neighbouring points. The stencil operations at individual points

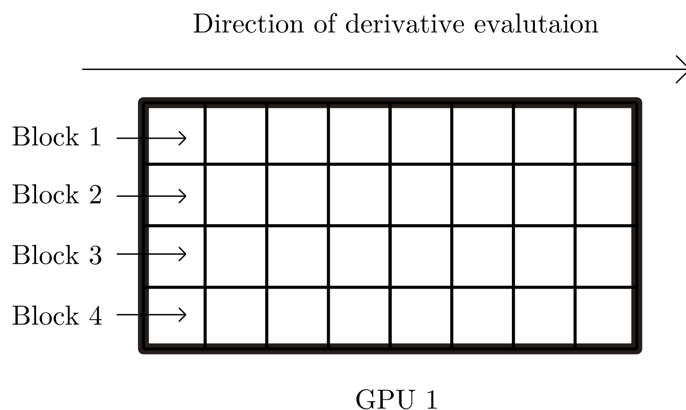

Figure 3.2: Compact finite differences - single GPU

are independent, making the overall calculation highly parallelizable. The solution of the tridiagonal systems is of course parallelizable, as has been discussed in previous sections.

Without loss of generality, we consider the parallelization of compact finite difference evaluations for 2-dimensional problems. We assume that the derivative is being calulated for the coordinate direction along which elements are stored contiguously in memory, i.e., the “fastest” coordinate direction.

3.2.1 Single GPU

For a single GPU (Fig. 3.2), when calculating the right hand sides, each point in the grid is mapped to a single thread, and each thread applies the required stencil operation to compute the right hand side at that point. We note that threads near the left and right boundaries apply a different stencil from the interior threads. The implementation of GPU kernels for stencil operations such as these is a topic of wide study. The most important considerations were brought out in the paper by Micikevicius et al. [18]. For solving the tridiagonal systems, each thread block of the GPU is mapped to a different *grid line* aligned along the direction the derivatives are

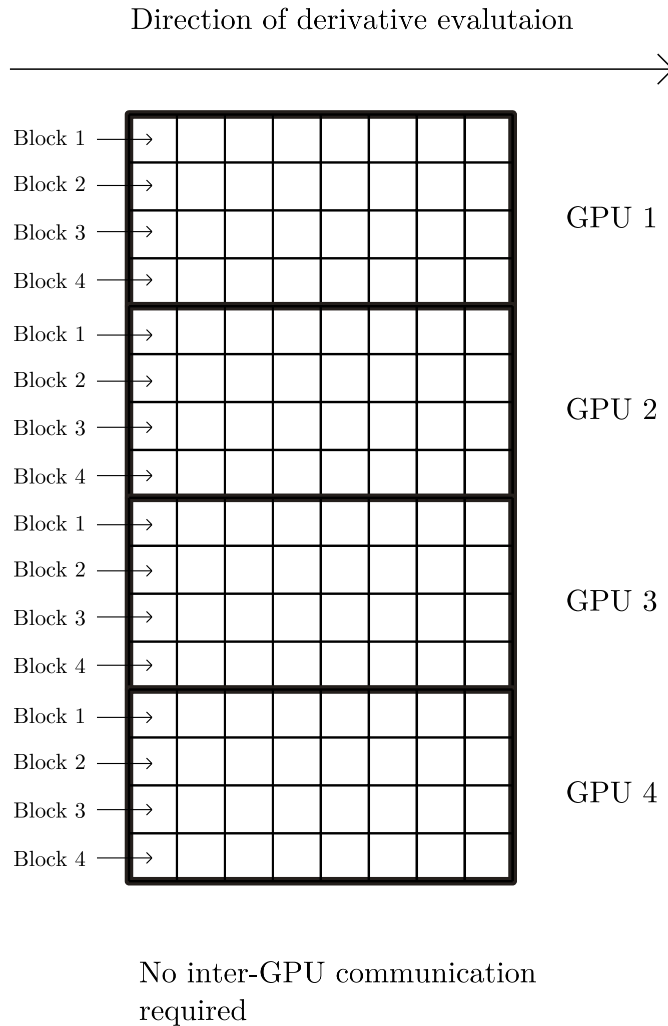

Figure 3.3: Compact finite differences - multiple GPUs on same node

being calculated. The right hand sides are stored contiguously along these grid lines (as in Fig. 2.2), and the modified cyclic reduction algorithm developed can be used to solve for the derivatives.

3.2.2 Multiple GPUs on a single node

Here, we consider the case of multiple GPUs on a single shared memory node, i.e., multiple GPUs attached to the same PCI-e bus. In this case, every GPU is visible

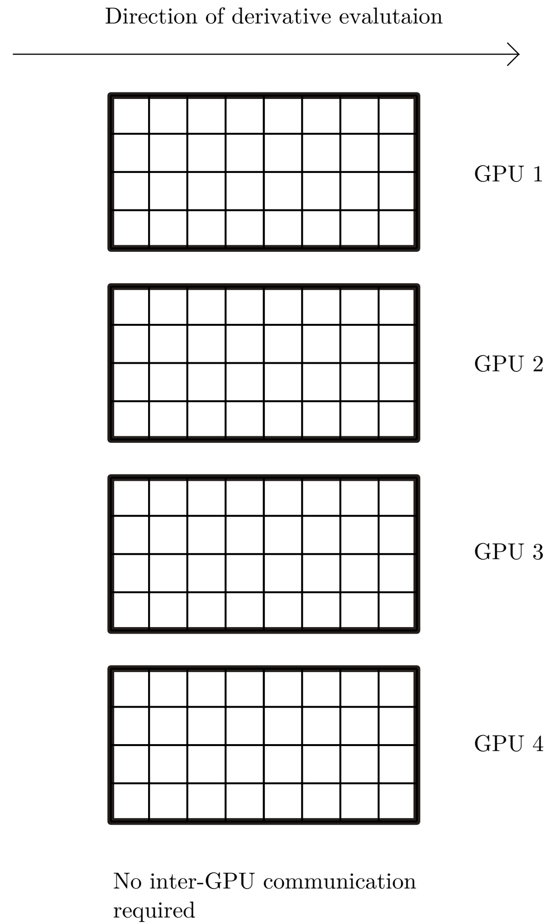

Figure 3.4: Compact finite differences - distributed GPUs and restricted in one direction

to the host, and the GPUs read from and write into the same host memory space. The domain is divided into a number of "subdomains," as shown in 3.3. The domain decomposition is done such that only a single subdomain is used along the coordinate direction of the derivatives. This is the method presented by Sakharnykh et al. [25], and it has the advantage that no coordination between GPUs is required: each GPU is assigned an independent set of grid lines to solve.

