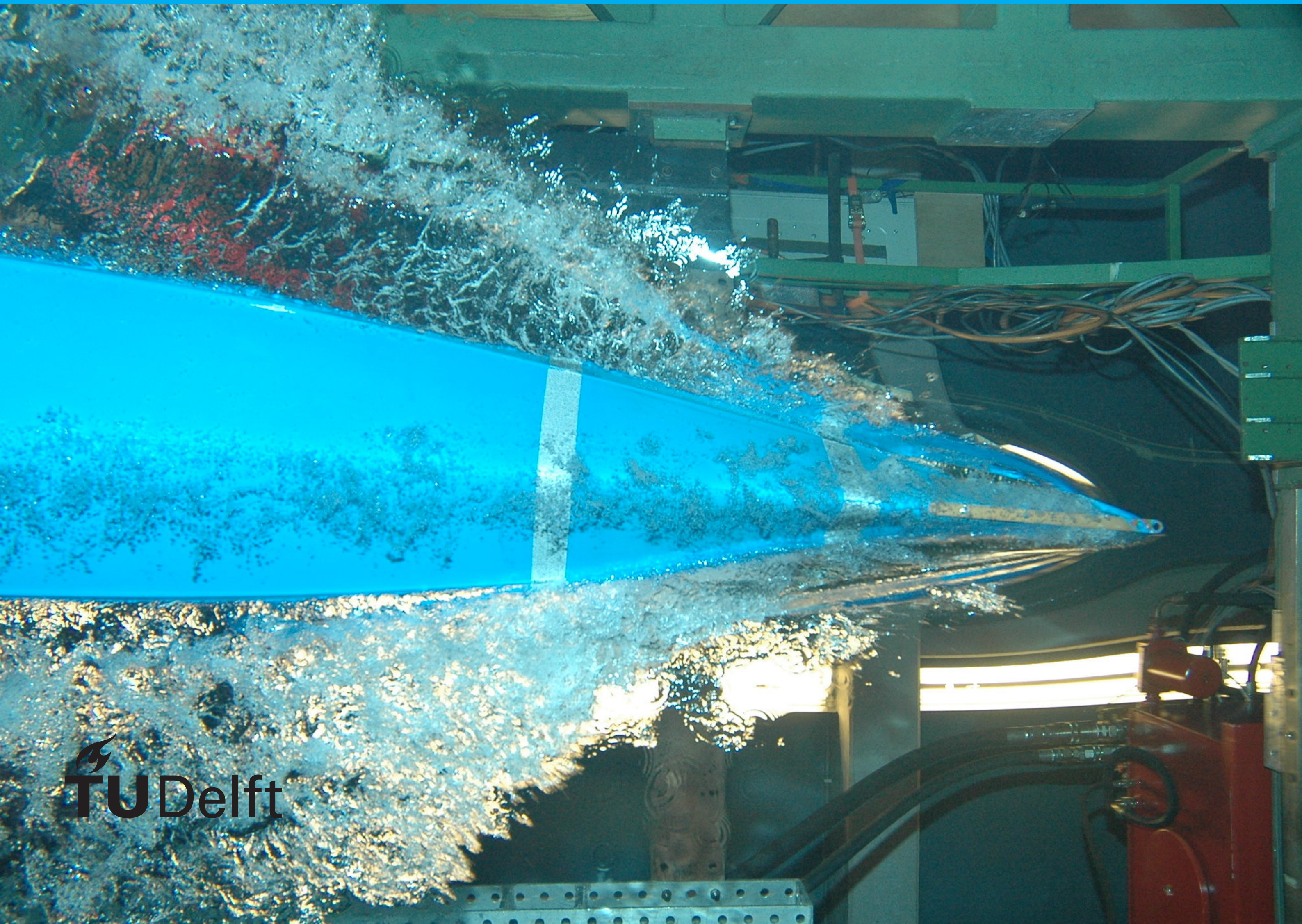
Implementing Finite Difference Schemes on Graphic Processing Units

Philip Lippmann

ME55035 MSc Thesis



TU Delft

IMPLEMENTING FINITE DIFFERENCE SCHEMES ON GRAPHIC PROCESSING UNITS

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of

Master of Science

by

Philip Lippmann

January 2022

This thesis is also available electronically at http://repository.tudelft.nl/.

The work in this thesis was made in the:

Delft University of Technology

Fluid Dynamics of Energy Systems Group

Faculty of Mechanical, Maritime and Materials Engineering (3mE)
Delft University of Technology
The Netherlands

Chair: Dr. Rene Pecnik

Co-readers: Dr. Jurriaan Peeters
Asif Hasan

ABSTRACT

The continued development of improved algorithms and architecture for numerical simulations is at the core of increased computational performance and, therefore, the ability to perform more complex and precise numerical simulations in less time in areas such as Computational Fluid Dynamics. Employing faster algorithms on more efficient processing units, such as Graphics Processing Units (GPUs), can reduce not only the time spend per simulation but also the energy required to perform these computations as well. This will be of significant benefit to different areas of research and engineering and through improvement achieved in these to society at large as well.

As simulations grow in dimensions and accuracy the wall clock time is bound to increase substantially, reaching days and potentially months depending on the parameters and geometry of the chosen simulation. The performance of different finite difference solvers with different degrees of optimization on different types of compute hardware was investigated and the achieved speedups assessed. Specifically, one serial CPU-based solver was presented as a baseline, which was then transitioned to a GPU-based solver. This in turn was then optimized further with regards to improved memory redundancy. To make comparisons fairer all of the solvers used the same temporal and spatial discretization techniques. Further, a benchmarking scenario was proposed to be used for the different solvers across used hardware, including the relevant geometry, gird, and initial and boundary conditions.

The speedups between the different solvers were observed and contextualized with regard to the effort that went into implementing the solvers and the capability and cost of the used hardware. The speedups at different problem sizes were investigated with the aim to establish how the performance gain from parallelizing and optimizing solvers scales with the chosen number of grid points and therefore the computational load.

Very significant speedups were achieved between the regular CPU solver and its GPU implementation, clearly showing the possible performance gains when moving from a serial to a parallel implementation running on an accelerator. The speedups between the two GPU-based solvers were more modest but still significant when considering the possible time spend on one simulation, depending on the chosen number of grid points.

Finally, an additional performance analysis was performed on two Navier-Stokes, one of which was the optimized version of the other, to investigate whether the performance increases were in line with prior findings and what magnitude of a reduction of wall clock time was possible for a state-of-the-art finite difference Navier-Stokes solver.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, Dr. Rene Pecnik, for agreeing to supervise my thesis, his help in finding a topic, and his continuous feedback and advice throughout the past nine months. I would also like to thank him for his assistance in all administrative matters regarding this work.

Beyond this, I thank Simone Silvestri for taking the time to help with many technical problems encountered in the early stages of this thesis, for agreeing to let me use his Navier-Stokes solver in this work, and for proof-reading early drafts of this work.

Finally, I would like to thank Asif Hasan for agreeing to let me use his optimization of a Navier-Stokes solver in this work and for the help with the troubleshooting its implementation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Computational Fluid Dynamics (CFD) is concerned with modelling and predicting the behaviour of fluids. The cornerstone of CFD are the Navier-Stokes (NS) equations, which are used to model the flow of viscous fluids by applying the conservation of momentum and conservation of mass to fluids. To study the performance of different implementations of solvers for numerical simulations, a sub-problem of the NS equations will be the focus of the majority of this work: the heat equation.

The continued development of improved algorithms and architecture for simulating fluids is at the core of increased computational performance and, therefore, the ability to perform more complex and precise numerical simulations in less time. Furthermore, faster algorithms on more efficient processing units, such as GPUs, can reduce the energy required to perform these computations. This will be of significant benefit to different areas of research and engineering and through improvement achieved in these to society at large as well.

As simulations grow in dimensions and accuracy the wall clock time is bound to increase substantially, reaching days and potentially months depending on the parameters and geometry of the chosen simulation. Moreover, memory requirements will increase as well with the size of the problem. This makes High Performance Computing (HPC) for CFD, through the use of optimised parallel algorithms running on more capable hardware, appealing in a bid to make the most demanding simulations more accessible. The main source of potential performance increase in the 21st century are accelerators, purpose-build processing units designed with many more computational cores compared to standard CPUs, which make speedups of an order of magnitude or more for certain applications possible. The most widely used of these is the Graphic Processing Unit (GPU), which was initially developed for efficient computer graphics processing and visualisation in the 1970s. The highly parallel architecture found on GPUs makes them very effective at solving problems for which large blocks of data may be processed in a parallel fashion. Characteristically GPUs have a high floating-point operations per second performance, implying that they are capable of performing a large number of computations per second. This, in combination with GPUs being readily available in most computers, justifies an investigation into whether a significant speedup in research CFD simulations can be achieved through the use of accelerators and how this speedup may best be achieved. GPUs have become increasingly popular in the past ten years due to their particular architecture lending itself to speeding up the parallel computations typically performed during Monte Carlo simulations [36], machine learning [42], or molecular dynamics [43].

The programming framework of choice for accessing GPU computing is Nvidia's Compute Unified Device Architecture (CUDA), which allows programmers to write code directly on the GPU in a variety of languages, such as Python or FORTRAN, with its default being a C-style language (simply referred to as CUDA in the rest of this paper).

1.1 THESIS STATEMENT OF PURPOSE

The primary aim of this thesis is to design, evaluate, and optimize a CUDA-based solver for the three-dimensional heat equation which runs on a GPU and provides a significant speedup over CPU-based methods. To accomplish this, the solver must scale strongly with the parallelization offered when GPUs are being used. The intended approach is build on a CPU-based solver and involves designing an equivalent in CUDA, before trying to optimize this approach further in order to increase performance, taking full advantage of parallelizing an algorithm and the increase in computational power such a system provides, while trying to minimise the performance bottlenecks that come with a parallel solver run on an accelerator. Whether the project was successful will be judged by the speedup achieved by the solvers, as well as their ease of use, and whether insights can be obtained about how feasible it is to parallelize different numerical simulations.

1.2 CPU VS GPU COMPUTING

The CPU can be viewed as the brain of the computer, it is in nature general-purpose orientated and its key tasks include handling arithmetic calculations and logic operations, as well as input/output operations. Virtually all operations a computer performs are build upon these building blocks. The CPU has build-in memory it can access more quickly, similar to that of a GPU in structure and operation, but is also connected to Random Access Memory (RAM) which is installed separately in the computer. The build-in memory, also known as cache, can be accessed significantly faster than the separate RAM due to being located closer to the processing units. Much like the GPU, the CPU's caches are also divided into a memory hierarchy, with some caches being closer to the processing units than others.

1.2.1 Serial Example

Beyond the difference in architecture, which will be explained in detail in a later chapter, the main differentiating factor between the two categories of computing hardware is the number of cores and the core clock frequency, which is a measurement of how many operations the processor can perform each second. It is typically faster for the cores of the CPU compared to those of the GPU. A typical CPU core runs in the single digit GHz range, while

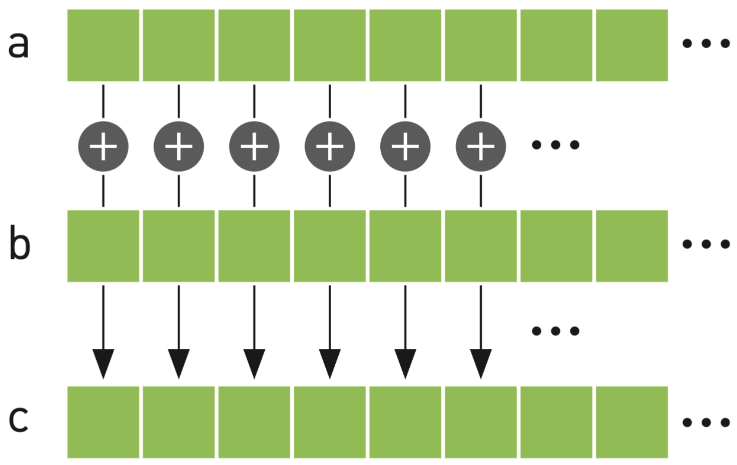


Figure 1.1: Two vectors are summed at each index and their resulting value is stored in a third vector. Each block represents one address of memory which holds the value of the corresponding vector. Figure courtesy of Nvidia [39].

a typical accelerator core will have a clock speed in the low thousand MHz range. This, in combination with the aforementioned differences in the number of available cores, results in the great possibility presented by optimizing algorithms for the GPU to make use of these many cores. To illustrate the differences between CPU and GPU (and in turn serial and parallel) computing more effectively, a series of algorithmic examples will follow, performing vector addition purely algorithmically, in serial C++ implementation, and in parallel CUDA code. While an understanding of C-style languages is beneficial to understand the presented examples, it is not required to grasp the underlying concepts. Vector addition is the most common, as well as one of the simplest, examples to highlight the differences between serial and parallel computation. The following algorithms are loosely based on the book CUDA by Example by Sanders and Kandrot of Nvidia [39]. Algorithm 1 outlines a simple vector addition through the use of pseudo code. Here the vectors a and b get summed to vector c at each index i, where the index is increased by one every iteration. The loop stops when the index is the size of the predefined constant N. This is graphically represented in figure 1.1, where each block represents the memory that holds a value of one of the vectors mentioned above.

Algorithm 1 Pseudo code of straightforward vector addition.

```
while $i < N$ do
    c[i] = a[i] + b[i]
    i += 1
```

This algorithm is serial in nature, with each iteration of the loop only performed once the previous iteration has finished. Algorithm 2 shows a very rudimentary implementation of this vector addition in C-style code, designed to be executed on a CPU in serial fashion. Here, once the vectors are declared and populated the *add()* function is called, where the summation occurs until the loop is finished. The memory allocation and read and write operations occur between the CPU and RAM and are managed by the compiler.

Algorithm 2 Simple C++ code of serial CPU vector addition implementation.

```
#define N 10

void add( int *a, int *b, int *c ) {
    for (int i=0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}

int main() {
    int a[N], b[N], c[N];

    // populate arrays a and b with chosen numbers in CPU memory
    for (int i = 0; i < N; i++) {
        a[i] = i;
        b[i] = -i;
    }

    add( a, b, c );

    // display the results
    for (int i = 0; i < N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    return 0;
}
```

1.2.2 Parallel Example

To implement the algorithm effectively in CUDA it must first be parallelized. This will be done through build-in functions in this example, allowing the programmer to split the problem into one sub problem to be run on a single thread for each addition. This parallel CUDA version can be found in algorithm 3.

While the syntax used for CUDA is similar to any C-style language, there are some key differences. The *__global__* keyword preceding the *add()* function identifies it as a global function, i.e. a function to be run on the GPU which you can call from the host side using CUDA kernel call semantics. Separate arrays have to be declared on the host side to be used on the device for the vectors, where they receive the *gpu_* prefix within this code to clearly label them as such. The *cudaMalloc()* function is used to allocate memory on the GPU side from the host and *cudaMemcpy()* can be used to copy memory from the CPU to the GPU side or vice versa, which is crucial since the host and device are not able to access each others memory directly. The kernel for the *add()* function is launched on line 30 with the corresponding number of blocks and threads allocated by the programmer, the details of which will be explained later. The blocks each contain the number of specified

Algorithm 3 CUDA implementation of parallelized vector addition for the GPU.

```
#define N 10

__global__ void add(int *a, int *b, int *c) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) {
        c[tid] = a[tid] + b[tid];
    }
}

int main() {
    int a[N], b[N], c[N];
    int *gpu_a, *gpu_b, *gpu_c;

    // allocate memory on the GPU
    cudaMalloc( (void**)&gpu_a, N * sizeof(int) );
    cudaMalloc( (void**)&gpu_b, N * sizeof(int) );
    cudaMalloc( (void**)&gpu_c, N * sizeof(int) );

    // populate arrays a and b on the CPU
    for (int i = 0; i < N; i++) {
        a[i] = i;
        b[i] = -i;
    }

    // copy arrays a and b to from the CPU to the GPU
    cudaMemcpy( gpu_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( gpu_b, b, N * sizeof(int), cudaMemcpyHostToDevice );

    // call CUDA kernel
    add<<<1,N>>>( gpu_a, gpu_b, gpu_c );

    // copy the array c back from the GPU to the CPU
    cudaMemcpy( c, gpu_c, N * sizeof(int), cudaMemcpyDeviceToHost );

    // display the results
    for (int i = 0; i < N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    cudaFree( gpu_a );
    cudaFree( gpu_b );
    cudaFree( gpu_c );

    // and on the CPU
    free( a );
    free( b );
    free( c );

    return 0;
}
```

threads and each thread executes the function called through the kernel. The syntax for launching a kernel is as follows, $<<< M, T >>>$, where M represents the blocks allocated to the kernel and T the number of threads per block. Once the kernel has finished, the results are copied back into CPU memory before they are printed by the CPU. Finally, after printing the result, the memory allocated on the GPU is freed. As many threads are launched simultaneously, each thread performs a single call to the global function of the kernel, summing the two values of the initial vectors and storing the computed result in memory.

The memory for the reading of the initial values and for the final storing of the results, as well as the individual thread responsible for the computation, are determined via the appropriate global *tid* index. This index is computed in line 4, using *threadIdx.x*, *blockIdx.x*, and *blockDim.x*, indicating the index of the individual thread, the index of the block the thread is found in, and the dimensions of this block. It is determined for each unique threads every time the function which contains it is called. This unique index, corresponding to a thread which is able to access a place in memory, makes it possible to parallelize the code effectively without many changes, as it allows for each thread across the different blocks to have a unique identifier. Through this identifier, it becomes possible to run the same function, *add()* in this case, in parallel through operating on different data stored in designated memory addresses.

1.3 ORGANIZATION OF THIS THESIS

In chapter 2 the architecture of accelerators, specifically Graphics Processing Units manufactured by Nvidia, is introduced and what makes them particularly suitable for the task at hand is described in detail. This includes both hardware and software considerations that are necessary to be aware of in order to utilize the full potential of accelerators to speed up CFD simulations.

Chapter 3 presents a short literature review on general CFD research and its history, before reviewing the specific aspects of CFD relevant to this thesis. Finally, the use of accelerators in this field of research is reviewed.

In chapter 4 all the relevant theory of this thesis is introduced, starting with the Navier-Stokes equations followed by their relation to the heat equation. Following this, the heat equation is discretized in the spatial and temporal domains, using finite differences and Runge-Kutta 3 respectively, to achieve equations that can be implemented computationally. Finally, the used boundary conditions are introduced.

Chapter 5 first introduces the different solvers to be used in the simulations. Here, initially the CPU-based solver is introduced before explaining the structure of a similar GPU-based solver and finally the optimized GPU-based solver. Following this, the validation scenarios are explained and the results for all three solvers are analyzed in order to fully validate all three solvers.

Chapter 6 begins with the benchmark case employed across all three solvers being introduced, including the chosen initial and boundary conditions, grid, and other relevant parameters. Then the used hardware of the different workstations is discussed and contextualized. Finally, the results of all three solvers are shown and their performance is compared and studied.

In chapter 7 a further performance analysis is performed to contextualize the previously studies solvers. This is based on a different GPU-based solver, which does not focus on the sub-problem of the heat equation, but rather the entirety of the Navier-Stokes equations.

Chapter 8 summarizes the work presented throughout the different chapters, assessing what the key takeaways are, before giving some recommendations for further work on this topic.

2 LITERATURE REVIEW

This chapter aims to gather and summarise the previous work done that is relevant to this project. More specifically, the previous work which focuses on CFD, the numerical schemes and problems thematized in this work, and these being implemented on GPUs. An uninitiated reader will be provided with a choice of past publications needed to gain a deeper understanding of the relevant topics, while simultaneously selecting those that are relevant to put the findings later encountered in this work into context. Section 2.1 gives a general overview of CFD literature, including the application of the finite difference method and the heat equation, and previous work on DNS. Section 2.2 outlines how GPUs have been used more recently to tackle computationally intensive simulations, including CFD. Finally, section 2.3 gives an overview of different performance analysis related to CFD, including across different hardware and schemes, as well as some of the achieved speedups.

2.1 THE ORIGINS OF CFD RESEARCH

Computers have been used as an aid in solving fluid dynamics problems almost since the inception of programmable computers in some shape or form, with the first exploration specifically into Navier-Stokes based CFD numerical methods occurring in the 1950s at Los Alamos National Laboratory [17]. Today, CFD is a well established and active field with a plethora of documented use-cases beyond fundamental research, ranging from aerodynamics [23] to the food industry [52].

The concepts and assumptions which lie at the base of CFD and fluid dynamics have been validated through experimental tests and are well documented in literature. Further, the different approaches and ideas used in CFD to solve partial differential equations, such as finite volume methods and numerical dissipation, have been studied extensively in other fields dealing with numerical modeling [26] [49].

