A Classic Linear System Solver on Modern Hardware Architecture for Sparse Systems

Nils Urmersbach

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Matemática Aplicada
Orientador: Prof. Dr. Alexandre Megiorin Roma

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, Outubro de 2016

A Classic Linear System Solver on Modern Hardware Architecture for Sparse Systems

Abstract

URMERSBACH, N. **A Classic Linear System Solver on Modern Hardware Architecture for Sparse Systems**. 2016. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2016.

In this work we present our implementations for the Jacobi Method for general sparse linear systems in the Compressed Sparse Row (CSR) format using OpenMP, OpenACC and CUDA. We apply these implementations to the linear system derived from the central finite difference discretization of the two-dimensional Poisson Equation on rectangular domains, and compare the performance of the CSR implementations to the performance of a direct Poisson Equation solver using the five-point stencil. For our case study, we consider five different grid size (with up to ~67.1 million unknowns), both in single precision and double precision, and a variety of thread numbers for the OpenMP implementation, resulting in 300 different configurations in total that were executed for this work. We discuss the scaling behaviour of the different implementations and present some profiling results of our parallelized programs.

Keywords: OpenMP, OpenACC, CUDA, Compressed Sparse Row Format, Jacobi Method, Poisson Equation.

Resumo

URMERSBACH, N. **Um solver de sistemas lineares clássico na arquitetura moderna de hardware para sistemas esparsos**. 2016. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2016.

Nesse trabalho apresentamos as nossas implementações do Método de Jacobi para sistemas lineares esparsos gerais no formato de Compressed Sparse Row (CSR) usando OpenMP, OpenACC e CUDA. Aplicamos essas implementações no sistema linear derivado da discretização de diferenças finitas centrais da Equação de Poisson em duas dimensões em domínios retangulares e comparamos o desempenho das implementações de CSR com o desempenho de um solver direto da Equação de Poisson usando o estêncil de cinco pontos. Para nosso estudo de caso nós consideramos cinco tamanhos diferentes de malhas (com até ~67.1 milhões desconhecidos), ambos precisão simples e dupla, e uma variedade de números de threads para a implementação de OpenMP, resultando em 300 configurações diferentes executadas para esse trabalho. Nós discutimos o comportamento de escalagem das implementações diferentes e apresentamos alguns resultados de perfilamento dos nossos programas paralelizados.

Palavras-chave: OpenMP, OpenACC, CUDA, Formato de Compressed Sparse Row, Método de Jacobi, Equação de Poisson.

Contents

List of Figures **ix**

List of Tables **xiii**

1 Introduction **1**

1.1 Motivation 1

1.1.1 Parallelism 1

1.1.2 Iterative Solvers for Linear Systems 2

1.2 Objective 4

1.3 Organization 4

2 Parallel Computing **5**

2.1 Basic Processor Architecture – The von Neumann Model 5

2.2 Parallel Computing on Shared-Memory Parallel Computers 6

2.2.1 Shared-Memory Architecture 6

2.2.2 Programming Shared-Memory Parallel Computers: OpenMP 8

2.3 Parallel Computing with GPUs 12

2.3.1 GPU Architecture 12

2.3.2 Programming GPUs: CUDA 15

2.3.3 Programming GPUs: OpenACC 18

2.4 Performance Metrics for Parallel Programming 21

3 Mathematical Problem **25**

3.1 Discretization of the Two-Dimensional Poisson Equation 25

3.2 Iterative Methods 28

3.2.1 Jacobi Method 30

4 Implementations **33**

4.1 The Compressed Sparse Row Format 33

4.2 The Jacobi CSR and “modified” Jacobi CSR Implementations 34

4.3 The Five-Point Jacobi Implementation 35

5 Results **37**

6 Conclusion **55**

Appendices **57**

A Essential Definitions and Theoretical Results **59**
A.1 Vector Norms 59
A.2 Matrix Norms 62
A.3 Similarity and Diagonalizability of Matrices 68
A.4 Unitary and Normal Matrices 70

B Profiling of OpenMP Implementations using `perf` **75**

Bibliography **85**

List of Figures

1.1 "Normalized Power versus Normalized Scalar Performance for Multiple Generations of Intel Microprocessors", Grochowski and Annavaram. The normalization is based on the values of the i486 processor. 2

1.2 Intel CPU introductions, taken from Sutter [2005]. 3

1.3 Comparison between a single core and an idealised duo-core processor whose capacity is exactly double the capacity of the single core processor, and whose frequency and voltage are exactly half of the frequency and voltage of the single core processor. 3

2.1 A sketch of the von Neumann architecture, based on Kirk and Hwu [2013] and Shiva [2007]. 6

2.2 Block diargram of a generic, cache-based dual core processor. Chapman et al. [2007] . . . 7

2.3 Comparison between UMA (a) and NUMA (b) platforms, compare Barney [2016]. . . . 7

2.4 The Fork-Join Model. 8

2.5 Taken from Chapman et al. [2007]. "Schedule kinds supported on the schedule clause – The `static` schedule works best for regular workloads. For a more dynamic work allocation scheme the `dynamic` or `guided` schedules may be more suitable." 9

2.6 Overheads of several common OpenMP directives and constructs. Taken from Chapman et al. [2007]. Note that the overhead of reduction operations for this compiler is unreasonably high, and not representative for other compilers. Chapman et al. [2007] 11

2.7 Overheads of different kinds of OpenMP loop schedules. Taken from Chapman et al. [2007]. 11

2.8 A comparison of the schematic architectures of CPUs and GPUs. Taken from Kirk and Hwu [2013] 12

2.9 NVIDIA Kepler Block Diagram. Taken from Wolfe [2012]. 13

2.10 Block diagram of the GK110 chip architecture. Taken from NVIDIA [2012]. 14

2.11 Block diagram of a single Streaming Multiprocessor of the GK110 chip. Taken from NVIDIA [2012]. 14

2.12 Graphic representation of a warp scheduler. Taken from NVIDIA [2012]. 15

2.13 Accelerator model used for OpenACC. Taken from Kirk and Hwu [2013]. 19

2.14 OpenACC fork/join process. Taken from Kirk and Hwu [2013]. 20

2.15 Maximum speedup S achievable according to Equation (2.1) as a function of the parallel fraction of the program f_{par} and the number of processors used p. 22

3.1 Plot of (3.15) in dependence of p and q. 32

4.1 The five-point stencil for the laplacian operator. 36

5.1 Graphical representation of the converging process. Depending on how the iteration converges to the exact discrete solution, $\hat{u}_{discr.}$, (for example due to the choice of of the initial guess u_0) it may be necessary to adapt the chosen tolerance, ϵ, in order to let the iterative process terminate. An appropriate choice for the tolerance is given in the error bound Equations (3.3) and (3.4). 38
5.2 OpenMP speedup plot for the (modified) Jacobi CSR implementation when using single precision. Based on the execution time of the Jacobi method only. 38
5.3 OpenMP speedup plot for the (modified) Jacobi CSR implementation when using double precision. Based on the execution time of the Jacobi method only. 38
5.4 OpenMP efficiency plot for the (modified) Jacobi CSR implementation when using single precision. Based on the execution time of the Jacobi method only. 39
5.5 OpenMP efficiency plot for the (modified) Jacobi CSR implementation when using double precision. Based on the execution time of the Jacobi method only. 39
5.6 Relative execution time difference between the modified and unmodified Jacobi CSR implementations using single precision. 41
5.7 Relative execution time difference between the modified and unmodified Jacobi CSR implementations using double precision. 41
5.8 OpenMP speedup plot for the five-star Jacobi implementation when using single precision. Based on the execution time of the Jacobi method only. 42
5.9 OpenMP speedup plot for the five-star Jacobi implementation when using double precision. Based on the execution time of the Jacobi method only. 42
5.10 OpenMP efficiency plot for the five-star Jacobi implementation when using single precision. Based on the execution time of the Jacobi method only. 43
5.11 OpenMP efficiency plot for the five-star Jacobi implementation when using double precision. Based on the execution time of the Jacobi method only. 43
5.12 Graphical representation of Table 5.2. 45
5.13 Graphical representation of Table 5.3. 45
5.14 Relative execution time difference between the modified and unmodified Jacobi CSR implementations for the OpenACC and CUDA programs for both single and double precision. All times based on the Jacobi kernel execution times only. 46
5.15 Plot of the iteration fraction of the modified to unmodified CSR implementation. 46
5.16 Bandwidth utilization of the GPU L1 cache for all implemented Jacobi kernels. 48
5.17 Bandwidth utilization of the GPU L2 cache for all implemented Jacobi kernels. 48
5.18 Bandwidth utilization of the texture memory for all implemented Jacobi kernels. 48
5.19 Bandwidth utilization of the device memory for all implemented Jacobi kernels. 49
5.20 Plot of active warps for all implemented Jacobi kernels, using single precision. 49
5.21 Plot of active warps for all implemented Jacobi kernels, using double precision. 49
5.22 Occupancy plot for all implemented Jacobi kernels, using single precision. 50
5.23 Occupancy plot for all implemented Jacobi kernels, using double precision. 50
5.24 Relative total execution time differences between single and double precision cases for all implementations used. 53

A.1 Plots of $||\mathbf{x}||_p = 1$ on $\mathbb{R}^2$ for different values of p. 60

A.2 Geometric presentation of the Norm Equivalence Theorem for the l_∞ and l_2 norms on $\mathbb{R}^2$: $\frac{\sqrt{2}}{2}||\mathbf{x}||_\infty \leq ||\mathbf{x}||_2 \leq ||\mathbf{x}||_\infty$ 62
A.3 Geometric representation of $\text{lub}(A) = \max_{||\mathbf{x}||=1} ||A\mathbf{x}||$ (for the l_2 norm). . . . 64

B.1 "Number of CPUs utilized" for the unmodified single precision Jacobi CSR implementation. 76
B.2 "Number of CPUs utilized" for the unmodified double precision Jacobi CSR implementation. 76
B.3 "Number of CPUs utilized" for the modified single precision Jacobi CSR implementation. 77
B.4 "Number of CPUs utilized" for the modified double precision Jacobi CSR implementation. 77
B.5 Average CPU frequency achieved for the unmodified single precision Jacobi CSR implementation. . . . 77
B.6 Average CPU frequency achieved for the unmodified double precision Jacobi CSR implementation. . . . 77
B.7 Average CPU frequency achieved for the modified single precision Jacobi CSR implementation. . . . 78
B.8 Average CPU frequency achieved for the modified double precision Jacobi CSR implementation. . . . 78
B.9 Percentage of the total cache misses (based on all cache references) for the unmodified single precision Jacobi CSR implementation. . . . 78
B.10 Percentage of the total cache misses (based on all cache references) for the unmodified double precision Jacobi CSR implementation. . . . 78
B.11 Percentage of the total cache misses (based on all cache references) for the modified single precision Jacobi CSR implementation. . . . 79
B.12 Percentage of the total cache misses (based on all cache references) for the modified double precision Jacobi CSR implementation. . . . 79
B.13 Percentage of the L1 cache misses (based on all L1 cache references) for the unmodified single precision Jacobi CSR implementation. . . . 79
B.14 Percentage of the L1 cache misses (based on all L1 cache references) for the unmodified double precision Jacobi CSR implementation. . . . 79
B.15 Percentage of the L1 cache misses (based on all L1 cache references) for the modified single precision Jacobi CSR implementation. . . . 80
B.16 Percentage of the L1 cache misses (based on all L1 cache references) for the modified double precision Jacobi CSR implementation. . . . 80
B.17 Percentage of the branch misses for the unmodified single precision Jacobi CSR implementation. . . . 80
B.18 Percentage of the branch misses for the unmodified double precision Jacobi CSR implementation. . . . 80
B.19 Percentage of the branch misses for the modified single precision Jacobi CSR implementation. 81
B.20 Percentage of the branch misses for the modified double precision Jacobi CSR implementation. 81
B.21 "Number of CPUs utilized" for the single precision five-point Jacobi implementation. . . . 81
B.22 "Number of CPUs utilized" for the double precision five-point Jacobi implementation. . . 81
B.23 Average CPU frequency achieved for the single precision five-point Jacobi implementation. 82
B.24 Average CPU frequency achieved for the double precision five-point Jacobi implementation. 82
B.25 Percentage of the total cache misses (based on all cache references) for the single precision five-point Jacobi implementation. . . . 82

B.26 Percentage of the total cache misses (based on all cache references) for the double precision five-point Jacobi implementation. 82
B.27 Percentage of the L1 cache misses (based on all L1 cache references) for the single precision five-point Jacobi implementation. 83
B.28 Percentage of the L1 cache misses (based on all L1 cache references) for the double precision five-point Jacobi implementation. 83
B.29 Percentage of the branch misses for the single precision five-point Jacobi implementation. 83
B.30 Percentage of the branch misses for the double precision five-point Jacobi implementation. 83

List of Tables

2.1 CUDA variable type qualifiers for different memories. Kirk and Hwu [2013] 17

5.1 Number of iterations needed to achieve a tolerance of $\epsilon = 0.01$ for the problem given in Equation (5.1) and an initial guess of $u^{(0)} \equiv 0$. 38
5.2 Summary of all execution times using single precision floating point numbers. For the OpenMP results 64 threads are used. 44
5.3 Summary of all execution times using double precision floating point numbers. For the OpenMP results 64 threads are used. 44
5.4 Grid Sizes for the (Modified) Jacobi CSR Implementations 47
5.5 Grid Sizes for the Five-Point Stencil Jacobi Implementations 47
5.6 Kernel Limiters: (Modified) Jacobi CSR, Single Precision. The ^{1} superscript marks the primary limiter. 47
5.7 Kernel Limiters: (Modified) Jacobi CSR, Double Precision. The ^{1} superscript marks the primary limiter. 51
5.8 Kernel Limiters: Five-Point Stencil Jacobi, Single Precision. The ^{1} superscript marks the primary limiter. 51
5.9 Kernel Limiters: Five-Point Stencil Jacobi, double precision. The ^{1} superscript marks the primary limiter. 51
5.10 Register restriction: OpenACC, (Modified) Jacobi CSR, Single Precision 52
5.11 Register restriction: OpenACC, (Modified) Jacobi CSR, Double Precision 52
5.12 Register restriction: CUDA, (Modified) Jacobi CSR, Single Precision 52
5.13 Register restriction: CUDA, (Modified) Jacobi CSR, Double Precision 52
5.14 Register restriction: OpenACC, Five-point Jacobi, Single Precision 52
5.15 Register restriction: OpenACC, Five-point Jacobi, Double Precision 52
5.16 Register restriction: CUDA, Five-point Jacobi, Single Precision 52
5.17 Register restriction: CUDA, Five-point Jacobi, Double Precision 52

Chapter 1

Introduction

1.1 Motivation

1.1.1 Parallelism

As H. Sutter stated in his publication with the same name Sutter [2005]: “The free lunch is over”. In former times, in order to let one’s program run faster all one had to do was to buy a new, faster processor and let the program run on it. This was the case because until recently the clock speed (the frequency of the processor) increased with every generation, as can be seen representatively in Figure 1.2. Moreover, it can be seen in Figure 1.2 that the power consumption increased with the clock speed, and even worse, the power consumption increased faster than the clock speed: In Grochowski and Annavaram, a relationship between power consumption and performance[1] is found to be $P = perf^{1.74}$ based on the data of a few Intel single core processors, compare Figure 1.1. This behaviour describes a *power wall*: For single core processors boosting performance gets prohibitively expensive when considering the power consumption. To understand why a faster processor has a higher power consumption than a slower one, let as consider the following. By letting C donate the capacity, V donate the voltage and q donate a charge, we have

$$C = \frac{q}{V} \text{ or } q = CV. \tag{1.1}$$

Furthermore, by letting W denote the work, we get

$$W = qV = CV^2. \tag{1.2}$$

Since the power P is defined to be work per unit time, we thus receive

$$P = \frac{W}{t} = \frac{CV^2}{t} = CV^2 f, \tag{1.3}$$

where $f = 1/t$ is the frequency. Hence the power consumption of a processor is linear in both capacity and frequency, and quadratic in voltage. Therefore the faster the clock speed of a processor, the more power it consumes. However, since the power wall has been hit there was a need to find a way to improve the performance per watt.

For this reason, today a well established practice is to rather increase the number of cores on a processor chip rather than the frequency of a single core. To understand why this helps reducing the power consumption of a processor consider Figure 1.3 which portrays a single core and an idealised duo-core processor. The dual core processor is idealized in the sense that it is is built from two single core processors of half the frequency of the original one connected in parallel in such a way that the capacity is twice the capacity of the single core processor and without voltage spillage, such that each of the processors in parallel operate with half of the voltage of the single core processor. In this scenario, the duo-core processor only

[1]The performance in the work of Grochowski and Annavaram is based on SPECint, the integer performance testing component of the Standard Performance Evaluation Corporation (SPEC) test suite.

