UNIVERSITY OF CALIFORNIA,
IRVINE

Laminar Unsteady Navier-Stokes Flow on Multicore and GPU Architectures

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Engineering

by

Bahareh Mostafazadeh Davani

Dissertation Committee:
Assistant Professor Aparna Chandramowlishwaran, Chair
Professor Nader Bagherzadeh
Professor Feng Liu

2016

DEDICATION

To my Mom, who inspires me everyday.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to my committee chair, Professor Aparna Chandramowlishwaran, who has conveyed a spirit of adventure in regard to research and has been the most wonderful teacher in the past few years, both in and outside of the classroom. Without her guidance and persistent help, this dissertation would not have been possible.

I would like to thank my committee members, Professor Nader Bagherzadeh and Professor Feng Liu, for guiding and supporting me throughout these years.

I would also like to thank Ferran Marti for all his help and his patience at teaching.

ABSTRACT OF THE DISSERTATION

Laminar Unsteady Navier-Stokes Flow
on Multicore and GPU Architectures

By

Bahareh Mostafazadeh Davani

Master of Science in Computer Engineering

University of California, Irvine, 2016

Assistant Professor Aparna Chandramowlishwaran, Chair

Computational Fluid Dynamics (CFD) is a popular tool in engineering applications and an active research area in the field of Fluid Mechanics, yet today's state-of-the-art CFD solvers lack the ability to perform adequately on modern architectures. Design of an efficient CFD solver, that takes advantage of available computational resources on both CPU and GPU architectures, requires a comprehensive analysis of the solver while considering the different architectural aspects of these processing units. In this thesis, we present our efforts in designing an efficient and scalable implementation that solves the fluid motion of compressible viscous flow at transonic speeds. Stencil computation, which is the core computational pattern of these solvers, has been vastly studied and optimized for different architectures. However, these optimizations have mostly been focused on applications that primarily consist of a single stencil pattern. We tailor these optimizations to our specific application which involves multiple stencil patterns with different characteristics. Our optimizations also include unique numerical and application-specific improvements, while building up on well-known parallelization techniques for GPU and multicore CPU implementation.

Chapter 1

Introduction

Computational Fluid Dynamics (CFD) is a branch of fluid mechanics that uses numerical analysis and algorithms to solve and analyze problems that involve fluid flows. Computers are used to perform the calculations required to simulate the interaction of liquids and gases with surfaces defined by boundary conditions. These simulations are a widely popular tool, that reduces testing requirements while also providing new insight into important physical phenomena. Despite the advances in this field, these solvers are currently bound by computational resources. Grids which contain a large number of points (due to the high accuracy of the problem, or due to a large problem domain), or require a high temporal resolution to capture high-frequency motions have not been solved yet.

Today's supercomputing platforms are based on massively parallel processor designs to deliver petascale performance, yet current state-of-the-art solvers lack the ability to take full advantage of these computational resources. The design of numerical algorithms and software that scale on such platforms is critical for solving next-generation science and engineering problems and utilizing the computation power of new architectures effectively. This work aims for the design and implementation of a more efficient CFD solver, that maps well onto both multicore CPU and GPU architectures and capitalize on modern supercomputers. The different aspects of the CPU and the GPU architectures would require separate implementation and optimizations, based on an in-depth analysis of the solver's structure. In this chapter, we provide an overview of some architectural aspects of the GPU that we later refer to. Additionally, we briefly describe stencil computations, the core computational pattern of this solver. We also discuss previous work in both optimizing stencil computations and also efficient CFD solver implementations.

1.1 Stencil Computation

The core computational pattern in these solvers is stencils. Stencil computations sweep over an entire grid multiple times updating each grid point based on a fixed pattern, using its nearest neighbors. The main characteristic of stencil computation is that they are memory bound applications, based on the roofline model[23]. The roofline model uses arithmetic intensity to categorize different applications to provide performance estimates and present potential optimizations best suited for each category. Arithmetic intensity is presented as the ratio of the total number of floating point operations(flop) to total data movement(in bytes). While applications such as Dense Linear Algebra, have a high flop to byte ratio, stencil computations fall within the low-end of arithmetic intensity spectrum. This is mostly due to the fact that a single update of a cell, performs few floating point operations while most commonly, a considerable number of reads are required to access neighboring values. This highly impacts the performance and scalability of stencil computations, while also indicating the need for optimizations mostly aimed at increasing the flop to byte ratio.

1.2 Architecture Overview

In the past decade, GPUs have moved beyond their initial domain of Graphics Processing, and have been adapted into numerous different fields and shown astonishing improvements in terms of performance. With this growth in the use of GPUs and also with the rapid advances in their architecture and performance, it is necessary to adapt to their use. The high peak performance of GPUs could potentially be used in order to develop the next generation of CFD solvers. However, harvesting this computational power, presents a number of challenges. One of the main differences between the CPU and the GPU is that GPUs are specialized for compute-intensive, highly parallel computations mostly since this processing unit is composed of a large number of threads, at the cost of less cache allocation.
Considering these aspects, the main challenge of efficient GPU implementation is to have a large number of threads working independently and minimizing their need for synchronization.
We use Nvidia's CUDA parallel programming model to implement our GPU solver. In this model, the program would launch a sequence of kernels, where each kernel would spawn a user defined number of thread blocks with a given size. Each thread would thus be mapped by a unique thread ID within the thread block and a block ID that specifies which block it belongs to. We use these indexes to construct a one-to-one mapping of our GPU thread grid to our data grid.

Each thread block will be mapped onto a single multiprocessor at execution time and thus the threads within a thread block can synchronize and share data. The scheduler would map different thread blocks to different multiprocessors dynamically. These thread blocks are completely independent and might be scheduled concurrently or at different times, thus there's no guarantee of the order of the execution of thread blocks. Each block has a hard limit of up to 1024 threads, and these threads will be executed one warp at a time, where each warp consists of 32 threads.

1.3 Previous Work

Stencil computations are some of the well-studied parallel computing patterns [5, 4]. These computations, perform sweeps through an entire data structure that is much larger than the capacity of the data cache. As a result, stencil computations achieve a low fraction of theoretical peak performance, since data from main memory cannot be transferred fast enough to avoid stalling the computational units on modern processors. Reorganizing these computations to take full advantage of memory hierarchies has been the subject of much investigation. These studies have primarily focused on tiling optimizations [5, 4, 19, 18] to exploit locality by performing operations on cache-sized blocks of data. While most traditional tiling optimizations use domain decomposition to improve spatial locality, studies have also focused on exploiting the locality in the time dimension [6, 21, 15, 25]. An extension to single time step blocking is the time-skewing algorithm [21, 25] which blocks in both space and time while respecting stencil dependencies. This work on designing parallel algorithms and a computational software infrastructure to capture compressible flows differs from the preceding work in a couple of ways. First, we attempt to optimize the entire solver, not just a single stencil kernel. Typically, numerically accurate solvers that capture real physics consist of multiple terms, and therefore many different stencil calculations in each time step. This makes the problem extremely challenging. Second, our goal is to build a larger software infrastructure that can ultimately support numerous numerical schemes, turbulence models, and architectures to enable domain scientists to study complex phenomena in CFD by mixing and matching these methods.

While there have been efforts in mapping CFD solvers to modern multicore architectures, most of the work in this field has only applied a parallel implementation using either OpenMP or MPI, as opposed to our work where we focus on both efficient parallelization and stencil specific optimizations. The most popular choice when developing a parallel version of a CFD code is using MPI, mainly due to the fact that MPI is able to run in shared and distributed memory systems, and also that achieving a scalable OpenMP implementation requires some effort, as we mention in Section 3.1. However as indicated in [9], there might be some advantages in a hybrid approach where they obtained a speedup of 36$\times$ with 64 threads and 13$\times$ with 16 threads. [16] obtained a speedup of 8-11$\times$ with 16 threads using OpenMP. [20] showed a clear advantage of using a hybrid MPI-OpenMP versus single MPI when extending the number of cores beyond 500. The speedup with single MPI reached a flat value of 4.5 times the speedup of 120 cores, while the hybrid speedup kept increasing reaching values of 10 times the speedup of 120 cores when using 1300 cores.

The CFD community has traditionally run their codes on a single node or many-node homogeneous CPU architectures depending on the problem size. However, there have been some recent advancements using homogeneous GPU codes and even heterogeneous CPU-GPU systems showing excellent results. Lopez-Morales et al. [14] developed a multi-CPU-GPU code for explicit time-stepping, using high order methods and unstructured grids. Jude and Baeder [10] used a high order implicit scheme with structured grids. Multigrid methods are commonly used to accelerate convergence. At each coarser level of the multigrid cycle, the work drops by a factor of 8 and solutions are needed for GPUs since they start to become idle and the need to hide communication costs for a multi-GPU system gains importance [7].

Chapter 2

Formulation and Algorithms for Unsteady Flow

2.1 Mathematical Formulation

This section describes the Navier-Stokes equations and the numerical schemes that are used for simulating the complex realistic geometries of flows that form the basis for the high-fidelity CFD software infrastructure.

This work is based on the laminar implementation of an existing time accurate 3-D Navier-Stokes code. The code is called ParCAE [13, 26] and solves the Unsteady 3-D Reynolds Averaged Navier-Stokes (URANS) equations on structured grids using a cell centered finite-volume method. A time-accurate solution is obtained through a dual-time stepping scheme proposed by Jameson [1]. The set of 5 governing equations can be written as follows.