Beyond this, Moin and Mahesh wrote on how DNS is a key tool of accurately modelling turbulence in CFD, focusing on potential numerical issues such as boundary conditions and the importance of accurate spatial and temporal discretization [29]. The authors also went into detail on how DNS has been used in CFD research to gain a better understanding of the physics underlying turbulent flows.

A great way to see the progress in performance of CFD, and therefore the increase in the usability of CFD when applied to problems with increased complexity, can be found in the body of work by Moser et al. [31] [22] [30]

[25]. This revolves around DNS of turbulent single phase channel flow and was carried out over the span of over four decades. The hydraulic Reynolds number (Re_h) they were able to resolve has increased from a value of 11,960 in their earliest work in 1984 to a value of 500,000 in 2015, using the hydraulic channel diameter as their characteristic spatial scale.

2.1.1 Finite Difference Method

The finite difference method is a tool that leverages finite differences for solving differential equations by way of approximating derivatives. It makes it possible to solve partial differential equations computationally by turning them into a system of linear equations to be solved. According to Grossmann [15], the finite difference method is widely employed when trying to find the numerical solution to partial differential equations, whether that be in one, two, or three dimensions. The finite difference method has been computationally applied to three dimensional fluid dynamics problems for decades at this point [38]. More detail regarding the exact chosen implementation of the finite difference method will be given in a later chapter.

Beyond fluid dynamics, where applications of high-order finite difference schemes are plentiful, such as [21] and [40], finite differences have been used in a range of different areas, such as finance [51], where finite difference approximations of derivatives are required.

2.1.2 Heat Equation

The heat equation is a partial differential equation which was initially developed by Fourier in 1822, which describes a volume at a specific point and determines whether it will increase or decrease in temperature, due to this being proportional to the difference in temperature to the surrounding volume [50]. The heat equation is often applied to idealise the modelling of heat flow in a one dimensional rod in applied mathematics [7], but can also be extended to higher dimensions easily to model a three dimensional volume. The heat equation is a key part of fluid dynamics and is often simulated computationally. There exist plenty of use cases where the heat equation was applied computationally, such as modeling heat transfer in pipes [44] and for air heat exchangers [53].

The diffusion equation, another partial differential equation, can be used interchangeably with the heat equation when the diffusion coefficient of the diffusion equation is constant.

2.2 NUMERICAL SIMULATIONS ON GPUS

The CFD community was an early adopter of GPU computing, with Corrigan et al. [10] implementing a Runge-Kutta based unstructured grid solver for the 3D compressible Euler equations, which are a simplification of the NS equations, as early as 2009 and reporting speedups by a factor of 9.5 for the single GPU implementation compared to a parallelized CPU implementation and of 33 when compared to a serial CPU implementation. They also used Nvidia GPUs and CUDA as their programming framework of choice.

Nvidia, the largest manufacturer of GPUs, was promoting GPU computing for fluid simulation applications using its in-house programming language as early as 2009 as well [8]. There they laid out a road map to transition from legacy CFD code to parallel GPU code. Further, they benchmarked FORTRAN code for a Rayleigh-Bénard Convection, with a 384 by 384 by 192 grid, for double precision and second order accuracy, where they were able to achieve a speedup of factor 8.5. This benchmark was performed on equally priced CPU and GPU computational nodes.

Niemeyer and Sung reviewed the progress and challenges for GPU computing for CFD in 2014 [32]. Within this review, they presented comparisons between CPU- and GPU-based solvers for the incompressible NS equations. Further, GPU implementations of the Lattice Boltzmann method, laminar and turbulent solvers, and Laplace equations were discussed. Finally, recommendations for parallelizing CFD code for GPU computing are given and potential opportunities in the field were discussed.

As CFD on GPUs became more established and grew in scale, the potential of optimizing CPU-GPU systems to improve performance became clear, especially in HPC and large cluster applications. Posey wrote on parallel CFD, co-processing between CPUs and GPUs, and achieving high throughput on nodes with up to 8 GPUs in 2013 [35].

Finally, there are also plenty of examples of finite difference methods being run on GPUs, such as [6] and [48], as well as of the heat equation being solved on a GPU [5].

2.3 CFD PERFORMANCE ANALYSIS

There have been many scaling analyses and benchmarks related to accelerators used for CFD purposes in the last ten years. While they cover many different solvers and configurations, it is still useful to study them to obtain a complete view of the research area.

For CFD, there are ample comparisons of CPU-based solvers against accelerator-based ones, such as the one by Antz et al. comparing performance for conjugate gradient routines through looking at runtime against problem size, though only small problem sizes and a small number timesteps were investigated [3]. Beyond this, Horvátha and Liebmann undertook a performance analysis of CPUs and GPUs for the Euler equations on unstructured meshes [19]. Janßen et al. compared different CFD solvers based on the Lattice Boltzmann Method and on the fully-nonlinear Boussinesq equations with finite volume and finite difference methods, run on CPU and on GPU, for two dimensional grids of size 10^5 [20]. Crespo et al. undertook a performance analysis for a mesh-free particle method on single processors across a range of accelerators and CPUs [11]. Here, they achieved a speedup of 64 for their fastest accelerator when compared to their slowest tested processor.

Aissa et al. benchmarked a solver for non-linear partial differential equations using an explicit or an implicit scheme, comparing time per iteration for two CPUs and two GPUs. This implementation based around a RANS simulation across a range of number of cells. They achieved speedups of up to 136.4 between their slowest hardware (Intel Xeon E5-2640 CPU) using an implicit solver and their fastest hardware (Nvidia GTX 780 GPU) and an explicit solver. For their slower GPU (Nvidia K40) a speedup of only 6.09 was recorded when comparing implicit methods, at a grid size of $5 \cdot 10^5$ [1].

A benchmark of the two-dimensional heat equation was undertaken by Belhaous et al. comparing the C++ library SkelGis and a CUDA-based GPU implementation of a finite difference approach to the heat equation. They achieved only a moderate speedup of 12 based on a similar amount of compute [4].

There are also industry standard benchmarks for the testing of GPU clusters with multiple nodes, such as the Himeno CFD benchmark [34], which has been benchmarked for both GPU- and CPU-based systems by Matsuoka et al. for single units, as well as clusters [27].

3 COMPUTING WITH ACCELERATORS AND GPUS

Hardware accelerators are made for a unique purpose, which is the aim to perform specific computations in a more effective manner than the Central Processing Unit (CPU). The CPU is designed with a broad spectrum of applications in mind, seeing as it has to handle virtually all tasks performed by different applications on computers build for different purposes, thus sacrificing performance in some areas for breadth of use. Accelerators, on the contrary, are designed from the ground up with specific computations in mind, making to possible to achieve a higher degree of performance in targeted tasks. Examples of hardware accelerators include the Graphics Processing Unit (GPU), used for graphics rendering and image processing, and more recently the Tensor Processing Unit (TPU), an application-specific integrated circuit to be used specifically for machine learning software libraries. The reason GPUs have been so successful at tasks within and beyond graphics is that their architecture was not designed with general computations in mind, but rather for the parallel computations required during graphics rendering. This particular architecture lends itself to CFD simulations, which are not exactly embarrassingly parallel, i.e. an algorithm in which the tasks can be divided between different processes without the need for them to communicate at any time during the execution of the program, due to boundary conditions and halo communication, but do greatly benefit from parallelization.

This property makes it possible to use the strengths of the GPU, namely its large number of cores intended to perform many similar smaller computations in parallel, to speed up CFD simulations once the algorithm has been altered to create many parallel tasks which can be run concurrently, instead of performing all tasks in a serial nature. Even though the cores, and specifically the arithmetic logic units (ALUs) they contain and which are

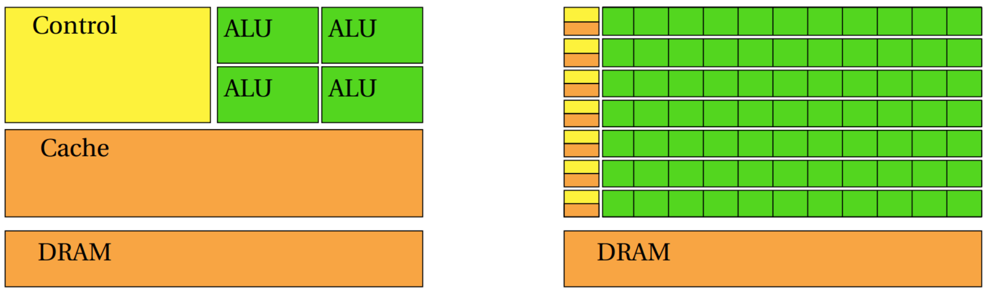


Figure 3.1: Conceptual representation of CPU (left) and GPU (right) for Control, ALU, Cache, and DRAM elements. Figure courtesy of Nvidia [33].

responsible for the actual computations, of a CPU have significantly faster clock frequencies, the cores of the GPU are multiple orders of magnitude more numerous. This difference is shown conceptually in figure 3.1. To summarize: while the CPU is optimized for flexibility, the GPU is optimized for throughput of parallel computations.

This chapter will initially outline the possible benefits and drawbacks of using GPUs for scientific computations like CFD, before moving on to describe the software and hardware architecture of Nvidia accelerators. Finally, some performance considerations will be highlighted and the opportunities of multi-GPU systems will be explained.

3.1 CUDA AND GPGPUS

Throughout their continued development over the past decades, graphics processing was the main area GPUs were applied to. The GPU has more recently been successfully employed to tasks beyond graphics as well, such as CFD and machine learning. These areas outside of graphics are in the domain of General Purpose GPUs (GPGPU), where the thousands of cores of a GPU can be used to perform other computational tasks provided that the task is parallelizable, which is the case for CFD. Previously, programming GPUs was achieved through shader languages, which had very few features and did not support many important aspects needed for wide-ranging GPGPU operation. Key aspects, such as floating point data for example, were missing. Today, this can more easily be achieved through the use of a programming language tailored to making the architecture of the hardware more accessible. The main language used for this purpose is Compute Unified Device Architecture (CUDA) which is developed by Nvidia and is exclusively available for their GPUs. There exist less popular alternatives to CUDA, such as the open source OpenCL, which can be used on any hardware platform irrespective of manufacturer. The default CUDA programming language is very similar to C-style languages - in both syntax and operation - with some GPU specific features. Furthermore, CUDA implementations of other common languages, such as FORTRAN and Python, with similar features to the default CUDA language exist. CUDA is build on the Single Program Multiple Data (SPMD) paradigm, where different threads of the GPU perform the same calculations on different sets of data. CUDA code can largely be separated into two kinds, host-side code which is run on the CPU and device-side code which is run on the GPU. Here, the host is responsible for the commands send to the device and allocating and copying memory, while the device runs kernels, which perform the actual calculations. These kernels may need to be invoked by the CPU initially. It should be noted that the memory of the host and the device are completely separate, both physically and computationally, i.e. the CPU cannot access the GPU memory directly and vice versa. Thus if some data stored in the memory of either the CPU or GPU is needed by the other it must copied over, which can be a bottleneck when trying to speed up the computation.

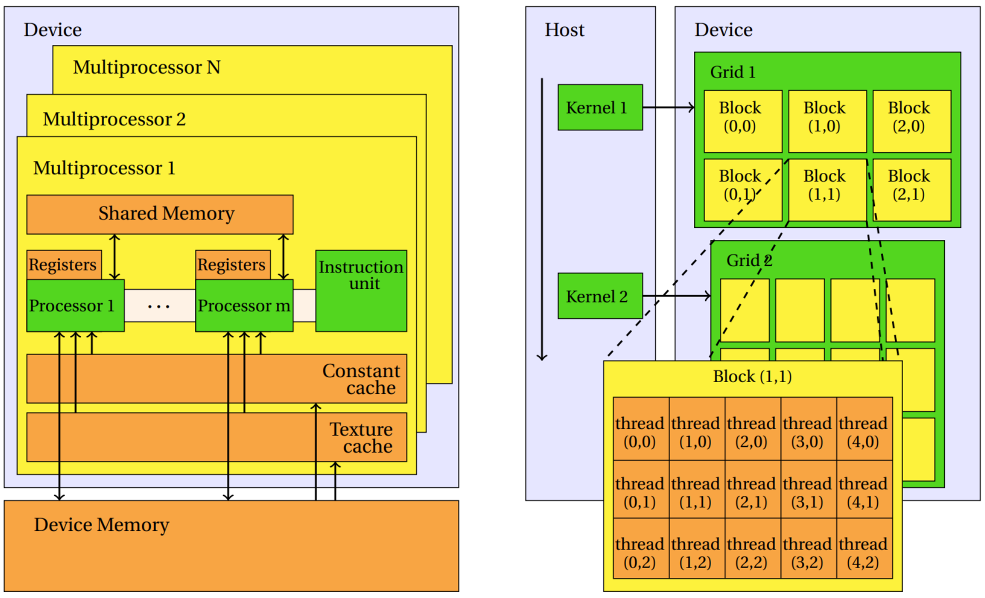


Figure 3.2: Schematic of GPU architecture (left) and how kernels from the host execute on the device in the grid/block/thread configuration (right). Figure courtesy of Nvidia [33].

3.2 GPU ARCHITECTURE

In order to get the highest possible performance increase out of using accelerators for scientific computations some level of knowledge of how the used accelerator is structured on an architectural level, i.e. how the hardware is organized and how instructions are executed, is required.

Every accelerator manufactured by Nvidia is given a compute capability rating, which is a two-digit number. For the GPU used later on in this thesis, the compute capability is 8.6. Here the first number, known as the major revision number, denotes the micro-architecture of the card, which in this case is the latest Ampere architecture, while the second number denotes the minor revision which is the version within that micro-architecture. Different architectures come with different features and compatibility in regards to the version of CUDA used, thus it is key to be aware of the corresponding compute capability of the used device.

GPUs are made up of a number of streaming multiprocessors (SMs), the exact number of which varies by card. SMs act independently from one another and are connected to the Dynamic Random Access Memory (DRAM) of the GPU. These SMs contain multiple cores each, which act in lock-step, i.e. they perform the same instructions on changing data in accordance with SPMD and are, therefore, parallel. The architecture of such SMs for Nvidia GPUs can be seen in figure 3.2. Each SM contains multiple streaming processors (SPs), which are the ALUs, or the actual part of the accelerator responsible for performing the computations. These SPs can be compared to the cores typically found in a CPU and their number varies across compute capabilities. The SM is made up of many different components beyond just SPs, most notably many different arrays of memory at different places of the memory hierarchy and the warp scheduler (also known as an instruction unit) which is responsible for scheduling the order and timing of computation of different warps of threads, which will be introduced later in this section. Accelerators with a more recent compute capability have two of these warp schedulers per SM, one dedicated to odd threads and one dedicated to even threads.

3.2.1 CUDA Execution

Figure 3.2 provides a more detailed look at the grids which exist in the CUDA execution model on the SMs (which are hardware) and the memory and processing units they are made up of. The CUDA execution model at its core is made up of computational threads which can be grouped into blocks which, in turn, then exist together in a grid. As the device executes a kernel for the host, it is passed to a grid which contains the threads required to perform the computations. The concept of a kernel was previously introduced in algorithm 3 and is fundamentally a C function making it possible to communicate from the host to the device exactly what instructions should be followed how many times. This is then allocated to the number of threads specified in the call to the kernel. A thread is a CUDA concept and constitutes what makes up the executions for the kernel, where every thread has a corresponding subset of the data to work with, tied to its identification, known as a thread index.

Within CUDA both threads and blocks can be grouped into one-, two-, or three-dimensional structures which may be more difficult to implement but can come with performance benefits. During our previous example in chapter 1.2, we simply used a one dimensional thread and block structure, denoted by the *.x* added to the end of *threadIdx.x*, *blockIdx.x*, and *blockDim.x*. Where a one-dimensional implementation puts all of these into a long 1D array, a two-dimensional implementation would split them into a matrix and use *.x* and *.y* suffixes. Doing such a restructuring allows for the data to match the computations according to whether they are a 1D array, matrix, or volume. During this work one- and two-dimensional structures will be investigated in CUDA.

The available number of threads for each block is not unlimited due to memory restrictions, as all threads are part of the same SP and therefore draw from the same pool of available memory. It is therefore desirable to choose blocks of equal dimensions to run on other SPs in parallel, which reside in a grid and are distributed to available SMs. A key factor when aiming for increased performance is to design the threads, blocks, and grid in a way that multiple threadblocks can be run without having to wait for others to complete their computation, making it possible to parallelize the program effectively. This is illustrated in figure 3.3.

Thread warps are the way CUDA allocates threads to perform computations and these warps are typically made up of 32 threads each, which are computed in parallel. They are the units executed in Single Instruction Multiple Threads (SIMT) fashion and their execution is managed by the warp scheduler of the SMs, where the warp scheduler chooses consecutive thread indexes and schedules these threads for execution. Multiple warps can run

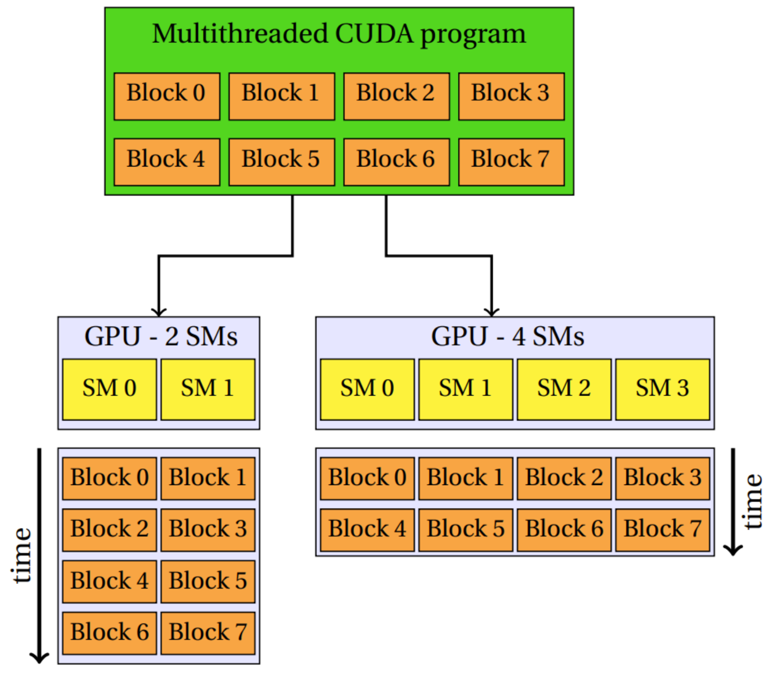


Figure 3.3: Schematic on how different blocks are distributed among SMs over time. Figure courtesy of Nvidia [33].

concurrently on a SM the exact number of which is dependent upon the computing capability of the accelerator.

As the device executes a kernel, it is passed to a grid, which contains the threads which perform the computations. The kernel is finished when all blocks have completed their assigned computations. The unique type of memory available to each thread are registers, as well as a general purpose cache, also known as local memory.

Since threads may not be perfectly in sync within a block, i.e. they finish their computation at not precisely the same time, the *__syncthreads()* command can be used to synchronize all threads that belong to the same block.

3.2.2 Memory

The memory found on the GPU is made up of a many different kinds of memory, each with their own purpose, size, and speed as can be seen in figure 3.2. There it is also indicated what parts of the grid are able to access what parts of the memory located on the GPU directly, which is commonly referred to as memory hierarchy.

The most high-level and largest memory is the global memory, to which all threads can write and from which all threads can read data, as well as the read-only constant memory and texture memory. It can be accessed by all SMs, and therefore all SPs, for read and write operations. Access latency is the largest out of all types of memory and the size usually is in the range of several GB. Beyond this, each SM is equipped with multiple other kinds of memory. There is the read-only constant cache and texture cache, which is available to all blocks and their threads for each SM. Each block has its own shared memory, usually up to 48 KB, which is completely managed by the programmer that can also be accessed by the threads of that block. Shared memory often is used when looking to increase performance of a solver, as it sits in a sweet spot between latency and size. The unique type of memory available to each thread are registers, with each register corresponding to a single thread. Registers have the lowest latency out of all memory but are significantly smaller in memory size compared to other forms of memory. Generally speaking, the larger the memory is, the further it is located from the processing units, and therefore the longer is the latency to access this memory. For example, threads accessing global memory comes with a two orders of magnitude larger latency compared to shared memory, with the size of global memory being in the order of Gigabytes and that of shared memory in the order of Kilobytes.