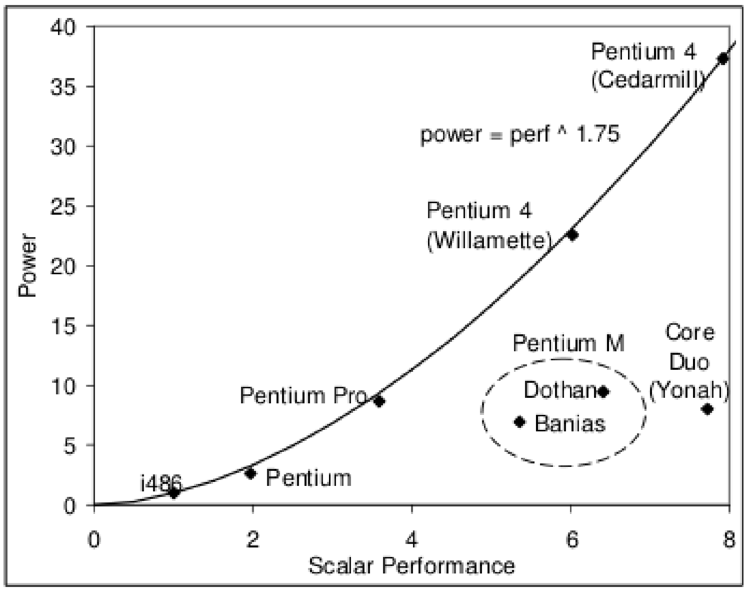


Figure 1.1: *“Normalized Power versus Normalized Scalar Performance for Multiple Generations of Intel Microprocessors”, Grochowski and Annavaram. The normalization is based on the values of the i486 processor.*

consumes a forth of the power of the single core processor, while both can potentially operate at the same overall frequency. However, the natural question to ask is whether an n-core processor is n times as fast as an equivalent single core processor (or just as fast as a single core processor with the n-fold frequency.)

In the spirit of this questions, is it possible to have an even better performance when using a GPU with a massive amount of (notably inferior) cores, instead of using a CPU (multicore processor) processor, even when there are additional data transfer costs? Relatively recent advances in the GPU architecture made it possible to perform regular computations in a massively parallel manner on the GPU.

Whether computations are done on multicore processors, GPUs or other devices or systems utilizing parallelism, they all have in common that it is now the user’s responsibility to optimize the code such that it can be efficiently run on the system. The time in which the user can rely on the hard and smart work of the hardware engineers has passed, the free lunch is over. *Multithreading* is here to stay, and it is now the software developer’s task to write multithreaded programs in order to make use of modern architecture.

1.1.2 Iterative Solvers for Linear Systems

Solving linear systems is among the basic tasks of scientific computation: Many scientific problems especially in the fields of physics and engineering can be described by differential equations (which involve an infinite number of unknowns). In many cases, in order to be able to at least approximate the solution of those differential equations, they will be transferred into a discrete system with a finite number of unknowns. However there are many other other applications of systems of linear equations in different fields of science that are not based on differential equations. As those systems for many applications tend to be large, direct methods become prohibitively expensive (both in data storage and computation) and are therefore solved by using iterative methods.

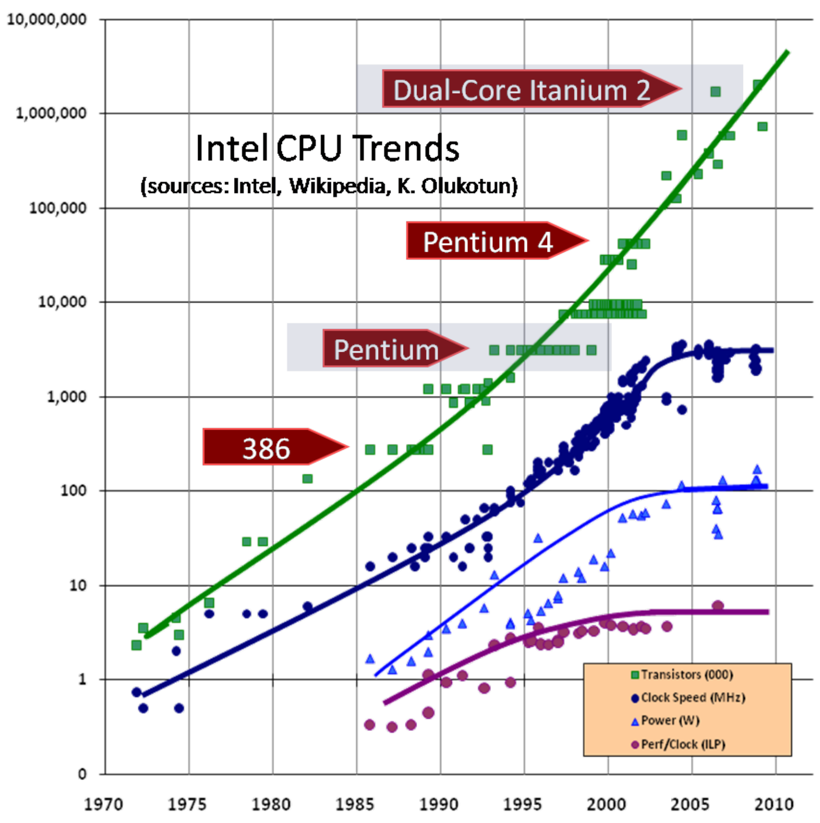


Figure 1.2: *Intel CPU introductions, taken from Sutter [2005].*

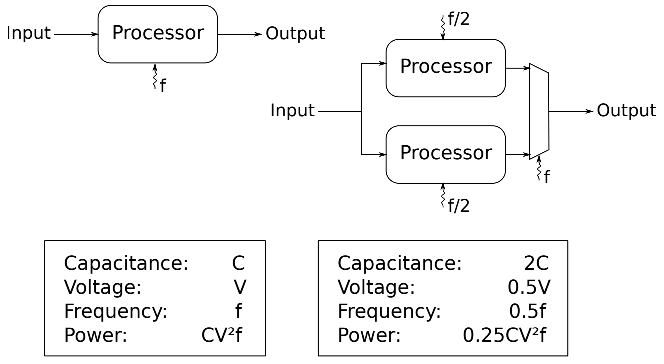


Figure 1.3: *Comparison between a single core and an idealised duo-core processor whose capacity is exactly double the capacity of the single core processor, and whose frequency and voltage are exactly half of the frequency and voltage of the single core processor.*

1.2 Objective

As we have motivated in the previous section, parallelism is here to stay and hence will be a necessity for future generations of developers. By the same token, sparse and big linear systems arise in many scientific areas and the resolution of these systems can make up a major portion of scientific applications. Parallelism can help to substantially decrease the time needed to resolve linear systems, though to make efficient use of the hardware and to optimize the performance, automatic parallelization as offered by several compilers will not be sufficient, and it will take a certain insight into hardware considerations as well as how the parallel methodologies are implemented. Specifically how different processors or processor cores have access to data can not only affect the performance of a program but may also lead to wrong results when not handled appropriately.

The objective of this work is to implement general sparse linear system solvers using the Jacobi Method for parallel execution on different types of modern hardware (shared memory parallel computers and GPUs), and to compare the performance of these implementations applied to a Poisson Equation with the performance of a direct Poisson solver implemented on the same systems. Furthermore, we investigate the performance behaviour for systems of different size, as well as try to investigate the limitations of the performances due to the implementations themselves and due to the hardware being used.

1.3 Organization

In Chapter 2, we introduce the hardware offering parallel computation that was used for this work, and the methodologies that can be used to let one's programs run on them in parallel, as well as some performance considerations. This chapter is then finished with a brief discussion on the metrics commonly used to measure parallel performance.

In Chapter 3, we discuss the mathematical problem that is considered and solved in parallel in this work, namely the (discrete) Poisson Equation, along with some theoretical considerations of iterative linear solvers in general and the Jacobi Method in specific.

Chapter 4 starts with a brief introduction into the Compressed Sparse Row (CSR) storage format, and continues with a presentation of the implementations of the Jacobi Method capable of either solving general sparse linear systems (if certain convergence criteria presented in Chapter 3 are fulfilled), or only the Poisson Equation.

In chapter 5, the results of these implementations using OpenMP, OpenACC, and CUDA are presented, compared, and discussed.

This thesis finishes with the conclusion of this work in Chapter 6.

Additional background that may be required for Chapter 3 is given in the Appendix in A.1 – A.4. Furthermore, some profiling results for the OpenMP codes are given in Appendix B.

Chapter 2

Parallel Computing

In this chapter we present the basic concepts of parallel programming, in regard to both architecture as well as to how to program these architectures, using OpenMP, CUDA and OpenACC. However it is not aimed to give complete introductions to them. For proper proper introductions and/or tutorials see Chapman et al. [2007], Barney [2016], Kirk and Hwu [2013], as well as http://www.openacc.org/content/education. The complete Application Programming Interface for OpenMP can be found in Board [2015], for OpenACC in OpenACC-Standard.org [2015].

We only present the architectures used in this work, namely shared memory systems and GPUs. For the sake of completeness note however that there is another very important parallel programming architecture – *distributed memory systems*. Their communication is commonly controlled with MPI.

2.1 Basic Processor Architecture – The von Neumann Model

The so called von Neumann model was developed in the 1940s and has been "the foundational blueprint to virtually all modern computers", Kirk and Hwu [2013]. It describes the architecture of electronic digital computers, which can be broadly divided into a processing unit containing the arithmetic and logic unit (ALU), a control unit, processor registers, a memory, and input/output devices. A characteristic of the von Neumann model is that both program instructions and data are saved in the same memory, consisting of a Memory Address Register (MAR) and a Memory Data Register (MDR). Note that both instructions and data are passed from the memory to the processor (consisting of processing and control unit), or vice versa, via the same path, Shiva [2007].

The control unit maintains a program counter (PC) and an instruction register (IR). The PC contains the memory address of the next instruction to be executed, moreover it fetches an instruction into the IR. Those instructions are then used to "determine the action to be taken by all components of the computer", Kirk and Hwu [2013]. A sketch of the von Neumann model is given in Figure 2.1.

The von Neumann model describes an architecture that can execute various programs without modifications to the hardware. The execution on it is completely sequential (one instruction is executed after another), a deterministic program sequence is guaranteed. However, there is one prominent weakness: As instructions and data share the same path to (and from) the processor, and since the throughput between the memory and the processor is limited, especially compared to the processor frequency, there is a data/instruction latency and the processor idles for several cycles in situations in which it has not received the needed data/instruction. This is called the *von Neumann bottleneck*, a term coined by John Backus in his 1977 ACM Turing Award lecture:

> "Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically

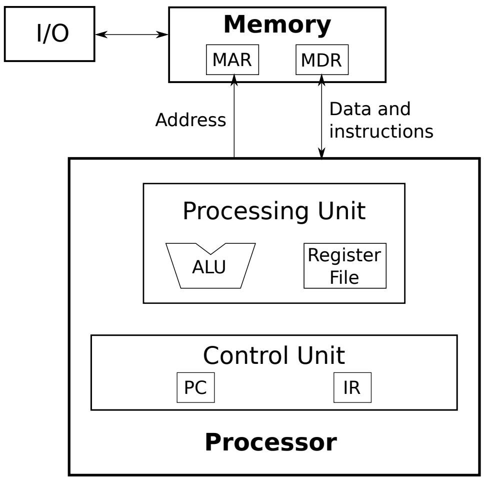


Figure 2.1: *A sketch of the von Neumann architecture, based on Kirk and Hwu [2013] and Shiva [2007].*

planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it." Backus [1978]

2.2 Parallel Computing on Shared-Memory Parallel Computers

The first shared-memory parallel computers, which we will abbreviate in accordance with Chapman et al. [2007] as SMPs in the following text, were already manufactured in the 1980s and have gained popularity in the server market in where they maintained major importance. There are systems using over a thousand CPUs, however in recent years those systems gained importance on a much smaller scale, too. Today, virtually every reasonably modern laptop or desktop computer has a multi-core CPU. No matter the scale, as the name suggests, all processors in those systems have access to (and thus share) the same memory.

In this section, we will present the basic architecture of SMPs and give an introduction to how one can take advantage of them for one's programs using OpenMP.

2.2.1 Shared-Memory Architecture

The setup of a shared-memory architecture is fairly simple. As an example, consider Figure 2.2, which portrays the architecture of a dual core processor. In this example there are private level 1 caches: the instruction cache, the data cache, and a Translation-Lookaside Buffer (TLB) which is an address cache. In level 2 there is a shared unified cache, meaning that both cores have access to it and that the chache hosts both data and instructions. A higher level number means more distance to the core, which implies that the caches are bigger but slower. Additionally both cores have equal access to the main memory. This is an example of a Uniform Memory Access (UMA) architecture, in which all the processors or cores have the same distance and therefore the same access speed to all the data in the memory. Figure 2.3 shows a UMA and a Non-Uniform Memory Access (NUMA) platform. A problem of NUMA systems is *memory consistency*: "When a processor of an SMP stores results of local computations in its private cache, the new values are accessible only to code executing on that processor. If no extra precautions are taken, they will not be available to instructions executing elsewhere on an SMP machine until after the corresponding block of data is displaced from cache. But it may not be clear when this will happen. Infact, since the old values might still be in other private caches, code executing on other processors might continue to use them even then", Chapman et al. [2007]. Most modern NUMA systems however have mechanisms to prevent such behaviour, i.e. updates to data on one processor will be communicated to all other processors if needed. Those systems are hence called *cache coherent* NUMA (or simply cc-NUMA) systems.

SMP systems can be classified as *multiple instruction, multiple data* (MIMD), allowing completely independent control of each of the processors, both in instructions to be executed and data to be processed.

In the following we will only say "processor" even though we may actually refer to a core of a processor.

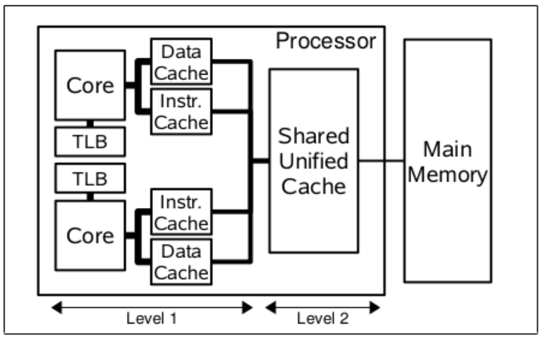


Figure 2.2: *Block diargram of a generic, cache-based dual core processor.* Chapman et al. [2007]

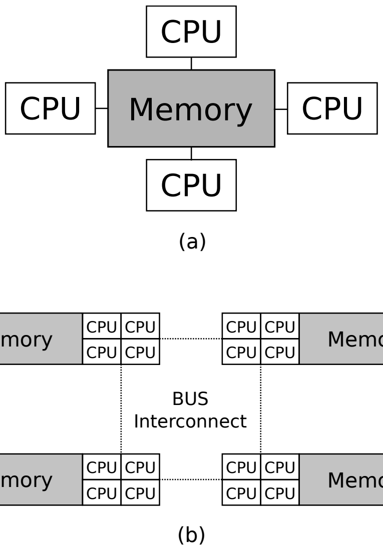


Figure 2.3: *Comparison between UMA (a) and NUMA (b) platforms, compare* Barney [2016].

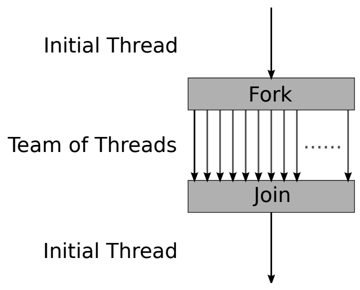


Figure 2.4: *The Fork-Join Model.*

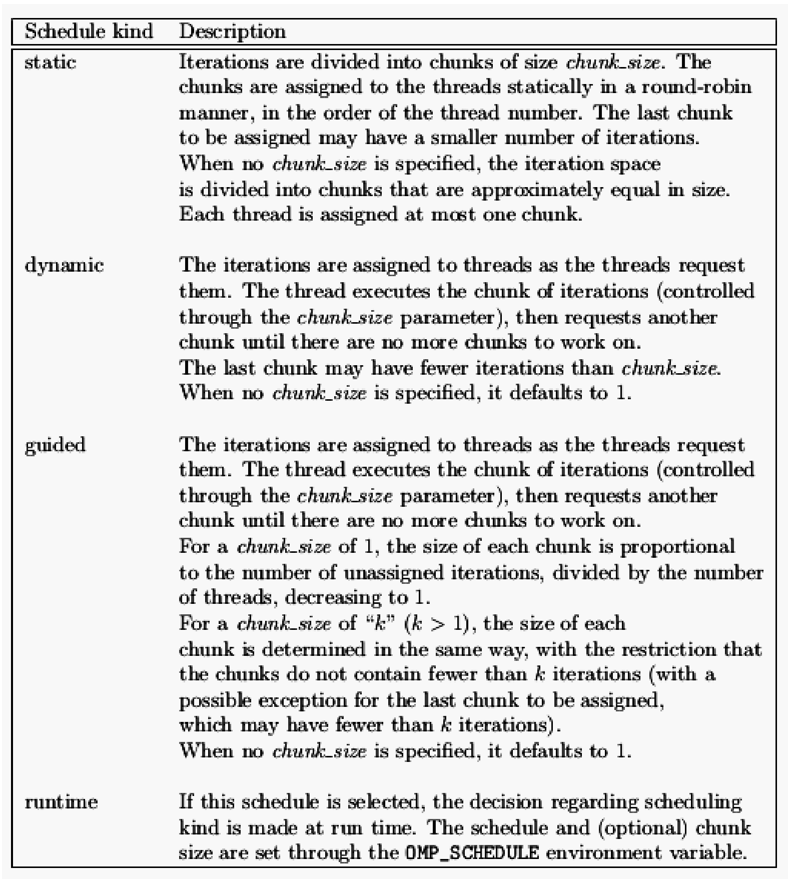
2.2.2 Programming Shared-Memory Parallel Computers: OpenMP

OpenMP is a shared-memory application interface (API), applicable to codes written in Fortran and C/C++. Rather than being a new programming language it is a set of so-called compiler *directives* that can be added to the serial code, as well as runtime library routines and environment variables. OpenMP directives are instructions in a special format that can only be understood by OpenMP compilers, meaning that for non-OpenMP compilers (or compilations without the respective OpenMP flag) those instructions will be ignored. The directives have the appearance of comments (`!$omp`) for regular Fortran compilers, or of pragmas (`#pragma omp`) for regular C/C++ compilers. The directives describe how the work is shared among threads executed on different processors, as well as how data is shared among different processors. A *thread* is "a run-time entity that is able to independently execute a stream of instructions" Chapman et al. [2007], or, according to Kirk and Hwu [2013], "[a] thread in modern computers is a virtualized von Neumann processor".