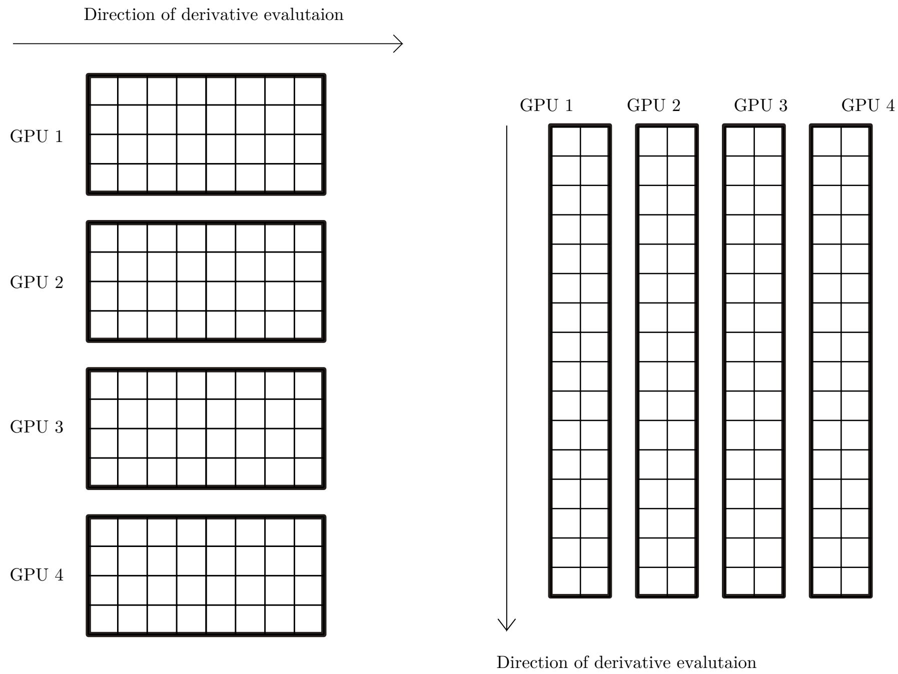

Figure 3.5: Compact finite difference evaluation in both coordinate directions

3.2.3 Distributed GPUs - restricted in one direction

For larger problems, a distributed system is nearly always required. Here, the simplest strategy is to use a domain decomposition as shown in Fig. 3.4. Here again, no inter-GPU communication is required. By restricting the distribution along one coordinate direction, the ease of solution of the tridiagonal systems is maintained.

3.2.4 Distributed GPUs in all directions

The above domain decomposition strategies are convenient, but can be impractical for some cases. For instance, let us consider the evaluation of derivatives in the other coordinate direction (Fig. 3.5). Because the grid lines aligned along this direction must reside on the same GPU, it follows that:

1. A *global* transposition or rearrangement of the data is required, such that each

GPU now contains data for grid lines aligned in the direction orthogonal to the previous direction. For distributed systems, this transposition can be an extremely expensive process.

2. For domains that are much longer along one coordinate direction compared to the other(s), the subdomains may become impractically *slender*.

Such decomposition strategies are therefore, generally applicable when compact finite difference schemes are used only in a single direction. For the other coordinate directions, explicit finite difference schemes may be used, which have the disadvantages discussed earlier. To accommodate compact finite difference schemes in all the coordinate directions, the domain decomposition generally must be performed in all directions, as shown in Fig. 3.6. In this strategy, the grid lines in all coordinate directions are interrupted by the subdomain boundaries. Thus, inter-GPU communication is required. For the right hand sides evaluation, each GPU must communicate information at the subdomain boundaries with neighbouring GPUs. For example, in Fig. 3.6, the right-most grid points in the subdomain of GPU 2 require data from the left-most grid points in the subdomain of GPU 6 when evaluating derivatives in the horizontal direction. Similarly, the left-most grid points in the subdomain of GPU 6 require data from the right-most grid points in the subdomain of GPU 2. Thus, a "swapping" of the boundary information is required at each of the subdomain boundaries. The solution of the distributed tridiagonal systems involves much more complexity, and is discussed in detail in the next section.

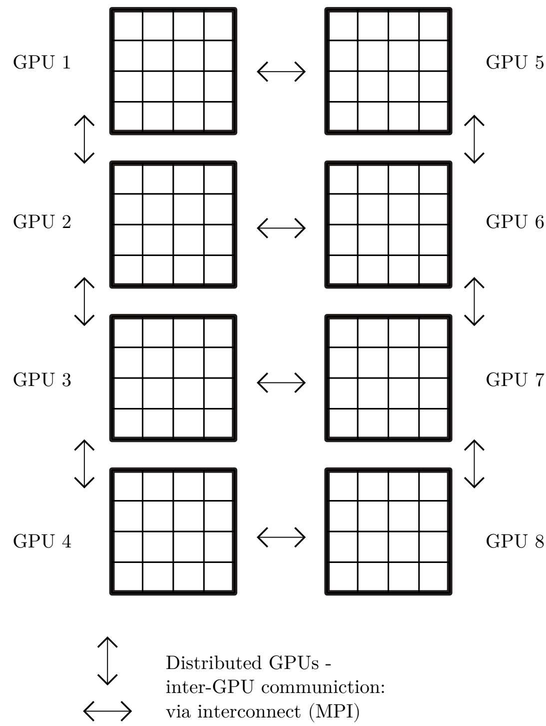

Figure 3.6: Compact finite differences - distributed GPUs in all directions

3.3 Distributed tridiagonal solver

3.3.1 General algorithm

We use the algorithm proposed by Mattor et al. [17] for solving the distributed tridiagonal systems. For a general tridiagonal system $A\boldsymbol{x} = \boldsymbol{r}$, the algorithm begins by partitioning the matrix and right-hand side among the P processes. Each process p is then associated with the following *subsystems*

$$A^p \boldsymbol{u}^p = \boldsymbol{r}_{\boldsymbol{u}}^p \tag{3.1}$$

$$A^p \boldsymbol{l}^p = \boldsymbol{r}_{\boldsymbol{l}}^p \tag{3.2}$$

$$A^p \boldsymbol{x}_{\boldsymbol{r}}^p = \boldsymbol{r}^p \tag{3.3}$$

expanded as:

$$\begin{bmatrix} b_1^p & c_1^p & & & & \\ a_2^p & b_2^p & c_2^p & & & \\ & a_3^p & b_3^p & c_3^p & & \\ & & a_4^p & b_4^p & c_4^p & \\ & & & & \ddots & c_{m-1}^p \\ & & & & a_m^p & b_m^p \end{bmatrix} \begin{bmatrix} x_{r,1}^p \\ x_{r,2}^p \\ x_{r,3}^p \\ x_{r,4}^p \\ \vdots \\ x_{r,m}^p \end{bmatrix} = \begin{bmatrix} r_1^p \\ r_2^p \\ r_3^p \\ r_4^p \\ \vdots \\ r_m^p \end{bmatrix} \tag{3.4}$$

(3.5)

$$
\begin{bmatrix}
b_1^p & c_1^p & & & & \\
a_2^p & b_2^p & c_2^p & & & \\
 & a_3^p & b_3^p & c_3^p & & \\
 & & a_4^p & b_4^p & c_4^p & \\
 & & & & \ddots & c_{m-1}^p \\
 & & & & a_m^p & b_m^p
\end{bmatrix}
\begin{bmatrix}
u_1^p \\ u_2^p \\ u_3^p \\ u_4^p \\ \vdots \\ u_m^p
\end{bmatrix}
=
\begin{bmatrix}
-a_1^p \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0
\end{bmatrix}
\tag{3.6}
$$

$$
\begin{bmatrix}
b_1^p & c_1^p & & & & \\
a_2^p & b_2^p & c_2^p & & & \\
 & a_3^p & b_3^p & c_3^p & & \\
 & & a_4^p & b_4^p & c_4^p & \\
 & & & & \ddots & c_{m-1}^p \\
 & & & & a_m^p & b_m^p
\end{bmatrix}
\begin{bmatrix}
l_1^p \\ l_2^p \\ l_3^p \\ l_4^p \\ \vdots \\ l_m^p
\end{bmatrix}
=
\begin{bmatrix}
0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ -c_m^p
\end{bmatrix}
\tag{3.7}
$$