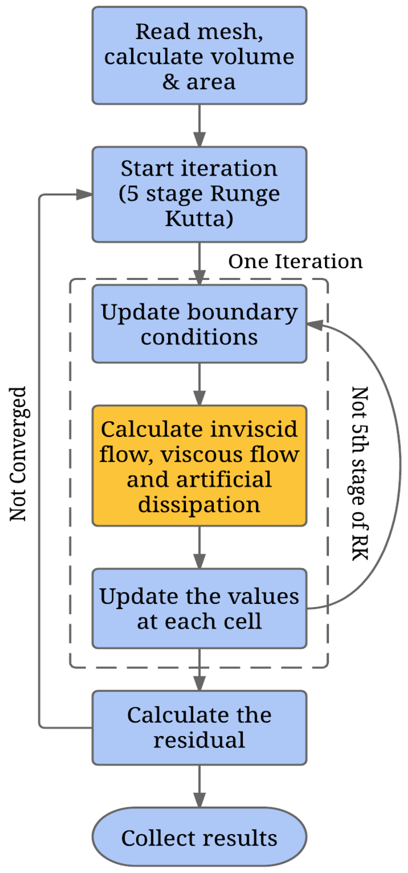
Read mesh, calculate volume & area
Start iteration (5 stage Runge Kutta)
One Iteration
Update boundary conditions
Calculate inviscid flow, viscous flow and artificial dissipation
Update the values at each cell
Not 5th stage of RK
Not Converged
Calculate the residual
Collect results

Figure 2.1: High-level overview of the URANS solver. The dashed box denotes one iteration where we solve for $\vec{W}_{i,j,k}$. The yellow box highlights the calculation of the fluxes which is the computational core of the solver.

$$\frac{d(\vec{W}\Omega)_{i,j,k}}{dt} = -\vec{R}_{i,j,k} \tag{2.1}$$

where $\vec{W}_{i,j,k}$ is the vector of conservative variables (conservation of mass, momentum in each direction, and energy) for cell (i, j, k), Ω is the cell volume, and $\vec{R}$ is the vector of residuals.

The time derivative is discretized by a backward-difference scheme of second-order accuracy,

$$\frac{3(\vec{W}\Omega)^{n+1}_{i,j,k} - 4(\vec{W}\Omega)^{n}_{i,j,k} + (\vec{W}\Omega)^{n-1}_{i,j,k}}{2\Delta t} = -\vec{R}_{i,j,k}(\vec{W}^{n+1}) \tag{2.2}$$

At each time step the problem is reformulated as the following steady-state problem in pseudo time t^*.

$$\frac{d(\vec{W}^*\Omega)^{n+1}_{i,j,k}}{dt^*} = -\vec{R}^*_{i,j,k}(\vec{W}^*) \tag{2.3}$$

where $\vec{W}^* = \vec{W}^{n+1}$ when the solution has converged, and

$$\vec{R}^*_{i,j,k}(\vec{W}^*) = \vec{R}_{i,j,k}(\vec{W}^*) + \frac{3(\vec{W}\Omega)^{*}_{i,j,k} - 4(\vec{W}\Omega)^{n}_{i,j,k} + (\vec{W}\Omega)^{n-1}_{i,j,k}}{2\Delta t}. \tag{2.4}$$

For each real time step, the set of equations are marched to a steady state in pseudo time. The pseudo time marching is implemented as a sequence of pseudo time steps, each of them discretized with a multi-stage Runge-Kutta scheme as shown in Figure 2.1. The solution at stage k is

$$\vec{W}^k_{i,j,k} = \vec{W}^0_{i,j,k} - \frac{\alpha_k \Delta t^*_{i,j,k}}{\Omega^*_{i,j,k}} \left[1 + \frac{3\alpha_k \Delta t^*_{i,j,k}}{2\Delta t}\right]^{-1} * \\ \left[\vec{R}_{i,j,k}(\vec{W}^{k-1}) + \frac{3(\vec{W}\Omega)^{0}_{i,j,k} - 4(\vec{W}\Omega)^{n}_{i,j,k} + (\vec{W}\Omega)^{n-1}_{i,j,k}}{2\Delta t}\right] \tag{2.5}$$

where α is the Runge-Kutta coefficient for the corresponding stage.

The spatial discretization of the residual is 2^{nd} order accurate and is composed of the sum of the face fluxes.

$$\vec{R}_{i,j,k}(\vec{W}) = \sum_{m=1}^{N_F} [(\vec{F}_c - \vec{F}_v)_m \vec{n}_m S_m]_{i,j,k} \tag{2.6}$$

with N_F equal to the number of cell faces (6 for a 3D hexahedral grid), S the face surface, n the normal vector to the face, and $\vec{F}_c$ and $\vec{F}_v$ are the convective and viscous fluxes accounting for inertial and viscous effects respectively.

Defining the conservative variables at a face (e.g. at the face between cell i and cell $i+1$) as

$$\vec{W}_{i+1/2,j,k} = \frac{1}{2}(W_{i,j,k} + W_{i+1,j,k}). \tag{2.7}$$

and avoiding the j, k notation, the contribution to the residual of the convective fluxes at face $i + 1/2$ is

$$[\vec{F}_c \vec{n} S]_{i+1/2} \approx \vec{F}_{inv}(\vec{W}_{i+1/2})(\vec{n}S)_{i+1/2} - \vec{D}_{i+1/2} \tag{2.8}$$

where the inviscid fluxes $\vec{F}_{inv}$ are discretized with 2nd order central difference, and to avoid numerical instabilities, local artificial dissipation is added with $\vec{D}$ following the classic JST scheme [8].

$$\vec{D}_{i+1/2} = \hat{\Lambda}^S_{i+1/2}\,[\epsilon^{(2)}_{i+1/2}(\vec{W}_{i+1} - \vec{W}_i) - \epsilon^{(4)}_{i+1/2}(\vec{W}_{i+2} - 3\vec{W}_{i+1} + 3\vec{W}_i - \vec{W}_{i-1})] \tag{2.9}$$

The contribution of the viscous fluxes is more challenging since it requires an auxiliary grid. The formulation in the viscous terms contains gradients of velocity. Using Green's theorem, and defining an auxiliary grid centered on the vertices of the original grid, the gradients of velocity at such vertices are defined as (e.g. for $\partial u/\partial x$)

$$\left(\frac{\partial u}{\partial x}\right) \approx \frac{1}{\Omega_{aux}} \sum_{m=1}^{N_F} (u\, n_{x_{aux}}\, S_{aux})_m \tag{2.10}$$

where the summation is on the faces of the auxiliary grid. After computing vertex velocity gradients using Equation 2.10, the face values of the viscous fluxes on the original grid are recovered by averaging the obtained vertex values.

2.2 Algorithm: Multi-stencil kernels

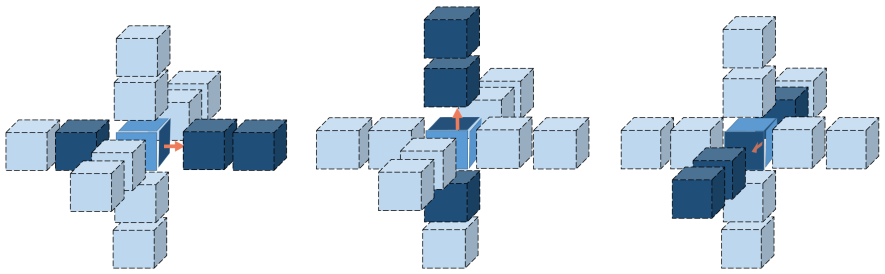

Figure 2.2: Memory access patterns in artificial dissipation calculations, where we read the $\vec{W}$ of three neighboring cells shown in dark blue to find the flux at the dark blue surface of the cell in the center.

In this section, we describe the main computational challenges in solving the above equations.

Variable	Description
$\vec{F}_{inv}$	Inviscid fluxes
$\vec{D}$	Fluxes of artificial dissipation
$\vec{F}_c$	Convective fluxes
$\vec{F}_v$	Viscous fluxes
$\vec{W}$	Conservative variables
$\vec{R}$	Residuals
Ω	Cell volume
S	Face surface
$\vec{n}$	Face normal vector
Δt	Real time step
Δt^*	Pseudo time step
α	Coefficient for Runge-Kutta scheme
$\hat{\Lambda}^S$	Spectral radii of convective flux Jacobian
ϵ	Artificial dissipation coefficients
u	Component of velocity vector in x direction

Table 2.1: Summary of solver parameters

The overall structure of the URANS solver is shown in Figure 2.1 where, we solve for the updated values of $\vec{W}_{i,j,k}$ in Equation 2.5 at each stage (dashed box). The calculation of the fluxes (yellow box) is the computational core and most time consuming phase of the solver. It consists mainly of three flux calculations namely, viscous flux, inviscid flux, and artificial dissipation.

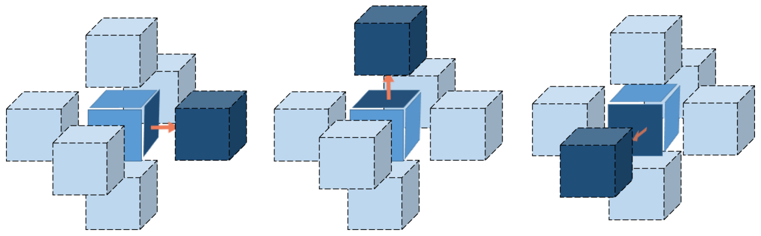

Figure 2.3: Memory access patterns in inviscid flux calculations. To compute the flux at the dark blue surface of the cell in the center, $\vec{W}$ of the dark blue neighbor is read.

Even though all the flux calculations have a stencil pattern, we claim that (1) real solvers consist of not just a single stencil kernel but multiple stencils, (2) the stencils themselves have different memory access patterns depending on their categorization, and (3) the variables described in Section 2.1 (summarized in Table 2.1), all need to be stored for each cell and accessed for each flux calculation making this problem extremely challenging.

To further elaborate on the above observation, we divide the stencil patterns in flux calculations into two categories namely, cell-centered and vertex-centered. Artificial dissipation ($\vec{D}$) and inviscid fluxes ($\vec{F}_{inv}$) are in the former category whereas viscous fluxes ($\vec{F}_v$) are vertex-centered.