We use OpenMP to exploit parallelism for the available SMP, however there are other ways to achieve this, for example POSIX threads (Pthreads) or the Message Passing Interface (MPI), among others. Many compilers also provide flags for automatic program parallelization, however the compiler often lacks the necessary information to do a good job. The top priority for the compiler is to ensure the correctness of the results, thus when it is in doubt whether parallelization might be infringing the correctness of the computations (for example when there may be data dependence) the compiler will opt for not parallelizing the piece of code (Chapman et al. [2007]).

A big advantage of OpenMP is that one can parallelize a given sequential code incrementally: It is possible to parallelize one portion of the initially serial program at a time, thus making sure that the parallelization does not affect the correctness of the program. Furthermore, precisely because its directives are ignored by non-OpenMP compilers, it lets parallelized programs remain executable on older compilers not offering OpenMP. OpenMP supports the so-called fork-join model, portrayed in Figure 2.4: The program begins sequentially with an *initial thread* until an OpenMP `parallel` construct is encountered and the initial thread is creating a team of threads (this is referred to as the *fork*). Within the team of threads, the initial thread turns into the *master thread* of the team. All the members of the team collaborate to execute the part of the code that is enclosed by the parallel construct. At the end of that construct only the master thread remains (turning itself back to the initial thread), while all the other threads terminate. This is the *join*. The part within the parallel construct is called the *parallel region*. In general, the different threads will take a different amount of time to reach the end of the parallel region, and the master thread may have to wait at the end of that region until all the other threads of the team terminate before it can continue to execute sequentially.

By default, all data is considered to be shared among the threads (`shared`). This means that all the data is copied into the caches of the processors and updated in all caches whenever the data changes. Though for certain situations this behaviour can be a problem and it is desired to give each thread a private copy of some variable from the master thread (`firstprivate`) or to create variables within the parallel region (or within some construct inside the parallel region) that stay private to each thread (`private`). Observe that private data is undefined before that construct is entered or exited. If it is necessary to preserve the value of a variable after such a construct, `lastprivate` will need to be used. The default behaviour can be changed using the `default` clause, though.



Schedule kind	Description
static	Iterations are divided into chunks of size *chunk_size*. The chunks are assigned to the threads statically in a round-robin manner, in the order of the thread number. The last chunk to be assigned may have a smaller number of iterations. When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size. Each thread is assigned at most one chunk.
dynamic	The iterations are assigned to threads as the threads request them. The thread executes the chunk of iterations (controlled through the *chunk_size* parameter), then requests another chunk until there are no more chunks to work on. The last chunk may have fewer iterations than *chunk_size*. When no *chunk_size* is specified, it defaults to 1.
guided	The iterations are assigned to threads as the threads request them. The thread executes the chunk of iterations (controlled through the *chunk_size* parameter), then requests another chunk until there are no more chunks to work on. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned iterations, divided by the number of threads, decreasing to 1. For a *chunk_size* of "k" ($k > 1$), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations (with a possible exception for the last chunk to be assigned, which may have fewer than k iterations). When no *chunk_size* is specified, it defaults to 1.
runtime	If this schedule is selected, the decision regarding scheduling kind is made at run time. The schedule and (optional) chunk size are set through the `OMP_SCHEDULE` environment variable.

Figure 2.5: *Taken from Chapman et al. [2007]. "Schedule kinds supported on the schedule clause – The* `static` *schedule works best for regular workloads. For a more dynamic work allocation scheme the* `dynamic` *or* `guided` *schedules may be more suitable."*

Another important aspect of parallel programming (at least for SMPs and GPUs) is that thread execution is not in order. This means that given a set of processors $p_1, \ldots, p_n$ and a set of threads $t_1, \ldots, t_n$ one cannot foresee which thread t_i is run on which processor p_j, or even which thread is run first and so forth. Moreover, when rerunning the programming in parallel it is observable that the order in which the threads are run (or on which processor they are run) is arbitrary once more. For this reason it is important to write parallel programs whose results do not depend on the order of execution of the threads. The errors in results due to this order dependence are known as *(data) race conditions*.

Sometimes it is important for all threads to wait until all threads terminated a given portion of the program before they enter the next one in order to prevent data corruption, so that data is not read before it has been completely (created or) modified, or vice versa. This is known as *synchronization* or as a *barrier*. The `barrier` directive offers this in OpenMP. However there is an implied barrier after work sharing constructs (`do`/`for`, `sections`, `single`, or `workshare` (Fortran only)) as well as after a parallel region, too. If the implicit barrier after work-sharing constructs is not desired, one can put the `nowait` directive after it to disable synchronization at that point.

In many applications it is desirable to parallelize loops in which each loop iteration is independent of the other iterations. It is necessary to know beforehand how many loop iterations will be performed; `while`-loops are not parallelizable in general. There are different ways how a loop can be parallelized with OpenMP, using different schedule clauses, compare Figure 2.5. The parallelization of a given loop can be classified as SIMD, or *single instruction, multiple data*. For tightly nested loops it is possible to let the compiler collapse them (or a portion of them) into one single big loop using the `collapse` directive which is then executed in parallel according to the schedule clause.

Reduction operations significantly prosper from parallelism, too. A reduction operation is an operation that reduces an array of values into a single value while using only one single mathematical operation like

$+, -, *, /$, min, or max, among others. A variable that will be reduced needs to be declared as such at the beginning of the corresponding parallel region in the same way a private or shared variable needs to be declared. In fact, it can be considered to be a special kind of merger between a shared and private variable: "At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable." Barney [2016]

In the following we will briefly discuss some performance considerations for programs using OpenMP. For further details we refer to Chapman et al. [2007]. It should be no surprise that achieving good performance for parallel programs requires even more care than for serial programs. For both serial and parallel programs memory access patterns have a great impact on performance, however for programs parallelized using OpenMP there are additional factors, for example:

- Fraction of work in the parallel region that is sequential or replicated;
- Amount of time handling OpenMP constructs (directives and routines in OpenMP come with overheads);
- Load imbalance between synchronization points;
- Other synchronizations costs.

Replicated work are instructions in the parallel section that already have been performed by the initial thread before forking. Work is being serialized within a parallel region by using the `critical`, `atomic`, `single` or `master` constructs. The OpenMP overhead is additional work or computational cost incurred by the creation and handling of parallelism like starting and ending parallel regions, sharing work among threads or all kinds of synchronization. The sources of overheads include

- starting up threads and creating their execution environment;
- encapsulation of a parallel region within a separate function;
- computing schedules;
- (un)blocking threads;
- threads fetching work and signalling that they are ready.

The amount of overheads depend on the OpenMP translation strategy used by the compiler, the characteristics of the run-time library routines and the way they are used, the target platform, and how the compiler optimizes the code. Different examples of overheads are portrayed in Figures 2.6 and 2.7 which are measured on the EPCC microbenchmarks for the first version of the OpenUH compiler. Note that, for example, even though the dynamic schedule is substantially more expensive than the static schedule, its use may still lead to a better overall performance due to a reduction of thread idle times in situations of high load imbalance.

There are some general broad rules of thumb of how to achieve good performance with OpenMP. For example, as discussed previously, synchronization operations are expensive and should therefore be used as little as possible though as much as necessary to guarantee correct computations, for instance, skipping implicit barriers whenever they are not explicitly required. Furthermore, it is advisable to maximize parallel regions, to avoid large critical regions, to avoid parallel regions in inner loops (as this leads to potentially extensive overheads), as well as to address poor load balance. Whereas those rules of thumb will generally yield a better performance, there are some considerations that may have a notable impact on the performance, though depend on different factors like target platform, the compiler, and the code itself. For example one may gain performance by using the `single` rather than the `master` construct (or vice versa), or by letting certain data be `private` rather than `shared` (or vice versa).

Another more intricate aspect is the so-called *false sharing*, which is a side effect of cache line granularity of cache coherence implemented on SMPs. In order to understand what false sharing is consider the following situation. When different caches contain the same cache line of some shared data and a bit

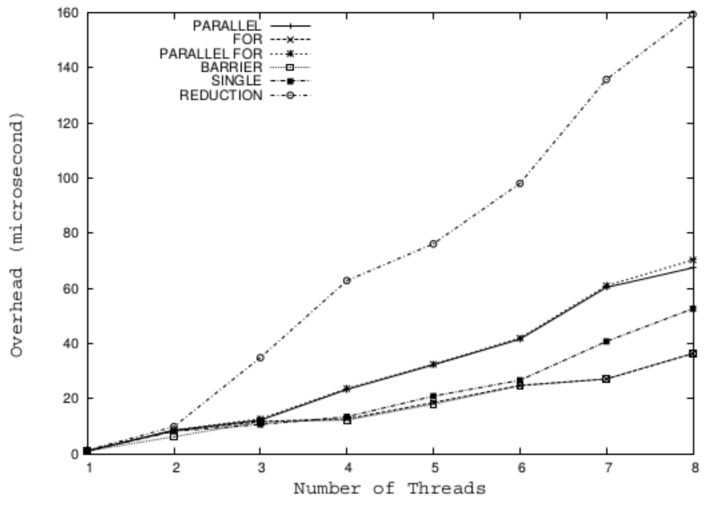


Figure 2.6: *Overheads of several common OpenMP directives and constructs. Taken from Chapman et al. [2007]. Note that the overhead of reduction operations for this compiler is unreasonably high, and not representative for other compilers. Chapman et al. [2007]*

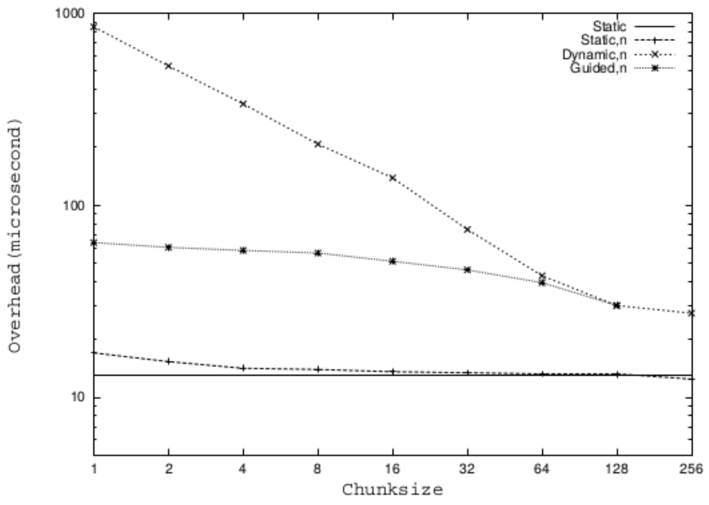


Figure 2.7: *Overheads of different kinds of OpenMP loop schedules. Taken from Chapman et al. [2007].*

(or more) of this cache line is modified on one cache then the "state bits" of that line on the other caches indicates that data within the cache line is no longer valid. There is no mechanism to indicate that only a portion (or even which portion) of the cache line is invalidated, hence the entire cache line will be updated. Depending on the implementation this update may be fetched from the cache of another processor or from the main memory. False sharing is the effect caused by two or more threads updating different data elements in the same cache line. An extreme example would be an array `arr` (partially) stored in a given cache line and every element of `arr` in that cache line being modified by a different thread. In this scenario, after one thread modified an element of `arr` in that cache line within its cache, the cache line in all other caches needs to be updated before the next thread could modify another element of `arr` in this cache line, and so forth. It is stated in Chapman et al. [2007] though that modest amounts of false sharing do not have a significant impact on performance, but will do if all of the following three conditions are satisfied:

1. Shared data is modified by multiple threads;
2. The access pattern is such that multiple threads modify the same cache line(s);
3. These modifications occur in rapid succession.

It is noted that private data substantially reduces the risk of false sharing[1], and that read-only data does not create false sharing.

In order to achieve high performance for an OpenMP program both experimentation and analysis are needed, especially for the more important program regions. Even though OpenMP directives can be directly added to an existent serial code, sometimes it might be advantageous to rewrite certain parts of the code in a way that more parallelism can be exploited. For example optimizations of the parallelization of loops may include loop unrolling (and jamming), loop interchanges, loop fusion/fission, or loop tiling, compare Chapman et al. [2007].

2.3 Parallel Computing with GPUs

In this section we will treat CUDA GPUs. CUDA (an acronym for Compute Unified Device Architecture) is both a parallel computing platform and an programming model created by NVIDIA which was first "unveiled" in 2006[2].

The GPU is used as an accelerator, meaning that a program is still started and processed on the CPU which then sends data-parallel functions, so called *kernels*, and the necessary data for their execution to the GPU. The GPU then runs the kernels in an ideally highly parallel manner.

The GPU is commonly referred to as *device* whereas the rest of the computer is called *host*.

2.3.1 GPU Architecture

The architecture of GPUs is tremendously more intricate than of SMP systems. The CUDA architecture varies notably from one generation to the next (this seems to be especially true for the GPU chip), however some architecture aspects remain constant. For example, as can be seen in Figure 2.8, considerably more transistors are used as ALUs (or cores in general) on a GPU than on a CPU.

"A GPU is connected to a host through a high speed IO bus slot, typically PCI-Express in current high performance systems", Wolfe [2012]. Off-chip, the GPU possesses its own DRAM memory up to several gigabytes. It is used for the global (often called device memory) and local memory of the GPU. Between the GPU and the host memories, data is usually transferred using direct memory access (DMA), see Figure 2.9. "The device memory supports a very high data bandwidth using a wide data path. On NVIDIA GPUs, it's 512-bits wide, allowing sixteen consecutive 32-bit words to be fetched from memory in a single cycle", Wolfe [2012]. For GPUs it is therefore of major importance to access memory consecutively, as strided accesses cause severe effective bandwidth degradation.

Note however that "[t]here is a trend to integrate CPUs and GPUs into the same chip package, commonly referred to as *fusion*. Fusion architectures often have a unified memory space for host and devices. There are new programming frameworks, such as GMAC, that take advantage of the unified memory space and eliminate data copying costs", Kirk and Hwu [2013].

As an example for a general CUDA capable GPU architecture, we will focus on the architecture of the Kepler GK110 GPU chip NVIDIA [2012], which is built in the Tesla K40 GPU used for our work. The

[1]According to Chapman et al. [2007] false sharing can still happen when there are different private data in the same cache line.
[2]See http://www.nvidia.com/object/cuda_home_new.html, accessed 01.09.2016

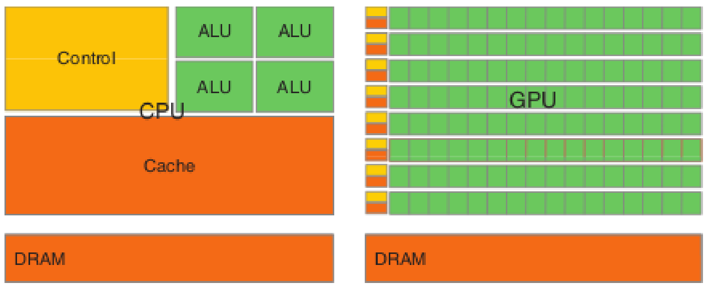


Figure 2.8: *A comparison of the schematic architectures of CPUs and GPUs. Taken from Kirk and Hwu [2013]*

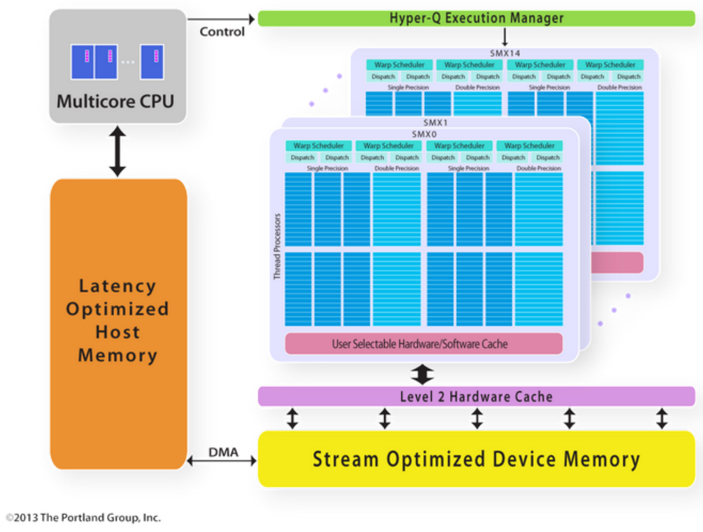

Multicore CPU
Control
Hyper-Q Execution Manager
SMX14
SMX1
SMX0
Warp Scheduler
Dispatch
Single Precision
Double Precision
Thread Processors
User Selectable Hardware/Software Cache
Latency Optimized Host Memory
Level 2 Hardware Cache
DMA
Stream Optimized Device Memory
©2013 The Portland Group, Inc.