We refer to the subsystem in Eq. (3.4) as the "primary" system, and the subsystems in Eqs. (3.6) and (3.7) as the "secondary" systems. Additionally, the following "reduced" system must be constructed andsolved:

$$
\begin{bmatrix}
l_m^1 & -1 & & & & & & \\
-1 & u_1^2 & l_1^2 & & & & & \\
 & u_m^2 & l_m^2 & -1 & & & & \\
 & & -1 & u_1^3 & l_1^3 & & & \\
 & & & u_m^3 & l_m^3 & -1 & & \\
 & & & & \ddots & \ddots & \ddots & \\
 & & & & & -1 & u_1^P
\end{bmatrix}
\begin{bmatrix}
\beta^1 \\ \alpha^2 \\ \beta^2 \\ \alpha^3 \\ \beta^3 \\ \vdots \\ \alpha^P
\end{bmatrix}
=
\begin{bmatrix}
x_{r,m}^1 \\ x_{r,1}^2 \\ x_{r,m}^2 \\ x_{r,1}^3 \\ x_{r,m}^3 \\ \vdots \\ x_{r,1}^P
\end{bmatrix}
\tag{3.8}
$$

The local part of the solution is obtained as a linear combination of the solutions to the primary and secondary systems:

$$
\boldsymbol{x}^p = \boldsymbol{x}_r^p + \ \alpha^p \boldsymbol{u}^p + \beta^p \boldsymbol{l}^p \tag{3.9}
$$

where α^p and β^p are obtained from the solution of the reduced system.

3.3.2 Specialization for compact finite difference evaluations

The algorithm described in Sec. 3.3.1 is applicable to a single tridiagonal system distributed across several processors. When solving a distributed tridiagonal system for several distributed grid lines, we note that the secondary systems [Eq. (3.6) and Eq. (3.7)] are dependent only on $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$, the tridiagonal coefficients of the global tridiagonal system. These are the same for all local grid lines, so they are solved for a single local grid line in each subdomain. Only the system in Eq. (3.4) is solved for all the grid lines in a subdomain. For a single tridiagonal system, the reduced system requires *global* communication of the values $\{u_1^p, u_m^p\, l_1^p, l_m^p\, x_{r,1}^p x_{r,m}^p\}$ from each process p. When solving for several grid lines, the left hand side may be

assembled for a single grid line. The right hand sides, however, must be assembled for every grid line, and the system is solved for each right hand side.

3.4 GPU implementation

The algorithm for our distributed *nearo-Toeplitz* solver (NEATO) is outlined in Fig. 3.7, and is described for the case of evaluating derivatives in the direction along which successive function values in a subdomain are stored contiguously in memory, i.e., the "fastest" coordinate direction. Each step of the algorithm must be implemented on the GPU to avoid data transfer to and from the CPU, which is prohibitively expensive for large problems. Thus, we have several kernels to implement the algorithm. For communication of data between processes, we use the Message Passing Interface (MPI), and leverage the NVIDIA GPUDirect Technology for GPU-GPU communication. MPI is interfaced via the mpi4py [7] Python library. The purpose of each CUDA kernel and MPI call used is described in Table 3.1.

The secondary systems [Eq. (3.6) and Eq. (3.7)] are solved on the CPU, and the results are transferred to the GPU. The primary system [Eq. (3.4)] must be solved for each of the local grid lines in a subdomain, as the right hand sides are different at each of the local grid lines. The evaluation of the right hand sides are pointwise stencil computations, which require communication of the function values at the boundaries of the subdomains. This is achieved using a halo-swapping technique with dedicated contiguous halo arrays for each subdomain, as communication of non-contiguous MPI data types is expensive on the GPU. The primary system is solved for all the local grid lines using the NEATO solver. The set up and solution of the reduced system requires global communication of the boundary information from the solutions $\boldsymbol{u}^p$, $\boldsymbol{l}^p$ and $\boldsymbol{x}_{\boldsymbol{r}}^p$. The tridiagonal coefficients of the reduced system are set up easily, by communicating

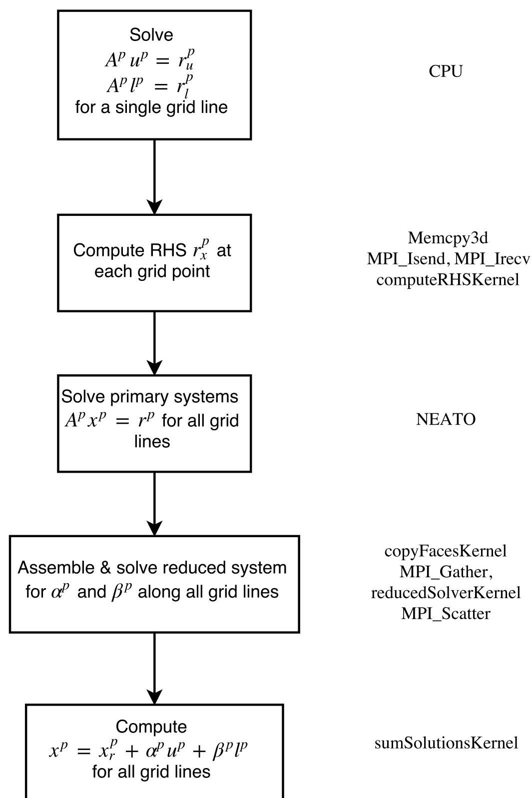

Figure 3.7: Algorithm for evaluating compact finite differences on multiple GPUs, (right: CUDA kernels and MPI calls used)

Table 3.1: Purpose of kernels and MPI calls in compact finite difference application

Memcpy3d	Copy noncontiguous boundary information of the function values to and from contiguous halo arrays
ISend, IRecv	Perform halo swaps with i-1 and i+1 processes
computeRHSKernel	Apply pointwise stencil operator to compute RHS at each grid point in the subdomain, using the halo values near the boundaries
NEATO	Solve the primary near-Toeplitz tridiagonal systems for the computed right hand sides, giving x_r
copyFacesKernel	Copy the left and right faces of x_r into a single contiguous array
MPI_Gather	Gather the data required to assemble the reduced system at rank 0
reducedSolverKernel	Solve the reduced systems for parameters α^p and β^p for each grid line
MPI_Scatter	Scatter the parameters α^p and β^p from rank 0 to all the processes
sumSolutionsKernel	Sum the primary and secondary solutions to compute the local part of the solution