There are some specific hardware features which must be kept in mind when programming a GPU using CUDA. When considering the parallelization of the code and how to allocate the computational load, it is key to understand the execution and memory limitations outlined above. The key numbers to consider can be found in table 15 of the CUDA Programming Guide (available freely through Nvidia) and can depend on the technical specification for the compute capability. As a rule of thumb, for every GPU a block can be made up of a maximum of 1024 threads and the maximum number of registers per thread is 255. The maximum number of registers per SM is 65536. The maximum amount of local memory per thread is 512 KB and the amount of constant memory is 64 KB. All of these limitations are set by the hardware and must not be violated by the programmer. If, for example, the number of registers for an SM has reached 65536, then no additional blocks can be scheduled to it for the time being. Code should be designed with the most frequently accessed data being stored in the low-level memory, such as registers, with care being taken to use as much of the available memory as possible. Operations on non-gird located memory should be limited as they come with a higher time cost. For example, constant memory, which is cached, should be used for declaring constants at the start of a program. At the time of programming, each kernel must be individually invoked with the chosen number of threads per block and the number of blocks needed. The values specified should be within the previously stated hardware limitations.

3.3 IMPROVING PERFORMANCE THROUGH CUDA

In order to understand how to properly improve performance using CUDA, the distributed memory model is used, which is usually applied to distributed computer systems but can also be applied here to outline the communication that happens inside a GPU during a simulation. The time taken to perform a parallel computation can be stated as

$$t_{total} = t_{comp} + t_{comm}, \tag{3.1}$$

where t_{total} is the total time spend on the solution, t_{comp} represents the time dedicated to pure computation, and t_{comm} denotes the entirety of the parallel overheads, i.e. the time taken to communicate information within the system. These parallel overheads can be further broken down into

$$t_{comm} = a + bN, \tag{3.2}$$

where a is the latency cost of information and is a constant factor for each transmission operation, b is the inverse bandwidth cost which grows linearly with the size of the transmitted data, and N is the number of the pieces of data transmitted. Note that in practice $a > b$.

Using this information as well as information from previous sections of this chapter on memory and architecture, a few guidelines can be outlined that should be followed in order to design a CUDA program that allows for the best possible speedup of over a CPU-based one for numerical simulations. As t_{comp} is limited by hardware and cannot be improved, our goal is to minimise t_{comm} as much as we can, which is done through eliminating as much data being copied inside the accelerator as possible, through correct and lean allocation. This ties in with the maximum possible use of fast memories, such as shared memory or preferable registers, removing as much latency as possible. Finally, all data transfer between the device and the host should be kept to a minimum, as it is the most time consuming of all data transfers. The parameters that require copying from one to the other should be carefully chosen and the time data is transferred should be limited (ideally to two transfers, one initial one from the host to the device and one final one with the results back to the host).

3.4 GPUS AND SCIENTIFIC COMPUTING

Scientific computing, as well as CFD simulations, often boil down to solving a system of equations, in the CFD case usually discretized governing equations, at successive timesteps. Memory reads and writes pose the major obstacle for scientific GPU computing at a reasonable scale. These take away from the main strength of a GPU which is its ability to perform many floating point operations per second in parallel. This advantage makes them particularly suitable for CFD simulations, as these usually require equations to be solved for a large amount of data as the problem size increases and thus can take advantage of the GPUs ability to perform more operations per second than a CPU under the right conditions.

GPUs, due to their history of being used for graphics processing, are mostly focused on single precision (32-bit) operations, whereas scientific computing usually requires double precision (64-bit) accuracy. Depending on whether a consumer-grade GPU or a data center-focused GPU is used, there may be a focus on single or double precision in the architecture, respectively. Double precision operations can still be performed on a consumer-grade GPU, though this comes at a performance penalty. The simulations performed in this thesis will be of double precision accuracy.

3.5 MULTI-GPUS AND SUPERCOMPUTERS

To capitalise more on these strengths, the next step is to divide the computational load over even more cores. This can be done through using multiple GPUs simultaneously within a multi-GPU system. Such multi-GPU systems combine many GPUs (and usually also CPUs) into connected computational clusters. One can easily judge how successful this approach is by observing the TOP500, a list which aims to rank the most powerful supercomputers around the world. The majority of the systems on the list, and especially those at the top, are based on clusters of CPUs and many GPUs working together. The TOP500 ranking aims to test supercomputers using high performance LINPACK benchmarks for double-precision floating-point operations [46]. Accelerators are especially prevalent at the top of the list. Of the current top ten ranked supercomputers, seven use some form of accelerator to complement the present CPUs, while of the entire list, 353 systems do not use accelerators of any kind. The Tesla V100 accelerator by Nvidia is the single most popular across the list, being used in 90 different supercomputers. Another area accelerators excel in is energy efficiency, with GPUs being up to 10x more energy efficient compared to CPUs, due to energy consumption rising rapidly with the need to sustain the increased clock speeds of CPUs [14]. The organisation which publishes the TOP500 also publishes the Green500 list, an efficiency ranking of supercomputers by GFLOPS per Watt using the same benchmarks as those used for the main ranking [45]. In the most recent ranking at the time of writing, eight of the top ten most efficient supercomputers are based on accelerators and 37 out of the top 40 systems using some kind of accelerator.

The main bottleneck in multi-GPU systems is usually not the computing power, measured in Floating Point Operations per Second (FLOPS), but rather the inability for the GPUs to make full use of memory and communicate with one another or the CPU effectively when compared to the possible computational throughput of the system. All data transfer usually occurs through the CPU, with memory instructions transferred through standard PCIe connectors, thought more recently, Nvidia has introduced NVLink to connect multi-GPU systems directly.

4 THEORETICAL BACKGROUND: NAVIER-STOKES, HEAT EQUATION, AND DISCRETIZATION

In this chapter the equations and theory relevant to the simulations which will be performed later are introduced. Here, the governing equations for the problem are initially introduced, followed by the spatial and temporal discretization that are required to implement them on a CPU or GPU. Further, the applied boundary conditions are discussed at the end of this chapter.

4.1 NAVIER-STOKES EQUATIONS AND SUB-PROBLEM OF UNSTEADY HEAT EQUATION

According to Kundu et al. [24] compressible flow occurs where changes in momentum within a flow produce important variations in pressure and density of a fluid, and the thermodynamic characteristics play a direct role in the development of the flow. In compressible flows, the sound speed in the fluid becomes an important parameter and cannot be treated as infinite (as done in the incompressible flow limit). This branch of fluid dynamics deals with flows at large velocities. These include external flows such as flows around projectiles, rockets, re-entry vehicles, and airplanes; and internal flows in ducts and passages such as nozzles and diffusers used in jet engines, rocket motors, and compressed gas systems.

The Mach number is commonly used to determine whether a flow can be treated as compressible or incompressible and is a dimensionless quantity given by

$$Ma = \frac{U}{c}, \tag{4.1}$$

where U is the local velocity of the fluid and c is the local speed of sound, given by

$$c^2 = \left(\frac{\partial P}{\partial \rho}\right)_s, \tag{4.2}$$

where P denotes the pressure and ρ the density of the fluid and the s subscript alludes to the fact that the partial derivative is taken at constant entropy.

Incompressible flow occurs at a $Ma = 0$, where the density is independent on the changes of pressure in the flow (though in practice, at a value of

$Ma < 0.3$ the flow may still be considered incompressible if it is quasi-steady and isothermal). Beyond that limit, compressible flow takes place at all Mach numbers, with a further characterisation into subsonic flow occurring at $Ma < 1$, followed by supersonic flow at $Ma > 1$, and finally hypersonic flow at $Ma > 3$.

4.1.1 Direct Numerical Simulation

Turbulence within a flow is represented by a complicated and unsteady solution of the Navier-Stokes equations. There are no analytical solutions to turbulent flows, hence a complete description of a turbulent flow, where flow variables such as velocity and pressure are known as a function of space and time, can only be obtained by numerically integrating the NS equations [29]. This is known as Direct Numerical Simulation (DNS) and requires a great amount of computational resources since the NS equations describing the flow must be resolved for all scales in space and time, which, depending on the problem, can be quite complex numerically. DNS does not use a separate turbulence model, unlike the previously mentioned RANS and LES, where turbulence modelling assumptions are essential to the method.

The main reason not to use DNS instead of some type of approximation is limited computational power. Just a few decades ago it was only rarely possible to carry out a full DNS; through recent increases in computational resources and algorithmic efficiency DNS have been increasingly used in research [47] [12] [13]. While the numerical algorithms underpinning DNS of incompressible flows have been established in previous decades, the formulations of compressible DNS are still evolving, as different formulations are possible [18] [9]. An advantage of DNS over real-world experiments is the ability to freely change parameters while offering more degrees of control over the flow conditions [37].

4.1.2 Compressible Navier-Stokes Equations

To adequately model a flow, the fully compressible NS equations have to be solved, which use the conservation of mass, momentum and energy principles to describe a flowing fluid. These equations can be expressed as

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0, \tag{4.3a}$$

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} = -\frac{\partial P}{\partial x_i} + \frac{\partial \sigma_{ij}}{\partial x_j} + f\delta_{i1}, \tag{4.3b}$$

$$\frac{\partial \rho E}{\partial t} + \frac{\partial \rho u_j H}{\partial x_j} = \frac{\partial q_j}{\partial x_j} + \frac{\partial \sigma_{ij} u_i}{\partial x_j} + f u_1, \tag{4.3c}$$

where t is time, $u_i = (u, v, w)$ denotes velocity in the x, y, and z directions, respectively, and $x_i = (x, y, z)$ denotes the corresponding coordinate direction in three dimensions. The energy per unit mass of the fluid, E, can be obtained by $E = c_v T + u_i u_i / 2$, where c_v is the specific heat at constant volume and T is the temperature of the fluid. The f terms denote the forcing term and σ is the stress tensor. The total enthalpy, H, can be determined using $H = E + P/\rho$. Further, the heat flux vector is given by

$$q_j = -k\frac{\partial T}{\partial x_j}, \tag{4.4}$$

where the thermal conductivity k is given by $k = c_P\mu/Pr$. Here c_P is the specific heat capacity and μ is the dynamic viscosity of the modelled fluid. Pr is the non-dimensional Prandtl number, which gives an indication about which form of heat transfer is dominant in the flow, either heat transfer by convection when the momentum diffusivity is dominant and $Pr > 1$ or by conduction when the thermal diffusivity is dominant and $Pr < 1$. When $Pr > 1$ in a boundary layer of the flow, then the velocity boundary layer is more prominent then the thermal boundary layer, which indicates momentum dissipation exceeding heat dissipation. The viscous stress tensor σ may be expressed as

$$\sigma_{ij} = \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}\right) \tag{4.5}$$

and describes the viscous stress, i.e. the rate of change of deformation in time, at a point of the fluid.

4.1.3 Heat Equation from Navier-Stokes

Instead of treating the entirety of the Navier-Stokes equations here, we will instead focus on the sub-problem of the three-dimensional heat transfer equation. It is connected to the equations above through the diffusion term of the Navier-Stokes equations. Namely, we will use equation 4.3c and make use of its diffusion term. To arrive at the heat equation we will make the assumption of the velocity, u, being zero. This simplification will make it possible to write the equation with only two remaining terms, which did not contain the velocity, as

$$\frac{\partial \rho E}{\partial t} = \frac{\partial}{\partial x_j}k\frac{\partial T}{\partial x_j}. \tag{4.6}$$

Since the energy per unit mass, E did contain the velocity, it can now be simplified as well due to the absence of the velocity, resulting in

$$E = \rho e, \tag{4.7}$$

where e is the internal energy and can be stated as

$$e = c_v T. \tag{4.8}$$

These expressions can then be substituted in to rewrite the equation 4.6 as

$$\frac{\partial\left(\rho c_v T\right)}{\partial t} = \frac{\partial}{\partial x_j}k\frac{\partial T}{\partial x_j}, \tag{4.9}$$

which can finally be rearranged and written out as the heat equation in three dimensions

$$\frac{\partial T}{\partial t} = \frac{k}{\rho c_v}\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right), \tag{4.10}$$

where $k/\rho c_v$ is the thermal diffusivity.

4.2 DISCRETIZATION

To benchmark the solver we will simulate the heat equation in three dimensions, assuming a medium which is both isotropic and homogeneous. The heat (or diffusion) equation governs the temperature distribution within a medium, i.e. it is a partial differential equation (PDE) describing how temperature changes in both space and time. Now, we will introduce the heat equation in differential form, followed by discretizations in both space and time.

4.2.1 Differential Heat Equation

The heat equation can be stated as a PDE, which relates a function of two or more variables to its partial derivatives, and is of the form

$$\frac{\partial \phi}{\partial t} = \alpha \Delta \phi, \tag{4.11}$$

where ϕ is temperature as a function of space and time, α is a positive coefficient which represents the thermal diffusivity of the medium, and Δ denotes the Laplacian for n dimensions

$$\Delta = \sum_{i=1}^{n} \frac{\partial^2}{\partial x_i^2}, \tag{4.12}$$

which for this three dimensional problem becomes

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}. \tag{4.13}$$

Therefore the three dimensional heat equation for modeling heat propagation in an isotropic and homogeneous medium in differential form is

$$\frac{\partial \phi}{\partial t} = \alpha\left(\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2}\right), \tag{4.14}$$

4.2.2 Spatial Discretization Heat Equation

To make the above stated equations usable for our simulations, they must first be discretized. This discretization implies changing continuous equations into discrete ones, which can be used for numerical evaluation. For spatial discretization, this means calculating the solution for a limited number of points rather that in the entire domain.

Here, the finite difference spatial discretization approach is used to discretize the continuous equations above in space. This finite difference approach uses an approximation of the derivative at each given grid point. This approximation is achieved through summation of values at adjacent grid points to the one for which the new value is calculated. This summation is governed by a set of coefficients, which depend on both the order of the employed scheme and the order of the derivative. One of the most intuitive implementation to understand this for the heat equation, which requires the second derivative, is the second-order finite difference scheme with a Taylor expansion which can be stated as

$$\frac{\partial^2\phi}{\partial x^2}_{x_i,t_n} \simeq \frac{\frac{\partial^2\phi}{\partial x^2}_{x_{i+1/2},t_n} - \frac{\partial^2\phi}{\partial x^2}_{x_{i-1/2},t_n}}{\Delta x} \simeq \frac{\phi_{i-1,n} - 2\phi_{i,n} + \phi_{i+1,n}}{(\Delta x)^2}, \tag{4.15}$$

where the aforementioned finite difference coefficients are 1, -2, and 1, as can be seen in equation 4.15 above, to obtain the new value.

The same second derivative for an eight-order scheme can be calculated with the coefficients $-1/560$, $8/315$, $-1/5$, $8/5$, $-205/72$, $8/5$, $-1/5$, $8/315$, $-1/560$ and can be written as

$$\frac{\partial^2\phi}{\partial x^2}_{x_i,t_n} \simeq \frac{Z}{(\Delta x)^2}, \tag{4.16}$$

where

$$\begin{aligned} Z = &-\frac{1}{560}\phi_{i-4,n} + \frac{8}{315}\phi_{i-3,n} - \frac{1}{5}\phi_{i-2,n} + \frac{8}{5}\phi_{i-1,n} \ \cdots \\ &- \frac{205}{72}\phi_{i,n} + \frac{8}{5}\phi_{i+1,n} - \frac{1}{5}\phi_{i+2,n} + \frac{8}{315}\phi_{i+3,n} - \frac{1}{560}\phi_{i+4,n}. \end{aligned}$$

The number of adjacent points used here, and therefore also the number of coefficients, dictates the order of the scheme. For different orders of schemes, different stencil sizes have to be used. Stencils are a geometric arrangement of a number of nodes, here assumed to be a one dimensional array, which relates to the point of computation. For an eight-order scheme, such as the one above, this results in *stencil* = 4, indicating the use of four points on each side from the center point in the x-, y-, and z-directions. This is demonstrated in a simplified two-dimensional view in figure 4.1, where in order to obtain the value at point (x, y), a number of nodes equal to *stencil* are taken into consideration in each direction up to and including $(x, y \pm 4)$ and $(x \pm 4, y)$. Here, h is the step size between any two nodes of the same axis and is constant, as the nodes are equidistant, both between points on the

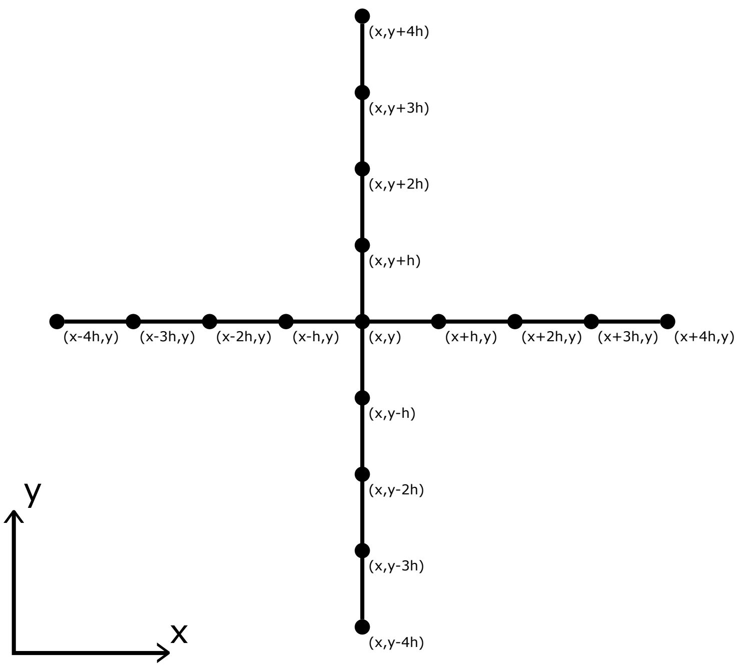


Figure 4.1: Two dimensional representation of the evenly-spaced grid used for $stencil = 4$.

same axis and between different axes. A higher order scheme, which takes more points into consideration, can be expected to be more accurate, but will increase the computational load as more points are considered during the computation.

4.2.3 Temporal Discretization Heat Equation

For temporal discretization the Runge-Kutta 3 (RK3) time stepping scheme is used to obtain approximations for the solutions to the previously stated differential equations. Runge-Kutta methods use more than one point to extrapolate the value to be calculated for the following timestep. The Runge-Kutta family of iterative methods are used for numerically estimating solutions to differential equations of the form $\frac{dy}{dt} = f(t, y)$, which can be used for temporal discretization and are here used for time stepping. The RK3 scheme can be written in the form

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 4k_2 + k_3)\,h, \tag{4.17}$$

$$t_{i+1} = t_i + h, \tag{4.18}$$

where h is the size of the timesteps, also called the step-size, and $h > 0$. This can be used to calculate the next following time t_{i+1} using the time at the previous step t_i. Further, y_{i+1} represents the RK3 approximation of $y(t+1)$, i.e. the approximation of the function value at the following timestep. This new value is determined by the present value y_i in addition to a weighted average of three increments, here represented by k_1, k_2, and k_3 respectively, each of which results from the estimated slope given by function f on the right-hand side of the differential equation and the step-size, h. These used increments can be described as follows: k_1 is the slope at the beginning of the interval, using y, k_2 is the slope at the middle of the interval, using y and k_1, and k_3 is the slope at the end of the interval, using y, k_1, and k_2. It can be seen that the weight given to the slope at the middle increment is four times the magnitude of the other increments, therefore this increment is more impactful for determining the result of that timestep. These intervals for $n = 0, 1, 2, 3, ...$ are given by

$$k_1 = f\left(t_n, y_n\right), \tag{4.19}$$

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right), \tag{4.20}$$

$$k_3 = f\left(t_n + h, y_n + 2k_2 - k_1\right). \tag{4.21}$$

The local truncation error (LTE), which is the error that is induced at every time-step, of the second order Runge-Kutta method is $O(h^3)$, while the LTE of the RK3 method is $O(h^4)$, and the LTE of the RK4 is $O(h^5)$, where h is the used step size, which is assumed to be constant. Therefore, as the order of the RK method is increased, the LTE changes. Evidently, a higher order method will result in a lower LTE for the same h. As the order of the RK method is increased, the LTE is reduced (provided a step size < 1), resulting in a more accurate simulation, while the time to compute is increased significantly. This presents the researcher with an important balance to find, the chosen method has to be accurate enough for the simulation's needs while also keeping computational constraints in mind to achieve good performance. This LTE is different from the total global error, which is defined as the absolute value of the difference between the true solution and the computed solution. If the exact solution is unknown then the global error cannot be evaluated. However, if we neglect the round off errors, it is reasonable to assume that the global error at the n-th time step is n times the LTE, since n is proportional to $1/h$, the global error should be proportional to LTE$/h$. This implies that for a k-th order method, the global error scales with kh.

Different temporal approaches have different benefits and drawbacks, with Runge-Kutta methods providing the right balance between accuracy and performance for this work. For example, compared to one of the most common time stepping techniques, known as Forward Euler, it provides far greater accuracy, as Forward Euler has an LTE of $O(h^2)$ due to being a first order method.