Cell-centered stencils. For calculating the artificial dissipation, we use Equation 2.9, which is a blend of second order and fourth order differences. The fourth order term at each surface uses $\vec{W}$ of the neighboring cells as shown in Figure 2.2, where cell i is shown with a different color in the center. The artificial dissipation at the surface shown in dark blue is calculated by reading the $\vec{W}$ of the dark blue neighbors to find the second term in Equation 2.9. Considering the three other fluxes not shown in the figure, the calculation of this term for cell i consists of a 13 point stencil. Similarly, the calculation of the second order term of artificial dissipation, and also the inviscid flux, where we read the value of one neighboring cell to compute the flux at each surface, use a 7 point stencil as shown in Figure 2.3.

Vertex-centered stencils. As described in Section 2.1, we calculate the velocity gradients at the vertices of each cell in order to compute the viscous flux. This calculation is illustrated in Figure 2.4, where the colored cubes are the cells in our original grid and the dashed cubes represent the cells of the auxiliary grid. The red point is the cell center of the auxiliary grid and the vertex of the original grid. We first calculate the velocity gradients at the vertices using Equation 2.10, then we use these values to calculate the flux at each surface. The complete calculation of the viscous flux can be described as a two stage stencil. First, we use eight 8 point stencils to calculate the gradients at the vertices, and finally use another 8 point stencil to find the sum of all the fluxes.

To clarify the reason for classifying these stencils into two categories, it is worth mentioning what differentiates them. Aside from the fact that these stencils use different values of the neighboring cells based on the equation, they have two major differences. First, considering the memory access of the two categories, vertex-centered stencils have a more complex memory access pattern than cell-centered stencils. While the latter was previously discussed, the calculation of gradients at each vertex requires reading at least four new cell values where each cell itself includes $\vec{W}$, *pressure, volume* and multiple values of *surfaces* for different faces, thereby, more expensive and less cache-efficient than the cell-centered stencils. Second, the viscous flux calculation requires two separate traversals over the entire grid, first to find the gradients and a second pass to compute the flux whereas the cell-centered stencils require only one pass over the grid. The two passes in the vertex-centered stencil can be merged into one but it results in redundant computation and exacerbates the memory access complexity because it requires reading 27 cells for calculating the flux at one cell.

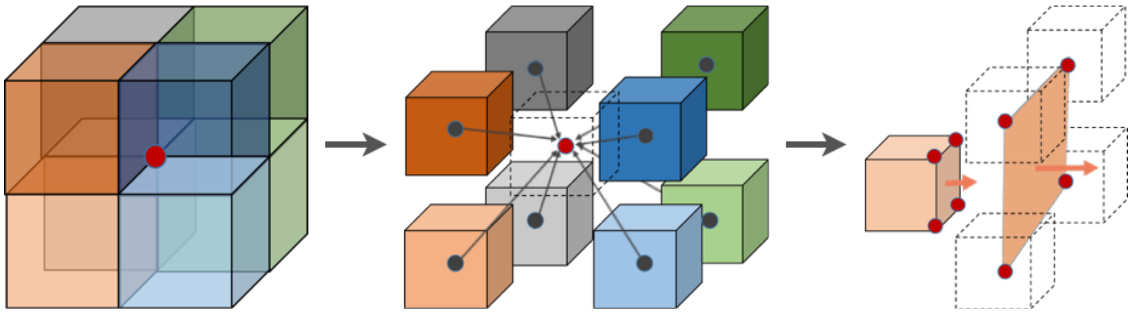

Figure 2.4: Memory access and computational patterns in viscous flux calculation. The colored cells are read to compute the velocity gradient at the center of the auxiliary grid, shown as red dots. These gradients are then used to compute the flux at the surface for the original grid.

These different stencil patterns and their memory accesses, result in a high bandwidth-to-compute ratio, which limits the scalability of URANS solver and adds to the challenges of optimization.

Chapter 3

Multi-core Implementation

Our code is based on an existing time accurate 3-D Navier-Stokes solver, which is written in Fortran[13]. As a first step, we implemented the laminar unsteady flow in C++. The overall structure of the code is as shown in Figure 2.1, where at each iteration, we run 5 stages of Runge-Kutta. Each stage mainly consists of the artificial dissipation, inviscid, and viscous flux calculations followed by updating the values of $\vec{W}$ and *Pressure* for each cell of the grid, and finally updating the boundary conditions.

Since the fluxes are calculated at each surface and then summed to reach the cell-centered value, it is redundant to compute the flux at all six surfaces of each cell. Instead, for cell (i, j, k), we only calculate the three outgoing fluxes as shown in Figures 2.2 and 2.3. The incoming flux from the other three surfaces are calculated at cells $i-1$, $j-1$, and $k-1$. So, our solver traverses the grid one dimension at a time, calculating the outgoing flux at each cell. This scheme of traversal not only reduces redundant computation but also exploits spatial locality. For viscous flux calculation, we first traverse the entire grid to find the gradients and then calculate the flux at each surface. We iterate through these stages until the residual drops below a certain threshold, at which point we obtain a converged solution.

3.1 Optimizations

We implement a variety of optimizations to improve the performance of our baseline solver, which we discuss in detail in this section. Most of the optimizations are focused on improving the memory bandwidth utilization because this solver like any other stencil based computation is highly memory bound.

3.1.1 Single Core Optimizations

As a first step, we apply some well-known single core optimizations to improve the performance of the sequential solver. These optimizations are detailed as follows.

Loop Interchange

We rearrange the loops in order to have unit stride access within the innermost loop, which improves locality and could potentially assist the compiler with auto-vectorization.

Strength Reduction

Math operations such as exponentiation was initially one of the most time consuming bottlenecks of every simulation. We replaced these expensive operations with cheaper alternatives. This includes replacing exponentiation with multiplication and addition. Division for both floating points and integers were replaced with multiplication. While this results in a slight loss of accuracy (less than 1%), it considerably improves the execution time.

Variable	Description	Size
$\vec{F}_{inv}$	Inviscid fluxes	Grid size * 5^1
$\vec{D}$	Fluxes of artificial dissipation	Grid size * 5^1
$\vec{F}_v$	Viscous fluxes	Grid size * 5^1
$\vec{W}$	Conservative variables	Grid size * 5^1
Ω	Cell volume	Grid size
S	Face surface	Grid size * 6^2
Δt^*	Pseudo time step	Grid size

Table 3.1: Size of each parameter

Cache Blocking

As shown in previous work [12, 5], cache blocking is an important optimization especially in stencil computation kernels. This is even more critical for our solver because we need to store multiple values for each cell unlike simple stencil kernels. These variables are summarized in Table 3.1.

To efficiently utilize the different levels of the cache hierarchy, we decompose the entire grid into multiple blocks. This decomposition is illustrated in Figure 3.1. We first divide the grid into blocks of size $LL\_X \times LL\_Y$ and then each one of these blocks is subdivided into chunks of size $L1\_X \times L1\_Y$. These values are then tuned in such a way that the initial block fits into the last level of cache, and the second level could be stored entirely in the L1 cache. This size depends not only on the size of the cache, but also on the floating point precision we are using.

This optimization is designed in such way as to map to any cache-based architecture. Not only can we tune the size of the blocks for each architecture, we can also choose to only implement one level of blocking. This could be useful for cases where a small grid is being simulated on a system with a large enough last level cache that could store the entire grid.

[1] Multiplication by 5 is because of the five conservative variables

[2] Multiplication by 6 is because the surface values are stored for faces in each of the three dimensions, and each face consists of components in three directions

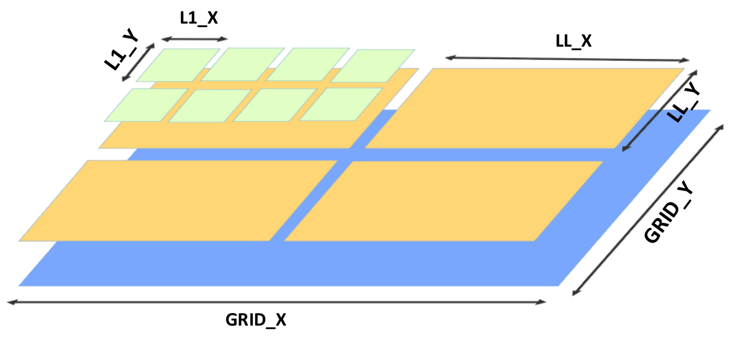

Figure 3.1: The domain is decomposed into two levels to utilize L1 (green blocks) and last level cache (orange blocks).

Vectorization

The width of vector registers has increased from 128 to 256 bits in modern processors with AVX support and 512 bits in AVX-512 instruction set. This trend of increasing the size of register file is expected to continue in future architectures, thus the potential gain of a fully vectorized code is not negligible. Assuming a 256-bit wide register, a fully vectorized implementation is potentially 4× to 8× faster for double and single precision floating points respectively. While we do not expect significant performance improvements from vectorization for a memory bound application, it is still important to have a vectorized code in order to take full advantage of emerging hardware.

To achieve better code portability and performance improvement for different architectures with different vector register size, we preferred obtaining a compiler generated vectorized code to manual vectorization using intrinsics. Since the compiler had initially failed to auto-vectorize the code for the most part, we implemented a number of optimizations and transformations that helped auto-vectorization and achieved a fully vectorized code.

We briefly discuss some of these modifications in this section. It is worth noting that due to better initial performance in auto-vectorization (compared to the GNU compiler), we use the Intel compiler, and its vectorization guide [2], while applying these optimizations.

(I) Loop Unrolling. Loop unrolling is a simple transformation, and the compiler is able to apply this optimization to most loops. However, in the case of nested loops, compiler only vectorizes the inner most loop. This is not efficient in cases where the inner loops has very low iteration counts. In such cases, we manually unroll the small loops, in order to force the compiler to vectorize the outer loop.