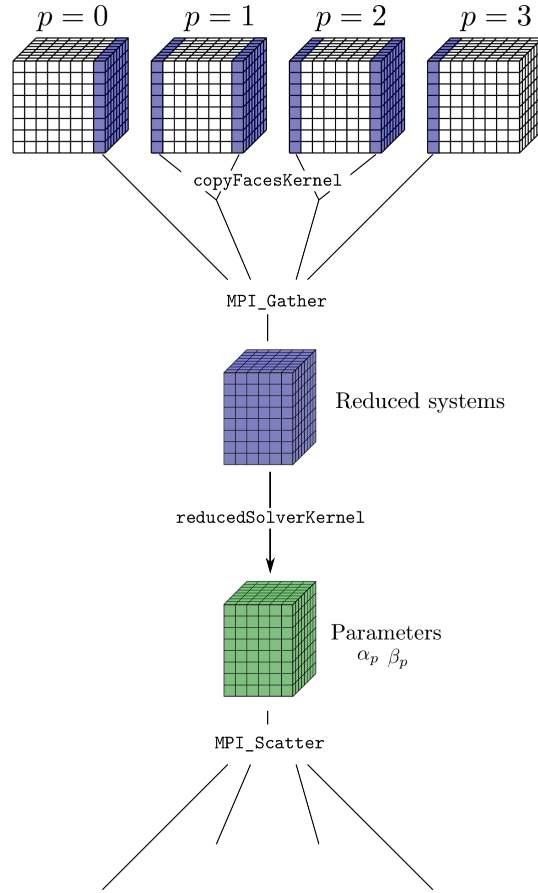

Figure 3.8: Construction of the reduced system and scattering of parameters.

the boundary elements from $\boldsymbol{u}^p$ and $\boldsymbol{l}^p$, which are the same for every local grid line in each subdomain. The right hand sides require significantly more communication, as they are assembled from the boundary elements of $\boldsymbol{x}_{\boldsymbol{r}}^p$, which are different for each local grid line in each subdomain. Thus, the boundary "faces" of each subdomain need to be communicated. These faces are first copied into a contiguous array, and these arrays are gathered at rank 0 to assemble the right-hand sides of the reduced system (Fig. 3.8). This communication strategy has the effect of producing *interleaved* right hand sides aligned along the "slowest" co-ordinate direction, i.e., the right hand side values for neighbouring grid points are located far apart in memory. As the reduced

systems are quite small relative to the primary systems,, we use the convenient, but rather inefficient p-Thomas algorithm to solve the systems. The solution of the reduced systems produces the parameters α^p and β^p, which are scattered back to the respective processes, p. Finally, the summing of the solutions is a pointwise operation that is easily implemented on the GPU. For evaluation of compact finite differences in other coordinate directions, a local permutation of the data is performed on the input data (function values) before applying the above algorithm. and again on the output data (derivative values).

Chapter 4

Results

4.1 Performance of GPU tridiagonal solver

In this section we present a performance overview of our NEATO solver against a multi-threaded Intel MKL solver and the CUSPARSE GPU solver (`dgtsv` and `dgtsvStridedBatch` respectively). The MKL solver uses Gaussian elimination with partial pivoting, and the CUSPARSE solver uses a combination of Cyclic Reduction and Parallel Cyclic Reduction as described by Zhang et al. [30]. These solvers represent the most straightforward way to compute solutions for tridiagonal systems and are highly optimized for performance on underlying architectures. boundary conditions may prevent the matrix from being symmetric and/or diagonally dominant, precluding the use of more specialized tridiagonal solvers.

The CPU code is compiled with the Intel C compiler (version 15.0), and run (with OpenMP support) on up to 16 independent cores of the the same shared-memory node (one thread per core). The CPU is an Intel Xeon Processor E5-2670 v2 (2.50 GHz, 25 MB Smart Cache). GPU code is compiled with the CUDA toolkit (version 6.5.14), and run on the NVIDIA Tesla K20 Accelerator. When measuring

GPU performance (both CUSPARSE and NEATO), we do *not* include the cost of data transfer between the CPU and GPU. This is because the tridiagonal solver is expected to be part of a larger application. This is in keeping with the timing strategies in related literature.

The `-O2` level compiler optimizations are turned on for both CPU and GPU code; no further optimization options are enabled in either case. Of course, we use double precision for all solvers. The timings reported are kernel *execution* times, i.e., the time for all kernel(s) to execute completely before returning the program control to the CPU. All timings are averaged over 100 tridiagonal solves.

4.1.1 NEATO: global memory v/s shared memory performance

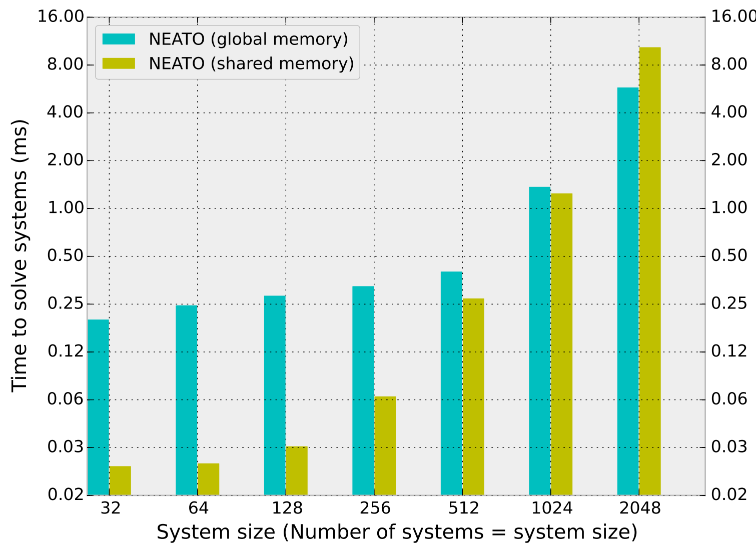

Figure 4.1: Comparison of global memory and shared memory implementations of NEATO (2D problems).

In Figs. 4.1 and 4.2, we report the performance of the two solvers for the case

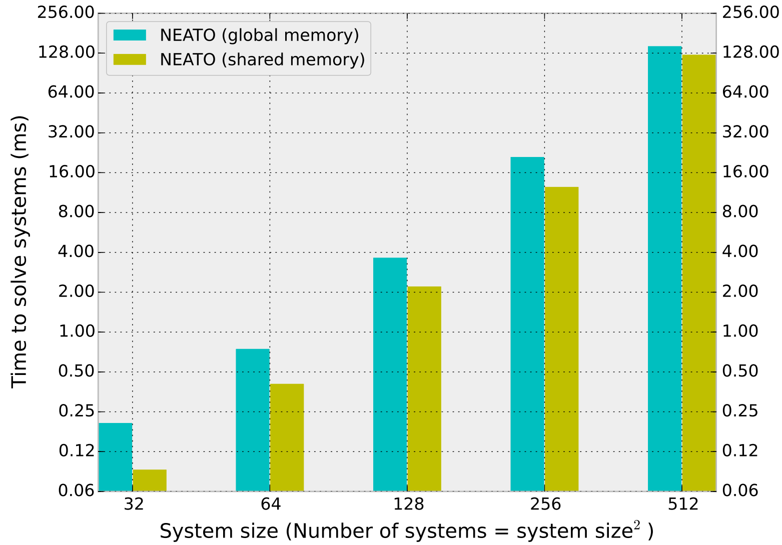

Figure 4.2: Comparison of global memory and shared memory implementations of NEATO (3D problems).

$N_{rhs} = n$ and $N_{rhs} = n^2$. These cases correspond to tridiagonal systems arising in 2-D and 3-D problems respectively. We note that the shared memory implementation offers better performance in nearly all cases. However, the relative speedup from using shared memory diminishes with increasing problem size. For larger problem sizes, the synchronization costs associated with inactive threads leads to poor shared memory performance.