Figure 2.9: *NVIDIA Kepler Block Diagram. Taken from Wolfe [2012].*

GK110 GPU chip consists of 15 streaming multiprocessors (in this specific generation called SMX, though normally referred to as SM), an L2 cache, memory controllers, and the GigaThread engine (which schedules thread blocks to various SMs, see Section 2.3.2), compare Figure 2.10. A more detailed block diagram of the architecture of a single SM is given in Figure 2.11. The SM mostly consists of 192 single precision and 64 double precision cores (in Figure 2.11 labelled as Core and DP Unit respectively), as well as various different kinds of memory: The instruction cache, register files, shared memory/L1 cache, read only cache, and the texture memory (Tex) which is another read only memory. Observe that the shared memory is a user- (or software-) managed data cache, and that the shared memory/L1 cache can be configured to be 32 kB/32 kB, 48 kB/16 kb, or 16 kB/48 kB. Furthermore the SM has 32 special function units (SFU) (for example for functions like the sine, tangent, or the exponential) and 32 load/store units (LD/ST). Additionally, the SM inhibits four warp schedulers and eight instruction dispatch units. This means that for the GK110 chip four warps can run concurrently per SM with two independent instructions per cycle, compare Figure 2.12. A *warp* is a group of 32 parallel threads. NVIDIA calls the parallelism in which a warp executes its work SIMT (single instruction, multiple threads). SIMT is very similar to SIMD parallelism, though handles conditional operations somewhat differently than SIMD (see the discussion on control divergence in Section 2.3.2). As is shown in Figure 2.8, a substantial part of the transistors of a CPU is used for the cache. And as a matter of fact, a great deal of the performance of a CPU is due to cache optimizations: All data is ideally available in some level of the cache whenever it is needed by the computation of the CPU. That is, CPUs are *latency optimised*, where latency is the time needed from the request of some data until it is available to the processing unit. Even though GPUs have been equipped with L1 and L2 caches in recent CUDA architectures, they remain to be designed for throughput computing and therefore handle memory performance differently than CPUs. GPUs work with *latency hiding*: Even though a memory request to the device memory can take up to several hundred clock cycles, GPUs tolerate this high latency by using a high degree of parallelism or *multithreading*. For example, the GK100 architecture supports up to 64 active warps on each SM (though only four warps can work on an SM concurrently at any given time). Hence, when a warp stalls due to a memory operation, another warp is selected by the control unit that has all the data available to perform its computations. As long as there is enough work for the data available to the cores, they will be kept busy and the memory operation will virtually not affect the overall performance.

Note that different GPU generations may have a different number of streaming multiprocessors, as well as a varying amount of single/double precision cores per streaming multiprocessor and differently sized memories, to name only a few architecture changes. Additionally, earlier GPU generations may not carry an L2 cache.

The newest CUDA architecture (as of the time of this writing), namely the P100 chip, can be found at https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. To emphasize the architectural difference to the GK100 chip, note that the P100 includes 56 SMs of 64 single precision and

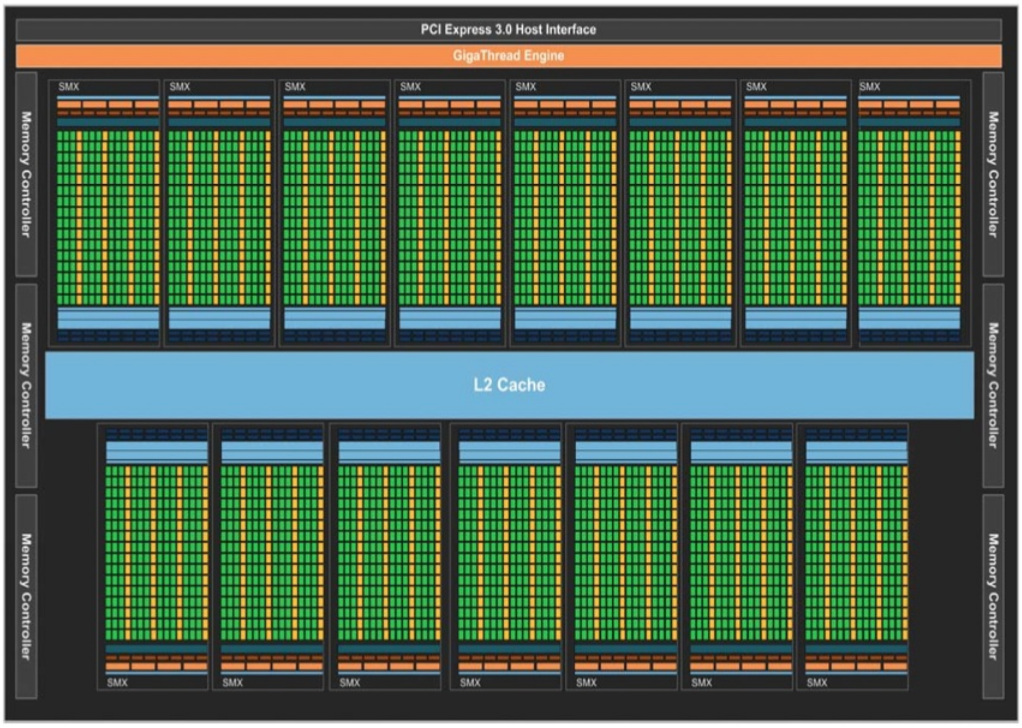


Figure 2.10: *Block diagram of the GK110 chip architecture. Taken from NVIDIA [2012].*

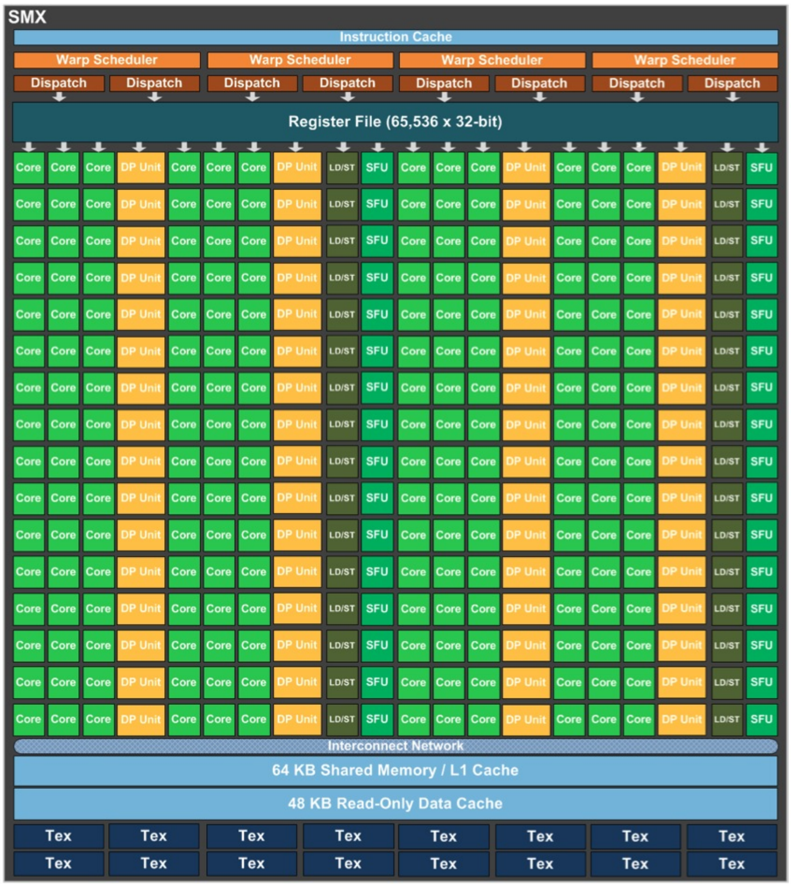

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Figure 2.11: *Block diagram of a single Streaming Multiprocessor of the GK110 chip. Taken from NVIDIA [2012].*

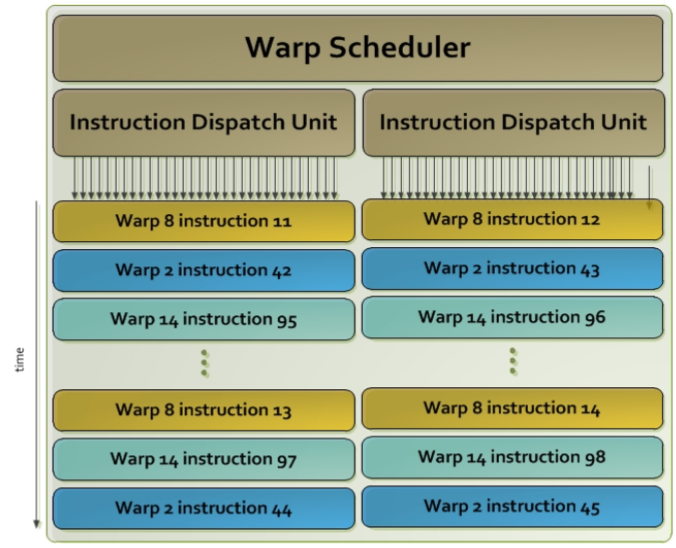


Figure 2.12: *Graphic representation of a warp scheduler. Taken from NVIDIA [2012].*

32 double precision cores each, for example.

2.3.2 Programming GPUs: CUDA

The programs using CUDA in this work were written in CUDA Fortran, which is based on the CUDA C runtime API. “There are a few differences in how CUDA concepts are expressed using Fortran 90 constructs, but the programming model for both CUDA Fortran and CUDA C is the same.

“The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the host refers to the CPU and its memory, while the device refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches *kernels* which are functions executed on the device. These kernels are executed by many GPU threads in parallel.

“Given the heterogeneous nature of the CUDA programming model, a typical sequence of operations for a CUDA Fortran program is:

1. Declare and allocate host and device memory.
2. Initialize host data.
3. Transfer data from the host to the device.
4. Execute one or more kernels.
5. Transfer results from the device to the host.” Ruetsch [2012]

In order to declare a device variable (or array) in the host code it is necessary to use the `device` variable attribute. This implies that any host variable cannot be used on the device and vice versa. It is therefore common practice that, for example, if a host variable has the name `var`, then the counterpart variable on the device will be called `var_d`. For CUDA Fortran it is possible to not only allocate the device memory in the host code but also to copy host data to or from the device directly (for example by writing `var_d = var`). For CUDA C, this option is not available and it is necessary to use the `CudaMemCpy` function (which is available to CUDA Fortran, too).

When launching a kernel one has to define a grid of thread blocks in the following way.

```
call kernel_name<<<grid,tBlock>>>(var-list...)
```

In between the triple chevrons is the so-called *execution configuration*. As a bare minimum one has to declare how many thread blocks shall be launched in the kernel’s grid (`grid` variable) as well as how many threads per thread block (`tBlock` variable). Both of these variables can either be one-dimensional or of the derived type `dim3` which contains the x, y and z components. The choice of dimensions for each of these variables is independent of one another, for example we can choose to have a one-dimensional grid of two-dimensional thread blocks. There is a hardware limit of how many blocks per SM and how many threads (warps) per SM can be launched, for example.[3] An optional though frequently used argument in the execution is the shared memory allocation size per thread block (in bytes). It is put after the blocks/grid and threads/block arguments.

As was discussed in Section 2.3.1, a group of 32 threads creates a thread unit called *warp* which operates in an SIMT fashion. It is possible to select the above variable `tBlock` in such a way that it is not a multiple of 32 in any of the components, however when the kernel is launched its thread block will be padded to form a multiple of 32 threads (and all of these threads will perform the same SIMT operations). In those situations special care must be taken that those extra threads do not modify data elements: Observe that every thread within thread block has local indices (`ThreadIdx%x`, `ThreadIdx%y`, `ThreadIdx%z`). Together with the block indices (`BlockIdx%x`, `BlockIdx%y`, `BlockIdx%z`) and the block dimensions (`BlockDim%x`, `BlockDim%y`, `BlockDim%z`) it is possible to calculate the global index of a thread:

$$\texttt{id\_x} = \texttt{ThreadIdx\%x} + \texttt{BlockIdx\%x} * (\texttt{BlockDim\%x} - 1)$$

or, for C/C++,

$$\texttt{id\_x} = \texttt{ThreadIdx.x} + \texttt{BlockIdx.x} * \texttt{BlockDim.x}$$

In this manner it is possible to let different threads access different elements of the same data. But those threads that were created to pad thread blocks into multiples of 32 have those local thread indices, too. Therefore, unless we specifically exclude them from taking any action (after letting them determine their global indices), they will access data in the same way as all the other requested threads do. This can be done in the following way (for a one-dimensional block and grid):

```fortran
attributes(global) subroutine kernel(num_threads_requested, ...)
implicit none
integer :: num_threads_requested
...

id_x = threadIdx%x + BlockIdx%x*(BlockDim%x-1)
if (id_x <= num_threads_requested) then
   ! do some work
   ...
endif

end subroutine kernel
```

There are a couple of things to discuss from the above code snippet. First of all, note that the `attributes(global)` indicates that this subroutine is called from the host and executed on the device. Analogously, `attributes(device)` indicates that a subroutine is called from and executed on the device. There is also an optional `attributes(host)` qualifier stating that the subroutine is called from and executed on the host, however in practice this is rarely used (if ever). Next, we see that only the threads that were requested in the execution configuration will be performing the work of this kernel. Lastly, we already briefly mentioned in Section 2.3.1 that the SIMT parallelism handles conditionals somewhat different than SIMD. This is an important aspect of the performance of GPUs: The GPU cannot execute different conditional paths concurrently. It rather first executes all the threads for which the condition is true in the `if`-construct. Afterwards all threads will be executed that satisfy the `else if`-condition(s) (if there are any), and finally it executes all the threads in the `else` path. This means that all these paths will be executed sequentially one after another, hence degrading parallel performance substantially. For that reason it is advised to divide work among the warps such that ideally all threads of a given warp follow the same conditional path.

A different issue to keep in mind when launching a kernel on the device is that while the use of registers

[3]For the GK100, these limits are 64 warps/SM, 16 thread blocks/SM and 1024 threads per thread block.

Table 2.1: *CUDA variable type qualifiers for different memories. Kirk and Hwu [2013]*

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
Fortran: `integer, device, shared :: SharedVar` C: `__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
Fortran: `integer, device :: GlobalVar` C: `__device__ int GlobalVar;`	Global	Grid	Application
Fortran: `integer, device, constant :: ConstVar` C: `__device__ __constant__ int ConstVar;`	Constant	Grid	Application

and especially shared memory can be extremely helpful to reduce the access to global memory and therefore can potentially increase performance drastically (as variables residing in these memories can be accessed at “very high speed in a highly parallel manner”, Kirk and Hwu [2013]), extensive use of these resources can actually limit parallelism. This is due to the fact that these memory resources are limited on the SM. This implies that if a block exceeds these capacities less and less threads will be able to run concurrently on the SM. As an example, on the GK100 chip used for our work, each SM can host up to 2048 threads and 65536 registers. This means that to make use of all the threads available on the SM each thread can only use 32 registers, for if they would require using 33 registers, less threads will be available to run concurrently on the SM. Such a reduction in threads is realized at block granularity. For example, if each block launched in this kernel would consist of 512 threads, the need to use 33 registers would result in there only being $2048 - 512 = 1536$ threads available on the SM – a thread reduction of 25% for an increase of 3.125% of registers per thread. Analogous behaviour occurs for the shared memory. On the GK100 each SM has up to 48 kB of shared memory and can hold up to 16 thread blocks. To achieve the maximum number of thread blocks (assuming that every thread block has no more than 128 threads) each block must not use more than 3 kB of shared memory in this configuration (if the shared memory/L1 cache is not configured to be 48 kB/16 kB for this architecture, each block will have less shared memory available in order to let all 16 blocks run on the SM).