(II) Loop Unswitching. In this modification, we try to avoid using **if/else** statements within the body of the loops. Even though the compiler is able to vectorize, if such statements can be implemented as masked assignments [2], the performance gain would be considerably higher than without these statements. In order to remove the **if or else** clauses, we either move them outside the loop and duplicate the loop's body, or use the C++ conditional operators where possible.

(III) Loop Fission. This transformation is mostly useful when the vectorization fails due to dependencies. For instance, in cases where a value is updated, used, and then updated again, the compiler detects a **Write After Write** dependency (Intel compiler calls this an Output dependency) and fails to vectorize. The code shown in Loop 1 is an example of such a case.

Loop 3.1: Sample loop with dependencies

```
for(int i = 0; i = max_i; i++){
A[i] = SomeValue;
B[i] = /*Some calculation on A[i];*/
A[i] = UpdatedValue;
}
```

Breaking this loop into two separate loops as shown in Loop 2, allows the compiler to vectorize both loops since there are no dependencies within a loop.

Loop 3.2: Loop Fission example

```
for(int i = 0; i = max_i; i++){
A[i] = SomeValue;
}
for(int i = 0; i = max_i; i++){
B[i] = /*Some calculation on A[i];*/
A[i] = UpdatedValue;
}
```

This pattern occurs in viscous flux computations, where we calculate and then use values such as average velocity gradients. While this adds the overhead of a new loop, the cost is negligible compared to the performance gain from vectorization.

(IV) Pointer Declaration. One issue that arises with the use of pointers in C++ is that the compiler by default assumes that two distinct pointers could be pointing to the same location in memory. This assumption forestalls auto vectorization where there is a write to an array using pointers. This is because the compiler assumes dependencies between this write and any reads from any pointers. Using the *__restrict__* keyword in C++ for pointer declaration would specifically indicate that the target of that pointer will not be accessed through any other pointers. Even though using this keyword would require extra care while working with pointers, it is essential for ensuring auto vectorization.

3.1.2 Multicore Optimizations

In this section we discuss how we design our parallel algorithm and the optimizations we implement to achieve scalability.

Parallelization

We use OpenMP to implement the parallel solver. While it's mostly straightforward to have a parallel code using OpenMP pragmas, achieving a scalable implementation requires much more effort. In this implementation we divide the grid into $TX \times TY$ blocks of equal size, where TX is the number of blocks in the X dimension and TY is the number of blocks in the Y dimension and each of these blocks is assigned to one thread. Since all threads are working on blocks with equal size, this implementation is load balanced. Another benefit of this decomposition is the locality that is exposed to each threads work, since each cell needs the value of its neighboring cells for a single stencil calculation.

It should also be noted that this decomposition could count as a third level of domain decomposition, as illustrated in Figure 3.2. More specifically, we could first divide the grid into blocks of size $LL\_X \times LL\_Y$, then each block is divided into $TX \times TY$ blocks and lastly, each thread would work sequentially work on blocks of size $L1\_X \times L1\_Y$. However, we could choose to disable the first level of blocking, depending on the architecture and the grid size.

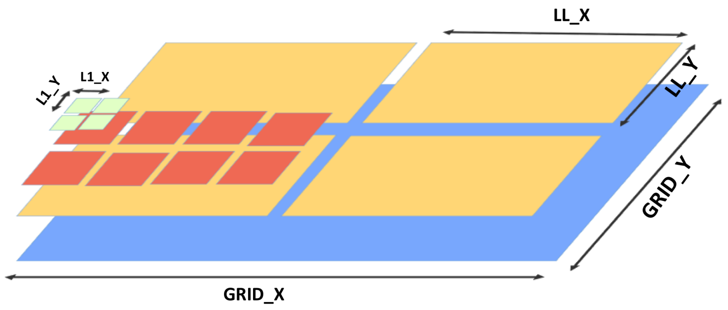

Figure 3.2: Three levels of decomposition applied to a grid. First level is to optimize accesses to the last level cache, second level is decomposition into thread blocks and the third level is optimizing L1 cache accesses.

We also implement the following optimizations, to achieve better scalability, specially for larger number of threads:

NUMA Optimizations

In Non-uniform memory access(NUMA) design, different processors will have different memory access time depending on whether the processor is accessing its local memory or not. Neglecting the effects of NUMA, while using systems with multiple NUMA nodes, would result in considerable performance degradation. Thus optimizing for efficient use of these nodes is rather crucial, and becomes even more so with larger computing clusters and supercomputers. Our NUMA optimization presumes a first-touch policy, thus we need to manage how we initialize the data. We use either of the following techniques to do so, and we choose the technique best suiting each application or architecture.

We could either assume a static thread to block mapping. Assuming ThreadID x is mapped to a fixed block, we could have the exact same domain decomposition for both the initialization and the computation section of the code. Thus each thread would initialize the section of data it would later calculate on, therefore it would be initialized on its corresponding node's memory.

Another approach we could take for diminishing NUMA effects, is to first launch C threads, where C is equal to the number of NUMA nodes we will be using in the program, and have each of these threads initialize a chunk of the domain on one node, and then launch $\frac{(X \times Y)}{C}$ threads to do the computation on each node. This mapping could easily be done using OpenMP thread affinity management run-time routines[3].

False Sharing Avoidance

In symmetric multiprocessor (SMP) systems, different processors each have a local cache that allows for faster access to commonly used data. The memory system guarantees cache coherency, which could be designed in many different ways. One common way to achieve

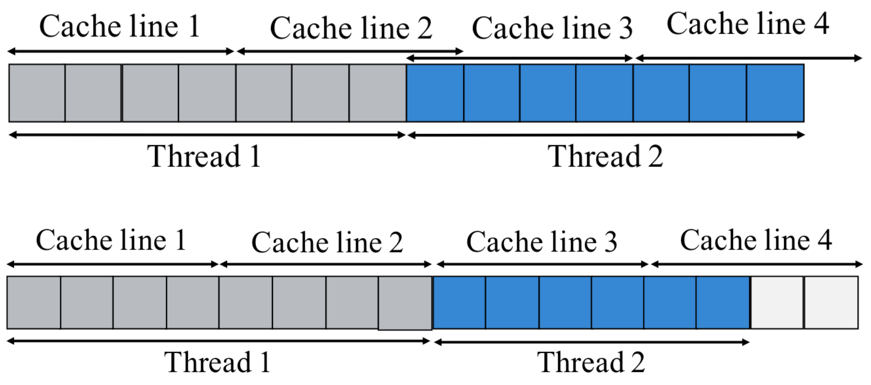

Figure 3.3: Top) This array illustrates the cache lines accessed by each thread before padding, where a write to line 2 would result in an invalidation of cache line 3 for Thread 2. Bottom) In the padded array, threads will access separate cache lines.

cache coherency is that when one processor writes to a value in its own cache, that same value is invalidated in any other cache. Thus any other processor would have to read from main memory again, in order to have the correct data. This leads to false sharing while using multiple processors in parallel. If processors modify different variables that reside in the same cache line, it will lead to a non-necessary invalidation of that cache line on all processors.

Avoiding false sharing among multiple threads is necessary in order to achieve scalability, especially since it could potentially increase as the number of threads grows.

One way to decrease the effects of false sharing, is to minimize shared data structure. In order to do so, we have modified the structure of the code, so that instead of storing the different fluxes(e.g. viscous flux, inviscid flux,etc.) for the entire grid, we store them for each block separately. Thus each thread would only access its own block while doing the calculations, and writes only the final values to a shared data structure. While this optimization improves the performance, it comes with a price; Which is the redundant calculation of the fluxes at the boundary of each block. While these fluxes have the same value for two blocks (outgoing flux for one block is the same as the incoming flux for another block at the same surface), they are calculated separately at each block.

There are some structures that are shared among all threads, such as all the components of W. The reason these data structures could not be separated is that since these values are updated after each stage and used in the next, we would have the added cost of communication among different blocks in order to maintain the correct values at the halo region. In order to avoid false sharing when writing to such data structures, we have implemented array padding. If each block writes to a chunk of the data that is a multiple of the cache line, there would be no overlapping of the data stored in cache for different threads, eliminating the problem of false sharing. While some initial grid sizes would have this property when divided into thread blocks, some might not; In which case, we pad the arrays to a multiple of the cache line size for each block.

Figure 3.3 illustrates how this could improve the performance. In the non-padded array, if Thread 1 writes to an element in cache line 2, Thread 2 would have to fetch an entire cache line(Cache line 3) from the next level in memory hierarchy, since the copy in its local cache wil be invalidated by cache coherence protocols.

Chapter 4

GPU Implementation

Production CFD codes operate at only 3–5% of the machine's peak performance. This is due to two main reasons – (a) most of the software was written to minimize computation, not communication, and (b) they were designed for homogeneous machines. The computing trend is relying on massive heterogeneous parallelism at lower clock speeds favoring problems with higher computational intensity (i.e.high flop:byte ratio). This is bad news for stencil patterns which exhibit low computational intensity and are severely memory bound.

Our goal is to transform the memory limited problem into a compute bound problem by

	DP	SP
Machine Peak (GPU)	4.96	14.89
Theoretical Peak	3.68	6.65
Baseline CPU	0.24	0.43
Baseline GPU	0.42	0.87
Spatial blocking	0.77	1.26
Fused (5-stage RK)	1.71	3.14
Fused (3-stage RK)	1.73	3.26
Temporal blocking	NA	3.58

Table 4.1: GPU optimizations and its computational intensity denoted as the ratio of flop to byte. DP and SP stand for double precision and single precision respectively.

leveraging both numerical and architectural optimizations. To observe the improvement with each optimization, we present the computational intensity (flop to byte ratio) in Table 4.1. Machine peak ratio refers to the ridge point in the roofline model [23] – the fraction required to hit the compute ceiling on that system. The ratios reported for the CPU baseline code and the theoretical peak for this solver are back-of-the-envelope estimates that roughly illustrate the improvements. The theoretical peak performance is calculated by assuming infinite size for shared memory, thus the grid is read once from the global to shared memory and written back once, while performing the same number of floating points as our current solver. We used Nvidia's profiling tool, **nvprof** to obtain the number of floating point operations and global memory transactions for each GPU optimization. The number of bytes read and written from global memory are calculated from the global memory transactions.