4.3 BOUNDARY CONDITIONS

In this work there were two types of boundary conditions used in conjunction with the equations above: Periodic and Dirichlet boundary conditions.

Periodic boundary conditions, which impose the same value on two sides, can be stated for the x-direction as

$$\phi_i = \phi_{i+N_x}, \tag{4.22}$$

$$\phi_{i+N_x+stencil} = \phi_{i+stencil}, \tag{4.23}$$

where N_x is the number of points in the x-direction and *stencil* is the used stencil size appropriate for the used scheme. Dirichlet, or prescribed temperature, boundary conditions directly prescribe a value at the boundary of the problem when applied to a PDE and can be stated for the x-direction as

$$\phi(x) = f(x), \tag{4.24}$$

where $f()$ is the function that dictates the value applied at x.

5 METHODS AND VALIDATION

This chapter will begin with explaining the three different solvers which share some characteristics and have key differences. The reasoning behind the choice of these approaches, their structure, as well as the differences and similarities between them are discussed in this chapter. Beyond that, this chapter outlines the methods used to simulate the theory introduced in the previous chapter, namely the governing equations of the heat equation in discretized form introduced in chapter 4.2.1, in order to validate the presented solvers. Here the selected benchmark scenarios are solved numerically using the spatial discretization shown in chapter 4.2.2 while the RK3 scheme shown in chapter 4.2.3 is used for time stepping.

5.1 SOLVER IMPLEMENTATIONS

Three different solvers were implemented to solve the governing equations numerically. Of these, one solver runs exclusively on a CPU and two solvers make use of an accelerator. From here on, the CPU-based solver may be referred to as the **first solver**, while the GPU-based implementation of that solver may be referred to as the **second solver**, and the optimized GPU-based solver may be referred to as the **third solver**.

Comparison of the three solvers			
	First Solver	**Second Solver**	**Third Solver**
Main Hardware	CPU	GPU	GPU
Computation	Serial	Parallel	Parallel
Stencil	1D	1D	3D
Spatial Scheme	8th-Order FD	8th-Order FD	8th-Order FD
Time Stepping	RK3	RK3	RK3
Optimization Effort	Simple	Difficult	More Difficult

Table 5.1: Summary of the characteristics of the three different solvers.

While the three solvers will be introduced separately in detail in the following sections, table 5.1 offers a quick summary to aid the reader in remembering which solver is which and what characteristics they may have.

5.1.1 CPU Solver (or First Solver)

The implementation which was first investigated in order to establish a baseline for performance was a standard CPU implementation of a finite difference solver, which does not take advantage of parallelization and accelerators. It was written in C++ due to the performance the language offers, as well as it being close in syntax to CUDA, which was used for the approaches that were investigated afterwards and will be presented in the following chapters 5.1.2 & 5.1.3.

The structure of this CPU-based solver can be summarised as follows: i) initially the memory is allocated through the CPU, where the needed number of bytes is allocated by the C++ language's *malloc()* function, ii) then the initial state of the system is defined, after which, iii) for each time step, the RK3 scheme is employed to solve the discretized equations to find the result of the heat equation for a corresponding node before moving to the next one, iv) before finally the data is saved and memory is freed again.

Algorithm 4 Pseudo code for the simplified structure of the serial first solver.

```
Input: Arrays initialized with initial condition, boundary conditions, as well
       as relevant constants
Output: Phi
for tstep ← 0 to nstep do
    /* RK Step 1                                                    */
    deriv2x(rhsx,phi,x)                     // second derivative in x
    deriv2y(rhsy,phi,y)                     // second derivative in y
    deriv2z(rhsz,phi,z)                     // second derivative in z
    for i ← 0 to $Nx * Ny * Nz$ do
        rhs1[i] = (rhsx[i] + rhsy[i] + rhsz[i]) * alpha;
    for i ← 0 to $Nx * Ny * Nz$ do
        phi[i] = phi[i] + rhs1[i]*dt;

    /* RK Step 2                                                    */
    deriv2x(rhsx,phi,x)                     // second derivative in x
    deriv2y(rhsy,phi,y)                     // second derivative in y
    deriv2z(rhsz,phi,z)                     // second derivative in z
    for i ← 0 to $Nx * Ny * Nz$ do
        rhs2[i] = (rhsx[i] + rhsy[i] + rhsz[i]) * alpha;
    for i ← 0 to $Nx * Ny * Nz$ do
        phi[i] = phi[i] + rhs2[i]*dt/4;

    /* RK Step 3                                                    */
    deriv2x(rhsx,phi,x)                     // second derivative in x
    deriv2y(rhsy,phi,y)                     // second derivative in y
    deriv2z(rhsz,phi,z)                     // second derivative in z
    for i ← 0 to $Nx * Ny * Nz$ do
        rhs3[i] = (rhsx[i] + rhsy[i] + rhsz[i]) * alpha;
    for i ← 0 to $Nx * Ny * Nz$ do
        phi[i] = phi[i] + rhs3[i]*dt;

    /* Finally the output Phi array is updated for the current iteration */
    for i ← 0 to $Nx * Ny * Nz$ do
        phi[i] = phi[i] + dt*(rhs1[i]+4*rhs2[i]+rhs3[i])/6;
```

The simplified version of the applied computational scheme can be seen in algorithm 4, where an outer loop iterates for every timestep and the RK3 scheme is performed with second derivatives in every direction. The called function used to calculate the derivative is shown in simplified from in algorithm 5 with *deriv2x()* serving as the example for the x-direction. The structure of the general solver loop is similar for all three solvers and is almost identical between the first and second solver, with the difference that all for-loops except the most outer one are foregone due to the parallelization of the routine, where the computational load is distributed using the *globalIdx* instead of performing serial loops.

Algorithm 5 Pseudo code of functions used for index and second derivative by first solver.

```
Function idx(i, j, k):
    return (((i) + (j)*Nx + (k)*Nx*Ny);)

Function deriv2x(rhs, phi, x, i, j, k):
    d2x = 1 / ((x[1] - x[0]) * (x[1] - x[0]))
    for k ← 0 to Nz do
        for j ← 0 to Ny do
            for i ← stencil to Nx + stencil do
                phiBound[i] = phi[idx(i-stencil,j,k)]

            /* Periodic boundary conditions */
            for i ← 0 to stencil do
                phiBound[i] = phi[idx(i+Nx-stencil,j,k)]
                phiBound[Nx+stencil+i] = phi[idx(i,j,k)]

            for i ← 0 to Nx do
                /* rhs is phi derivative in x */
                rhs[idx(i,j,k)] = coeffS[stencil] * phiBound[i+stencil]*d2x
                for idx ← 0 to stencil do
                    rhs[idx(i,j,k)] = rhs[idx(i,j,k)] + (coeffS[idx] *
                        (phiBound[i+idx] + phiBound[i+stencil*2-idx]))*d2x
```

The main limitation of this solver lies in the fact that each node is calculated one after another. Each step of the above described procedure is performed in a serial manner, with the calculation of new values, obtained through the solving of the discretized equations and timestepping schemes, occurring node by node. The solvers presented in the following subsections aim to improve upon this.

5.1.2 GPU CUDA Solver (or Second Solver)

The concept of serial versus parallel computing was already introduced in chapter 1.2 and its details will not be restated here. This first parallel approach using CUDA and GPGPU computing is similar in nature to the finite difference approach outlined in the previous subsection 5.1.1, with it being able to perform the same simulations while using the same general structure and timestepping scheme, for example. The main difference between them consists of how the calculations are performed. For the CPU-based solver, each point within the grid is iterated over one after the other for each timestep, effectively requiring three nested loops used to iterate over points in all three directions one by one. Meanwhile, the parallel solver presented here performs computations for multiple points simultaneously, by distributing the calculations out across the GPU architecture, as described in

3.2. It is key to split up the computation efficiently for each direction into appropriately sized chunks allocated to a number of threads per block and number of blocks, the exact number of which can be determined using the the number of grid points in the $x-$, $y-$, and $z-$directions with

$$THREADS_i = N_i, \tag{5.1}$$

$$BLOCKS_i = N_j N_k, \tag{5.2}$$

where i is the direction of interest and j and k are the two other directions. Memory is allocated on both the CPU- and GPU-side by the CPU before the computation begins and the required data must be copied over from the host to the device before the full computation is started and from the device to the host once it is completed. Provided suitable hardware is used, this move from many nested loops to a parallel implementation is expected to produce a significant speedup in the time taken to perform the complete simulation.

The implementation of this solver can be summarised as follows: i) the memory is allocated on the host and device through the CPU, through *malloc()* on the host and *cudaMalloc()* on the device, ii) then the initial state of the system is defined, after which, iii) the data required for computation is copied from the host to the device, iv) for each timestep the RK3 scheme is employed to solve the discretized equations to find the result of the heat equation for a corresponding node, v) data to be recorded is copied back from the device to the host, vi) before finally the data is saved and memory is freed again.

Algorithm 6 Pseudo code of function used for second derivative by second solver.

```
Function deriv2x(rhs, phi, x, globalIdx, i):
    d2x = 1 / ((x[1] - x[0]) * (x[1] - x[0]))
    __shared__ phiBound[Nx + stencil * 2]
    phiBound[i + stencil] = phi[globalIdx]


    /* Periodic boundary conditions                          */
    if i < stencil then
        phiBound[i] = phiBound[i + Nx]
        phiBound[Nx + stencil + i] = phiBound[i + stencil]

    temp = (coeffS[stencil] * phiBound[i + stencil]) * d2x
    for idx ← 0 to stencil do
        temp = temp + (coeffS[idx] * (phiBound[i + idx] + phiBound[i + sten-
          cil * 2 - idx])) * d2x
    rhs[globalIdx] = temp;
```

The called function used to calculate the derivative for the GPU-based implementation is similar in structure but was adapted to work with CUDA and is shown in simplified from in algorithm 6 with *deriv2x()* serving as the example for the x-direction. The differences between the same function for the first and second solver are clear, as the number of loops are greatly reduced in favor of using a *globalIdx* to distribute the computation across threads.

5.1.3 Optimized GPU CUDA Solver (or Third Solver)

Finally, a third approach, which is another GPU-based parallel implementation with a focus on minimising data access redundancy, i.e. minimising the amount of times a data point is loaded into memory, was investigated. This approach is based on an implementation of Micikevicius of Nvidia [28]. The core idea of this optimized approach is to streamline the use of memory bandwidth, and therefore improve total computational throughput, by reducing memory data access between iterations to remove this bottleneck as far as possible. This is done through improving memory access redundancy, which is defined as the ratio of elements loaded into memory for the current computation to the number of elements used for the computation.

Given a stencil of size k (i.e. for an 8th order stencil $k = 8$), as well as output tiles of dimension nxm, then the number of elements required of a shared memory array needed to accommodate the entirety of the data, can be calculated by $(n+k)(m+k)$. Since halo elements are read by at least two threadblocks, the read redundancy, or memory access redundancy (MAR), of loading the input data into shared memory arrays can be obtained through

$$MAR = \frac{nm + kn + km}{nm}, \tag{5.3}$$

while the redundancy for writing is simply one in this instance. Further, unlike the simpler 1D implementation used in the previous approach, a 2D grouping is used for threadblocks to effectively match the slice of data in memory, assigning one thread to each node of the tile.

This implementation also utilizes a stencil of the 8th order, though it is then used in 3D. This type of stencil is displayed in figure 5.1, where memory redundancy is visualised, with different colors indicating which elements in a stencil are only accessed by a single unique thread (typically found along the slowest direction), while those accessed by multiple threads are indicated in white and are found exclusively along the faster x- and y-directions. Here, only four threads are sufficient to cover the entirety of the two-dimensional slice of the visualized elements, while the elements along the slower z-axis would be stored in the corresponding thread's registers. Threads of a given threadblock coherently traverse the volume along the slowest direction, computing the output for each slice. Most elements in the current slice are used for computation by more than one thread, while elements in the slices preceding and succeeding the current z-position are used only by the threads corresponding to the elements' position in the x- and y-directions. So, four different threads would access all 32 elements in the 2D slice in shared memory, while the elements along the z-direction would be stored in corresponding thread's registers. When an entire threadblock's threads complete their computation and finish their write operation a shift occurs for the values in the registers, reading in a new element at distance $k/2+1$, where the registers and shared memory are used as a queue. The previous, naive GPU approach loads an entire stencil worth of new data points into memory for every single computed output value.

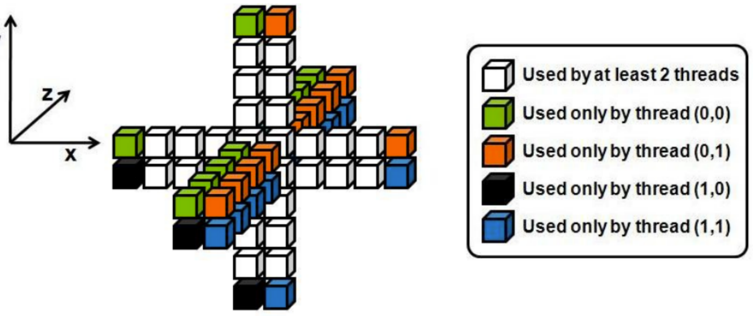

y
z
x
Used by at least 2 threads
Used only by thread (0,0)
Used only by thread (0,1)
Used only by thread (1,0)
Used only by thread (1,1)

Figure 5.1: Visualisation of 3D stencil with indication of elements without reuse. Figure courtesy of Nvidia [28].

The resulting redundancy achieved for this approach is 3 when loading a 16 by 16 slice into memory with a halo of four to each side, while the redundancy for the previous naive approach is 26. This improvement in memory access redundancy should help reduce a bottleneck of the naive solver and therefore improve overall performance.

As we have established earlier, the real advantage of GPUs is their potential for increased computational throughput compared to CPUs, but in order to achieve this potential we have to eliminate the main bottlenecks, which are usually related to memory read or write operations, wherever possible. This approach makes increased use of the faster shared memory, as well as registers, while also minimising data redundancy, with the hope of improving performance compared to the naive parallel implementation described above. The performance of both approaches will be compared against one another through a common benchmark in chapter 6 to study whether the increased complexity of this approach yields an increase in performance at all through alleviating some of the bottlenecks encountered in the naive implementation and whether these improvements are worth the extra effort needed to omit using a standard finite difference approach in favor of this optimized version. It should be noted that only the memory loads and writes during the computation are optimized here and that the initial memory allocation and copying process for this approach is identical to the that outlined for the other parallel method in the previous subsection.

Algorithm 7 Pseudo code of functions used for third solver.

```
Function kernelFD(rhs, phi, x, globalIdx, i):
    /* Advance local values (or slice), move threadfront forward in registers */
    behind4 = behind3
    behind3 = behind2
    behind2 = behind1
    behind1 = current
    current = front1
    front1 = front2
    front2 = front3
    front3 = front4
    front4 = input[inIdx]                         // Only global memory access
    stride = Nx*Ny

    /* Update indexes for global memory accesses */
    inIdx += stride
    outIdx += stride
    /* Check if top or bottom halo */
    if $threadIdx.y < radius$ then
        slice[ty][tx-1] = input[outIdx – radius * Nx]
        slice[ty][tx+1] = input[outIdx + 16 * Nx]
    /* Check if left or right halo */
    if $threadIdx.x < radius$ then
        slice[ty][threadIdx.x] = input[outIdx – radius]
        slice[ty][threadIdx.x+16+radius] = input[outIdx + 16]

    /* 16x16 internal data slice, update in shared memory so neighbors can use data */
    slice[ty][tx] = current

    div = coeffS[0] * current * d2x
    /* In computation two values from registers, four from shared memory */
    div += coeffS[1] * (front1 + behind1 + slice[ty-1][tx]+ slice[ty+1][tx]+
     slice[ty][tx-1]+ slice[ty][tx+1]) * d2x
    div += coeffS[2] * (front2 + behind2 + slice[ty-2][tx]+ slice[ty+2][tx]+
     slice[ty][tx-2]+ slice[ty][tx+2]) * d2x
    div += coeffS[3] * (front3 + behind3 + slice[ty-3][tx]+ slice[ty+3][tx]+
     slice[ty][tx-3]+ slice[ty][tx+3]) * d2x
    div += coeffS[4] * (front4 + behind4 + slice[ty-4][tx]+ slice[ty+4][tx]+
     slice[ty][tx-4]+ slice[ty][tx+4]) * d2x

    dphi[outIdx] = div
```

A simplified pseudo code implementation of the main kernel of this solver is given in algorithm 7.

5.1.4 Similarities and Differences Between Approaches

The most obvious difference between the solvers is the used hardware and the language they were implemented in, with the first solver being written in C++ due to running on a CPU and the second and third solvers were written in CUDA due to running on a GPU. The CPU and naive GPU solvers both use a 1D stencil, while the optimized GPU approach uses a 3D stencil for its computations.

All three of the approaches to be presented utilize RK3 as the timestepping scheme of choice, due to it striking the balance between accuracy, stability, and performance for our purposes. This also makes it easier to compare the performance between the methods, as this is kept constant, reducing variation. Further, all three solvers use the same finite difference approach for spatial discretization, with an 8th-order scheme being utilized.

5.2 VALIDATION WITH BENCHMARK SCENARIO

In order to validate the previously introduced solvers against one another, a single scenario will be simulated and the results will be compared. To do this, an initial condition in the form of a Gaussian function will be imposed on the volume and the corresponding values along a single axis will be studied. This function will diffuse through the volume over time according to the heat equation, which for the y-direction in the 1D case becomes

$$\frac{\partial \phi}{\partial t} = \alpha \frac{\partial^2 \phi}{\partial y^2}, \tag{5.4}$$

eventually resulting in a final state with a constant value for ϕ dependent on the initial condition along the entirety of the axis of interest, when no Dirichlet boundary condition is applied. The chosen initial Gaussian function has the form of

$$\phi = a \cdot exp\left(-\frac{(y-b)^2}{2c^2}\right), \tag{5.5}$$

where the coefficients used in the equation were chosen to be $a = 1$, $b = 0.5$, and $c = 0.1$. The chosen boundary conditions were periodic boundary conditions. This initial condition progressively diffuses, finally reaching a steady state equal along the entire axis. The time it takes for this steady state to be reached depends on the thermal diffusivity, α from equation 6.1, which describes the rate of diffusion through the volume.

5.2.1 Validation of Serial Implementation

First, the serial CPU solver, or first solver, was validated using the scenario described above. To start, at $t = 0$ the initial condition is applied, as is shown by the dotted line in figure 5.2. Diffusion occurs as time progresses, leading to a reduction at the grid points where there previously were higher values of ϕ compared to the average and an increase at the grid points where there previously was a smaller than average value, as the all values approach the steady state value. This transition state can be seen with the dash-dot line of figure 5.2 at 500 times steps of 0.000488 s each. Finally, after, in this case, 8000 total timesteps, a steady state is reached as expected, where all grid points have the same value of ϕ, as is shown by the flat dashed line of figure 5.2.

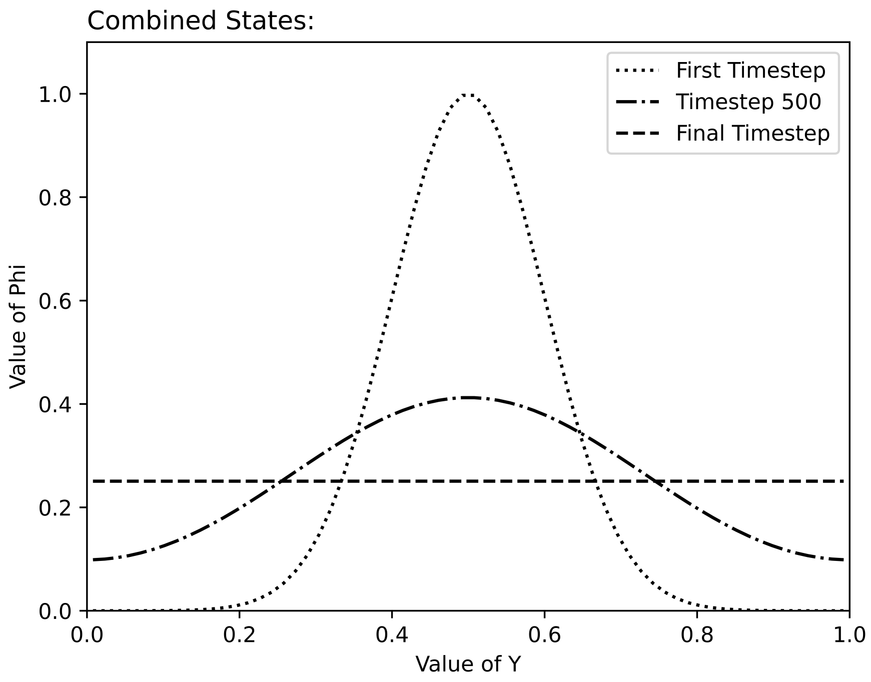


Figure 5.2: Visualisation for validation of CPU solver, or first solver, of all three chosen states.