Shared memory is very effective for non-coalesced memory accesses (for example for accesses along the rows in Fortran, or along the columns in C, or other strided memory accesses). One can load the elements into the shared memory in a coalesced manner from the global memory and then access those elements in whatever fashion in the shared memory, as accesses to shared memory are significantly less expensive than to global memory.

Table 2.1 lists how variables are put in different CUDA memories, and what their scope and lifetime is. Observe that the texture memory is omitted in this table as handling it is substantially more intricate. See Gupta [2013] for an introduction to using texture memory in CUDA.

Constant variables are stored in the global memory but are aggressively cached. It “supports short-latency, high-bandwidth, read-only access by the device when all threads simultaneously access the same location”, Kirk and Hwu [2013].

As was already discussed for OpenMP in Section 2.2.2, synchronization is one of the major aspects of parallel programming. For GPUs this is just as valid, though slightly more complicated as for SMPs, as “GPUs do not support a fully coherent memory model that would allow [SMs] to synchronize with each other” Wolfe [2012]. However, it is possible to synchronize threads within the same block. Synchronizing all threads of the grid is only possible at the end of a kernel, before a new kernel launched from the same stream starts. It is possible to let different kernels execute simultaneously when they are launched via different streams, however often the kernels are sufficiently large to fill the entire device one by one.

Threads of one SM cannot send results to threads of other SMs, nor is it possible to create “a critical section among all threads of the whole system. [. . .] Threads in a single block will be executed on a single

[SM], sharing the same [shared memory], and can synchronize and share data with threads in the same block [. . .]. [The threads of a different block] may be assigned to different [SMs] concurrently, to the same [SM] concurrently (using multithreading), or may be assigned to the same or different [SMs] at different times, depending on how the blocks are scheduled dynamically", Wolfe [2012]. As stated in Section 2.3, due to latency hiding, warps become inactive when executing global memory operations and other warps that have all the resources available for execution become active instead. Since warps cannot migrate from one SM to another, the inactive warp will resume its execution on the same SM after its memory operation has finished.

Note that as of the Kepler architecture generation it is possible for kernels to launch sub-kernels, which is called *dynamic parallelism* by NVIDIA. This helps the device to "generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths", NVIDIA [2012], all without involving the host.

The most severe difference between CUDA Fortran and CUDA C is probably that CUDA Fortran provides "kernel CUDA Fortran loop directives", also called CUF kernels, to parallelize loops or perform reductions on the device in the host code without having to write an explicit kernel. This is a huge advantage, as writing efficient reduction operations is no trivial task. As an example for CUF kernels consider the following code snippet.

```
!$cuf kernel do <<<*,*>>>
do i=1,N
   y_d(i) = y_d(i) + a*x_d(i)
   xsum = xsum + x_d(i)
enddo

y=y_d
```

Here, between the triple chevrons, once more we have the execution configuration. But unlike earlier when we invoked a proper CUDA kernel, by writing asterisks in it we can also opt for letting the compiler choose a configuration. If the left-hand side of an expression within a CUF loop kernel is a host scalar variable, a reduction operation will be performed on the device – in this loop, this is created by `xsum = xsum + x_d(i)`. Note that reduction operations on the GPU always include control divergence at some point of the process.

However, the one big disadvantage of CUDA Fortran is that there is only one (commercial) compiler offered for it, namely the PG Fortran compiler (at least this is the case at the time of this writing). In contrast, CUDA C is provided for free by the NVIDIA C comipler (`nvcc`), which results in a bigger community with more active and more extensive user forums. Additionally there are more debugging applications for CUDA C than for CUDA Fortran.

As stated in Wolfe [2012], "[p]erformance tuning on NVIDIA GPUs requires optimizing all these architectural features:

- Finding and exploiting enough parallelism to populate all the [SMs].
- Finding and exploiting enough additional parallelism to allow multithreading to keep the cores busy.
- Optimizing [global] memory accesses for contiguous data, essentially optimizing for stride-1 memory accesses.
- Utilizing the [shared memory] to store intermediate results or to reorganize data that would otherwise require non-stride-1 [global] memory accesses". Wolfe [2012]

2.3.3 Programming GPUs: OpenACC

If it is desired to parallelize a given serial code with CUDA it is not possible to simply add some directives, in contrast to using OpenMP, or even to let a CUDA code run sequentially if the compiler has no

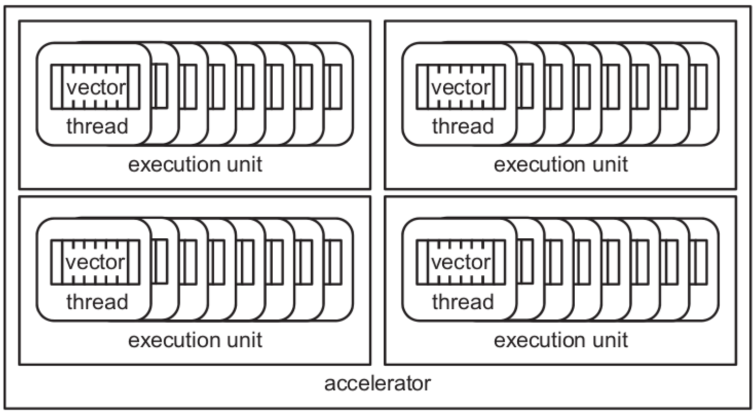


Figure 2.13: *Accelerator model used for OpenACC. Taken from Kirk and Hwu [2013].*

CUDA capability. OpenACC offers the same characteristics and advantages as OpenMP for GPUs. Equivalent to OpenMP, OpenACC is an API providing a set of compiler directives, library routines, and environment variables, and can be applied to codes written in Fortran and C/C++ in order to let them execute in parallel on accelerator devices, including GPUs. With OpenACC the threshold for scientists to let their code run on GPUs is diminished as they do not need to invest the time in learning CUDA and rewriting already existing code. Instead, it is possible to add some directives in the sequential codes in order to take advantage of the GPU. This is a great advantage for debugging purposes, too, because it is possible to incrementally parallelize an existing serial code and compare the results of the parallelized version with the sequential one. Its syntax is very similar to OpenMP. All of this is not very surprising when taking into consideration that the OpenMP Architecture Review Board (ARB) "has formed an accelerator working group to extend OpenMP support on accelerators [and that] [a]ll OpenACC founding members are members of this group", Kirk and Hwu [2013]. Furthermore, it is intended by this group to merge both specifications into a common one.

As OpenACC can parallelize serial code on different accelerator devices (not only GPUs), it considers a more general kind of architecture model, portrayed in Figure 2.13. In this architecture model, three levels of parallelism are considered. "At the outermost course-grain level, there are multiple execution units. Within each execution unit, there are multiple threads. At the innermost level, each thread is capable of executing vector operations", Kirk and Hwu [2013]. Due to its accelerator architecture model, its execution model is not divided into warps, blocks, and grids, but rather into vectors, workers, and gangs. "On a GPU, a possible implementation is to map a gang to a CUDA block, a worker to a CUDA warp, and a vector element to a thread within a warp. However, this is not mandated by the OpenACC specification and an implementation (compiler/runtime) may choose a different mapping based on the code pattern within an accelerator region for best performance", Kirk and Hwu [2013].

Another difference to CUDA is that some OpenACC directives are merely hints to the compiler, which may or may not be able to take full advantage of such hints. For this reason it is of paramount importance to check the information given on the parallelization by the compiler during compilation (for the PGI compiler this is done by the `-Minfo=accel` flag). The performance of an OpenACC program greatly depends on the capability of the OpenACC compiler, whereas for CUDA the parallelization of a program is expressed explicitly and therefore does not rely so much on the compiler. Additionally, the parallelization model of an OpenACC program is based on forking and joining, similarly to OpenMP. The joining process is the only way for synchronization with OpenACC. Compare Figure 2.14 for an example of an OpenACC fork/join process. Another characteristic of OpenACC is that there is no "reliable way to allow one execution unit to consume data produced by another execution unit. [...] Therefore, in OpenACC, different execution units are expected to work on disjoint memory sets. Threads within an execution unit can also share memory and threads have coherent memory", Kirk and Hwu [2013].

An OpenACC directive is preceded by `!$acc` (Fortran) or `#pragma acc` (C/C++). OpenACC offers two constructs that specify which part of the program is supposed to be executed on the accelerator: `parallel` and `kernels`. While both may be able to yield the same accelerator execution behaviour there are some

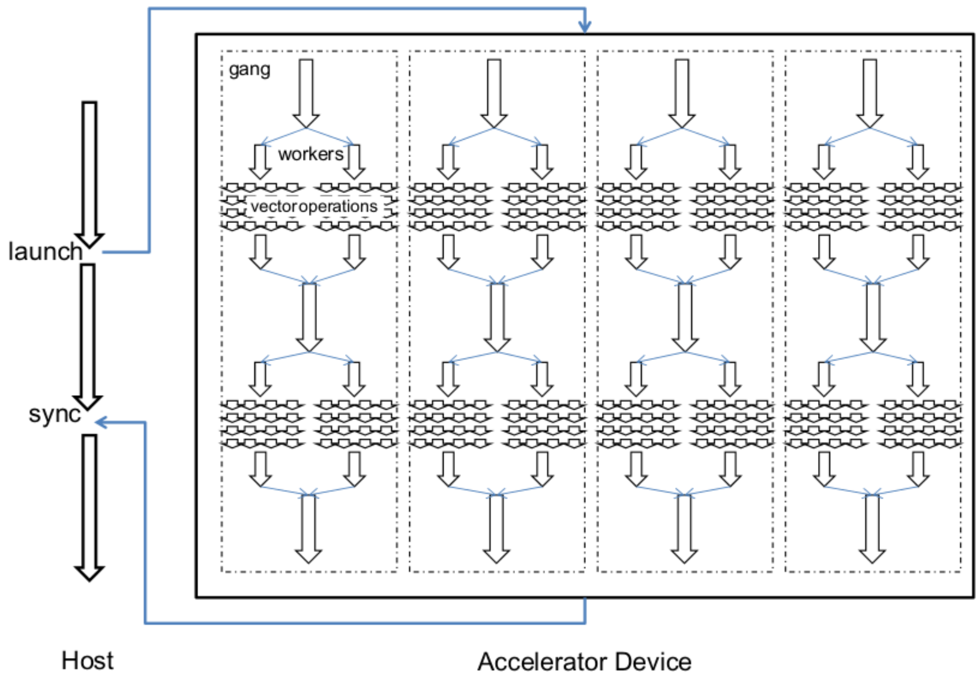


Figure 2.14: *OpenACC fork/join process. Taken from Kirk and Hwu [2013].*

important differences.

Using the `parallel` construct, a gang of workers are created, though only one worker will execute the parallel region initially and other workers will be deployed once there is more parallel work at an inner level. It is possible to specify the number of gangs, workers, as well as the length of the vector after the `parallel` construct using the `num_gangs`, `num_workers`, and `vector_length` clauses. If not specified by the user the implementation will select these values at runtime. In both cases these numbers are fixed for the remainder of the parallel section. Note while there is initially only one active worker per gang (in Kirk and Hwu [2013] described as *gang lead*) executing the code within the parallel region, unless specified otherwise, this code will be executed redundantly by all gangs (each using the gang lead) if the number of gangs is greater than one.

Similar to OpenMP's `do` (Fortran) or `for` (C/C++) constructs, OpenACC has a `loop` construct to parallelize loops in a parallel region. OpenACC also offers the `reduction` and `collapse` clause, and even a `tile` clause[4] which is not offered by OpenMP. As was mentioned earlier, OpenACC offers three levels of parallelism which also reflects on the loop constructs: The iterations of a loop can be be shared among gangs, workers and vectors, using the `gang`, `worker`, or `vector` clause afte the `loop` construct. The gang loop is shared among all gang leads and is used for outermost loops. When wanting to parallelize an inner loop one can use a worker loop which distributes the loop iterations among all workers within a gang. The vector loop is often used to parallelize the innermost loop in an SIMD fashion. This way of parallelizing loops is especially helpful for loops that are not tightly nested and therefore cannot be collapsed.

The `kernels` construct opens a region that may contain more than one kernel and each of these kernels

[4]From the OpenACC API OpenACC-Standard.org [2015] we have:

> "The `tile(`*`size-expr-list`*`)` clause specifies that the implementation should split each loop in the loop nest into two loops, with an outer set of *tile* loops and an inner set of *element* loops. The argument to the `tile` clause is a list of one or more tile sizes, where each tile size is a constant positive integer expression or an asterisk. If there are *n* tile sizes in the list, the `loop` construct must be immediately followed by *n* tightly-nested loops. The first argument in the *size-expr-list* corresponds to the innermost loop of the *n* associated loops, and the last element corresponds to the outermost associated loop. If the tile size is specified with an asterisk, the implementation will choose an appropriate value. Each loop in the nest will be split or *strip-mined* into two loops, an outer *tile* loop and an inner *element* loop. The trip count of the element loop will be limited to the corresponding tile size from the *size-expr-list*. The *tile loops* will be reordered to be outside all the *element* loops, and the *element* loops will all be inside the *tile* loops."

may have a different configuration of gangs, workers and vector lengths. For this reason, the `num_gangs`, `num_workers`, and `vector_length` clauses are optional on a `loop` construct within a kernels region but are not used on the `kernels` construct itself. The other big difference to the `parallel` construct is that the `kernels` construct is more *descriptive*, it tells the compiler the intentions of the user, however it is up to the compiler to create kernels within this region. There are two common reasons for the compiler to not generate a kernel for a `loop` construct: Safety and performance. Safety in this context means that the result of the parallel execution of the loop is the same as for the serial execution. If there is a data-dependency between two loop iterations this safety is not given.

The `kernels` construct is probably facilitating the porting of programs to OpenACC in a greater manner, however the "quality" of the generated accelerator code strongly depends on the capability of the compiler. The user has more control over the accelerator code using the more *prescriptive* `parallel` construct, which offers no safety net for data dependence however. Sometimes the compiler has not enough information available to guarantee that there is no risk of data dependence for a given loop. In those situations, the user can force parallelization using the `independent` clause to the `loop` construct, it is then the user's responsibility to guarantee that the parallelization indeed does not affect correct computation.

Efficient data management is even more crucial for OpenACC than for OpenMP, as data transfer from the host to the device (or vice versa) is highly expensive. It is therefore of major importance to let data reside as long as needed on the device memory. While it is possible to have data copied to and from the device for every kernel it is highly recommendable to set up data regions using the `data` construct before parallel/kernels regions. The `data` construct has the same data clauses available as other constructs such as `parallel`, `kernels`, or `loop`:

- `copyin` in order to copy data from the host to the device upon entering the data region;
- `copyout` in order to copy data from the device to the host immediately after the data region;
- `copy` which implies both `copyin` and `copyout`;
- `create` when only the allocation of data on the device memory is needed without copying the data back to the host after the data region.

Additionally, there is the `present` clause, however, from OpenACC 2.5 on all of the data clauses mentioned above behave like `present_or_copyin` etc. meaning that if there already is the desired data present on the accelerator, the present copy will be used, otherwise the data will be copied to or created on the device memory.

The same remarks regarding performance for OpenMP at the end of Section 2.2.2 are also valid for OpenACC (aside from false sharing).

2.4 Performance Metrics for Parallel Programming

For both, parallel programming on SMPs and GPUs there are certain metrics in order to gauge performance and to analyse potential ways to improve it. Due to the differences in architecture those metrics are not the same for these platforms either.

For example, for homogeneous SMPs in which one has various core within the same processors and/or various identical processors it makes sense to measure the *speedup* which compares the execution time of the serial program relative to the parallel one:

$$S = \frac{T_{ser}}{T_p},$$

where T_{ser} is the serial execution time and T_p the parallel one (with p denoting the number of processors used). The obvious question is whether a given program will get an arbitrary speedup if a sufficient amount of processors are available. The well-known *Amdahl's law*[5] answers this question from a theoretical point

[5]Amdahl's law is actually not a physical law but rather a model.