4.1.2 Comparison of NEATO with Intel MKL and CUSPARSE solvers

In Fig. 4.3 and 4.4, we provide the relative performance of Intel MKL and CUSPARSE solvers and compare against the NEATO shared memory implementation. The relative performance for each problem size is obtained by normalizing the solver timings by the timing for the NEATO solver for that problem size. Table 4.1

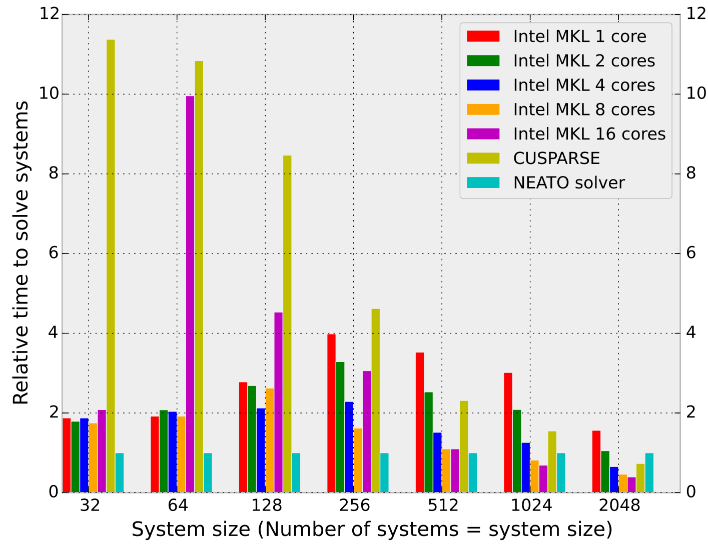

Figure 4.3: Relative solver performance for 2-D problems. Relative time defined as: Time taken by solver/Time taken by NEATO solver

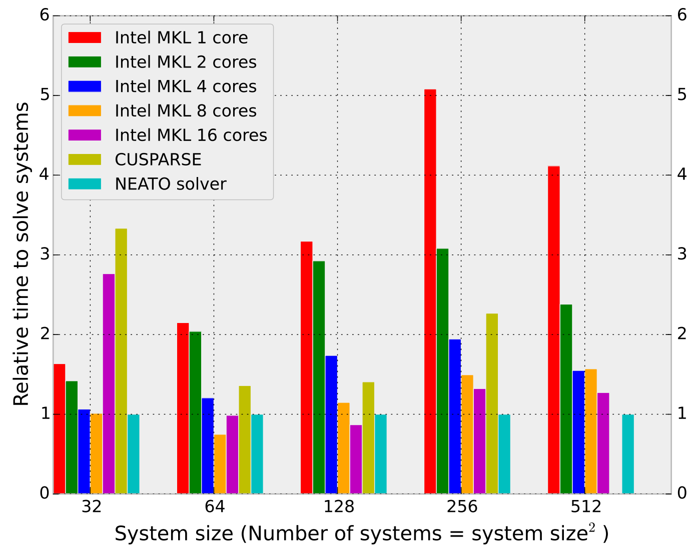

Figure 4.4: Relative solver performance for 3-D problems. Relative time defined as: Time taken by solver/Time taken by NEATO solver

shows the timings of the various solvers to solve different problem sizes. Note that data is missing for the CUSPARSE solver for the 512^3 3-D case, as the GPU was unable to accomodate this problem size—this is due to the large amount of scratch space required by the CUSPARSE implementation.

4.2 Performance of compact finite difference application

The timings for the compact finite difference application were measured on the Clemson University Palmetto Cluster, using NVIDIA Tesla K20 and K40 GPUs. The K40 GPUs were used for the largest problem sizes. Each compute node on the cluster is equipped with up to 2 GPUs, and nodes are connected by 56 Gbps Infiniband interconnect. We use Open MPI 1.8.1 configured with OpenFabrics support.

The timings reported are *wall clock* times with global synchronization between processes performed before and after evaluation of the derivatives. All timings are averaged over 100 evaluations of the function derivatives in each coordinate direction. We make it clear that our reported problem sizes represent the *actual size of problem data*. Although it may be considered sufficient to run tests on a single line of processes for measuring the compact finite difference solver performance, we set up and solve the problem for the entire computational domain. In the context of a larger simulation, global synchronization between the processes is typically performed before and after the evaluation of derivatives, and it is important to consider the related overhead.

Table 4.1: Performance of Intel MKL, CUSPARSE and NEATO solvers.

		Time to solve (ms)				
System size	Number of systems	MKL 1 core	MKL 8 cores	CUSPARSE	NEATO (global)	NEATO (shared)
32	32	0.045	0.042	0.273	0.201	0.024
64	64	0.048	0.048	0.271	0.247	0.025
128	128	0.089	0.084	0.271	0.284	0.032
256	256	0.263	0.107	0.305	0.326	0.066
512	512	0.959	0.299	0.629	0.403	0.272
1024	1024	3.775	1.023	1.939	1.375	1.252
2048	2048	16.272	4.823	7.607	5.811	10.407
32	1024	0.152	0.094	0.31	0.207	0.092
64	4096	0.879	0.306	0.556	0.751	0.409
128	16384	7.052	2.553	3.128	3.669	2.225
256	65536	63.858	18.792	28.495	21.148	12.565
512	262144	515.792	196.748		145.34	125.311

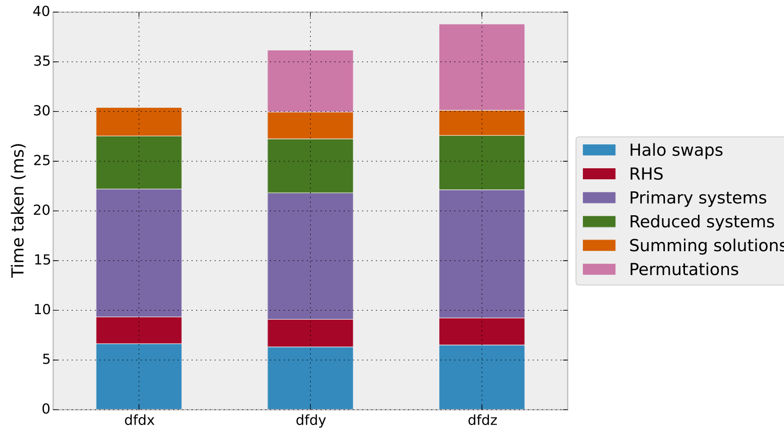

Figure 4.5: Solving problem sized 1024^3 on 64 GPUs

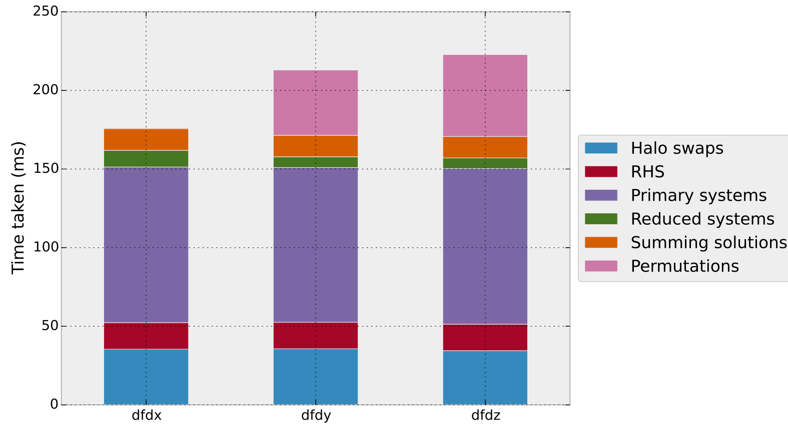

Figure 4.6: Solving problem sized 2048^3 on 64 GPUs

4.2.1 Performance profiling

Figures 4.5 - 4.6 show the time taken by the different steps of the compact finite difference solver. We note that for the larger problem size, the evaluation of the primary systems (using the NEATO solver) constitutes a larger majority of the total runtime, which justifies our efforts in optimizing the tridiagonal solver. For evaluation of the derivatives in the $y-$ and $z-$ directions, we note that a significant portion of the runtime is dedicated to performing permutations of the input and output data.