5.2.2 Validation of Parallel Implementation

The parallel GPU solver, or second solver, was also validated with the same method described above. Similarly, at $t = 0$ the initial condition is applied, as can be seen with the dotted line in figure 5.3. Diffusion occurs as time progresses, leading to a reduction at the grid points where the previous values of ϕ were above the average value and an increase at the grid points where there previously was a smaller than average value, as the all values approach the steady state value across the volume. This transition state can be seen with the dash-dotted line of figure 5.3 at 500 times steps of 0.000488 s each. Finally, after, in this case, 8000 total timesteps, a steady state is reached as expected, where all grid points have the same value of ϕ, as can be seen with the flat dashed line of figure 5.3.

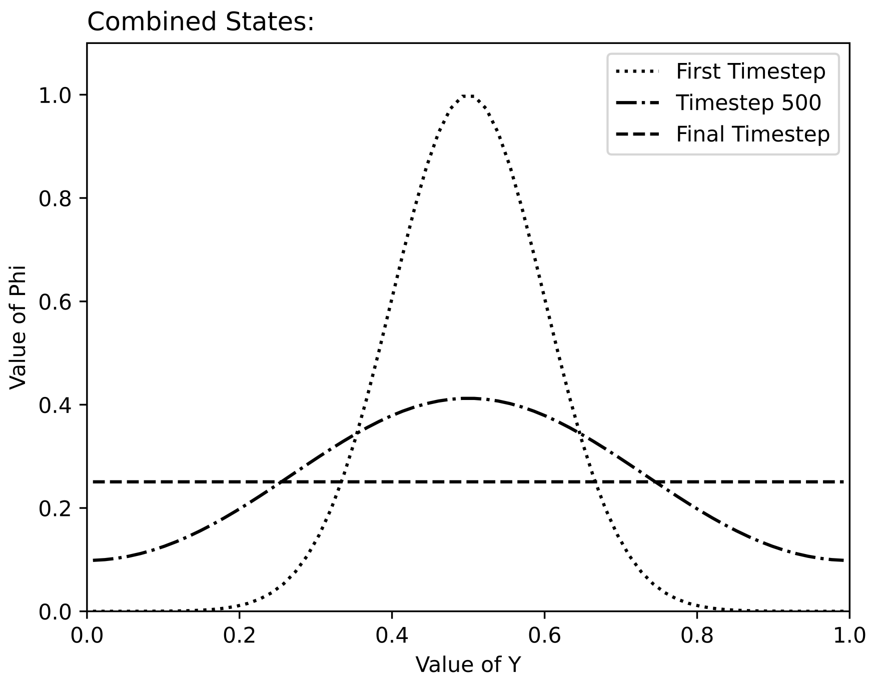


Figure 5.3: Visualisation for validation of GPU solver, or second solver, of all three chosen states.

5.2.3 Validation of Parallel Implementation with Redundancy

The optimized parallel GPU solver, or third solver, was also validated with the method described above. To start, at $t = 0$ the initial condition is applied, as can be seen with the dotted line in figure 5.4. Much like for the previous two solvers' validation, diffusion occurs as time progresses, leading to a reduction at the grid points where previous values of ϕ were above the average and an increase at the grid points where there previously was a smaller than average value, as the all values approach the steady state value. This transition state is shown by the dash-dotted line of figure 5.4 at 500 times steps of 0.000488 s each. Finally, after, in this case, 8000 total timesteps, a steady state is reached as expected, where all grid points have the same value of ϕ, as can be seen by the flat dashed line of figure 5.4.

5.2.4 Verifying Validation Across Solvers

Finally, the results at the three different previously defined states were compared across solvers. The purpose of this is to verify that each solver delivers the same expected results accurately for every time step. This cross-verification can be seen in figure 5.5, where the corresponding data is overlaid, with the data for the first solver taking the form of the previously used dotted, dash-dotted, and dashed lines, while the data for the second and third solvers is overlaid at the same timesteps using blue and white dots, respectively.

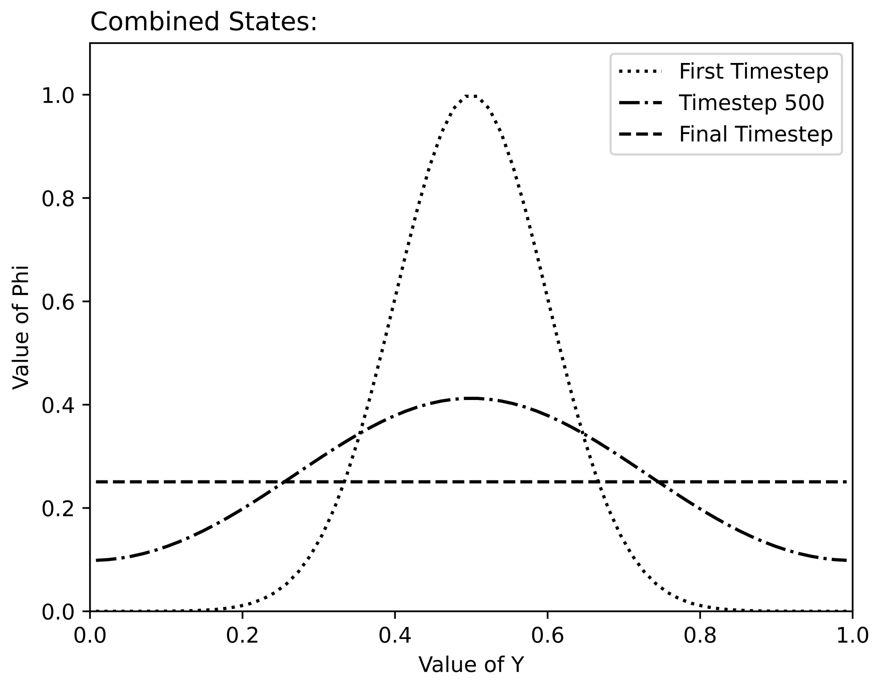


Figure 5.4: Visualisation for validation of optimized GPU solver, or third solver, of all three chosen states.

5.3 VALIDATION WITH ANALYTICAL SOLUTION

In order to validate the previously introduced solvers not just against one another but also against an analytical solution, a single scenario will be simulated and the outputs will be compared for which the solution is known. To do this, an initial condition will be imposed on the volume, as well as a constant set of boundary conditions. The initial condition is $\phi = 0$ at all grid points and the imposed set of boundary conditions are of the Dirichlet variety and can be stated as

$$\phi = \begin{cases} 1 & at\ y = 0, \\ 0 & at\ y > 0. \end{cases} \tag{5.6}$$

Then, the corresponding values along a single axis were be studied, similarly to the prior validation method, with the y-axis being used again. As time progresses, the value imposed at the boundary condition will lead to diffusion through our area of interest, according to the heat equation, which for the y-direction was stated above in equation 5.4.

To validate the simulations performed for the three solvers, we compare our obtained results to the analytical solution of this problem, which, according to Slingerland and Kump [41], can be stated as

$$\phi = \phi_0 \left[\sum_{n=0}^{\infty} erfc\,(2n\eta_1 + \eta) - \sum_{n=0}^{\infty} erfc\,(2(n+1)\eta_1 + \eta) \right], \tag{5.7}$$

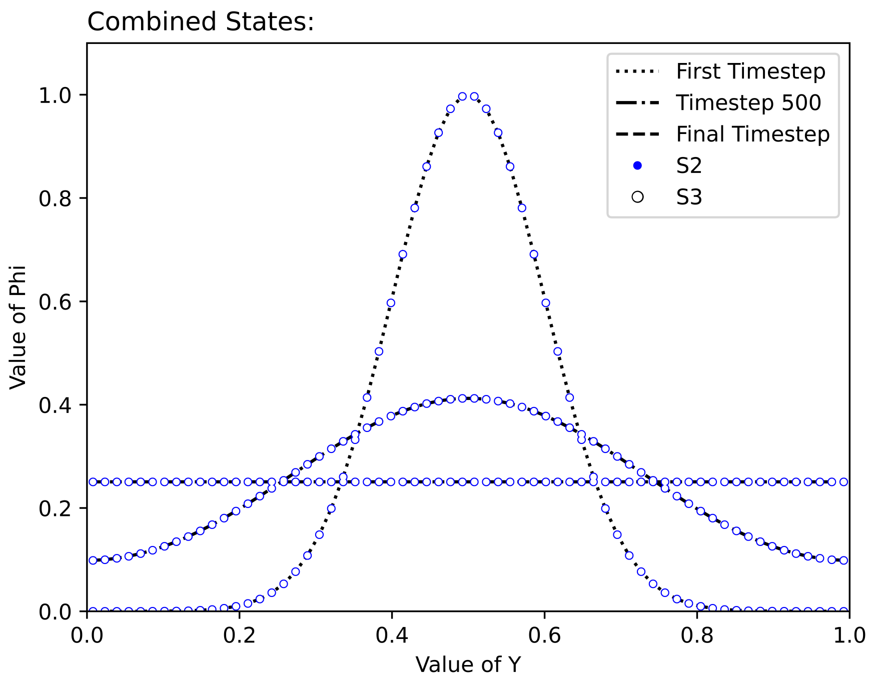


Figure 5.5: Validation for first (S1), second (S2), and third (S3) solver for the first, 500th, and final timestep. S1 is shown with lines, S2 with solid blue circles, and S3 with solid white circles. The circles of S2 and S3 can be observed to overlap with each other and the lines of S1.

where ϕ_0 is value of the imposed boundary condition. Here, $erfc$ is the complementary error function for each element of z and is defined by

$$erfc(z) = 1 - erf(z), \tag{5.8}$$

with erf being the error function. The complementary error function can further be stated as

$$erfc(z) = \frac{2}{\sqrt{\pi}} \int_{z}^{\infty} e^{-t^2} dt. \tag{5.9}$$

Further, η_1 and η in equation 5.7 above can be substituted for

$$\eta_1 = \frac{h}{2\sqrt{\alpha t}}, \tag{5.10}$$

$$\eta = \frac{y}{2\sqrt{\alpha t}}, \tag{5.11}$$

where h is the length scale of the problem and y is the positions of grid point along the y-axis.

5.3.1 Validation of Serial Implementation with Analytical Solution

First, the serial CPU implementation was compared with the analytical solution described above in order to validate this particular solver. To do this, the results of the implementation were compared to the analytical solution at three points in time: at the start of the simulation ($t = 0$) when only the initial condition is present, after 500 timesteps of 0.000122 seconds, a transition state between the initial and final solution, and finally at a large number of timesteps which guarantees that the final steady state of the problem has been reached, in this case 8000 timesteps. In figure 5.6 the simulation results represented by lines with a color corresponding to the time of the simulation the values were taken from are overlaid on the analytical solution, here represented as black dots.

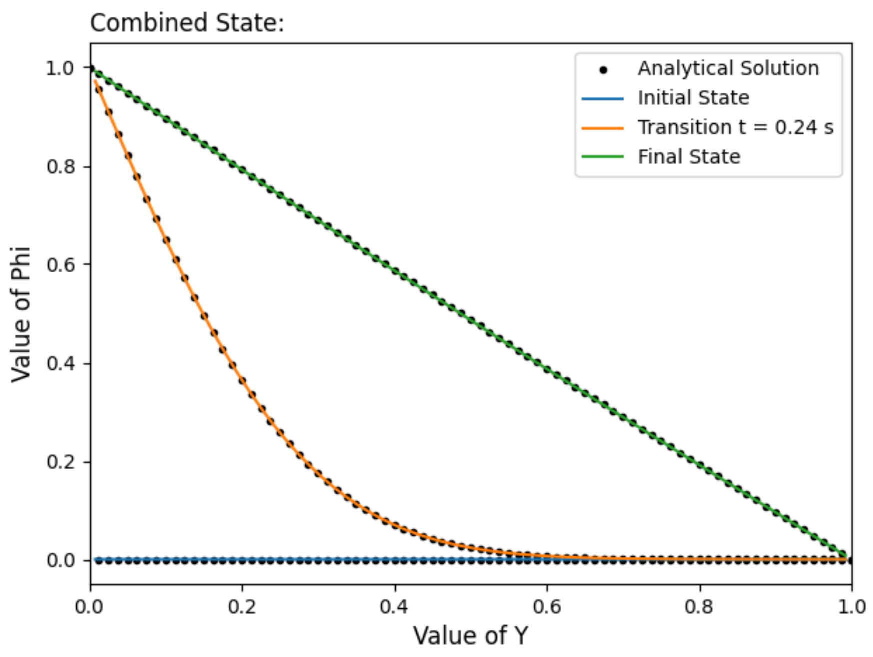


Figure 5.6: Visualisation for validation of CPU solver, or first solver, of all states against analytical solution.

5.3.2 Validation of Parallel Implementation with Analytical Solution

Subsequently, the parallel GPU implementation was also compared with the same analytical solution described above in order to validate the second solver. To do this, the results of the implementation were compared to the analytical solution at three points in time, the same three points that were chosen for the validation of the previous solver. In figure 5.7 the simulation results represented by lines with a color corresponding to the time of the simulation the values were taken from are overlaid on the analytical solution, here represented as black dots.

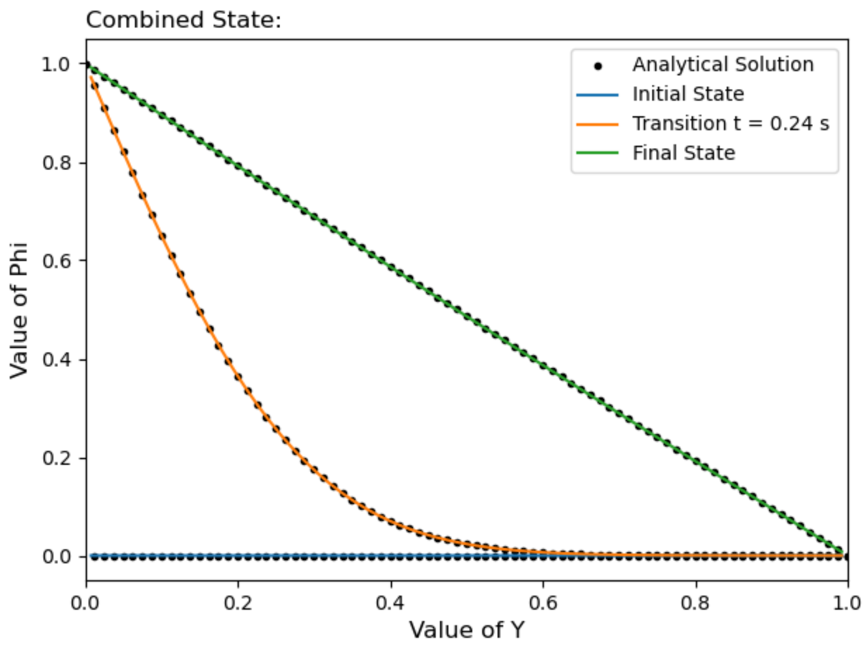


Figure 5.7: Visualisation for validation of GPU solver, or second solver, of all states against analytical solution.

5.3.3 Validation of Optimized Parallel Implementation with Analytical Solution

Finally, the optimized parallel GPU implementation was compared with the same analytical solution described above in order to validate the solver. To do this, the results of the implementation were compared to the analytical solution at three points in time, the same three points that were chosen for the validation of the previous two solvers. In figure 5.8 the simulation results represented by lines with a color corresponding to the time of the simulation the values were taken from are overlaid on the analytical solution, here represented as black dots. Just like for the two previous solvers, a match between the two sets of results can be seen at the different chosen timesteps and therefore the solvers can be considered to be fully validated.

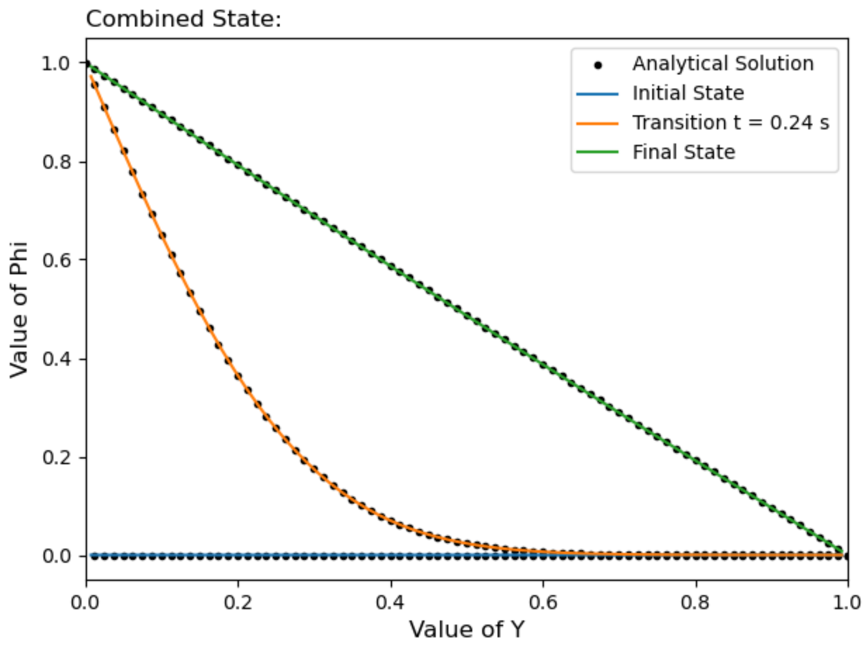


Figure 5.8: Visualisation for validation of optimized GPU solver, or third solver, of all states against analytical solution.

6 BENCHMARK RESULTS AND COMPARISONS

This chapter proposes a common scenario to be used as a benchmark to which the previously introduced and validated solvers can be applied. Further, the measured wall clock time to fully simulate this outlined scenario for each solver is investigated to yield a statistically valid overview of performance. Once performance has been determined for each solver individually, comparisons will be drawn between them. These comparisons between performances are also drawn in the context of the used hardware to give a more complete picture. The goal is to determine if a speedup can be achieved and, should this be the case, how significant such a speedup would be and if it justifies the undertaken optimizations and increased spending on additional hardware.

6.1 BENCHMARK CASE

The chosen test case is used for all simulations performed in this chapter and is based on a simple geometry and a set of boundary and initial conditions to be outlined in the following subsections. The chosen geometry is covered in an equidistant grid with an equal number of gird points in each direction, which can be altered to simulate more demanding meshes for benchmarking purposes.

6.1.1 Geometry

Due to the emphasis of this work being the study of solver performance, a simple geometry was chosen. The geometry for the used scenario, which is used across all solvers, consists of a shape with three equal lengths in each direction for each side, $L_x = L_y = L_z = 1$, resulting in a three-dimensional object bounded by six square faces, with three of these meeting at each vertex, or as it is more commonly known: a cube.

6.1.2 Grid

The chosen scenario was benchmarked on a variety of grid sizes to observe how the speedup achieved between solvers differed for number of grid points. Here, the expectation was that the speedup should increase as the gird size used for the computations increases, since the main advantage of high throughput GPU-based solvers lies in the ability to perform the

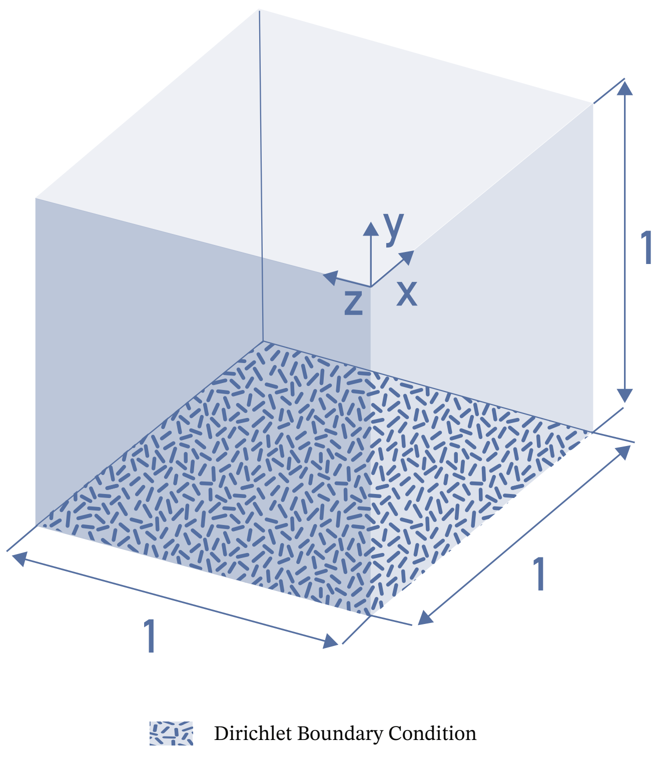


Figure 6.1: Chosen geometry including measurements and location of applied Dirichlet boundary condition. Figure courtesy of Malu Kawasaki.

iterations of the simulation more quickly, while the main drawback is the increased time required to transfer the data between host and device. So, as the time spend on computation increases, the speedup factor should increase as well. All chosen grids employed an equal number of grid points in each direction to give equidistant coverage of the above outlined geometry. The number of grid points in each direction will always be of the form 2^n to better align with architecture constrains outlined in chapter 3, with chosen values being, for example, 64 or 256 points in each direction, which would result in a total number of 262,144 or 16,777,216 grid points for the three-dimensional problem space, respectively. The grid points found at $index = 0$ and $index = N_k$, for direction k, represent the boundary of the problem, to which appropriate conditions may be applied.