4.1 Naïve Implementation

To implement the GPU code for our solver, we start with a baseline implementation where the calculation of each cell of the grid is assigned to one thread that reads and writes all the values from and to the global memory. The flux calculations on GPUs have one notable difference compared to CPUs. As discussed in Chapter 3, the fluxes are not calculated at all the surfaces for one cell, rather each flux is calculated once and used for two cells as shown in Figure2.3 and Figure 2.2. While this is advantageous on the CPU, the significant cost of synchronization among threads makes this optimization impact negatively on the GPU. Hence, we calculate all the fluxes for every cell to trade-off computation for communication.

4.2 Spatial Blocking

Blocking is a well-known technique for exploiting locality in memory-bound applications on both CPU (using caches) and GPU (using local store) platforms. However, unlike caches, shared memory is a scarce resource on GPUs, and thus a bottleneck as we will discuss in further detail in Chapter 6.

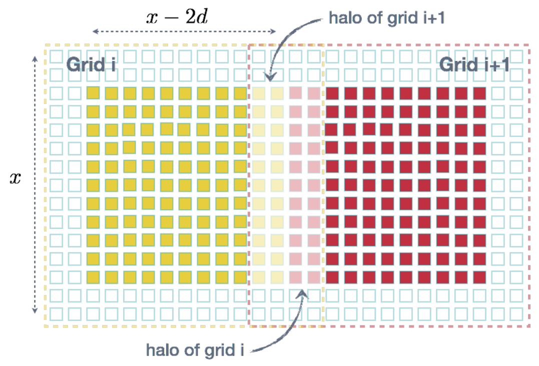

Figure 4.1: Two local grids of size $x \times x$, each with a halo region of size 2 in each dimension, d. Grid i (dashed yellow box) is updated by thread block i where each thread updates one yellow cell. Similarly, grid $i+1$ (dashed red box) is updated by thread block $i+1$.

To solve the equations in Section 2.1, each block requires a halo of two cells in all directions as shown in Figure 4.1. In this example, the two local grids, i and $i+1$ are assigned to one thread block each. Thread block i is responsible for updating the yellow cells and $i+1$ is responsible for updating the red cells. As illustrated in Figure 4.1, to update a block of $x - 2d \times y - 2d$, each thread block has to read a block size of $x \times y$ into shared memory.

The main disadvantage of this scheme is the cost of synchronization among thread blocks. The calculation of $\vec{W}$ at stage k is dependent on the updated values of $\vec{W}$ at stage $k-1$ (cf. Equation 2.5), Therefore, each thread block has to read the updated values of the halo region, which is calculated by its neighboring thread blocks. Figure 4.1 illustrates this issue, where at each stage, the yellow thread block reads the values of red cells that fall within the yellow dashed box (the memory block of the yellow thread block), and thus has to synchronize with the red thread block. Since thread blocks cannot communicate with each other, this results in a global memory synchronization at the end of each stage making this computation both severely storage and memory-bound even with spatial blocking.

Spatial blocking eliminates the global memory accesses needed to read each cell's values and write the updated results back, while performing the same number of floating point operations. This results in an increase in the computational intensity from 0.42 to 0.77 in double precision and 0.87 to 1.26 in single precision. However, the flop to byte ratio is still less than or close to 1 (as is the case for most stencil applications [23]).

4.3 Fused Time Steps

The synchronization after each stage of Runge-Kutta is one of the main reasons for the low arithmetic intensity even with spatial blocking. The next optimization is specifically aimed at reducing this overhead. Here, we fuse all 5 stages of Runge-Kutta and only synchronize once per time step. Eliminating the global synchronization among thread blocks reduces the accesses to global memory and significantly increases the flop to byte ratio. At the same time, the halo regions are now not updated with the correct values after each stage and hence, what we trade off to achieve this improved performance is the correctness of the result at each time step. At each stage, we introduce error to an outer region of size two in each direction (halo region). However, our solver is capable of damping out this error by performing more number of iterations and eventually converges to the final solution.

Table 4.1 shows that this optimization increases the computational intensity to 1.71 and 3.26 for double and single precision respectively. This is a big step towards our goal in making this solver more compute and less memory bound.

4.4 Numerical Optimization

The third optimization is yet another trade-off. The idea here is to change the time marching scheme from a 5-stage to a 3-stage Runge-Kutta. The main intuition behind this optimization is that merging 3 stages will introduce less error than 5 stages. As discussed in Section 4.3, two additional cells in each dimension will not have the correct result after each stage since their neighboring cells have not been updated. Therefore, reducing the number of stages will slow down the propagation of error.

However, changing the numerical scheme comes at the cost of a lower CFL number which results in a further increase in the number of iterations to converge. But, each time step would potentially take significantly less time due to fewer number of stages. Most engineering applications including the case studies discussed in this paper do not require a high CFL number. Therefore, we believe this to be a reasonable numerical optimization.

Comparing the fused 3 stage scheme to the fused 5 stage scheme, we only see a slight improvement in the arithmetic intensity as shown in Table 4.1. This is because of two main reasons – (1) the 3-stage scheme has fewer overall floating point operations, and (2) it has almost the same number of reads and writes from and to the global memory except for variables such as the surfaces of each cell which are read from the global memory at each stage. Therefore, a 3 stage scheme results in fewer number of such accesses. Note that the ultimate goal is to not only increase the computational intensity of this CFD solver but also reduce the overall runtime. Even though this optimization does not contribute significantly to the former goal, reducing the propagation of error results in faster convergence which results in a lower overall runtime.

4.5 Temporal Blocking

Since the computation of each cell in stencil computations is dependent only on its neighbors, there is limited opportunities for reuse using only spatial blocking. Therefore, to alleviate the communication and synchronization bottlenecks and still obtain the correct result at each time step, we augment spatial blocking with temporal blocking. In this technique, we compute multiple time steps simultaneously. Specifically, we load the same block of cells into shared memory, but perform redundant computation at the cost of avoiding synchronization at the end of each stage to obtain the right result. Figure 4.2 demonstrates this algorithm, where at each stage we update a smaller block. More precisely, all the colored cells will have the correct result at the end of stage 1. In stage 2, the red cells become the halo region and hence are not updated and so on. As shown in Figure 4.2, the size of the updated block is reduced by 4 in each dimension at every successive stage. For instance, if we start with a thread block of size $x \times y$, the final updated block will be of size $x-6d \times y-6d$ after 3 stages, and the rest of the colored cells are updated by another thread block. The intuition as to why temporal blocking across stages and time-steps would pay off is due to the increased cost of communication compared to computation especially on GPUs.

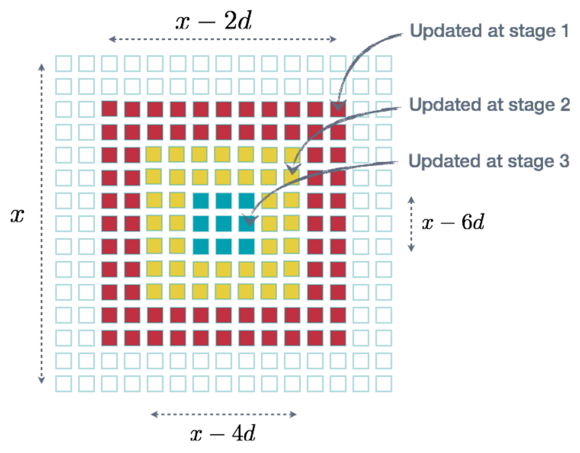

Figure 4.2: Temporal blocking illustrated. White cells are the halo for the first stage, red cells are updated at the first stage and read at the next two stages. Yellow cells are updated for the first two stages and finally, the blue cells are the only cells updated at the end of the third stage.

Comparing this scheme against the previous merged 3 stage Runge-Kutta, we see a slight improvement in the flop to byte ratio. It is evident that this scheme has more floating point operations due to redundant computations. But, the number of memory accesses for temporal blocking has also increased. This is because of the fact that the cells that are being redundantly calculated at one thread block are also read by other thread blocks. Thus, we have more number of total reads from global memory.

Chapter 5

Case Studies

In this chpater, we describe the experiments for steady and unsteady laminar flow to test the correctness of the solver and implementation.

5.1 Channel flow – steady

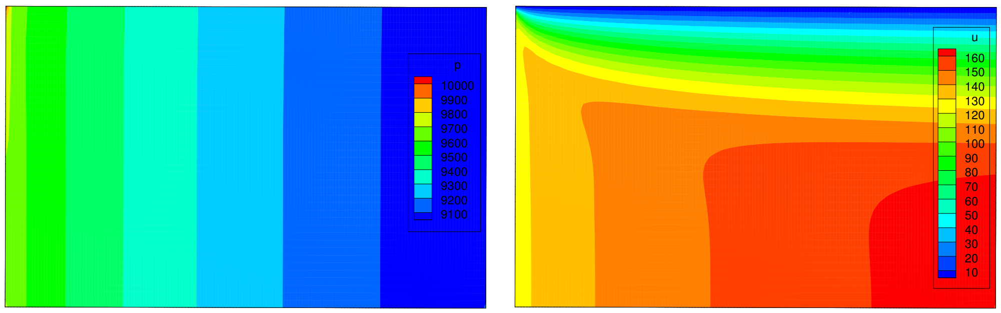

Figure 5.1: (a) Left: Pressure contour of channel flow. (b) Right: Velocity contour of channel flow.