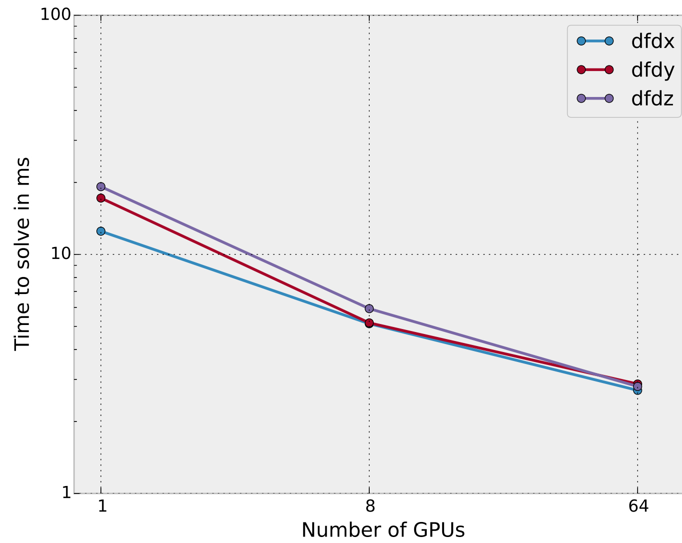

Figure 4.7: Strong scaling for multi-GPU compact finite difference, problem size: 256^3.

We attribute this to our naïve implementation of the permutation kernels (no shared memory usage).

4.2.2 Strong and weak scaling

Figures 4.7 - 4.10 show the strong and weak scaling of the compact finite difference solver for evaluating derivatives in all three coordinate directions. For the strong scaling measurement, we keep the problem size fixed and increase the number of GPUs used to solve the problem. For the weak scaling measurement, we keep the problem size *per GPU* fixed, and increase the number of GPUs used. The strong scaling for larger problems is somewhat better as the GPU is kept more busy.

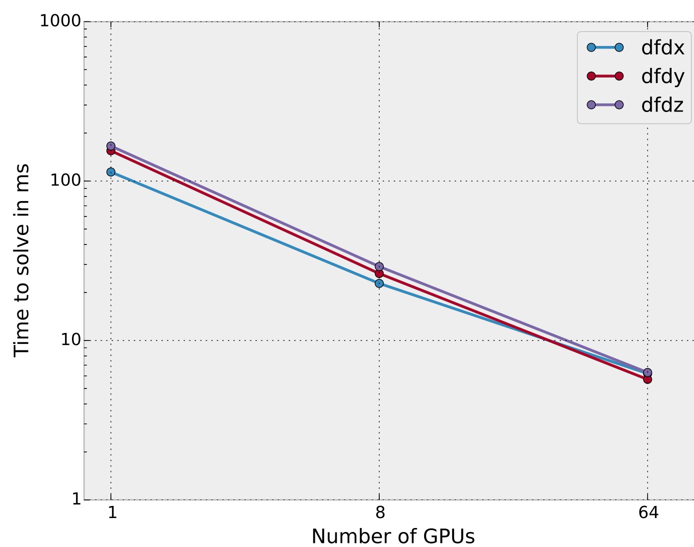

Figure 4.8: Strong scaling for multi-GPU compact finite difference, problem size: 512^3.

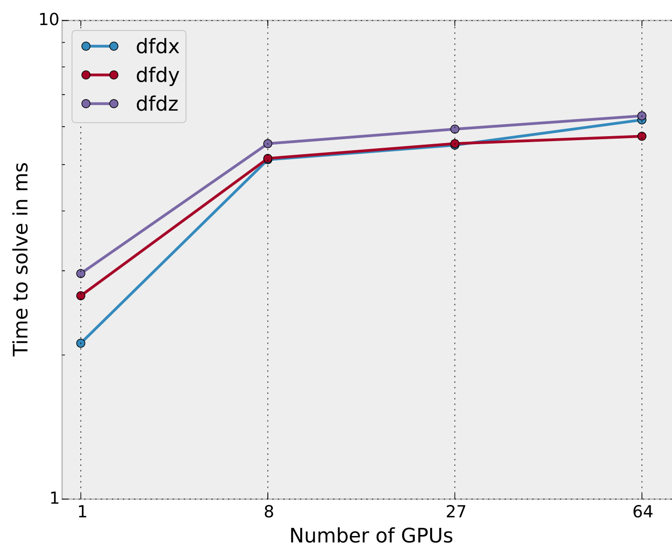

Figure 4.9: Weak scaling for multi-GPU compact finite difference, problem size: 128^3 per process.

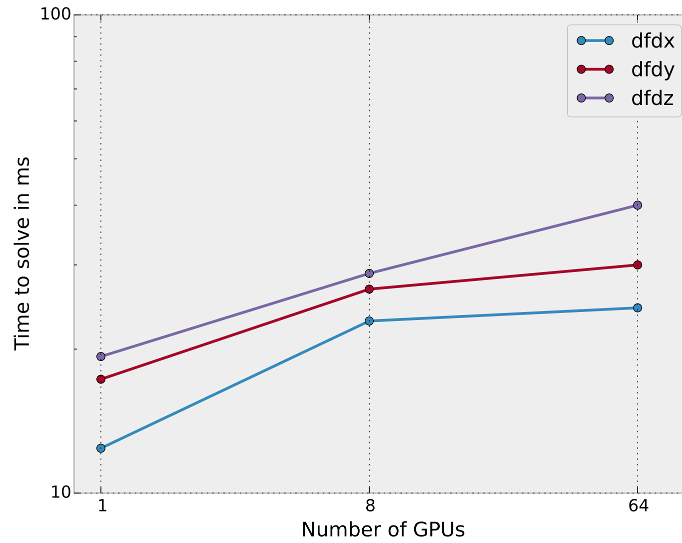

Figure 4.10: Weak scaling for multi-GPU compact finite difference, problem size: 256^3 per process.