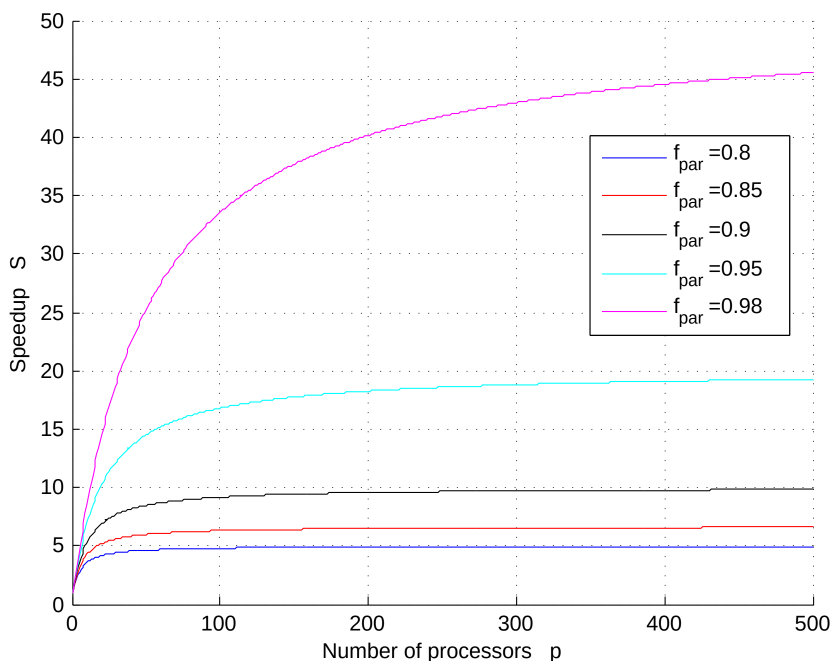


Figure 2.15: *Maximum speedup S achievable according to Equation* (2.1) *as a function of the parallel fraction of the program f_{par} and the number of processors used p.*

of view. The maximum speedup achievable due to Amdahl's law is

$$S = \frac{1}{\frac{f_{par}}{p} + (1 - f_{par})}, \tag{2.1}$$

with f_{par} being the parallel fraction of the program and p being the number of processors, Chapman et al. [2007]. For $f_{par} = 1$ Equation (2.1) yields $S = p$, and for $f_{par} = 0$ Equation (2.1) reduces to $S = 1$. Speedups according to Equation (2.1) with other choices of f_{par} and in function of p can be found in Figure 2.15.

It must be understood, though, that Amdahl's law is based on linear speedup in the sense of that the maximum speedup cannot be greater than the number of processors used. But as is stated in Chapman et al. [2007], "a parallel program has more aggregate cache capacity at its disposal, since each thread will have will have some amount of local cache. This might result in a *superlinear* speedup: the speedup exceeds the number of processors used." This positive effect can offset some of the performance loss caused by serial code and the various overheads.

Another popular metric used for SMP systems is the parallel *efficiency*

$$E = \frac{S}{p}.$$

Moreover, within the field of high performance computing, frequent metrics are strong scaling and weak scaling. Scalability in general describes how solving a bigger problem is handled with additional hardware resources. A program is said to be scalable if adding more processors reduces execution time in some proportionality to the processors added. In this context, *strong scaling* describes how the execution time of a program changes with the number of processors for a fixed total problem size. *Weak scaling*, on the other hand, describes how the execution time of a program changes with the number of processors for a fixed problem size per processor. With the given definitions it can easily be seen that speedup and strong scaling are identical.

For GPUs, the concept of speedup of a program does not work as for SMPs. This is due to differences in architecture, there is no connection between the execution time of the serial program on a CPU and the parallel execution on the GPU. Even though one is ultimately still interested in letting the program run as fast as possible using the GPU, the metrics used for GPU programming refer rather to the exploitation of the GPU architecture. For example one way to gauge GPU performance of a program is the *ratio of computation to global memory access* (CGMA): The more computation can be done for every unit of data retrieved from global memory, the higher the efficiency of the GPU code will be.

Furthermore, it is desired for a GPU application to yield as much giga floating point operations per second (GFLOPS) as possible on the given GPU. As was discussed in Section 2.3.2 the performance bottlenecks of a GPU are the global memory accesses (high latency, high but limited bandwidth), divergence, and the limited resources of an SM. All of these factors can result in lower GFLOPS of a program than the maximum possible GFLOPS of a GPU.

These and other metrics (for example how the resources of a SM are used) of how efficiently the GPU is utilized can be given via profiling software, like the NVIDIA CUDA Profiling Tools Interface (which already contains the NVIDIA Visual Profiler).

Overall, however, one is mostly interested in the acceleration of a program, meaning how fast it is possible to let the program execute with the help of the GPU.

Chapter 3

Mathematical Problem

In this chapter we discuss the mathematical problem we solve numerically. The necessary mathematical basis for this discussion can be found in the Appendix in Sections A.1 – A.4.

3.1 Discretization of the Two-Dimensional Poisson Equation

The Poisson Equation has many applications in steady state phenomena. For example it is used to describe the stationary pressure distribution in an incompressible flow field, stationary heat transfer via conduction, stationary mass transfer via diffusion, or for electrostatics and magnetostatics.

Let us now consider the continuous two-dimensional Poisson Equation with boundary conditions Φ on a rectangle

$$\begin{cases} -\Delta\tilde{u} = -(\tilde{u}_{xx} + \tilde{u}_{yy}) = f(x,y), & (x,y) \in \Omega = (0,a) \times (0,b) \\ \tilde{u}(x,y) = \Phi(x,y), & (x,y) \in \partial\Omega. \end{cases}$$

Here, the solution of the continuous problem is denoted with $\tilde{u}$, whereas we will denote the numerical solution of the discrete problem simply with u. We will discretize the derivations using centered finite differences:

$$\Delta_h u = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2},$$

with

$$h_x = \frac{a}{n+1}, \qquad h_y = \frac{b}{m+1},$$
$$u_{i,j} = u(ih_x, jh_y), \quad 1 \le i \le n, 1 \le j \le m,$$

and

$$u_{0,j} = \Phi(0, jh_y), \quad u_{N+1,j} = \Phi(a, jh_y),$$
$$u_{i,0} = \Phi(ih_x, 0), \quad u_{i,M+1} = \Phi(ih_y, b).$$

This way, the discrete problem can be stated as

$$\begin{cases} -\Delta_h u = f, & (x_i, y_j) \in \Omega_h = \{1 \le i \le n, 1 \le j \le m\} \\ u(x_i, y_y) = \Phi(x_i, y_j), & (x_i, y_j) \in \partial\Omega_h = \{0 \le i \le n+1, 0 \le j \le m+1\} \backslash \Omega_h. \end{cases}$$

In accordance to Isaacson and Keller [1996], let us now define

$$\delta^2 = \frac{h_x^2 h_y^2}{2(h_x^2 + h_y^2)}, \quad \theta_x = \frac{h_y^2}{2(h_x^2 + h_y^2)}, \text{ and } \quad \theta_y = \frac{h_x^2}{2(h_x^2 + h_y^2)},$$

so that we can write the discrete two-dimensional Poisson Equation as

$$
\begin{gathered}
A\mathbf{u} = \delta^2 \mathbf{F}, \\
A = \begin{bmatrix} B & -\theta_x I & & & \\ -\theta_x I & B & -\theta_x I & & \\ & \ddots & \ddots & \ddots & \\ & & -\theta_x I & B & -\theta_x I \\ & & & -\theta_x I & B \end{bmatrix}, \\
B = \begin{bmatrix} 1 & -\theta_y & & & \\ -\theta_y & 1 & -\theta_y & & \\ & \ddots & \ddots & \ddots & \\ & & -\theta_y & 1 & -\theta_y \\ & & & -\theta_y & 1 \end{bmatrix}, \\
\mathbf{u} = \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ \vdots \\ u_{n,1} \\ u_{1,2} \\ u_{2,2} \\ \vdots \\ u_{n,2} \\ \vdots \\ \vdots \\ \vdots \\ u_{n,m} \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} \mathbf{f}_1 + \frac{\mathbf{w}_1}{h_x^2} + \frac{\boldsymbol{\Phi}_0}{h_y^2} \\ \mathbf{f}_2 + \frac{\mathbf{w}_2}{h_x^2} \\ \vdots \\ \mathbf{f}_k + \frac{\mathbf{w}_k}{h_x^2} \\ \vdots \\ \mathbf{f}_{m-1} + \frac{\mathbf{w}_{[m-1]}}{h_x^2} \\ \mathbf{f}_m + \frac{\mathbf{w}_m}{h_x^2} + \frac{\boldsymbol{\Phi}_{m+1}}{h_y^2} \end{bmatrix}, \\
\mathbf{f}_i = \begin{bmatrix} f_{1,i} \\ f_{2,i} \\ \vdots \\ f_{n,i} \end{bmatrix}, \quad \mathbf{w}_i = \begin{bmatrix} \Phi_{0,i} \\ 0 \\ \vdots \\ 0 \\ \Phi_{n+1,i} \end{bmatrix}, \quad \boldsymbol{\Phi}_i = \begin{bmatrix} \Phi_{1,i} \\ \Phi_{2,i} \\ \vdots \\ \Phi_{n,i} \end{bmatrix},
\end{gathered}
\tag{3.1}
$$

where I in matrix A denotes the unit matrix of dimension $n \times n$. We know that this system has a unique solution, since A is real symmetric and therefore normal, which implies that A^{-1} exists, compare Section A.4 in the Appendix. For this reason the system $A\mathbf{u} = \delta^2\mathbf{f}$ has the unique solution $\mathbf{u} = \delta^2 A^{-1}\mathbf{f}$.

Often we set up and solve a discrete system trying to approximate the solution of the continuous problem. For this reason, the question of how good that approximate discrete solution actually is naturally arises. As in general the true solution is not known, it is not possible to answer this question directly, though it is possible to analyse how much they can differ at most via error bounds. Next we will give an error bound for the discretization of the two-dimensional Poisson Equation.

In accordance with Isaacson and Keller [1996], let us first introduce the *local truncation error* as

$$\tau\{\varphi(x,y)\} = \Delta_h \varphi(x,y) - \Delta\varphi(x,y),$$

where we assume that $\varphi(x,y)$ is sufficiently smooth on the considered domain. Then we have

$$\tau\{\tilde{u}(x,y)\} = \Delta_h \tilde{u}(x,y) - \Delta\tilde{u}(x,y) = \Delta_h \tilde{u}(x,y) + f(x,y), \quad (x,y) \in \Omega_h$$

but since both the continuous and discrete problem use the same right-hand side $f(x, y)$, this is equivalent to

$$\tau\{\tilde{u}(x, y)\} = -\Delta_h[u(x, y) - \tilde{u}(x, y)].$$

Moreover, $\tilde{u}$ and u respect the same boundary conditions, and thus

$$u(x, y) - \tilde{u}(x, y) = 0 \quad \text{(on the boundary)}.$$

Next, applying

Theorem 1. *Let $V(x, y) = \{V_{ij}\}$ be any net function defined on the sets Ω_h and $\partial\Omega_h$. Then*

$$\max_{\Omega_h} |V| \leq \max_{\partial\Omega_h} |V| + \frac{a^2}{2} \max_{\Omega_h} |\Delta_h V|. \quad \textit{Isaacson and Keller [1996]}$$

to our net function $u(x, y) - \tilde{u}(x, y)$ yields

$$\max_{\Omega_h} |u(x, y) - \tilde{u}(x, y)| \leq \frac{a^2}{2} \max_{\Omega_h} |\tau\{\tilde{u}(x, y)\}|$$

or, when defining $|| \cdot || = \max_{\Omega_h} | \cdot |$,

$$||u(x, y) - \tilde{u}(x, y)|| \leq \frac{a^2}{2} ||\tau\{\tilde{u}(x, y)\}||$$

which means that the error bound is proportional to the local truncation error. Assuming that $\tilde{u}$ has continuous fourth derivatives, we can Taylor-expand

$$\tilde{u}(x \pm h_x, y) = \tilde{u}(x, y) \pm h_x \frac{\partial \tilde{u}(x, y)}{\partial x} + \frac{h_x^2}{2!} \frac{\partial^2 \tilde{u}(x, y)}{\partial x^2} \pm \frac{h_x^3}{3!} \frac{\partial^3 \tilde{u}(x, y)}{\partial x^3} + \frac{h_x^4}{4!} \frac{\partial^4 \tilde{u}(x + \theta_\pm h_x, y)}{\partial x^4}, \qquad |\theta_\pm| < 1.$$

And with this we get

$$\frac{\tilde{u}_{i-1,j} - 2\tilde{u}_{i,j} + \tilde{u}_{i+1,j}}{h_x^2} - \frac{\partial^2 \tilde{u}}{\partial x^2} = \frac{h_x^2}{12} \frac{\partial^4 \tilde{u}(x + \theta h_x, y)}{\partial x^4}, \qquad |\theta| < 1$$

as well as the analogous result for the y-derivatives. And therefore

$$\tau\{\tilde{u}(x, y)\} = \frac{1}{12} \left(h_x^2 \frac{\partial^4 \tilde{u}(x + \theta h_x, y)}{\partial x^4} + h_y^2 \frac{\partial^4 \tilde{u}(x, y + \theta h_y)}{\partial y^4} \right),$$

or, when denoting $M_x^{(4)}$ and $M_y^{(4)}$ as the suprema for the respective fourth order derivatives,

$$||\tau\{\tilde{u}(x, y)\}|| \leq \frac{1}{12} \left(h_x^2 M_x^{(4)} + h_y^2 M_y^{(4)} \right). \tag{3.2}$$

With that we can finally express the error bound:

$$||u(x, y) - \tilde{u}(x, y)|| \leq \frac{a^2}{24} \left(h_x^2 M_x^{(4)} + h_y^2 M_y^{(4)} \right) \tag{3.3}$$

For the sake of completeness it shall be mentioned that Isaacson and Keller [1996] also deduces an error bound expression for the matrix formulation, which will not be derived but merely given here:

$$||\mathbf{u} - \tilde{\mathbf{u}}||_2 \leq \frac{a^2 b^2}{\pi^2 (a^2 + b^2)} ||\tau||_2 \cdot \left[1 + O(h_x^2 + h_y^2) \right]. \tag{3.4}$$

From Equation (3.2) we see that the chosen discretization scheme is consistent with order $m = 2$ for $\tilde{u} \in C^4(\Omega)$, and from Equation (3.3) we see that, moreover, this scheme is convergent (to the continuous solution), since $\|u(x_i, y_j) - \tilde{u}(x_i, y_j)\| \to 0$ as $h \to 0$ for all $1 \leq i \leq n,\ 1 \leq j \leq m$.

3.2 Iterative Methods

Let us consider a linear system

$$A\mathbf{u} = \mathbf{f} \tag{3.5}$$

which we desire to solve with an iterative method. We are going to split the matrix A into two parts which depend on the method itself:

$$A = M - N,$$

so that we can rewrite (3.5) as

$$(M - N)\mathbf{u} = \mathbf{f},$$

or, in terms of iterations

$$M\mathbf{u}^{(k+1)} = N\mathbf{u}^{(k)} + \mathbf{f}.$$

It is assumed that M is an invertible matrix, therefore we can write

$$\mathbf{u}^{(k+1)} = M^{-1}N\mathbf{u}^{(k)} + M^{-1}\mathbf{f} \equiv G\mathbf{u}^{(k)} + \mathbf{b}. \tag{3.6}$$

$G = M^{-1}N$ is called the iteration matrix of the respective method. Note that if $(I - G)$ is non-singular then $(I - G)\mathbf{u} = \mathbf{b}$ has a unique solution.

Now, representing the exact solution of (3.5) with $\hat{\mathbf{u}}$ and substituting $\hat{\mathbf{u}}$ in (3.6) yields a fixed point problem:

$$\hat{\mathbf{u}} = G\hat{\mathbf{u}} + \mathbf{b}. \tag{3.7}$$

Introducing the the error vector as $\mathbf{e}^{(k)} = \mathbf{u}^{(k)} - \hat{\mathbf{u}}$, we receive from (3.6)–(3.7) that

$$\mathbf{e}^{(k+1)} = G\mathbf{e}^{(k)} = \cdots = G^{k+1}\mathbf{e}^{(0)}. \tag{3.8}$$

Thus, if $G^k \to 0$, as $k \to \infty$, then the method will converge from any initial guess $u^{(0)}$.

Now we will analyse when it is true that $G^k \xrightarrow{k\to\infty} 0$. For the sake of simplicity it is supposed that G is normal (meaning that it commutes with its conjugate transpose, compare Section A.4 of the Appendix) and thus diagonizable, that is an eigenvalue λ_i of G of order n has exactly n corresponding eigenvectors $v_{i,1}, v_{i,2}, \ldots, v_{i,n}$. Then we can write

$$G = R\Lambda R^{-1},$$

where R is the (unitary) matrix of right eigenvectors of G, and Λ is the diagonal matrix of eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_m$, and

$$G^k = R\Lambda^k R^{-1},$$

with $\Lambda^k = diag(\lambda_1^k, \lambda_2^k, \ldots, \lambda_m^k)$, see Theorem 19 in Section A.3 of the Appendix.

With these assumptions and definitions we can analyse the rate of convergence of an iterative method. We have

$$\|\mathbf{e}^{(k)}\|_2 = \|G^k\mathbf{e}^{(0)}\|_2 \leq \|G^k\|_2\|e^{(0)}\|_2$$

and

$$\|G^k\|_2\|e^{(0)}\|_2 \leq \|R\|_2\|\Lambda^k\|_2\|R^{-1}\|_2\|\mathbf{e}^{(0)}\|_2 = \rho^k\kappa_2(R)\|\mathbf{e}^{(0)}\|_2, \tag{3.9}$$

where $\rho = \rho(G)$ is the spectral radius of G, and $\kappa_2(R) = \|R\|_2\|R^{-1}\|_2$ is the condition number of R using the euclidean norm. But since R is unitary we know that $\kappa_2(R) = 1$, as is shown in Theorem 21 in Section A.4 of the Appendix.