6.1.3 Initial and Boundary Conditions

The three-dimensional test case makes use of different conditions, either applied at the beginning of a simulation or for all simulated timesteps. At time $t = 0$, before the first iteration of the solver is performed, the chosen initial condition applied to the test case is defined by the equation

$$\phi = sin(\pi x)\, sin(\pi y)\, sin(\pi z), \tag{6.1}$$

for all values of x, y, and z within the geometry. Starting from the first iteration performed by the respective solver, two different kinds of boundary condition were used for this test case, one being of the Dirichlet variety, imposed at $y = 0$ and carrying a magnitude of $\phi = 1$, with periodic boundary conditions used at all surfaces of the geometry beyond that. Over time, the initial condition becomes less dominant in determining the observed profile of ϕ, as the impact of the imposed boundary condition starts diffusing through the medium. Here, the observed profile will be composed of the remnants of the initial condition, which will be mixed with the uniform profile diffusing from the imposed boundary condition. Eventually, a steady state will be reached, which will be made up of a profile that is determined entirely by the distance of the node of interest from the imposed Dirichlet boundary condition. Visually, this results in a smooth gradient profile, steadily decreasing in magnitude away from $y = 0$.

6.1.4 Chosen Parameters

For all simulations performed in this chapter, the values presented in table 6.1 are used throughout. A key problem to keep in mind when running the test case is the choice of an appropriate timestep size, in order to ensure stability in the calculations during simulation. The time step was calculated as the result of the chosen Courant–Friedrichs–Lewy (CFL) number multiplied by the distance between the grid spacing between two adjacent points in any direction, which ensures that the resulting timestep size scales with the chosen fineness of the grid.

Used Parameters			
Distance in x (L_x)	1	Boundary Condition in x	Periodic
Distance in y (L_y)	1	Boundary Condition in y	Dirichlet
Distance in z (L_z)	1	Boundary Condition in z	Periodic
Thermal diffusivity α	0.1	Number of timesteps n	2000
CFL number	0.4	Size of *stencil*	4

Table 6.1: Common parameters used across all three solvers.

Other parameters will be changed directly, such as the number of grid points in each direction, and their chosen value will be indicated where appropriate. Further, additional parameters will change as a result of another change, as their exact values are depended on some other parameter which is not identical for each simulation, such as the timestep size dt.

6.1.5 Context of Test Case and Performance Analysis

This test case relates to real world scenarios where heat transfer or diffusion occurs and the medium moves from an initial state to a new steady state. Here, the initial state, which wanes over time, and the imposed Dirichlet boundary condition at $y = 0$ which introduces heat into the volume both approximate plausible scenarios one might encounter in the real world applications where a medium without an internal source is heated from below.

Having a strong understanding of the required computational power and expected scaling performance will allow researchers to accurately assess their needs and possibilities when performing similar simulations to the same or greater level of accuracy. Further, by exploring the possible optimization by using a specialized solver it is possible for researchers to gauge whether spending an extended period of time optimizing elements of their own solvers when performing lengthy computations will be worth it.

6.2 HARDWARE

This section will outline all the hardware used to benchmark the solvers, including individual specifications. The CPU-based and GPU-based solvers will be run on their respective workstations, as outlined in table 6.2. This table is intended to guide the reader with respect to the performance of the hardware each solver is run on to make later drawn comparisons more meaningful, as differences in hardware specification are a given when comparing CPU- and GPU-based solvers. As previously explained in chapter 3, a large discrepancy between number of cores on a CPU and GPU can be observed, as well as a difference in achieved processor frequency. It should be noted that the GPU-related hardware on the CPU workstation is not used during computations and is just listed for completeness sake.

Used Hardware Specifications		
Component	CPU Workstation	GPU Workstation
CPU	Intel Core i7-9750H	Intel Core i5-9600K
CPU Frequency	2.6 GHz Base & 4.50 GHz Boost	3.70 GHz Base & 4.4 GHz Boost
CPU Cores	6	6
RAM	16 GB 2666 MHz DDR4	16 GB 3000 MHz DDR4
GPU	AMD Radeon Pro 5300M	Nvidia GeForce RTX 3070
GPU Frequency	1 GHz Base & 1.25 GHz Boost	1.50 GHz Base & 1.73 GHz Boost
GPU Cuda Cores	-	5888
GPU Architecture	Navi RDNA	Ampere
GPU Memory	4 GB GDDR6	8 GB GDDR6
GPU Bandwidth	192 GB/s	448 GB/s
CUDA Version	-	11.2
C++ Compiler	G++ 9.2.0	G++ 9.2.0

Table 6.2: Details of used hardware for different workstations. It should be noted that the GPU information for the CPU workstation is stated for completeness only.

6.3 A NOTE ON PERFORMANCE

There are two main indicators of performance which will be assessed here to understand whether there is an actual improvement and whether moving from a serial CPU implementation to a parallel GPU implementation or to a more optimized GPU implementation can be justified. The first being throughput efficiency, which is the time it takes to complete an iteration. It is clearly favourable to reduce the time per iteration for any solver, so we will compare the time to complete the chosen number of iterations across solvers and quantify the achieved improvements. Secondly, assessing the scalability of the new approach is key to understanding where a solver can bring the greatest increases in performance. Scalability is the expected increase in performance as more computational power is added on the hardware side. Scalability is usually measured for a single piece of hardware, as more pieces of additional hardware are added, such as in a Multi-GPU computational cluster. There is no straightforward way to compare scalability when different types of hardware are involved, as is the case here, so to compare the solvers we shall take their price points into consideration to get a estimate of how performance scales with cost. The CPU used for the serial solver individually retails for $395, while the used GPU retails for $499. These individual retail prices of the main pieces of hardware used for computation are obviously not everything that should be considered concerning hardware, as a GPU still relies on its host, a CPU, to perform some parts of the simulations required and beyond this a CPU still requires RAM and other components to function. Further, the specifications of these additional components can impact the achievable computational throughput, as they may bottleneck the computation. Thus, comparing individual components and their price points is not an exact science but brings us an additional piece of the puzzle needed to draw conclusions on the effectiveness of accelerating a simulation using a GPU and optimized solvers.

6.3.1 Amdahl's Law

A more formal way of assessing increases in performance is Amdahl's Law [2]. This gives the theoretical speedup, S_P, which is achievable when comparing the performance of a serial implementation to a parallel implementation of the same algorithm. Fundamentally, it aims to compare the execution time of the best possible serial implementation of the algorithm, t_S, to the execution time of the best possible parallel implementation of the algorithm, t_P, which in idealised form can be summarised as

$$S_P = \frac{t_S}{t_P}. \tag{6.2}$$

There exists a theoretical limit to increasing S_P, as the possible increase in throughput efficiency of a parallel system is limited by the time needed for the serial portion of the program to execute, assuming such a portion exists. Assuming that there are no memory reads and writes or any other practical hardware limitations, we can define the fraction executed in parallel as f and further assume P processing units are used to speed up the computation, then the speedup is given by

$$S_P = \frac{t_S}{f\frac{t_1}{P} + (1-f)\,t_S} = \frac{1}{\frac{f}{P} + (1-f)}. \tag{6.3}$$

When considering the fact that most parallel applications have a significant serial part in their computations, as do the ones outlined in this work, then equation 6.3 implies that the achievable speed up is limited by f and utilising more processing units will get us closer to that limit, though the efficiency will drop with each processor added. For example, if the fraction of the computation we can parallelize is 0.95, then the maximal possible speed up is 20 and each added processor gets us closer to approaching this limit.

If we assume an idealised problem, then equation 6.3 can be expressed as

$$S_P \leq \frac{1}{1-f}, \tag{6.4}$$

indicating that the possible speedup is merely a function of the fraction of the problem that is possible to compute in parallel.

Another view on this problem was put forward by Gustafson [16], who argued that the sequential fraction of a problem decreases as problem size increases. Therefore parallel computing is most suitable for solving problems of increasing size.

6.4 RESULTS OF SIMULATIONS USING SERIAL IMPLEMENTATION

Before considering any GPU-based implementations, we will consider the CPU-based serial implementation to establish a baseline for performance we can compare the following results to. This serial solver was used to simulate the test case elaborated on above and was run on the CPU workbench, the details of which are given in table 6.2. For the simulation, initially a grid of 128 by 128 by 128 grid points was chosen, resulting in a timestep size of 0.000122 s. The initial and boundary conditions introduced in section 6.1.3 are used. The results of these performed simulations are visualized in figures 6.2 - 6.5, at four distinct points in time, at the initial condition, at one third of the way to the final timestep or timestep 660, at one third of the way to the final timestep, namely timestep 1320, and at the final timestep 2000. This spacing between timesteps was chosen to visualize to the reader the progress of diffusion which occurs during the simulation as the initial condition ceases to be the dominant force determining the profile of ϕ in favor of the applied boundary condition.

Figure 6.2 shows the volume sliced in half along the y-axis, where the value of ϕ across the volume is entirely determined by the initial condition, resulting in a spherical profile with the maximum value of 1 being found at the centre of the cube. Here ϕ decreases as the observer moves away from the center of the cube in the x-, y-, or z-direction, reaching a value of 0 at the boundary. After 660 timesteps the initial profile has begun defusing outward, as can be seen in figure 6.3, resulting in the value observed at the center of the cube decreasing below 1. Further, the Dirichlet boundary condition has begun impacting the profile of ϕ in a noticeable way, as diffusion away from $y = 0$ occurs. This trend continues and after 1320 timesteps the initial profile has become less prominent while the applied boundary condition has resulted in more diffusion occurring along the y-axis, resulting in a smoother transition in the observed profile from the bottom of the cube towards the center. This is visualized in figure 6.4. Finally, the final timestep is reached, which can be seen in figure 6.5, where the initial profile is barely still visible and the observable profile is dominated by the Dirichlet boundary condition.

6.5 RESULTS OF SIMULATIONS USING PARALLEL IMPLEMENTATION

The parallel GPU-based solver was run on the GPU workbench, the details of which are given in table 6.2. For the simulations, initially a grid of 128 by 128 by 128 grid points was chosen, resulting in a timestep size of 0.000122 s. The initial and boundary conditions introduced in section 6.1.3 are used. The same behaviour can be observed for this solver as in the simulations of the same test case using the serial CPU solver, as can be seen in the visualization of figures 6.6 - 6.9, with the displayed images representing ϕ at the same timesteps as the ones of the CPU-based solver.

6.6 RESULTS OF SIMULATIONS USING OPTIMIZED PARALLEL IMPLEMENTATION

Finally, the parallel GPU-based solver with memory redundancy was run on the GPU workbench, the details of which are given in table 6.2. Similarly to the previous parallel implementation, initially a grid of 128 by 128 by 128 grid points was also chosen, resulting in a timestep size of 0.000122 s. The initial and boundary conditions introduced in section 6.1.3 are used. Again, the same behaviour can be observed as in the simulations of the same test case using the previous solvers, as can be seen in figures 6.10 - 6.13, with the displayed images representing ϕ at the same timesteps as the ones of the CPU-based solver.

6.7 PERFORMANCE COMPARISON

Multiple comparisons were performed in accordance with the principles outlined in earlier sections of this chapter in order to assess the possible performance gains between solvers. This section aims to contextualise the obtained results as well as visualise them in an intuitive way.

First, the efficiency gains which can be materialised were investigated between solvers, starting with a relatively small gird of 64 nodes in each di-

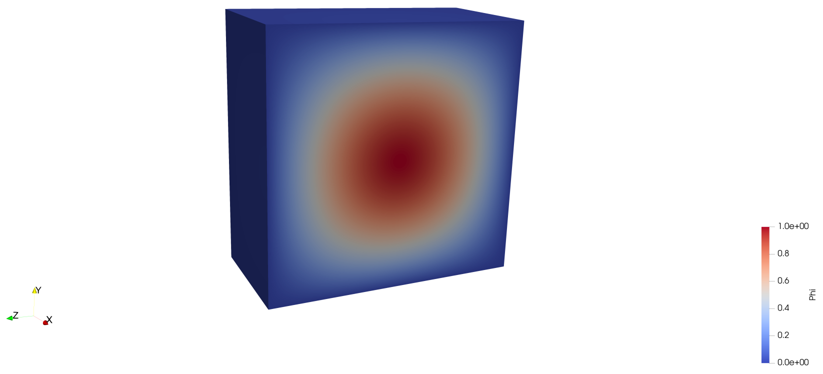


Figure 6.2: CPU solver, or first solver, at initial timestep.

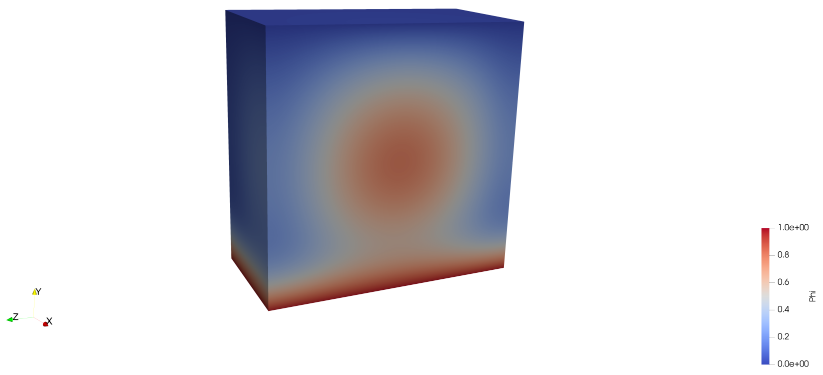


Figure 6.3: CPU solver, or first solver, at timestep 660.

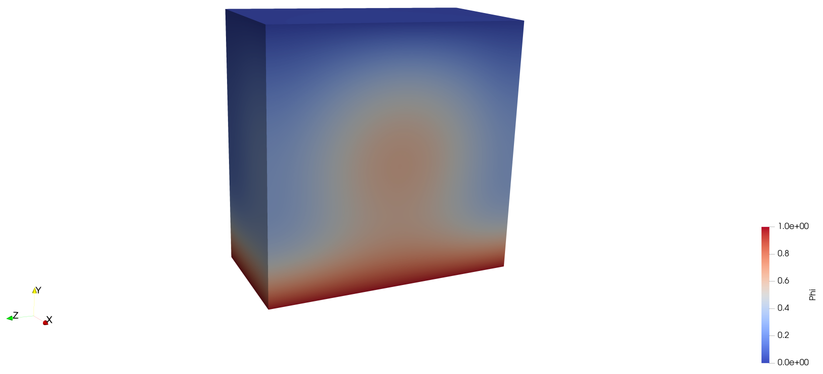


Figure 6.4: CPU solver, or first solver, at timestep 1320.

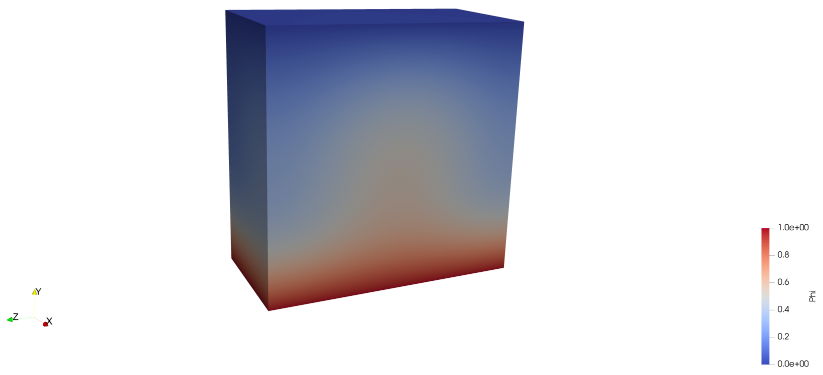


Figure 6.5: CPU solver, or first solver, at timestep 2000.

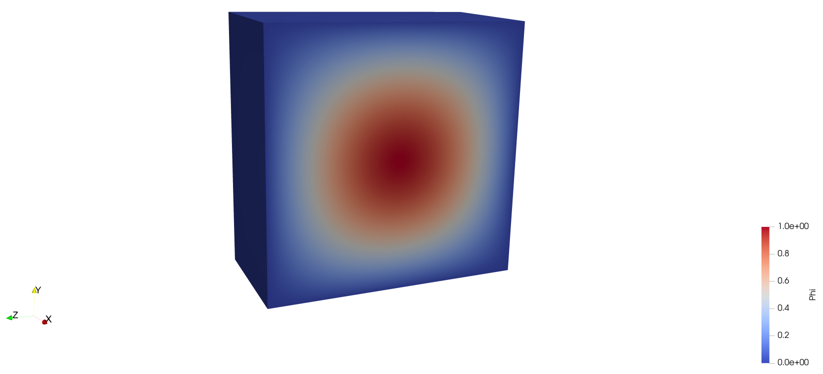


Figure 6.6: GPU solver, or second solver, at initial timestep.

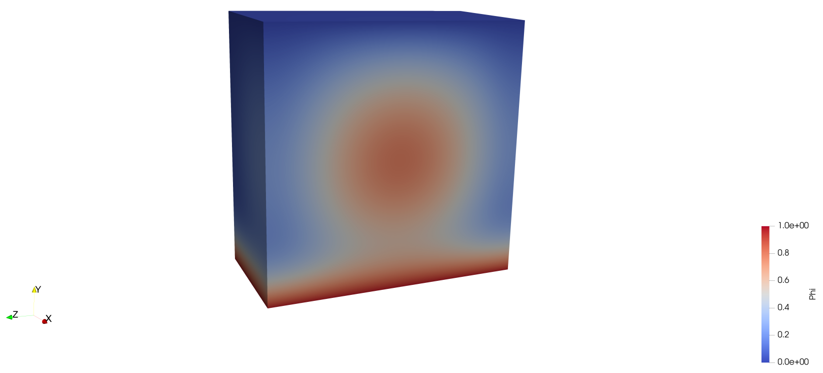


Figure 6.7: GPU solver, or second solver, at timestep 660.

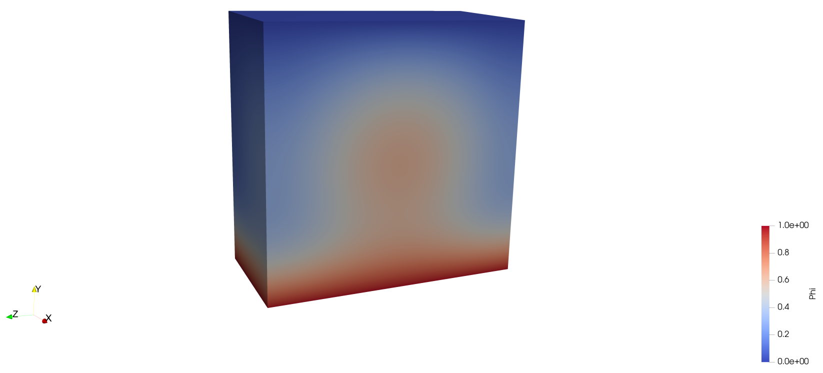


Figure 6.8: GPU solver, or second solver, at timestep 1320.

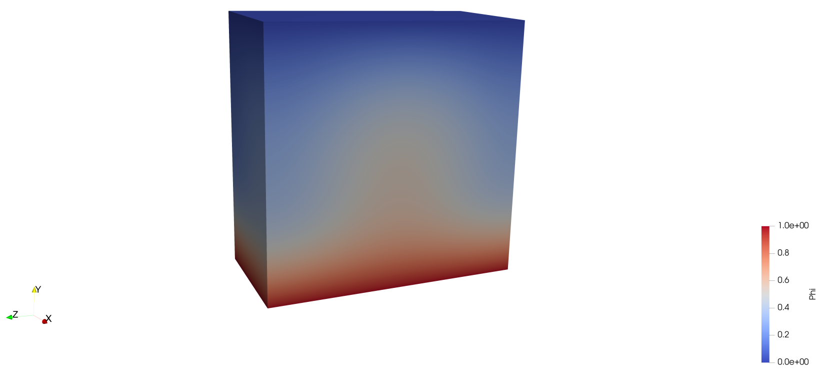


Figure 6.9: GPU solver, or second solver, at timestep 2000.