Table 4.2: Time (ms) to compute derivatives in the fastest coordinate direction - comparison with reference implementation [19]

Size	Ref. impl, #CPU cores			NEATO-based, #GPUs		
	8	64	512	1	8	64
256^3	79.5	20.8	11.1	19.9	5.17	2.79
512^3	556.8	146.5	29.2	164.5	23.24	5.62
1024^3	5188	1092	223.7	-	174.9	24.49
2048^3	-	-	1741	-	-	297.07

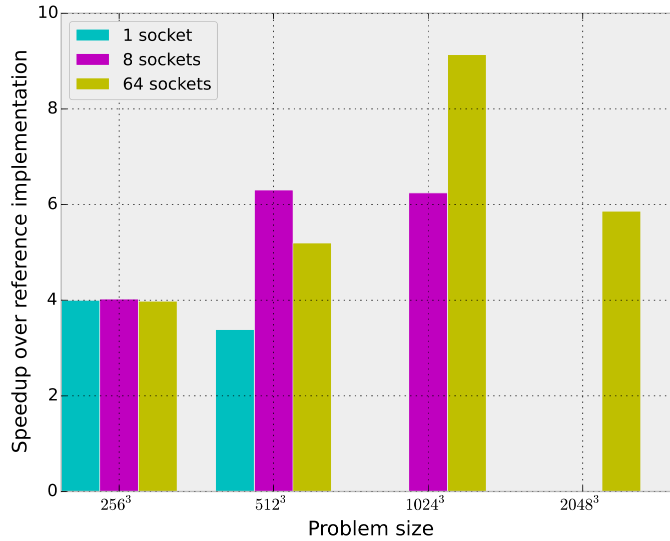

Figure 4.11: Speedups over reference implementation for computing derivative in the fastest coordinate direction

4.2.3 Comparison with a CPU-only approach

We also compare the performance of our compact finite difference solver with the approach described by Mohd-Yusof et al. [19], implemented for CPUs. The approach uses a distributed tridiagonal solver based on the LU decomposition specialized for tridiagonal systems. The problem is divided among individual CPU cores, communicating via MPI. For comparing timings, we use the number of CPU sockets as the basis. Each CPU socket uses 8 CPU cores and 1 GPU. Thus, we maintain a ratio of 1:8 between GPUs and CPU cores in our comparison. Table 4.2 shows timings for computing derivatives in the fastest coordinate direction for problems sized up to 2048^3, and Fig. 4.11 shows the respective speedup using our implementation.

Chapter 5

Conclusions and Future Work

We have presented an efficient approach for evaluating compact finite differences on GPU-accelerated clusters. At the core of our approach is a fast tridiagonal solver for the resulting linear systems. Here, we make use of the simple matrix structure to obtain better performance. The applicability of this strategy to the parallel cyclic reduction (PCR) algorithm, and to other hybrid approaches is yet to be studied.

The integration of this approach to our current DNS codes is yet to be performed. To ameliorate the cost of host-device transfers, it is likely that other portions of the code will also need to be ported to GPUs. However, most of the computational patterns followed in the rest of the code have been covered in the current work, and fewer challenges are likely to be faced in this process. Strategies for the evaluation of compact finite differences on other parallel architectures is of great interest, especially for the Intel Many Integrated Core (MIC) architectures.

Appendix

Here, we include the CUDA kernels for both the global memory and shared memory based implementations of the NEATO algorithm.

```
__global__ void forwardReductionKernel(const double *a_d,
                                const double *b_d,
                                const double *c_d,
                                double *d_d,
                                const double *k1_d,
                                const double *k2_d,
                                const double *b_first_d,
                                const double *k1_first_d,
                                const double *k1_last_d,
                                const int n,
                                int stride)
{
    int tix = threadIdx.x;
    int offset = blockIdx.x*n;
    int i;
    int j, k;
    int idx;
    double x_j, x_k;

    // forward reduction
    if (stride == n)
    {
        stride /= 2;
        j = log2((float)stride) - 1;
        k = log2((float)stride); // the last element
        x_j = (d_d[offset+stride-1]*b_d[k] - c_d[j]*d_d[offset+2*stride-1])/ \
                        (b_first_d[j]*b_d[k] - c_d[j]*a_d[k]);
```

```
28
29          x_k = (b_first_d[j]*d_d[offset+2*stride-1] - d_d[offset+stride-1]*a_d[k])/ \
30                          (b_first_d[j]*b_d[k] - c_d[j]*a_d[k]);
31          d_d[offset+stride-1] = x_j;
32          d_d[offset+2*stride-1] = x_k;
33      }
34      else
35      {
36          i = (stride-1) + tix*stride;
37          idx = log2((float)stride) - 1;
38          if (tix == 0)
39          {
40              d_d[offset+i] = d_d[offset+i] - \
41                  d_d[offset+i-stride/2]*k1_first_d[idx] - \
42                  d_d[offset+i+stride/2]*k2_d[idx];
43          }
44          else if (i == (n-1))
45          {
46              d_d[offset+i] = d_d[offset+i] - d_d[offset+i-stride/2]*k1_last_d[idx];
47          }
48          else
49          {
50              d_d[offset+i] = d_d[offset+i] - \
51                  d_d[offset+i-stride/2]*k1_d[idx] - \
52                  d_d[offset+i+stride/2]*k2_d[idx];
53          }
54      }
55  }
```

```
1   __global__ void backwardSubstitutionKernel(const double *a_d,
2                                    const double *b_d,
3                                    const double *c_d,
4                                    double *d_d,
5                                    const double *b_first_d,
6                                    const double b1,
7                                    const double c1,
8                                    const double ai,
9                                    const double bi,
10                                   const double ci,
11                                   const int n,
12                                   const int stride)
13
```

```
{
    int tix = threadIdx.x;
    int offset = blockIdx.x*n;
    int i;
    int idx;

    i = (stride/2-1) + tix*stride;

    if (stride == 2)
    {
        if (i == 0)
        {
            d_d[offset+i] = (d_d[offset+i] - c1*d_d[offset+i+1])/b1;
        }
        else
        {
            d_d[offset+i] = (d_d[offset+i] - (ai)*d_d[offset+i-1] - \
                (ci)*d_d[offset+i+1])/bi;
        }
    }
    else
    {
        // rint rounds to the nearest integer
        idx = rint(log2((double)stride)) - 2;
        if (tix == 0)
        {
            d_d[offset+i] = (d_d[offset+i] - \
                c_d[idx]*d_d[offset+i+stride/2])/b_first_d[idx];
        }
        else
        {
            d_d[offset+i] = (d_d[offset+i] - \
                a_d[idx]*d_d[offset+i-stride/2] - \
                c_d[idx]*d_d[offset+i+stride/2])/b_d[idx];
        }
    }
}
```

```
__global__ void shmemCyclicReductionKernel( double *a_d,
                                             double *b_d,
                                             double *c_d,
                                             double *d_d,
```

```
5                                  double *k1_d,
6                                  double *k2_d,
7                                  double *b_first_d,
8                                  double *k1_first_d,
9                                  double *k1_last_d,
10                                 const double b1,
11                                 const double c1,
12                                 const double ai,
13                                 const double bi,
14                                 const double ci)
15                                 {
16  /*
17
18  */
19  __shared__ double d_l[{{shared_size | int}}];
20
21  int tix = threadIdx.x;
22  int offset = blockIdx.x*{{n}};
23  int i, j, k;
24  int idx;
25  double d_j, d_k;
26
27  /* When loading to shared memory, perform the first
28     reduction step */
29  idx = 0;
30  if (tix == 0) {
31      d_l[tix] = d_d[offset+2*tix+1] - \
32                  d_d[offset+2*tix]*k1_first_d[idx] - \
33                  d_d[offset+2*tix+2]*k2_d[idx];
34  }
35  else if (tix == ({{(n/2) | int}}-1)) {
36      d_l[tix] = d_d[offset+2*tix+1] - \
37                  d_d[offset+2*tix]*k1_last_d[idx];
38  }
39  else {
40      d_l[tix] = d_d[offset+2*tix+1] - \
41                  d_d[offset+2*tix]*k1_d[idx] - \
42                  d_d[offset+2*tix+2]*k2_d[idx];
43  }
44  __syncthreads();
45
46  /* First step of reduction is complete and
```

```
47          the coefficients are in shared memory */
48
49      /* Do the remaining forward reduction steps: */
50      for (int stride=2; stride<{{(n/2) | int}}; stride=stride*2) {
51          idx = idx + 1;
52          i = (stride-1) + tix*stride;
53          if (tix < {{n}}/(2*stride)) {
54              if (tix == 0) {
55                  d_l[i] = d_l[i] - \
56                              d_l[i-stride/2]*k1_first_d[idx] - \
57                              d_l[i+stride/2]*k2_d[idx];
58              }
59              else if (i == ({{n}}/2-1)) {
60                  d_l[i] = d_l[i] - \
61                               d_l[i-stride/2]*k1_last_d[idx];
62              }
63              else {
64                  d_l[i] = d_l[i] - d_l[i-stride/2]*k1_d[idx] - \
65                              d_l[i+stride/2]*k2_d[idx];
66              }
67          }
68          __syncthreads();
69      }
70
71      if (tix == 0) {
72          j = rint(log2((float) {{(n/2) | int}})) - 1;
73          k = rint(log2((float) {{(n/2) | int}}));
74
75          d_j = (d_l[{{n}}/4-1]*b_d[k] - \
76                 c_d[j]*d_l[{{n}}/2-1])/ \
77              (b_first_d[j]*b_d[k] - c_d[j]*a_d[k]);
78
79          d_k = (b_first_d[j]*d_l[{{n}}/2-1] - \
80                 d_l[{{n}}/4-1]*a_d[k])/ \
81              (b_first_d[j]*b_d[k] - c_d[j]*a_d[k]);
82
83          d_l[{{n}}/4-1] = d_j;
84          d_l[{{n}}/2-1] = d_k;
85      }
86      __syncthreads();
87
88      idx = rint(log2((float) {{n}}))-2;
```

```
89      for (int stride={{n}}/4; stride>1; stride=stride/2) {
90          idx = idx - 1;
91          i = (stride/2-1) + tix*stride;
92          if (tix < {{n}}/(2*stride)){
93              if (tix == 0) {
94                  d_l[i] = (d_l[i] - c_d[idx]*d_l[i+stride/2])/\
95                              b_first_d[idx];
96              }
97              else {
98                  d_l[i] = (d_l[i] - a_d[idx]*d_l[i-stride/2] -\
99                              c_d[idx]*d_l[i+stride/2])/b_d[idx];
100             }
101         }
102         __syncthreads();
103     }
104
105     //When writing from shared memory, perform the last
106     //substitution step
107     if (tix == 0) {
108         d_d[offset+2*tix] = (d_d[offset+2*tix] - c1*d_l[tix])/b1;
109         d_d[offset+2*tix+1] = d_l[tix];
110     }
111     else {
112         d_d[offset+2*tix] = (d_d[offset+2*tix] - \
113                                 ai*d_l[tix-1] - ci*d_l[tix])/bi;
114         d_d[offset+2*tix+1] = d_l[tix];
115     }
116
117     __syncthreads();
118 }
```