For normal G we therefore have

$$\|\mathbf{e}^{(k)}\|_2 \leq \rho^k\|\mathbf{e}^{(0)}\|_2. \tag{3.10}$$

In the case of a normal G we can also estimate the approximate number of iterations needed to achieve a given precision ϵ in the same way as Isaacson and Keller [1996], namely

$$\|\mathbf{e}^{(k)}\|_2 \leq \epsilon\|\mathbf{e}^{(0)}\|_2, \tag{3.11}$$

or, written slightly differently, ϵ limits the relative error $\|\mathbf{e}^{(k)}\|_2/\|\mathbf{e}^{(0)}\|_2$. Substituting (3.11) into (3.10) then yields

$$\begin{aligned} & \epsilon\|\mathbf{e}^{(0)}\|_2 \leq \rho^k\|\mathbf{e}^{(0)}\|_2 \\ \Leftrightarrow \qquad & k \geq \frac{\ln(\epsilon)}{\ln(\rho)}, \end{aligned} \tag{3.12}$$

or, in other words, $\frac{\ln(\epsilon)}{\ln(\rho)}$ is an approximate lower bound for the number of iterations needed to achieve the *relative tolerance* ϵ according to (3.11). However, Equation (3.12) has one big flaw as it does not account for initial approximation effects: Whether one starts the iterative process with an initial approximation that equals the exact discrete solution or with an initial approximation that is far off, Equation (3.12) will yield the same number of iterations k.

In a similar though different approach we can demand that

$$\|\mathbf{e}^{(\hat{k})}\|_2 \leq \rho^{\hat{k}}\|\mathbf{e}^{(0)}\|_2 \overset{!}{\leq} \hat{\epsilon}.$$

which results in an estimate for the number of iterations needed to achieve an *absolute tolerance* $\hat{\epsilon}$ in the form of

$$\hat{k} \geq \frac{\ln\left(\hat{\epsilon}/\|e^{(0)}\|_2\right)}{\ln(\rho)}, \tag{3.13}$$

From the above discussion we see that $\|G^k\|_2$, or ρ^k, respectively, can be used as a strict upper bound estimate for the ratio $\|\mathbf{e}^{(k)}\|_2/\|\mathbf{e}^{(0)}\|_2$, when $\mathbf{e}^{(0)}$ is not the nullvector.

As we have seen in Equation (3.10), the iterative method will converge if $\rho(G) < 1$, moreover this implies that $G^k \xrightarrow{k\to\infty} 0$. It can easily be observed that the smaller ρ, the faster the method will converge. The above discussion is based on Isaacson and Keller [1996]. Ortega [1972] summarizes this result for general matrices G in the following

Theorem 2 (Fundamental Theorem of Linear Iterative Methods)**.** *Let $G \in \mathbb{C}^{n\times n}$ and assume that the equation $\mathbf{u} = G\mathbf{u} + \mathbf{b}$ has a unique solution $\hat{\mathbf{u}}$. Then the iteration $\mathbf{u}^{(k+1)} = G\mathbf{u}^{(k)} + \mathbf{b}$ converge to $\hat{\mathbf{u}}$ for any $\mathbf{u}^{(0)}$ if and only if $\rho(G) < 1$.*

Proof. This is an immediate result of Theorem 14 (Section A.2 of the Appendix) applied to Equation (3.8). □

Definition 1. *Let A and B be two $n \times n$ complex matrices. If, for some positive integer k, $\|A^k\| < 1$, then*

$$\mathfrak{R}(A^k) := -\ln[(\|A^k\|)^{(1/k)}] = -\frac{\ln[\|A^K\|]}{k}$$

is the average rate of convergence *for k iterations of the matrix A. If $\mathfrak{R}(A^k) < \mathfrak{R}(B^k)$, then B is iteratively faster than A for k iterations.*

From $\|\mathbf{e}^{(k)}\| \leq \|A^k\| \, \|\mathbf{e}^{(0)}\|$, $k \geq 0$, it can be seen that $\sigma := \left(\|\mathbf{e}^{(k)}\|/\|\mathbf{e}^{(0)}\|\right)^{1/k}$ is the *average reduction factor per iteration*. If $\|A^k\| < 1$, then by definition

$$\sigma \leq \|A^k\|^{1/k} = \exp(-\mathfrak{R}(A^k)).$$

Hence, $\mathfrak{R}(A^k)$ is the the exponential decay rate for a sharp upper bound of the average error reduction σ per iteration. Varga [1962]

Theorem 3. *Let A be a convergent (meaning convergent to the null matrix) $n \times n$ complex matrix. Then, the average rate of convergence for k iterations $\mathfrak{R}(A^k)$ satisfies*

$$\lim_{k \to \infty} \mathfrak{R}(A^k) = -\ln \rho(A) =: \mathfrak{R}_\infty(A),$$

where $\mathfrak{R}_\infty(A)$ is the asymptotic rate of convergence *of the matrix A.*

Corollary 1. *If A is an arbitrary convergent $n \times n$ complex matrix, then*

$$\mathfrak{R}_\infty(A) \geq \mathfrak{R}(A^k)$$

for any positive integer k for which $\|A^k\| < 1$.

According to Varga [1962], the asymptotic rate of convergence $\mathfrak{R}_\infty(A)$ for a convergent matrix A is by far the simplest practical measure of rapidity of convergence of a matrix which is in common use, though its indiscriminate use can give quite misleading information.

Missing proofs can be found in Varga [1962].

3.2.1 Jacobi Method

As discussed in Section 3.2, we can decompose matrix A into $A = M - N$. For the Jacobi Method we have seen above that $M = D$, $N = L + U$, where D is the diagonal of matrix A, and L and U are the strictly lower and upper triangular matrix parts of Matrix A, respectively. Hence, we get for the Jacobi iteration matrix

$$G_{Jac} = M^{-1}N = D^{-1}(L + U) = D^{-1}(D - A) = I - D^{-1}A.$$

Since the iteration matrix G_{Jac} of the matrix A derived in Section 3.1 is real and symmetric it is also normal, and therefore $\|G_{Jac}^k\|_2 \leq \rho^k$ because with normal G_{Jac} we have $\kappa_2(R_{Jac}) = 1$, compare (3.9). For that reason the approximations for the number of iterations given in the Equations (3.12) or (3.13), respectively are valid. In order to use this approximation for a given tolerance ϵ we still need to determine the spectral radius of G_{Jac}. Isaacson and Keller [1996] discusses the eigenvalues of A from (3.1):

$$\lambda_{p,q}(A) = 4\theta_x \sin^2\left(\frac{ph_x\pi}{2a}\right) + 4\theta_y \sin^2\left(\frac{qh_y\pi}{2b}\right), \qquad p = 1, 2, \ldots, N, \quad q = 1, 2, \ldots, M.$$

When applying the eigenvectors of A ($A\mathbf{v}_{p,q} = \lambda_{p,q}\mathbf{v}_{p,q}$) to G_{Jac} from our model problem from Section 3.1 we get

$$\begin{aligned} G_{Jac}\mathbf{v}_{p,q} &= \left[I - D^{-1}A\right]\mathbf{v}_{p,q} = \underbrace{(1 - \lambda_{p,q})}_{=\hat{\lambda}_{p,q}=\lambda_{p,q}(G_{Jac})} \mathbf{v}_{p,q} \\ &= 1 - 4\left[\theta_x \sin^2\left(\frac{ph_x\pi}{2a}\right) + \theta_y \sin^2\left(\frac{qh_y\pi}{2b}\right)\right]\mathbf{v}_{p,q} \qquad (3.14) \\ &= 1 - 2\left[\theta_x\left(1 - \cos\left(\frac{ph_x\pi}{a}\right)\right) + \theta_y\left(1 - \cos\left(\frac{qh_y\pi}{b}\right)\right)\right]\mathbf{v}_{p,q}, \qquad (3.15) \end{aligned}$$

where we used the identity $1 - \cos(2\alpha) = 2\sin^2(\alpha)$. From (3.14), or by looking at Figure 3.1, we see that by choosing $p = q = 1$ we obtain the spectral radius of G_{Jac}:

$$\rho_{Jac} = 1 - 2\left[\theta_x\left(1 - \cos\left(\frac{h_x\pi}{a}\right)\right) + \theta_y\left(1 - \cos\left(\frac{h_y\pi}{b}\right)\right)\right], \tag{3.16}$$

or in terms of n and m

$$\rho_{Jac} = 1 - 2\left[\theta_x\left(1 - \cos\left(\frac{\pi}{n+1}\right)\right) + \theta_y\left(1 - \cos\left(\frac{\pi}{m+1}\right)\right)\right].$$

In the special case of $a = b$ and $h = h_x = h_y$, which implies $\theta_x = \theta_y = \frac{1}{4}$ we have

$$\rho_{Jac} = \cos\left(\frac{h\pi}{a}\right) = \cos\left(\frac{\pi}{n+1}\right).$$

Hence, the Jacobi Method indeed converges for the system described in Section 3.1. However, there is an easier way to determine whether or not the Jacobi Method converges for a given system, Ortega [1972]:

Theorem 4. *Let $A \in \mathbb{R}^{n\times n}$ be an M-matrix*[1] *and let $\mathbf{b} \in \mathbb{R}^n$ be arbitrary. Then the Jacobi Method converges to $A^{-1}\mathbf{b}$ for any $\mathbf{u}^{(0)}$.*

Definition 2. *For $n \geq 2$, a matrix $A \in \mathbb{C}^{n\times n}$ is* reducible *if there exists permutation matrix $P \in \mathbb{C}^{n\times n}$ such that*

$$PAP^T = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix},$$

where A_{11} is an $r\times r$ submatrix and A_{22} is an $(n-r)\times(n-r)$ submatrix with $1 \leq r < n$. If no such permutation matrix exists, then A is irreducible. *If A is a 1×1 complex matrix, then A is irreducible if its single entry is nonzero, and reducible otherwise. Varga [1962]*

Definition 3. *$(a_{ij}) = A \in \mathbb{C}^{n\times n}$ is* diagonally dominant *if*

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \quad \forall 1 \leq i \leq n.$$

An $n\times n$ matrix A is strictly diagonally dominant *if strict inequality holds in the above equation. Furthermore, A is* irreducibly diagonally dominant *if A is irreducible and diagonally dominant, with strict inequality in the above equation for at least one i.*

Theorem 5. *Assume that $\mathbf{b} \in \mathbb{R}^n$ is arbitrary and that $A \in \mathbb{R}^{n\times n}$ is either strictly or irreducibly diagonal dominant. Then the Jacobi Method converges to $A^{-1}\mathbf{b}$ for any $\mathbf{u}^{(0)}$.*

Note that these Theorems are valid for the Gauss-Seidel Method as well. The proofs are given in Ortega [1972].

In this Section as well as in the previous Section (3.2) we stated Theorems and discussed results in a general manner for an arbitrary matrix A, however in this work it is desired to apply these results specifically to the matrix A from our discretization of the Poisson Equation in Equation (3.1). We showed that the spectral radius of the Jacobi iteration matrix of this system is $\rho(G_Jac) < 1$, and therefore, according to Theorem 2, the Jacobi iteration for this system will converge. We could have received the same result in a less cumbersome way without having to determine the spectral radius in Equation (3.16): Since A is an M-matrix and irreducibly diagonal dominant, the convergence of the Jacobi Method applied to A is also given by either Theorem 4 or 5, respectively.

[1]An M-Matrix is a matrix whose off-diagonal elements are non-positive and whose principal minor determinants are positive. Another definition can be found in Definition 7 in Section A.2 of the Appendix.

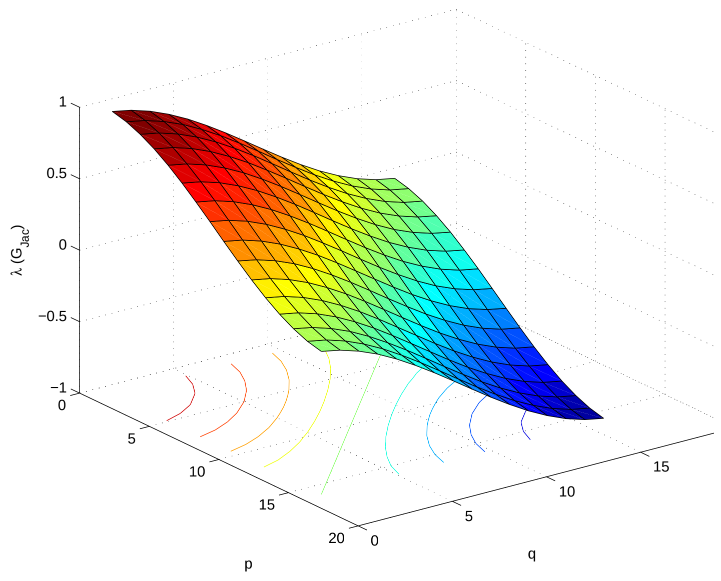


Figure 3.1: *Plot of* (3.15) *in dependence of p and q.*

On a different note, it can be seen that the Jacobi Method $\mathbf{u}^{(k+1)} = (I - D^{-1}A)\mathbf{u}^{(k)} + D^{-1}\mathbf{f}$ depends only on values of the last iteration $\mathbf{u}^{(k)}$ to calculate the solution vector of the new iteration $\mathbf{u}^{(k+1)}$. Since in every iteration step the values of $\mathbf{u}^{(k)}$ are readily available, every computation of $u_i^{(k+1)}$ is independent of $u_j^{(k+1)}$, $j \neq i$, and therefore these computations can be performed concurrently in any order. In parallel computing, a problem that can be broken into completely independent pieces that can run simultaneously is said to be *embarrassingly parallel*.

Chapter 4

Implementations

The goal of our work was to implement a linear solver for sparse linear systems of various forms using the Jacobi Method. In order to be able to solve sparse systems of different forms (that satisfy a certain convergence criterion as described in Section 3.2.1), we us the *Compressed Sparse Row* (CSR) format. The flexibility of such an implementation comes with a cost, as it is not possible to make use of optimizations for the specific sparse linear system. It is therefore of interest to know how expensive such a flexible system is when applied to a given system, compared to a first "optimization" for the same system. In our work we decided to solve the well-known Poisson Equation (see 3.1), and want to compare the performance of the flexible implementation against the specific implementation that can *only* solve the Poisson Equation. First we will introduce the CSR format and then we will describe our implementations.

4.1 The Compressed Sparse Row Format

For large sparse matrices saving the matrix in the memory of a computer "as is" can be prohibitively expensive, and in some instances the complete matrix may not even fit into the memory. Hence it is desired to only store that part of the matrix that really contains information and discard as many zero elements of the matrix as possible.

The Compressed Sparse Row (CSR) format is one of many formats that can be used to store a sparse matrix. In contrast to other formats (like the Diagonal (DIAG) format) it is relatively efficient for a variety of sparse matrices as it only stores the non-zero elements of a matrix together with their column indices and a vector that contains the pointers to the elements starting a new row.

For example, consider the following sparse matrix.

$$A = \begin{bmatrix} 1.0 & 0 & 0 & 2.0 & 0 \\ 0 & 0 & 3.0 & 0 & 4.0 \\ 0 & 5.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 7.0 & 0 \\ 8.0 & 0 & 0 & 0 & 9.0 \end{bmatrix}$$

The CSR format of A looks as follows:

$$\begin{aligned} \texttt{val} &= [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0], \\ \texttt{colInd} &= [1, 4, 3, 5, 2, 3, 4, 1, 5], \\ \texttt{rowPtr} &= [1, 3, 5, 6, 8, 10]. \end{aligned}$$

Here, the vector `val` stores the values of the non-zero elements in the order of how they appear in the columns, row per row. Obviously the size of this vector equals the number of non-zero elements (NNZ). The vector `colInd` is of the same size and stores the column indices of the elements stored in `val`. The vector `rowPtr` "points" to the element of `val` that starts a new row. If the matrix A has n rows and m columns, then `rowPtr` will store $n+1$ elements. Note that the $n+1$st element of `rowPtr` always is $NNZ+1$.

In general the vector `val` stores floating point numbers, whereas the vectors `colInd` and `rowPtr` store integer elements.

However, the more compact a matrix is stored, the more the information about its structure will be lost. For example, accessing the element $a_{i,j}$ of matrix A is a straight forward operation when storing a matrix in the usual way. However for the CSR format, for example, this operation becomes more expensive as one first has to "recover" the element $a_{i,j}$ from `val` using `colInd` and `rowPtr` (assuming that $a_{i,j}$ is not a zero element) as follows:

$$a_{i,j} \Longleftrightarrow \texttt{val}(k) \quad \text{with } \texttt{colInd}(k) = j \text{ and } \texttt{rowPtr}(i) \leq k \leq \texttt{rowPtr}(i+1).$$

For an introduction to other popular sparse matrix storing formats see Saad [2000], for example.