To test our steady laminar flow implementation, we use a 2D channel flow, with a grid size of 150×40. Even though this is a compressible code, we use a small Mach number which ensures a solution similar to the incompressible limit. The incompressible channel flow has an analytical solution for the fully developed laminar state. A parabolic velocity profile across the channel, and a linear pressure drop stream-wise should be obtained. Assuming constant density, the pressure drops linearly along the channel to balance the wall friction. Some non-ideal effects are expected at the entrance of the channel due to the boundary conditions not satisfying the analytical assumptions. A close-to-parabolic profile however should be retrieved farther downstream. The results of this simulation are shown in Figures 5.1.

5.2 Step flow – steady

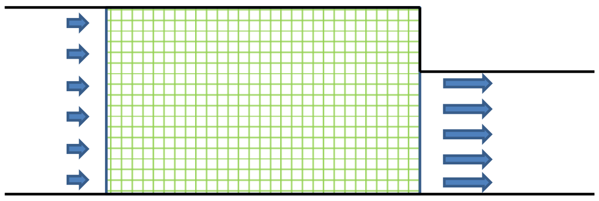

Figure 5.2: Illustration of step flow. The black solid lines are the physical domain, and the green mesh is the computational domain.

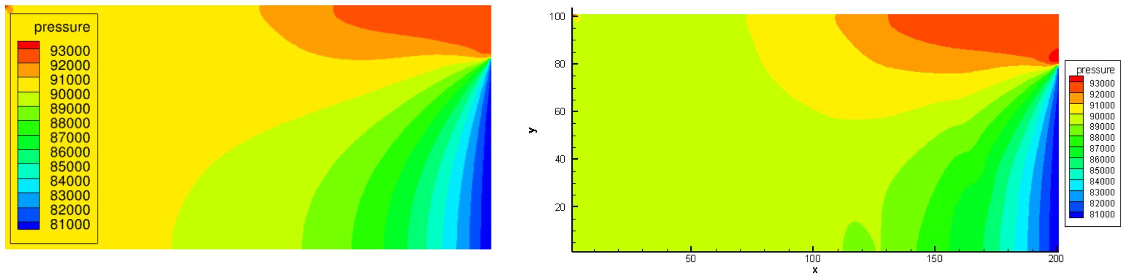

Figure 5.3: (a)Left, Pressure contour of steady step flow (b)Right, Pressure contour of unsteady step flow.

The step flow problem simulates a gas through a pipe of constant diameter, which suddenly encounters a reduction in its section as illustrated in Figure 5.2. This classical solution for laminar flows predicts a circulation bubble at the foot of the step whose length changes with the Reynolds number.

To save computational resources, a symmetric boundary condition is used for $y = 0$ where the centerline of the pipe would be. At $x = 0$, we use an inlet boundary condition which does not change in time, and we also use a steady outlet boundary condition for $x = x_{\max}$. The remaining boundaries are walls which use the non-slip boundary condition. A small inlet Mach number of 0.2 is chosen for this problem to avoid sonic flow at the outlet. The results of this simulation are shown in Figures 5.3.

5.3 Step flow – unsteady

To force an unsteady solution, the outlet boundary condition at $x = x_{\max}$ has an oscillating pressure value ($p_{\text{exit}}(t) = A * \sin(\omega t) + B$). The rest of the boundaries remain the same.

Since the maximum value of the exit pressure is always smaller than the inlet pressure, we prevent the flow from reversing its direction. The length of the circulation bubble oscillates at the exciting frequency of the exit pressure.

To accelerate the convergence of the solution to the periodic expected result, first we run a steady state step flow with non-oscillating pressure at the outlet, and we use that flow field to initialize the unsteady step flow problem. The simulation of the pressure contour is shown in Figure 5.3.

5.4 Cylinder – steady

The cylinder is an external flow problem, different from the previous test cases which are internal flows. Therefore, far field boundary conditions need to be implemented for the outer boundaries at j_{max}. The mesh wraps around the cylinder with a wall boundary condition for j_{min}, and connecting the cells at i_{min} with the cells at i_{max} on the same plane.

A simulation with Reynolds number of 50 and Mach number of 0.2 generates the steady solution shown in Figure 5.4.

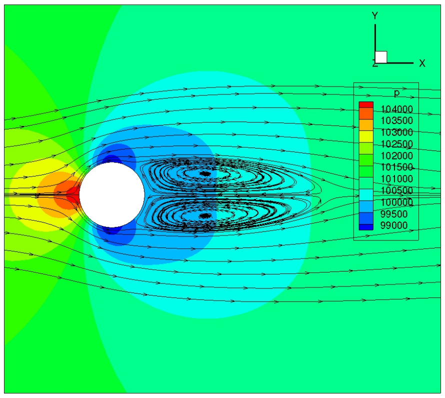

Figure 5.4: Streamlines and pressure contours for steady cylinder. Reynolds 50, Mach 0.2. Two symmetric circulation bubbles are formed behind the cylinder.

5.5 Cylinder – unsteady

By increasing the Reynolds number, the physics become unsteady without forcing any unsteady boundary condition. Laminar vortex shedding is captured for Reynolds number of 250. Figure 5.5 shows the streamlines and vorticity contours.

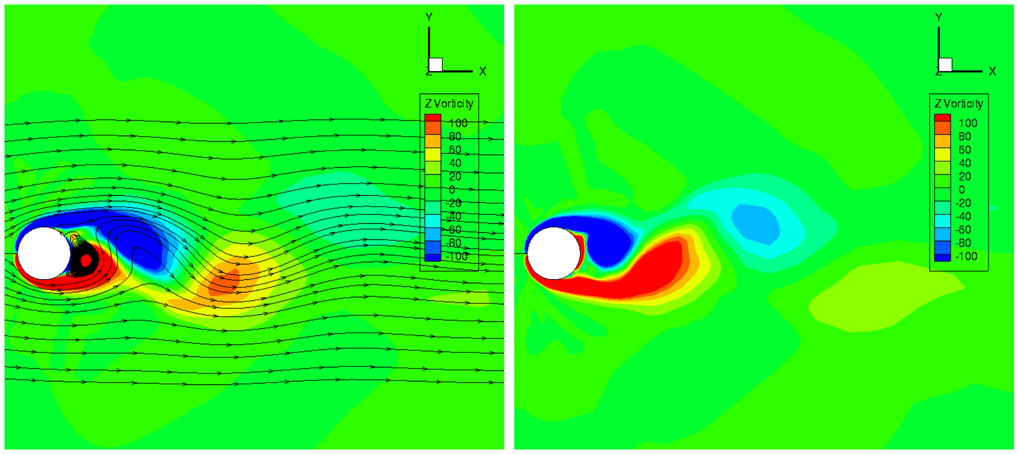

Figure 5.5: (a) Snapshot of streamlines and vorticity contour while shedding vortices, (b) Vorticity contours.

Chapter 6

Results and Discussion

We evaluate our implementations on two different systems with a dual-socket Intel Xeon processors. The key parameters of these system appears in Table 6.1.

	System1	System2
Architecture	Intel XE5-2630 v3 (Haswell-EP)	Intel XE5-2680 v3 (Haswell-EP)
Frequency	2.4 GHz	2.5 GHz
Sockets	2	2
Cores/Socket	8	12
Threads/Core	2	1
GFlop/s (DP, SP)	614.4, 1228.8	960, 1920
L1/L2/L3 cache	32/256/20480† KB	32/256/30720† KB
DRAM Bandwidth	59 GB/s	68 GB/s
Compiler	icpc 16.0.2	icpc 15.0.2

Table 6.1: Architectural Parameters. †shared among cores on a socket.

6.1 Multicore Implementation

We divide the performance analysis into two sections. First we discuss how each optimization improves the overall performance, and secondly we address the scalability of our parallel implementation.

6.1.1 Optimizations

To show the improvement of each optimization, we illustrate the GFlops achieved for different number of threads for a step flow simulation on a grid with 500000 cells. Figure 6.3 illustrates the performance on System 1. On this platform, the initial implementation had achieved 2.18 GFlops for single precision and 1.28 GFlops for double precision and our single threaded optimized implementation shows more than 5× improvement compared to the baseline, as a result of our single-core optimizations. The parallel implementation with 16 cores, achieves 38 and 74 GFlops respectively for double and single precision floating points. Since there are 8 cores per socket on this system, we need to consider NUMA for executions with more than 8 threads, as shown in Figure 6.3.

Figure 6.2 shows the same metric for System 2, which is briefly described in Table 6.1. Using all 24 cores of this system we reach 59 and 103 GFlops respectively for double and single precision with our fully optimized and vectorized implementation. On this system, the affects of NUMA optimizations are visible for executions with more than 12 threads, at which point the cores on both sockets will be in use.

It is also worth noting that both Figures 6.3 and 6.2, illustrate how the effects of false sharing increase with more number of threads and the false sharing avoidance optimizations thus show more improvement.

Both these systems have a Haswell architecture and support the AVX2.0 instruction set extension, which includes vector registers of 256 bits. As discussed in Section 3.1.1, full vectiorization has the potential of speeding up an execution 4 and 8 times for double and single precision floating points. For this application, the excessive number of memory access causes stalls and thus prevents a full vectorization and the perfect speedup. However, by applying the optimizations detailed in Section 3.1.1, we gain nearly 3× speedup for single precision and 2× for double precision floating points.