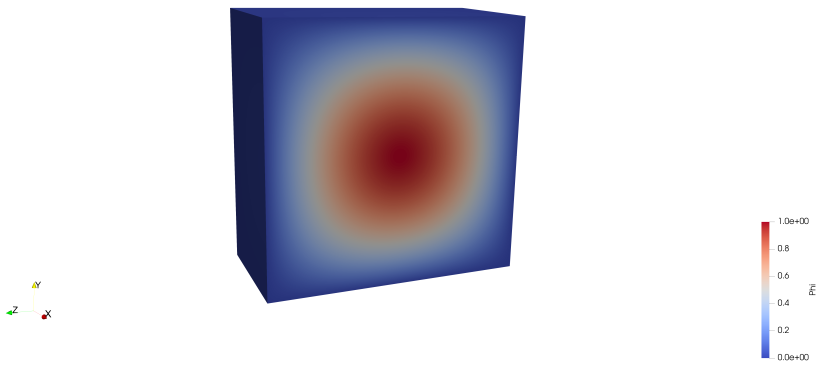


Figure 6.10: Optimized GPU solver, or third solver, at initial timestep.

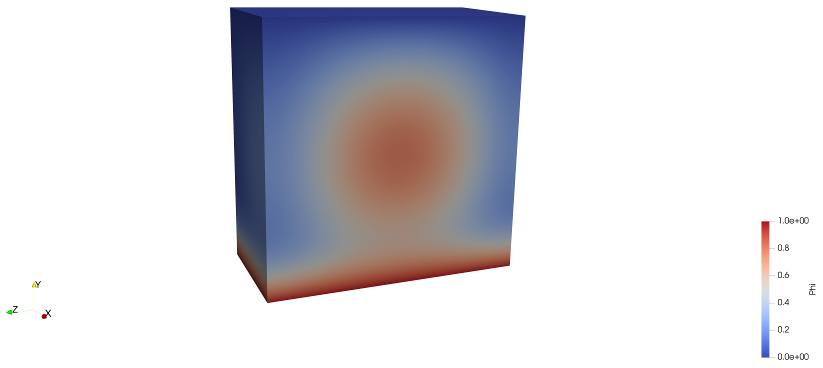


Figure 6.11: Optimized GPU solver, or third solver, at timestep 660.

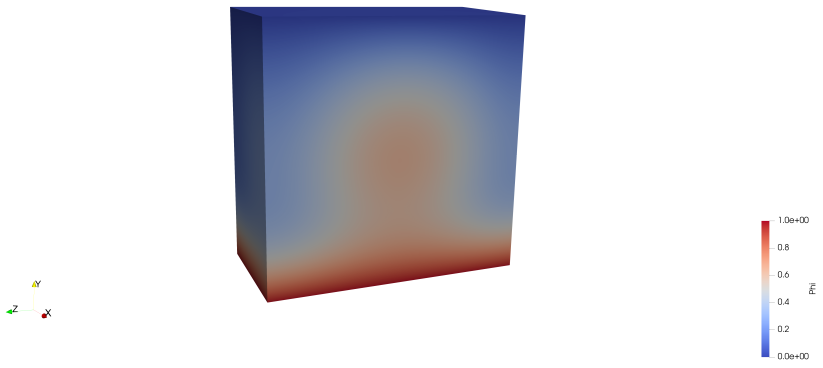


Figure 6.12: Optimized GPU solver, or third solver, at timestep 1320.

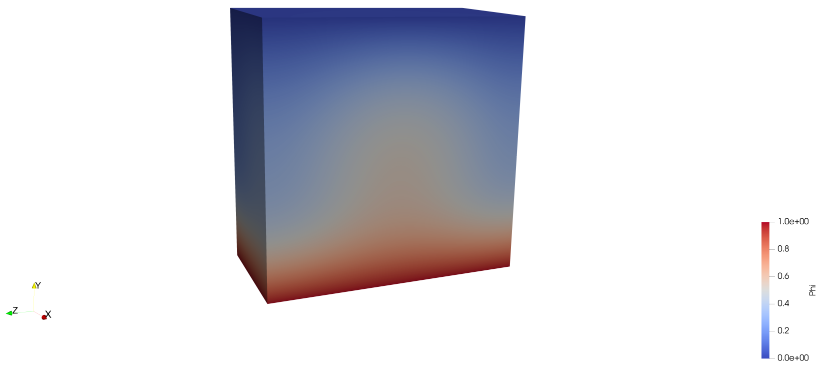


Figure 6.13: Optimized GPU solver, or third solver, at timestep 2000.

rection. Each solver was run ten times on the benchmark scenario outlined in section 6.1 and the resulting mean and standard deviations for the time taken by each solver are plotted in figure 6.14 in seconds for a relatively coarse grid of 64 by 64 by 64 points, with the error bars indicating the standard deviation. The goal of this comparison is to visualise the speedup in execution time achieved between the implementations as throughput efficiency improves. As can be seen, there is speedup of over ten for the execution time between the first and second solver (10.5 to be precise) with the second and third solvers achieving similar times. The encountered standard deviations are 3.67, 2.08, and 2.15, respectively. Recalling Amdahl's Law, this is still a relatively low speedup considering how much of the computation we are able to parallelize. To capture more of that possible upside, we need to focus more on the area where our GPU-based solvers have the advantage: the actual calculation part of our simulations. The less time we spend on anything but computing, the more of an advantage our GPU-based solvers will have. Thus, we should target problems of greater size by increasing the grid size and observing if the difference between the achieved wall clock times of the solvers increases, decreases, or remains the same.

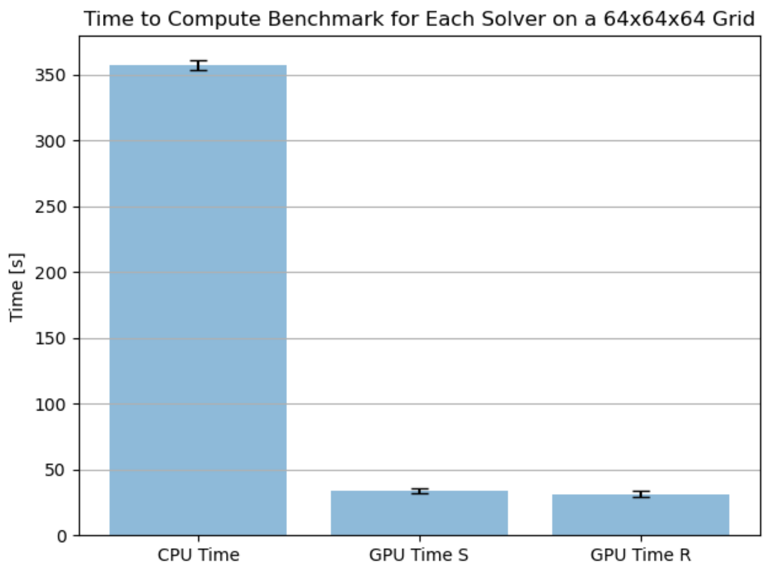


Figure 6.14: Wall clock time and error bar graph for timings of ten runs of each solver for a 64 by 64 by 64 grid.

For a computationally more intensive grid of 128 grid points in each direction the results for all three solvers are shown in figure 6.15. Here, a speedup of 18.3 was achieved between the first and second solver, a significant improvement over the previous more coarse grid, almost doubling the speedup relative to the previous grid. The encountered standard deviations were 5.44, 2.75, and 3.01, respectively. This confirms that as the size of the problem space grows, an increased speedup can be achieved as more time is spend on performing computations. Again, the difference in execution time between the two GPU-solvers was small, amounting to just a few seconds, though the difference was slightly larger than the one seen during simulations of the previous more coarse grid.

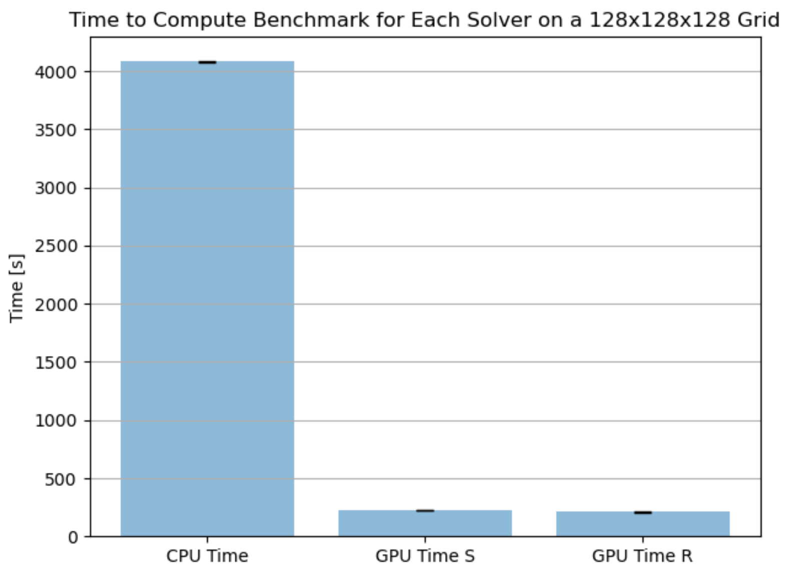


Figure 6.15: Wall clock time and error bar graph for timings of ten runs of each solver for a 128 by 128 by 128 grid.

Observing the drastic increase in achieved speedup between these two grids, one might pose the questions of how performance across all three solvers scales as the used grid size becomes more computationally demanding. Will the speedup continue to increase as we keep increasing the number of grid points used or will we reach a point beyond which the increase will be negligible before stopping completely, as predicted by Amdahl's law? To explore this further, the remainder of this chapter will be split into two parts: a performance scaling comparison between the first and second solver and a comparison between the second and third solver. The comparison between the first and second solver is intended to show the possible speedup gains when comparing two very similar, equally optimized solvers, with one running on a CPU and the other on a GPU. This should give us the clearest understanding of how performance scales with problem size for our particular applications between serial and parallel implementations. Next, two GPU-based solvers with different levels of optimization are compared, namely the second and third solvers. Here, the goal is to see how the achievable gains differ when a more efficient solver is used across different numbers of gird points. So far, the difference in execution times between them was limited, thus we are hoping to see if the optimized of the third solver will be rewarded with an increased speedup relative to the non-optimized second solver as the size of the used grid is increased.

6.7.1 Comparison Between Solver 1 and 2

As previously explained, the second solver is a GPU implementation almost identical in structure and approach to the first solver, which runs on a CPU. Thus a direct performance comparison between the two should show what difference in wall clock time simulating on a GPU really makes.

The achieved speedups, taken by comparing the wall clock time obtained when executing the same simulation, for grid of 64 and 128 grid points in each direction were given in the previous section. Figure 6.16 directly compares the two solvers across a range of grid sizes in order to answer the question posed at the end of the previous section. Here, grid sizes of 32, 64, 128, 256, 512, and 1024 grid points in each direction were considered, covering a wide range range of execution times, from just a few seconds when considering the coarsest grid to taking multiple days when considering the CPU-based solver and the finest grid.

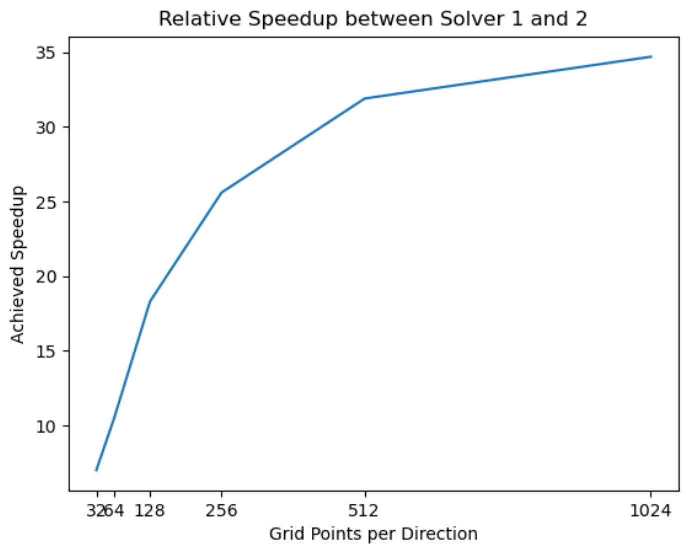


Figure 6.16: Comparison of wall clock speedup between CPU solver and regular GPU solver.

Initially, there are very large increases in achieved speedup between grid sizes, which can be seen in the steepness of the graph, due to moving away from simulations that spend a large portion on copying data between memory and other tasks not directly related to the computations to solve the discretized equations. As the time spend on calculations for each loop increases, the achieved speedup increases as well due to the advantage parallel implementations have over serial ones. Once we get to a grid size of 256 points in each direction there is a slowdown in increase in speedup, as we move from 18.3 to 25.6 when comparing the CPU to the GPU implementation. While speedup does continue to increase between 256 and 512 grid points in each direction, to a value of 31.9, the relative increase levels off further, indicating that we are spending less and less time of the computation on non-parallelizable tasks as the problem size grows and that we are therefore approaching the maximum achievable speedup for our simulation.

Finally, the increase in speedup for the two solvers between the largest two employed grid sizes is relatively moderate, only increasing to 34.7 at 1024 grid points. This indicates that the possible speedup limit is approaching, as the vast majority of the time spend is now focused on computing in parallel. Relative performance gains will be sparse beyond this point.

It should be noted that even thought the change in the achieved speedup between the second largest and largest grids was not that significant in relative terms, due to the long times involved in performing simulations at this point, the increase in total time savings is still very high.

Overall, there is a significant performance increase when parallelizing such a finite difference solver and moving it to appropriate hardware, especially once the problem size grows large enough that a significant part of the simulation is devoted to loops solving the discretized equations.

6.7.2 Comparison Between Solver 2 and 3

The second and third solvers are both GPU-based solvers, with the third solver being more optimized in regards to memory redundancy during computation. A direct performance comparison between the two will allow us to judge whether the optimizations undertaken will yield a measurable benefit, depending on number of used grid points.

Just as in the previous section, the achieved speedup in wall clock time for a chosen number of grid points was studied, which can be seen in figure 6.17. Initially, at a low number of grid points, there is only a slight difference of a few percent in wall clock time between the solvers leading to a speedup of 1.17, as the actual time spent on the simulation is very short and operations such as copying memory from the host to the device still take up a majority of wall clock time.

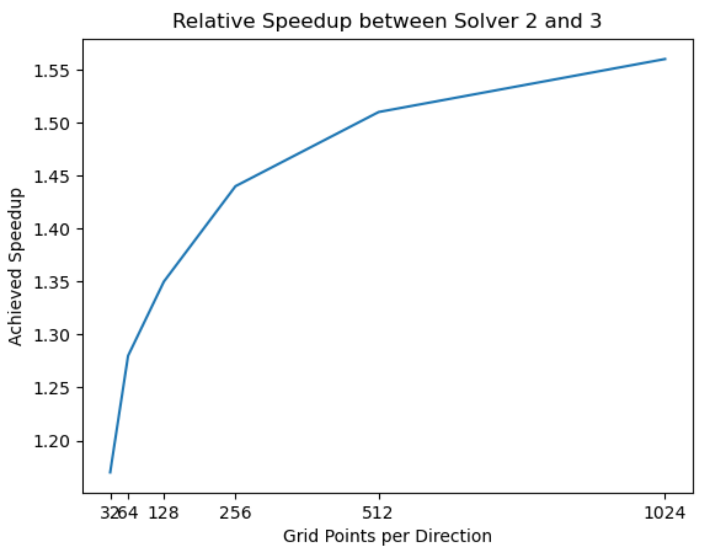


Figure 6.17: Comparison of wall clock speedup between GPU solver and optimized GPU solver.

As computations become more demanding due to an increase in grid size, the optimized solver continues pulling ahead, albeit only approximately a third at 128 grid points. Beyond this the achieved speedup continues to increase but at a progressively slower rate.

Finally, between the last two grid sizes the increase in speedup slows from 1.51 to 1.56, approaching the limit of how much the simulation can be sped up. For large grid sizes the maximum achievable speedup will not go far beyond 1.6, as the increase in speedup starts leveling off, which is not as significant of a speedup as when comparing CPU-based against GPU-based solvers, but still is a significant performance increase for the undertaken optimizations.

Again we can observe how the achieved speedup increases as the problem size grows, though the difference is not as drastic as the one between the similar CPU and GPU solvers, due to running on equal hardware. This confirms that the more the memory redundancy optimizations can be utilized, the higher the performance gains. Seeing as the the graph has a similar shape to the one discussed in the previous section, it can be observed that the increase in achievable speedup will only continue rising up to a point, which is approached as the number of used grid points increases and therefore more time is spend on computation alone compared to other tasks during the simulation.

7 PERFORMANCE ANALYSIS OF DNS NAVIER-STOKES SOLVERS

This chapter will focus on a performance analysis similar to the one performed in the prior chapter, in order to contextualize these previous findings further by comparing them to the types of speedups achieved over a broad range of grid sizes for a state-of-the-art numerical solver for the Navier-Stokes equations, developed my Dr. Simone Silvestri, and its optimized version, developed by Asif Hasan. The implemented solver is a more complex performance-orientated DNS numerical solver compared to the ones used in the previous chapter, focusing on the compressible Navier-Stokes equations. This comparison is especially relevant to the speedup previously achieved between the second and third solver, as they were also two similar solvers which differ by the degree of optimization and run on the same hardware, and will also give some insight into what kind of speedups can be achieved with available academic numerical solvers running on a GPU when properly optimized and see whether the findings agree with the results put forward in the prior chapters.

This chapter will initially give a short overview of the used solvers, their characteristics, and what differentiates them, before examining the results of the undertaken analysis and comparing it to the one from last chapter.

7.1 USED SOLVERS

There were two solvers used for the simulations in this chapter: the baseline Navier-Stokes solver, the code for which is publicly available[1], which is a high-performance DNS code for the compressible Navier-Stokes equations, which lacks some of the features utilized to minimize memory redundancy and other performance-critical aspects of computation, and the optimized version of this solver. This optimized version makes use of different techniques to minimize global memory accesses and utilizes further changes in the computation procedure to improve performance.

The baseline DNS Navier-Stokes solver is for compressible flows in 3D running on a CUDA-capable GPU and uses the same timestepping scheme, RK3, as the previously introduced solvers and the same spatial discretization, also at 8th-order. This baseline solver, much like the second solver from last chapter, already makes use of shared memory and some GPU-related optimizations. The optimized Navier-Stokes solver has the same characteristics as the baseline solver as far as structure and discretization techniques

1 https://github.com/simone-silvestri/cuda-solvers

are concerned but builds on this with further optimizations. These improvements include but are not limited to:

- Coalesced memory transfer by utilizing a tiling approach that uses shorter and wider two-dimensional tiles (or slices) for global memory accesses
  - When some data is accessed more data is read then is really needed, leading to a bottleneck and high redundancy
  - Cover one warp of threads in as few memory reads as possible
  - Use tiling approach to get 2D slice that reads as much relevant data as possible, instead of 1D array, to minimize reads per warp, i.e. minimize data read redundancy
  - Here, coalesced memory transfers of the optimized solver are necessary for making use of the most possible memory bandwidth available by reducing the number of memory loads for the required data and thus increasing the possible speedup, as per Amdahl's law
- Moving additional computations inside the x-, y-, and z-kernels that were previously outside them
  - These kernels are the part of the solver, running on the GPU, responsible for the computation for the respective direction
  - More speedup can be achieved through offloading additional operations onto the GPU
- Implementation of Flipping Block algorithm to coalesce shared memory access and minimize tendency of shared memory bank conflicts

and are intended to alleviate bottlenecks during the simulation to reduce wall clock time.

Used Parameters			
Distance in x (L_x)	2.0	Boundary Condition in x	Non-Uniform
Distance in y (L_y)	3.0	Boundary Condition in y	Periodic
Distance in z (L_z)	10.0	Boundary Condition in z	Periodic
Reynolds Number	7500	Adiabatic Constant	1.4
Prandtl Number	0.70	Number of timesteps n	500
Mach Number	0.7	Viscous Fluxes Stencil	2
CFL number	0.75	Size of *stencil*	4

Table 7.1: Parameters used across all simulations for both solvers.

The used parameters needed to reproduce the simulations are given above in table 7.1.