Bibliography

[1] NVIDIA GPU accelerated libraries. https://developer.nvidia.com/gpu-accelerated-libraries.

[2] OpenACC Web page. http://www.openacc.org/.

[3] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2015.

[4] Dario Bini and Beatrice Meini. On cyclic reduction applied to a class of Toeplitz-like matrices arising in queueing problems. In WilliamJ. Stewart, editor, *Computations with Markov Chains*, pages 21–38. Springer US, 1995.

[5] Li-Wen Chang, John A Stratton, Hee-Seok Kim, and Wen-Mei W Hwu. A scalable, numerically stable, high-performance tridiagonal solver using GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 27. IEEE Computer Society Press, 2012.

[6] Li-Wen Chang and W Hwu Wen-mei. A guide for implementing tridiagonal solvers on GPUs. In *Numerical Computations with GPUs*, pages 29–44. Springer, 2014.

[7] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[8] Andrew Davidson and John D. Owens. Register packing for cyclic reduction: A case study. In *in Workshop on General Purpose Processing on Graphics Processing Units*, pages 1–6, 2011.

[9] Vahid Esfahanian, Behzad Baghapour, Mohammad Torabzadeh, and Hossain Chizari. An efficient GPU implementation of cyclic reduction solver for high-order compressible viscous flow simulations. *Computers & Fluids*, 92:160–171, 2014.

[10] Bengt Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of computation*, 51(184):699–706, 1988.

[11] Dominik Göddeke and Robert Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. *IEEE Transactions on Parallel and Distributed Systems (TPDS), Special Issue: High Performance Computing with Accelerators*, 22(1):22–32, January 2011.

[12] Christopher A Kennedy and Mark H Carpenter. Several new numerical methods for compressible shear-layer simulations. *Applied Numerical Mathematics*, 14(4):397–433, 1994.

[13] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[14] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[15] A.G. Kravchenko and P. Moin. On the effect of numerical errors in large eddy simulations of turbulent flows. *Journal of Computational Physics*, 131(2):310 – 322, 1997.

[16] Sanjiva K Lele. Compact finite difference schemes with spectral-like resolution. *Journal of Computational Physics*, 103(1):16–42, 1992.

[17] Nathan Mattor, Timothy J Williams, and Dennis W Hewett. Algorithm for solving tridiagonal matrix problems in parallel. *Parallel Computing*, 21(11):1769–1782, 1995.

[18] Paulius Micikevicius. 3d finite difference computation on GPUs using CUDA. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pages 79–84. ACM, 2009.

[19] J Mohd-Yusof, D Livescu, and T Kelley. *Adapting the CFDNS Compressible Navier-Stokes Solver to the Roadrunner Hybrid Supercomputer*. DEStech Publications, Inc, 2010.

[20] J. Nickolls and W.J. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56–69, March 2010.

[21] NVIDIA. NVIDIAs next generation CUDA compute architecture: Kepler GK110. 2011.

[22] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[23] D. W. Peaceman and Jr. Rachford, H. H. The numerical solution of parabolic and elliptic differential equations. *Journal of the Society for Industrial and Applied Mathematics*, 3(1):pp. 28–41, 1955.

[24] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2007.

[25] N. Sakharnykh. Efficient tridiagonal solvers for adi methods and fluid simulation. *NVIDIA GPU Technology Conference*, September 2012.

[26] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[27] Xian-He Sun. Application and accuracy of the parallel diagonal dominant algorithm. *Parallel Computing*, 21(8):1241–1267, 1995.

[28] Bulent Tutkun and Firat Oguz Edis. A GPU application for high-order compact finite difference scheme. *Computers & Fluids*, 55:29–35, 2012.

[29] Zhangping Wei, Byunghyun Jang, Yaoxin Zhang, and Yafei Jia. Parallelizing alternating direction implicit solver on GPUs. *Procedia Computer Science*, 18:389–398, 2013.

[30] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. *SIGPLAN Not.*, 45(5):127–136, January 2010.