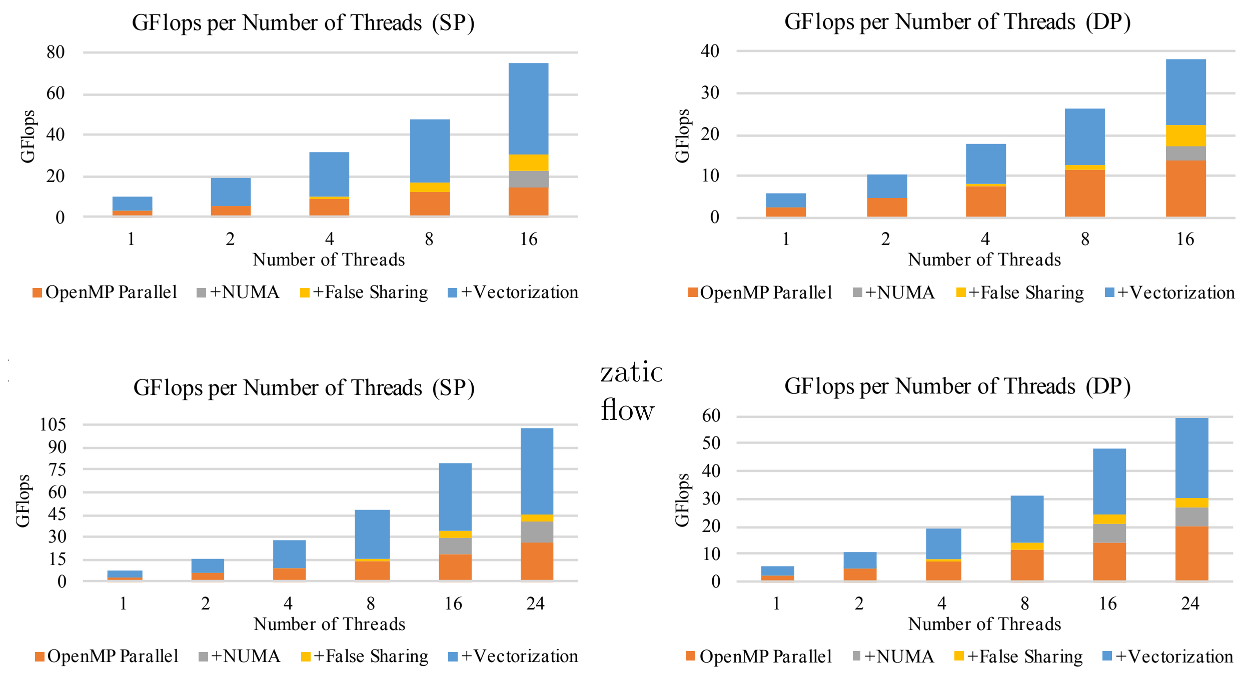

Figure 6.2: GFlops improvement per optimization for different number of threads on System 2. These results were obtained from a step flow simulation on a grid with 50000 cells.

6.1.2 Scalability

To showcase how our parallel implementation scales, we run both the step flow and the cylinder flow until the residual has dropped four orders of magnitude and the simulation has converged to a final solution.

These results were obtained on a grid with 50000(1000x500) cells for the step flow, and a grid with 10400 (130x80) cells for the cylinder flow. Tables 6.2 and 6.3 show the speedup we gain by increasing the number of threads for the fully optimized solver on System 1 and System 2 respectively.

The step flow simulation achieves nearly linear scalability for the step flow simulation. However, the cylinder flow scales slightly worse. This is mainly due to the fact that for this

Number of Threads:	2	4	8	16
Step, Single Precision	1.83	3.14	5.21	8.49
Step, Double Precision	1.84	3.26	5.53	8.85
Cylinder, Single Precision	1.84	3.38	5.83	8.25
Cylinder, Double Precision	1.89	3.39	5.85	8.37

Table 6.2: Speedup achieved for different number of threads on System 1, for a step flow simulation on a grid with 50000 cells and cylinder flow simulation on a grid with 10400 cells.

Number of Threads:	2	4	8	16	24
Step, Single Precision	1.85	3.26	5.68	9.43	12.19
Step, Double Precision	1.92	3.41	5.79	10.19	12.9
Cylinder, Single Precision	1.75	3.02	5.83	8.1	10.11
Cylinder, Double Precision	1.86	3.11	5.86	8.77	10.13

Table 6.3: Speedup achieved for different number of threads on System 2, for a step flow simulation on a grid with 50000 cells and cylinder flow simulation on a gid with 10400 cells.

case, we read the entire grid and the values of the surfaces for each cell from a pre-generated file, as opposed to the step flow simulation where generate the grid by specifying the values of the surfaces. These values are currently read sequentially from a grid file, and thus the initialization is not NUMA aware, which leads to off-chip memory accesses for the surface data structures.

6.2 GPU Implementation

We evaluate our GPU implementation on an NVIDIA K40 GPU, which has the peak single precision performance of 4.3 TFlop/s and the peak bandwidth is approximately 288 GB/s from global memory. Both of these performance values are distributed across 15 symmetric streaming multiprocessors. The local store or shared memory size on the Tesla is 48 KB per thread block. This greatly dictates the size of the thread block for the URANS solver. Table 6.4 summarizes the architectural parameters of this GPU.

Architecture	NVIDIA GK110B (Tesla K40)
Core Clock	745 MHz
CUDA Cores	2880
GFlop/s (DP, SP)	1430 , 4290
Memory Clock	3.0Ghz
Local Store	48 KB
DRAM Bandwidth	288 GB/s
Compiler	nvcc 7.5

Table 6.4: Architectural Parameters of Tesla K40 GPU.

Impact of optimizations

To show the performance impact of each optimization, we present results for the first three case studies discussed in Chapter 5. The baseline is our C++ code, and the naïve GPU code is as discussed in Chapter 4. The first optimization is spatial blocking, where we block the grid to fit in shared memory. However, due to shared memory limitations of 48 KB per thread block in Tesla K40, we only store $\vec{W}$ and *pressure* at each cell. Other variables such as surfaces and volumes are read from the global memory. Storing only these values, we can use thread blocks of size 32×32 for single precision and blocks of 32×24 for double precision floating point in our simulations. Given these thread block size restrictions, temporal blocking with the 5 stage Runge-Kutta scheme results in a final block of size 16×16. This results in lower performance due to the excessive amount of redundant calculations and read memory traffic. Thus, we only implement temporal blocking with a 3 stage Runge-Kutta scheme where we have a final block of 24×24 for single precision.

Figure 6.3 shows the speedup of all the optimizations compared to the baseline CPU code for the steady flow case studies. The first two groups of bars shows the performance of a channel flow simulation, with grid size of 160×40 and CFL number 1.5. For single precision floating point, we achieve $3.5\times$ speedup with the naïve implementation and $4.92\times$ speedup with spatial blocking. Merging the 5 stages together increases the speedup to $9.56\times$ compared to the baseline code. The fused 3 stage scheme achieves $15.98\times$ speedup and outperforms

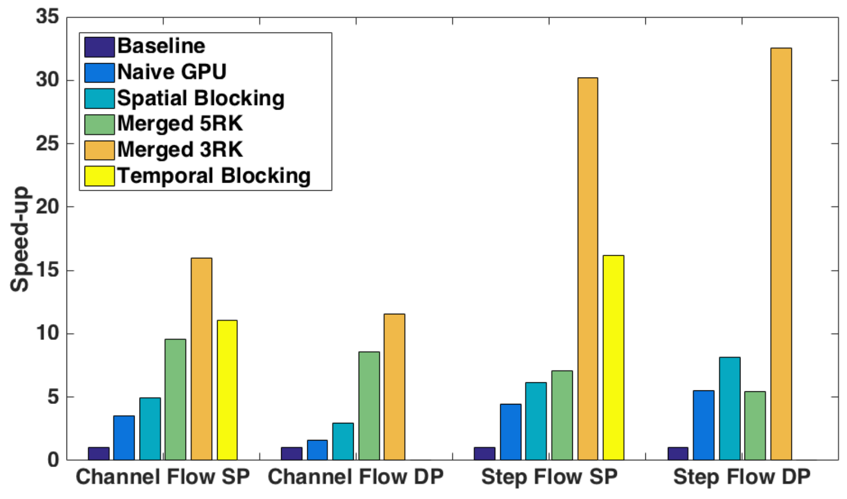

Figure 6.3: Performance results for different optimizations where each bar shows the speedup achieved compared to the baseline C++ code.

temporal blocking with the same number of stages, even though we perform more iterations to converge. This is because temporal blocking has a few disadvantages – it requires more thread blocks, redundant computation, and reads more cells since some parts of the grid are read by multiple thread blocks to simulate the same grid. In the end, it updates a smaller final block size. Another disadvantage of the temporal blocking scheme is the thread divergence in each block since only certain threads write the results at each stage. Temporal blocking shows 11.03× speedup against the baseline.

The second two groups of bars in Figure 6.3 show the performance of the optimizations for steady step flow case study with a grid of size 200×100 and CFL number 1.5. The naive implementation and spatial blocking show 4.46× and 6.18× speedup respectively for single precision and 5.49× and 8.13× for double precision. The merged 5 stage scheme exhibits poorer performance achieving 7.0× speedup compared to the baseline for single precision. This is due to the fact that in this scheme, there is more error being introduced and it takes longer for our solver to damp out this error. This problem is exacerbated for the double precision simulation, where the original block size is 32×24 and therefore at the end of the fifth stage, only a block of size 24×8 will have the correct results and thus this scheme achieves only $5.43\times$ speedup. The 3 stage merged scheme shows $30.23\times$ and $32.53\times$ speedup for single and double precision respectively, which is due to both the numerical optimization which results in taking fewer iterations to converge, and the reduced memory access optimizations that speeds up each iteration. Temporal blocking scheme has all the advantages of the merged 3 stage, yet looses performance due to the disadvantages discussed for the channel flow and thus this scheme only results in $16.15\times$ speedup against the baseline for single precision.