7.2 RESULTS OF PERFORMANCE STUDY

Again, a performance analysis was undertaken regarding changing grid sizes, similar to what was done in the previous chapter, to give some insight into the scaling of the selected solvers and whether they behave similarly to those analysed previously. The used grid sizes were made up of between $4 \cdot 10^5$ to $2 \cdot 10^7$ grid points, giving a similar range to the one used for previous analyses, though they were not evenly spaced. The solvers were benchmarked for a common three-dimensional scenario with all parameters kept constant across simulations.

The GPU workstation outlined in chapter 6 was used for the simulations and therefore a single GPU was utilized, even though the solvers presented in this chapter are capable of being run on a multi-GPU system. The same hardware was used for both solvers and all parameters aside from the number of grid points was kept constant across simulations, similarly to when the second and third solvers were compared in the prior chapter.

The results for the speedups achieved across the used grid sizes are shown in figure 7.1. Here, similarities to the development of the speedups from last chapter, particularly those between the second and third solver, can be observed, with an initial rapid increase in speedup which slows over time as the maximum possible gain from the optimizations is approached. The initial speedup value is close to one, indicating that still a significant part of the computation is not devoted to calculation at such as small problem size. A maximum speedup of 1.49 was observed for a problem size of $2 \cdot 10^7$ grid points.

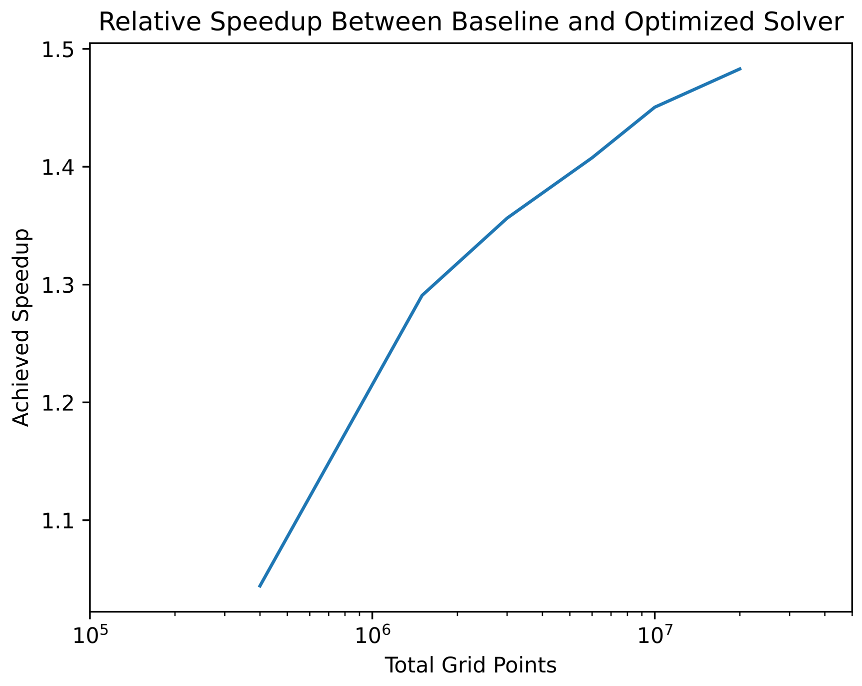


Figure 7.1: Relative speedup between Navier-Stokes solvers for different grid sizes.

It should be noted that larger speedups may be possible on larger problem sizes, as the speedup is still increasing slightly towards the end of the tested range of grid points. A larger problem size could not be simulated due to memory limitations of the GPU of the workbench. Beyond this, a limitation that comes with a consumer-grade graphics card is a lower memory bandwidth, which for the accelerator in this work is 448 GB/s, while server-based graphics cards can have memory bandwidths up to four-times that. This can lead to a memory bottleneck that will lower the achieved speedups when comparing solvers.

7.3 COMPARISON TO PREVIOUS SOLVERS

The performance analysis in this chapter resulted in similar findings to those of last chapter. This strengthens the view that for certain optimizations related to memory access on a GPU there is a predictable performance gain that scales with the problem size when applied to numerical simulations. Further, these performance gains are only fully realized at appropriate problem sizes, when significant parts of the time spent on the simulation are devoted to computations on the GPU within the kernels. On the contrary, when smaller problem sizes with lower expected wall clock times are to be simulated, then there is little reward for wide ranging optimizations of the used solvers.

The decease in wall clock time for both optimized solvers, when considering the time required for a full simulation of a problem with a significant number of grid points, represents a large cost reduction. This reduction of cost is quite significant when considering the price of GPU compute for problems with more grid points, as a 50% reduction of needed compute time can be seen as a significant pay off for the time invested into optimizing the solver, if the solver is to be run extensively for research purposes.

A maximum speedup of 1.49 was achieved between the baseline and optimized Navier-Stokes solvers, which is similar to the value observed for the speedup between the second and third solvers at a comparable problem size. Even though the two optimizations were not identical, they both focused mainly on improving memory access redundancy in order to minimize memory read and write bottlenecks and therefore make better use of a GPUs superior computational throughput.

8 DISCUSSION & CONCLUSION

In this chapter the previously presented findings are summarised and recommendations for relevant further work are made.

8.1 SUMMARISING CONCLUSIONS AND DISCUSSION

The purpose of this work has been to investigate the performance of different finite difference solvers with different degrees of optimization on different types of compute hardware and assess the achieved speedups. Specifically, the move from a CPU-based to a GPU-based solver and optimization with regards to improved memory redundancy were investigated.

After supplying the reader with a short literature review, this work focused on explaining the relevant hardware and software architecture which ones needs to be aware of to tackle the problem that was the focus of this thesis. Following this, the relevant theory for the chosen problem was introduced, before the methods used for the computational implementation were stated. These included a detailed explanation of the three separate solvers used, as well as other relevant information, such as the used temporal and spatial discretization schemes and the chosen validation scenarios.

A benchmarking scenario was proposed to be used for the different solvers across used hardware, including the relevant geometry, gird, and initial and boundary conditions. This aforementioned hardware was also introduced, including how to quantify performance beyond the achieved speedups with the cost of the used hardware and the complexity of the undertaken optimizations to make it possible to form a judgement on whether the resources, namely time and money, spent on them can be justified. Then, the obtained results were introduced and visualized for the three solvers, before moving onto the performance comparisons. The first performance comparison showed the time to complete the previously mentioned benchmark on two grid sizes, made up of 64 or 128 grid points in each direction, for all three solvers, showing a very significant speedup between the CPU- and GPU-based solvers and a moderate difference between the two GPU-based solvers. Beyond this, a comparison between the two solvers similar in structure but run on different hardware, the first and second solvers, and the two solvers run on the same hardware but optimized to different degrees, the second and third solvers, was undertaken in order to give a better overview of the achievable speedups for different types of hardware and degrees of optimizations in isolation. For the first and second solver, the speedup was measured over a range of different grid sizes, starting from 32 grid points in all directions up to 1024 grid points, resulting in speedups of less than five all the way to 34.7. Here, one was able to observe that the increase in achieved speedup slowed as the problem size increased, indicating that the improvements from parallelization were approaching full utilization, where less time was spent on computation. For the second and third solver a similar comparison across different grid sizes was undertaken, painting a similar picture of decreasing improvements in speedup as the problem size grows, though the achieved speedups were more modest, reaching a speedup of below 1.6 for 1024 grid points in each direction. This smaller speedup can still be considered significant when remembering the wall clock time of the simulations that treat large problem sizes and the cost of compute.

Very significant speedups were achieved between the first and second solver, clearly showing the possible performance gains when moving from a serial to a parallel implementation. The achieved speedups can justify the increased cost of the used hardware for the different workstations, as well as the time spent on the GPU implementation, given that a problem with a larger number of grid points is simulated.

The difference between the second and third solver was not as significant as the one between CPU and GPU implementations, but there still was a significant speedup, especially at larger grid sizes, which may justify the resources spent on undertaken optimizations. Whether spending time optimizing a solver is truly worth it also depends on the complexity of that solver, as more complex solvers may take additional care so that increased performance can be achieved.

It should be noted that these speedups are very dependent on problem size, as was shown in chapter 6. When considering the largest grid with over one billion nodes for the first and second solver, then the speedup is significant and the reduction in wall clock time clearly justifies the move from a CPU-based to GPU-based implementation, as many days of computing time can be saved. Though when considering the smallest tested grid size of 32 grid points in each direction, even though the wall clock time spend on the simulation can be decreased, the actual savings are not very significant since the simulation does not take an extended amount of time in the first place. Therefore, the effort would not be worth it considering the gains would be negligible.

Beyond this, two DNS Navier-Stokes solvers were compared against one another, one being an optimization of the other, to observe what kind of speedups can be achieved when a more complex research-orientated solver is optimized and run on the same hardware and whether these findings were in line with previously undertaken analyses. Speedups of similar magnitude were observed for comparable problem sizes.

Something that was mentioned earlier in this work but not investigated in detail is the move from double precision to single precision accuracy for the solvers. While using single precision accuracy will undoubtedly improve performance, especially on a consumer-grade GPU, it is nonetheless not applicable for most scientific applications, as they require a high level of accuracy, and therefore is not a viable alternative when considering the intended purpose of this work. Further, as the relative speedup was a key measure, the move from double to single precision would speed up all the different solvers, though, admittedly, to a different degree. Therefore, the relative speedup would not be impacted as much as the move from double to single precision would suggest.

8.2 RECOMMENDATIONS FOR FUTURE WORK

Finally, through conclusion of this work new possible areas have been identified that could be investigated to further contextualize the findings presented here. What follows are three suggestions for further continuation of this work.

8.2.1 Benchmarking Further Optimized Solvers

The solvers presented in this work follow an obvious progression from a simple CPU-based solver to a GPU-based solver of a similar structure to a GPU-based solver that has been optimized. To offer a more complete overview of the possible performance increases between CPU- and GPU-based solvers, as well as not optimized to optimized solvers, a broader range of solvers could be implemented and benchmarked. Comparing GPU-based solvers could help identify the currently existing solvers with the most promising performance, though care would have to be taken to standardize all parameters and other key factors such as discretization techniques to ensure a comparison that is as fair and representative as possible. Finally, research into new, more effective solvers could give a sizable performance increase and thus further research into this area could yield better performance compared to existing solvers on current hardware.

8.2.2 Considering a Wider Range of Accelerators

Only a small sample size of hardware was considered in this work. To gain further insight into the performance of the presented solvers additional accelerators could be used for the simulations in the future to measure performance as the compute of the hardware and the iteration of the architecture increases or decreases. What makes this increasingly relevant is the fact that a consumer-grade GPU from the GeForce line of Nvidia products was used for this work, instead of the server-based accelerators with a scientific computing focus, such as the Nvidia A100. This would make it possible to recommend the optimal configuration for price to performance for such simulations.

Beyond this, using GPUs with increased memory would make it possible to use grids even larger in scope then the ones used in this work. Using an accelerator like the Nvidia Titan RTX with 24 GB of memory, or the aforementioned server-based A100 with 80 GB of memory, would greatly increase the possible problem size on a single GPU. These would also alleviate the memory bandwidth bottleneck discussed in the last chapter.

8.2.3 Multi-GPU Systems

To further increase performance, a multi-GPU system could be employed, where the computation is not only parallelized across a single accelerator, but across multiple connected GPUs working together to share one computational load. Breaking down simulations in this way could lead to a further increase in performance, similar to the one found between simulations run on CPUs and GPUs. Making a CUDA-based solver ready to be run on a multi-GPU system takes a significant amount of optimization to ensure the maximum performance can be achieved when the computation is distributed across the system. Suitable hardware also has to be available, including the GPUs and NVLink communication bridges, which may come at a large cost.

BIBLIOGRAPHY

[1] M. Aissa, T. Verstraete, and C. Vuik. Toward a gpu-aware comparison of explicit and implicit cfd simulations on structured meshes. *Computers Mathematics with Applications*, 74(1):201–217, 2017. 5th European Seminar on Computing ESCO 2016.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[3] H. Anzt, M. Baboulin, J. Dongarra, Y. Fournier, F. Hulsemann, A. Khabou, and Y. Wang. Accelerating the conjugate gradient algorithm with gpus in cfd simulations. In I. Dutra, R. Camacho, J. Barbosa, and O. Marques, editors, *High Performance Computing for Computational Science – VECPAR 2016*, pages 35–43, Cham, 2017. Springer International Publishing.

[4] S. Belhaous, Z. Hidila, S. Baroud, S. Chokri, and M. Mestari. *An Execution Time Comparison of Parallel Computing Algorithms for Solving Heat Equation*, pages 283–295. 06 2020.

[5] Y. D. Bhadke, M. R. Kawale, and V. Inamdar. Development of 3d-cfd code for heat conduction process using cuda. In *2014 International Conference on Advances in Engineering & Technology Research (ICAETR-2014)*, pages 1–5. IEEE, 2014.

[6] M. Biazewicz, K. Kurowski, B. Ludwiczak, and K. Napieraia. Problems related to parallelization of cfd algorithms on gpu, multi-gpu and hybrid architectures. In *AIP Conference Proceedings*, volume 1281, pages 1301–1304. American Institute of Physics, 2010.

[7] J. Cannon and F. Browder. *The One-Dimensional Heat Equation*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1984.

[8] J. Cohen. Fluid Simulation with CUDA. https://www.nvidia.com/content/GTC/documents/SC09_Fluid_Sim_Cohen.pdf, 2009. [Online; accessed 19-March-2021].

[9] G. Coppola, F. Capuano, S. Pirozzoli, and L. de Luca. Numerically stable formulations of convective terms for turbulent compressible flows. *Journal of Computational Physics*, 382:86–104, Apr 2019.

[10] A. T. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid based cfd solvers on modern graphics hardware. 2009.

[11] A. C. Crespo, J. M. Dominguez, A. Barreiro, M. Gómez-Gesteira, and B. D. Rogers. Gpus, a new tool of acceleration in cfd: Efficiency and

reliability on smoothed particle hydrodynamics methods. *PLOS ONE*, 6(6):1–13, 06 2011.

[12] S. Elghobashi. Direct numerical simulation of turbulent flows laden with droplets or bubbles. *Annual Review of Fluid Mechanics*, 51(1):217–244, 2019.

[13] J. Fang, J. J. Cambareri, C. S. Brown, J. Feng, A. Gouws, M. Li, and I. A. Bolotnov. Direct numerical simulation of reactor two-phase flows enabled by high-performance computing. *Nuclear Engineering and Design*, 330:409–419, 2018.

[14] F. Foertter and J. Wells. Accelerating Science and Engineering with Titan, the World's Fastest Supercomputer. https://on-demand.gputechconf.com/gtc/2013/presentations/S3470-Accelerating-Science-Engineering-Titan.pdf. [Online; accessed 19-March-2021].

[15] C. Grossmann, H.-G. Roos, and M. Stynes. *Numerical treatment of partial differential equations. Revised translation of the 3rd German edition of 'Numerische Behandlung partieller Differentialgleichungen' by Martin Stynes*. 01 2007.

[16] J. L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[17] F. H. Harlow. Fluid dynamics in group t-3 los alamos national laboratory: (la-ur-03-3852). *Journal of Computational Physics*, 195(2):414–433, 2004.

[18] A. E. Honein and P. Moin. Higher entropy conservation and numerical stability of compressible turbulence simulations. *Journal of Computational Physics*, 201(2):531–545, 2004.

[19] Z. Horváth and M. Liebmann. Performance of cfd codes on cpu/gpu clusters. 1281, 09 2010.

[20] C. F. Janßen, D. Mierke, M. Überrück, S. Gralher, and T. Rung. Validation of the gpu-accelerated cfd solver elbe for free surface flow problems in civil and environmental engineering. *Computation*, 3(3):354–385, 2015.

[21] T. Kajishima, T. Ohta, K. Okazaki, and Y. Miyake. High-order finite-difference method for incompressible flows using collocated grid system. *JSME International Journal Series B Fluids and Thermal Engineering*, 41(4):830–839, 1998.

[22] J. Kim, P. Moin, and R. Moser. Turbulence statistics in fully developed channel flow at low reynolds number. *Journal of Fluid Mechanics*, 177:133–166, 1987.

[23] T. Kobayashi and K. Kitoh. A review of cfd methods and their application to automobile aerodynamics. In *SAE Technical Paper*. SAE International, 02 1992.

[24] P. Kundu, I. Cohen, and D. Dowling. *Fluid Mechanics*. Science Direct e-books. Elsevier Science, 2012.

[25] M. Lee and R. D. Moser. Direct numerical simulation of turbulent channel flow up to $Re_ø \approx 5200$. *Journal of Fluid Mechanics*, 774:395–415, 2015.

[26] H. Lomax, T. Pulliam, and D. Zingg. *Fundamentals of Computational Fluid Dynamics*. Scientific Computation. Springer Berlin Heidelberg, 2003.

[27] S. Matsuoka, T. Aoki, T. Endo, A. Nukada, T. Kato, and A. Hasegawa. Gpu accelerated computing–from hype to mainstream, the rebirth of vector computing. *Journal of Physics: Conference Series*, 180:012043, 08 2009.

[28] P. Micikevicius. 3d finite difference computation on gpus using cuda. pages 79–84, 01 2009.

[29] P. Moin and K. Mahesh. Direct numerical simulation: a tool in turbulence research. *Annual review of fluid mechanics*, 30(1):539–578, 1998.

[30] R. D. Moser, J. Kim, and N. N. Mansour. Direct numerical simulation of turbulent channel flow up to re=590. *Physics of Fluids*, 11(4):943–945, 1999.

[31] R. D. Moser and P. Moin. Direct numerical simulation of curved turbulent channel flow. NASA STI/Recon Technical Report N, Oct. 1984.

[32] K. E. Niemeyer and C.-J. Sung. Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *J. Supercomput.*, 67(2):528–564, 2014.

[33] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2017. Version 8.0.2.

[34] E. H. Phillips and M. Fatica. Implementing the himeno benchmark with cuda on gpu clusters. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, 2010.

[35] S. Posey. Considerations for gpu acceleration of parallel cfd. *Procedia Engineering*, 61:388–391, 2013. 25th International Conference on Parallel Computational Fluid Dynamics.

[36] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. Gpu accelerated monte carlo simulation of the 2d and 3d ising model. *Journal of Computational Physics*, 228(12):4468–4477, 2009.

[37] T. Prosperetti. *Computational Methods for Multiphase Flow*. Cambridge University Press.

[38] T. Pulliam and J. Steger. On implicit finite-difference simulations of three-dimensional flow. 02 1978.

[39] J. Sanders and E. Kandrot. Cuda by example: An introduction to general-purpose gpu programming. 01 2011.

[40] C.-W. Shu. High-order finite difference and finite volume weno schemes and discontinuous galerkin methods for cfd. *International Journal of Computational Fluid Dynamics*, 17(2):107–118, 2003.

[41] R. Slingerland and L. Kump. *Mathematical Modeling of Earth's Dynamical Systems: A Primer*. 12 2011.

[42] D. Steinkraus, I. Buck, and P. Y. Simard. Using gpus for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pages 1115–1120 Vol. 2, 2005.

[43] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten. Gpu-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling*, 29(2):116–125, 2010.

[44] S. Thakre and J. Joshi. Cfd modeling of heat transfer in turbulent pipe flows. *AIChE journal*, 46(9):1798–1812, 2000.

[45] TOP500. Green500 Ranking November 2020. https://www.top500.org/lists/green500/2020/11/, 2020. [Online; accessed 19-March-2021].

[46] TOP500. TOP500 Ranking November 2020. https://www.top500.org/lists/top500/2020/11/, 2020. [Online; accessed 19-March-2021].

[47] G. Tryggvason, R. Scardovelli, and S. Zaleski. *Direct Numerical Simulations of Gas–Liquid Multiphase Flows*. Cambridge University Press, 2011.

[48] B. Tutkun and F. O. Edis. A gpu application for high-order compact finite difference scheme. *Computers & Fluids*, 55:29–35, 2012.

[49] J. F. Wendt, editor. *Computational Fluid Dynamics*. Springer Berlin Heidelberg, 2009.

[50] D. Widder. *The Heat Equation*. Pure and Applied Mathematics. Elsevier Science, 1976.

[51] P. Wilmott, S. Howson, S. Howison, and J. Dewynne. *The Mathematics of Financial Derivatives: A Student Introduction*. The Mathematics of Financial Derivatives: A Student Introduction. Cambridge University Press, 1995.

[52] B. Xia and D.-W. Sun. Applications of computational fluid dynamics (cfd) in the food industry: a review. *Computers and Electronics in Agriculture*, 34(1):5–24, 2002.

[53] Z. Xian, X. Tang, and H. Mo. Cfd simulation different inner structure of air heat exchanger. In *IOP Conference Series: Earth and Environmental Science*, volume 199, page 052032. IOP Publishing, 2018.