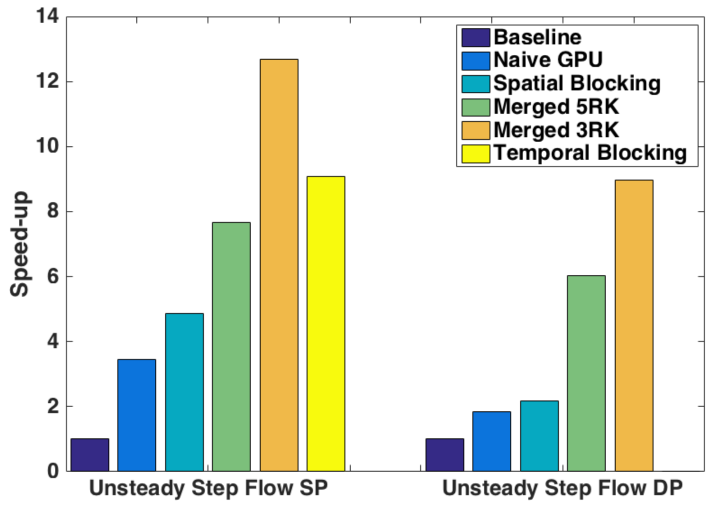

Figure 6.4: Performance results for different optimizations for unsteady step simulation where each bar shows the speedup achieved compared to the baseline C++ code.

Figure 6.4 illustrates the performance of our optimizations for the unsteady step case. Here, we read the results from a converged steady step flow solution, then execute 50 time steps with 500 iterations at each time step to reach the solution discussed in Section 5. In this case, since all the schemes are initialized with a converged steady solution, they all perform the same number of iterations to find the unsteady solution. Thus, the performance gain is only due to the improvement of time per iteration.

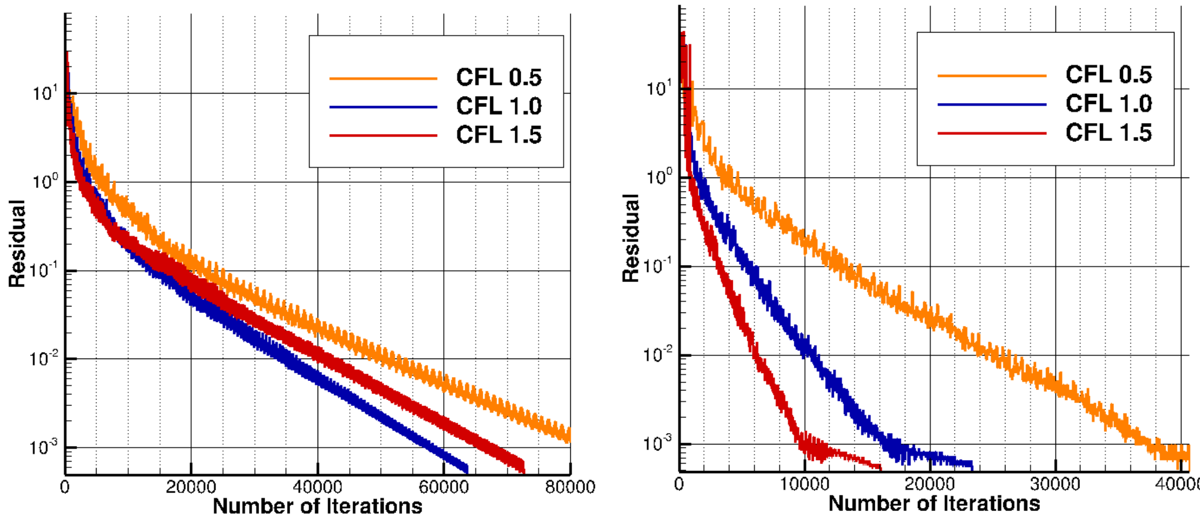

Figure 6.5: The residual drop plotted against the number of iterations. Figure on the left shows the residual drop for the 5 stage merged scheme and the right shows the drop for the 3 stage merged scheme. The 5 stage scheme take more time steps to damp out the error introduced by merging the stages together.

Trade-off between performance and correctness

As discussed in Chapter 4 , with the 5 stage and 3 stage merged schemes, we introduce some error to the solution in order to reduce the cost of communication within neighboring thread blocks. To further illustrate how this error has affected the the performance, Figure 6.5 represents the residual drop for the merged 5 stage and merged 3 stage Runge-Kutta, versus the number of iterations for different CFL numbers. Ideally, by increasing the CFL number, we take larger steps and thus converge to the final solution with fewer number of iteration. However as we can see in Figure 6.5, 5 stage merged scheme executes considerably more number of iterations to drop to the same residual compared to the 3 stage scheme.

5 stage scheme works for channel with roughly the same number of iterations as the CPU code but for step flow which is more complex we see performance degradation, same could happen with 3 stage going to more complex simulations without the shared memory limitations temporal blocking would be the best of both worlds.

Chapter 7

Conclusion

We introduced our CFD solver and the different optimizations we applied to best map this application onto both CPU and GPUs. These processing units have different characteristic and thus each one requires a distinctive set of optimizations and implementation details. We discussed parallelization techniques for multicore systems and how we have improved the performance with each optimization, using floating point operations per second(flops) as a metric.

GPU architectures are best utilized by compute-bound applications that fully saturate the GPU's immense processing powers. Thus we aimed to increase the arithmetic intensity of our solver with each optimization. Our optimized CPU implementation scales nearly linearly with increasing number of cores and achieves more than 50× speedup compared to a baseline implementation on 16 cores, while our GPU solver shows 30× speedup compared to the baseline. Despite our best efforts in efficiently mapping this solver on the GPU, our parallel CPU implementation out performs the GPU solver. This is mainly due to the high cost of synchronization on the GPU and given the fact that this solver and the numerical scheme requires multiple synchronizations at different levels of execution, in order to obtain correct results.

While some of the optimizations discussed in this work have been previously applied to stencil computations [5, 4, 17, 6], this is one of the first efforts in addressing the challenges of an entire solver which consists of multiple stencils. We discuss the different patterns of these stencils and the challenges they give rise to, along with an in-depth analysis of an important real-world application. Our GPU implementation integrates stencil computation optimizations [17, 11, 22, 24] into this solver while tuning different aspects of the numerical scheme to best match the architectural characteristics of GPU in order to obtain the best performance results.

Our efforts in design and implementation of a CFD solver that is optimized specifically for both CPU and GPU architectures to achieve their full potential would allow for faster simulations compared to current state of the art solvers which mostly lack the wide range of optimization we have applied. This work also lays the ground work for optimized solvers, that achieve a high percentage of machine peak performance on massively parallel architectures that would enable solving next-generation science and engineering problems.

Bibliography

[1] Time-dependent calculations using multigrid with application to unsteady flows past airfoils and wings, author = Jameson, A., journal = AIAA 91-1596, year = 1991, owner = ferran, timestamp = 2016.01.28.

[2] Mark Sabahi. A Guide to Auto-vectorization with Intel C++ Compilers. `https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers.`, 2012.

[3] OpenMP 4.5 API C/C++ Syntax Reference Guide, 2015.

[4] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, Nov 2008.

[6] M. Frigo and V. Strumpen. Cache Oblivious Stencil Computations. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[7] D. Jacobsen and I. Senocak. A Full-Depth Amalgamated Parallel 3D Geometric Multigrid Solver for GPU Clusters. In *49th AIAA Aerospace Sciences Meeting*, 2011.

[8] A. Jameson, W. Schmidt, and E. Turkel. Numerical Solution of the Euler Equations by Finite Volume Methods Using Runge-Kutta Time-Stepping Schemes. *AIAA Paper 81-1259*, 1981.

[9] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Comput.*, 37(9):562–575, 2011.

[10] D. Jude and J. Baeder. Extending a Three-Dimensional GPU RANS Solver for Unsteady Grid Motion and Free-Wake Coupling. In *AIAA SciTech, 54th Aerospace Sciences Meeting*, 2016.

[11] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[12] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60. ACM, 2006.

[13] F. Liu and X. Zheng. A Strongly-Coupled Time-Marching Method for Solving the Navier-Stokes and $k-\omega$ Turbulence Model Equations with Multigrid. *Journal of Computational Physics*, 128(2):289–300, 1996.

[14] M. Lopez-Morales, J. Bull, J. Crabill, T. Economon, D. Manosalvas, J. Romero, A. Sheshadri, J. W. II, D. Williams, F. Palacios, and A. Jameson. Verification and Validation of HiFiLES: a High-Order LES unstructured solver on multi-GPU platforms. *AIAA 32nd AIAA Applied Aerodynamics Conference*, 2014.

[15] J. Mccalpin and D. Wonnacott. Time Skewing: A Value-Based Approach to Optimizing for Memory Locality. Technical report, In http://www.haverford.edu/cmsc/davew/cache-opt/cache-opt.html, 1999.

[16] M. S. Mueller, B. M. Chapman, B. R. de Supinski, A. D. Malony, and M. Voss. *OpenMP shared memory parallel programming.* Springer, 2008.

[17] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–13, Nov 2010.

[18] G. Rivera and C.-W. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[19] S. Sellappa, , and S. Chatterjee. Cache-Efficient Multigrid Algorithms. *International Journal of High Performance Computing Applications*, 18:2004, 2001.

[20] C. Simmendinger, J. Jagerskupper, and R. Machado. A PGAS-based implementation for the unstructured CFD solver TAU. *In: 5th Conference on Partitioned Global Address Space Programing Models, Tremont House, Galveston Island*, 2011.

[21] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. *SIGPLAN Not.*, 34(5):215–228, May 1999.

[22] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 579–586, July 2009.

[23] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[24] M. Wittmann, G. Hager, and G. Wellein. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–7, April 2010.

[25] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 171–180, 2000.

[26] N. P. L. F. Xiong, J. and D. Papamoschou. Computation of High-Speed Coaxial Jets with Fan Flow Deflection. *AIAA Journal*, 48(10):2249–2262, 2010.