4.2 The Jacobi CSR and "modified" Jacobi CSR Implementations

We implemented two different Jacobi solver for sparse matrices that do satisfy the convergence conditions presented in Chapter 3 using the CSR storage format. The first one is the straight forward Jacobi Method applied to the CSR format, as is portrayed in Pseudo-Code Snippet 1. As was discussed in Chapter 2, it is not possible to parallelize loops that do not have a known number of iterations beforehand. For this reason, this implementation creates a parallel region in every single iteration which incurs mentionable overheads for systems that take a greater number of iterations to attain a desired precision (OpenMP/OpenACC). Note that for all the CUDA implementations in this work, computing $\texttt{u}_{\texttt{new}}$, checking for tolerance, and updating `u` are separate kernels to ensure synchronization of all CUDA blocks in the grid.

```
loop    (iteration loop)
    for k=1,n*m do    (loop in parallel)
        • determine line $i$ of matrix $A$ in the val vector
        • determine the diagonal element of $A$ in the val vector for line $i$ ($\texttt{A}_{\texttt{ii}}$)
        • compute $\texttt{u}_{\texttt{new}}(\texttt{k}) = \frac{\texttt{f(k)} - \sum_l \texttt{val(l)u(colInd(l))}}{\texttt{A}_{\texttt{ii}}}$
    end for

    • Check if the desired tolerance is reached (in parallel)

    for k=1,nm do    (loop in parallel)
        • update $\texttt{u}(\texttt{k}) = \texttt{u}_{\texttt{new}}(\texttt{k})$
    end for
end loop
```

Pseudo-Code Snippet 1: *First Jacobi CSR implementation.*

The second ("modified") Jacobi CSR implementation, shown in Pseudo-Code Snippet 2, introduces an *iteration-chunks* loop which has a fixed number of iterations enabling to open the parallel region around this loop, therefore reducing the parallelization overhead. Moreover, the attained precision will only be checked after each iteration-chunk, hence also decreasing the mathematical overhead.

```
loop    (outer loop)
    for iter=1,chunk_size do    (iteration-chunk loop)
        for k=1,n*m do    (loop in parallel)
            • determine line $i$ of matrix $A$ in the val vector
            • determine the diagonal element of $A$ in the val vector for line $i$ ($A_{ii}$)
            • compute $u_{new}(k) = \frac{f(k) - \sum_l val(l) u(colInd(l))}{A_{ii}}$
        end for

        for k=1,nm do    (loop in parallel)
            • update $u(k) = u_{new}(k)$
        end for
    end for

    • Check if the desired tolerance is reached (in parallel)

end loop
```

Pseudo-Code Snippet 2: *"Modified" Jacobi CSR implementation.*

4.3 The Five-Point Jacobi Implementation

The five-point Jacobi implementation, displayed in Pseudo-Code Snippet 3, differs from the CSR implementations in several ways: First of all, it is only able to solve the Poisson Equation. Since it does not rely on the CSR format the solution array u or u_{new}, repectively, can be set up to be 2 dimensional, the boundary conditions will be added to those arrays as extra elements, and in contrast to the CSR implementations not included in the right-hand side f, as described in Section 3.1. Moreover, u_{new} can be computed directly, as the terms can be directly taken from the five-point stencil (shown in Figure 4.1).

As for the previous implementations, the CUDA code for the five-point stencil is divided into 3 separate kernels in order to provide the synchronization needed for the Jacobi Method.

```
loop    (iteration loop)
    for j=1,m do    (loop in parallel)
        for i=1,n do    (loop in parallel)
            • compute $u_{new}(i,j) = 0.25(f(i,j) - (u(i-1,j) + u(i+1,j) +$
                                                  $+ u(i,j-1) + u(i,j+1)))$
        end for
    end for

    • Check if the desired tolerance is reached (in parallel)

    for j=1,m do    (loop in parallel)
        for i=1,n do    (loop in parallel)
            • update $u(i,j) = u_{new}(i,j)$
        end for
    end for
end loop
```

Pseudo-Code Snippet 3: *Jacobi implementation using the five-point stencil for the laplacian operator.*

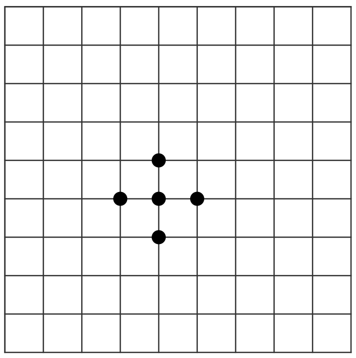


Figure 4.1: *The five-point stencil for the laplacian operator.*

Chapter 5

Results

In this Chapter we present the results of our work. We used three different implementations (Jacobi CSR, modified Jacobi CSR, five-point stencil Jacobi), three different parallelization paradigms (OpenMP, OpenACC, CUDA), five different grid sizes ($n = m = \{512, 1024, 2048, 4096, 8192\}$), and both single and double precision, yielding 90 different configurations. When additionally considering the different number of processors used in order to create the speedup and efficiency graphs (serial, $p = 1, 2, 4, 8, 16, 32, 64$ where p is the number of processors used in parallel), this even results in 300 different configurations run for this work.

The OpenMP results were obtained on a Silicon Graphics machine with four AMD Opteron 6376 processors with each 16 cores of 2.3 GHz, and with 8×2 MB L2 cache and 16 MB L3 cache, while the OpenACC and CUDA codes were run on a computer with an Intel Core i7-4770 processor with 3.4 GHz and 8 MB cache, and a NVIDIA Tesla K40 GPU. The Tesla K40 has 2880 single precision cores and 960 double precision cores, divided on 15 Streaming Multiprocessors with a clock rate of 745 MHz each. All four AMD Opteron 6376 processors use the same memory. The Silicon Graphics machine uses Debian 7.0 and gfortran 4.7.2 which supports the OpenMP 3.1 specification, whereas the computer with the Tesla K40 GPU uses Ubuntu 14.04 and pgfortran 16.9 which supports all OpenACC 2.0 features, except for the features *Declare Link* and *Nested parallelism*, as well as the CUDA 7.5 toolkit. No special flags have been used other than `-mcmodel=large` and `-mcmodel=medium` on the Silicon Graphics and the GPU machine, respectively, as well as the `-ta=tesla:cc35` flag for the GPU codes. For the execution of the OpenMP code we used the CPU affinity "$0 - 63 : \frac{64}{p}$" where p is the number of processor cores used. It shall be noted that we only run one thread on each processor core.

We consider the two-dimensional Poisson Equation with Dirichlet boundary conditions

$$\begin{aligned} u_{xx} + u_{yy} &= 2\pi^2 \sin(\pi x)\sin(\pi y), & x, y &\in \Omega = (0, 120) \times (0, 120), \\ u(x, y) &= 0, & x, y &\in \partial\Omega, \end{aligned} \tag{5.1}$$

whose exact solution is $u(x, y) = \sin(\pi x)\sin(\pi y)$. We chose the computational domain to be $[0, 120]\times[0, 120]$ as we know that the method will converge faster the smaller the spectral radius of the Jacobi iteration matrix ρ_{Jac}, (3.16), will be and that furthermore $\rho_{Jac} = 1 - O(h^2)$. Table 5.1 exhibits the number of iterations necessary to achieve an absolute error of $\|u_{i,j}^{(k)} - \sin(\pi i h_x)\sin(\pi j h_y)\|_\infty \leq \epsilon = 0.01$ for an initial guess of $u_{i,j}^{(0)} = 0$ for all $1 \leq i \leq N$, $1 \leq j \leq M$ on the considered regular grids. This means that we approximate the exact solution of the continuous problem with the solution of the discrete system, while the iterative process will converge to the exact solution of the discrete problem. For that reason it is not possible to achieve an arbitrarily small tolerance for the approximation to the exact continuous solution, compare Figure 5.1.

For the modified Jacobi CSR implementation, we chose two different iteration-chunk sizes: 10 for the grids with 512 and 1024 points in each direction, and 100 for the other grids. We chose a different number for the iteration chunk of the smaller grids because for a greater number of iterations the iterative process may not yield the desired tolerance to the exact continuous solution, but rather increases the error $\|u_{i,j}^{(k)} - \sin(\pi i h_x)\sin(\pi j h_y)\|_\infty$ with every iteration. This is due to the fact that the process will converge to the exact discrete solution, as mentioned before.

Table 5.1: *Number of iterations needed to achieve a tolerance of $\epsilon = 0.01$ for the problem given in Equation (5.1) and an initial guess of $u^{(0)} \equiv 0$.*

Grid points per direction	Number of iterations
512	10
1024	56
2048	257
4096	1071
8192	4333

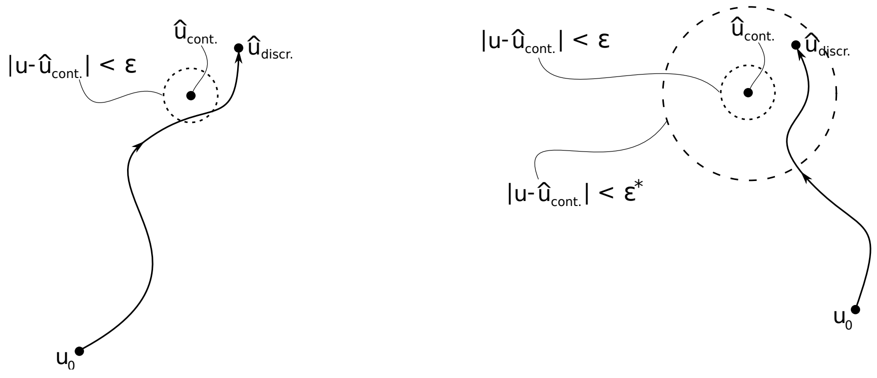

$|u-\hat{u}_{cont.}| < \varepsilon$
$\hat{u}_{cont.}$
$\hat{u}_{discr.}$
u_0
$|u-\hat{u}_{cont.}| < \varepsilon$
$\hat{u}_{cont.}$
$\hat{u}_{discr.}$
$|u-\hat{u}_{cont.}| < \varepsilon^*$
u_0

Figure 5.1: *Graphical representation of the converging process. Depending on how the iteration converges to the exact discrete solution, $\hat{u}_{discr.}$, (for example due to the choice of of the initial guess u_0) it may be necessary to adapt the chosen tolerance, ϵ, in order to let the iterative process terminate. An appropriate choice for the tolerance is given in the error bound Equations (3.3) and (3.4).*

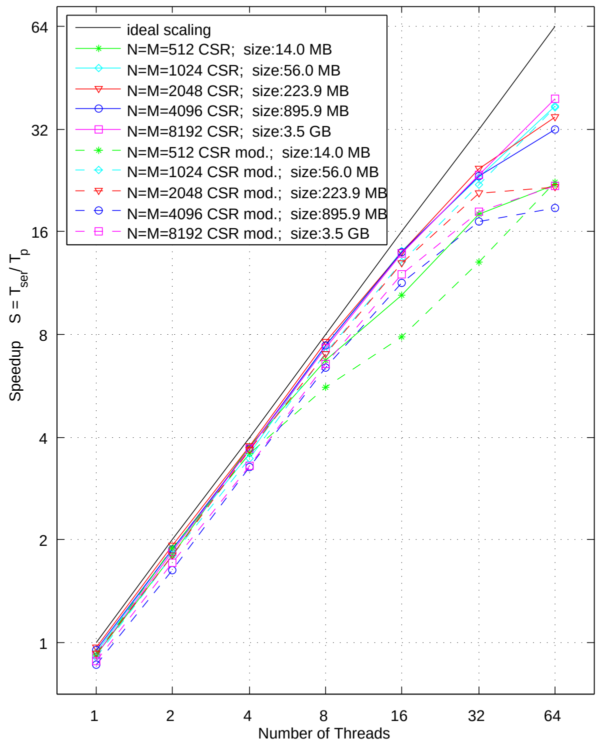


Figure 5.2: *OpenMP speedup plot for the (modified) Jacobi CSR implementation when using single precision. Based on the execution time of the Jacobi method only.*

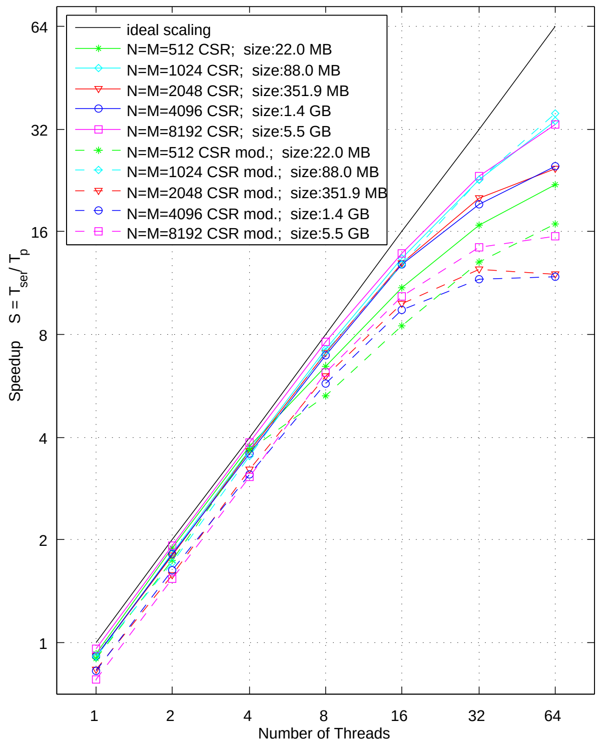


Figure 5.3: *OpenMP speedup plot for the (modified) Jacobi CSR implementation when using double precision. Based on the execution time of the Jacobi method only.*

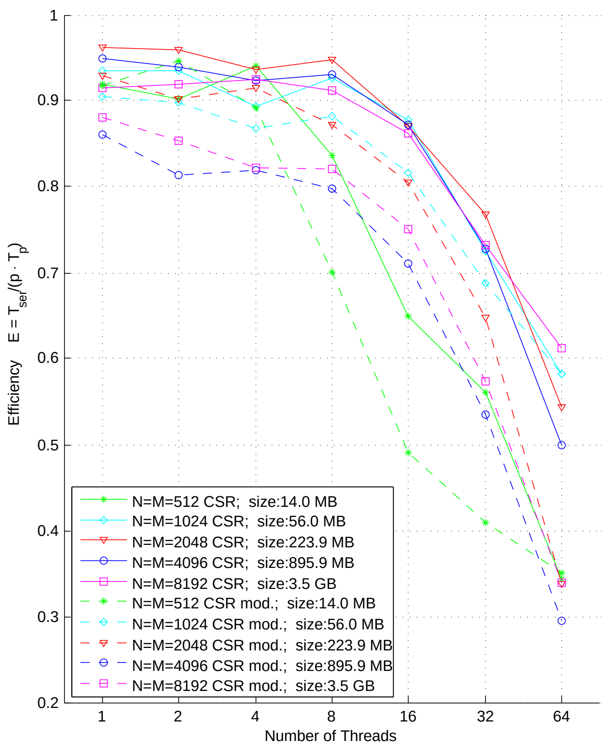


Figure 5.4: *OpenMP efficiency plot for the (modified) Jacobi CSR implementation when using single precision. Based on the execution time of the Jacobi method only.*

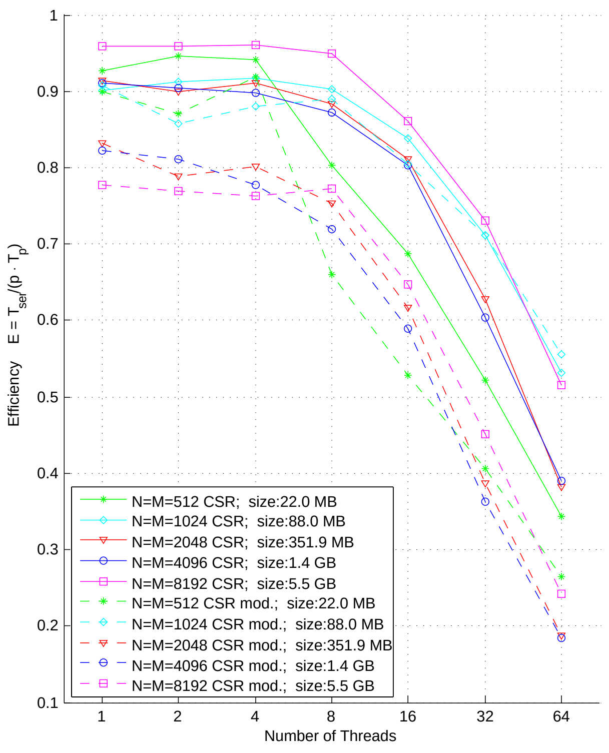


Figure 5.5: *OpenMP efficiency plot for the (modified) Jacobi CSR implementation when using double precision. Based on the execution time of the Jacobi method only.*

Note that all results presented here are based on a single run of each respective program and not averaged. Due to different background processes of the operating system, the programs run under different conditions and repeated execution of the programs yield somewhat varying results.

We first consider the results of the OpenMP implementations. For these implementations we analyse the speedup and efficiency behaviour of the respective Jacobi subroutines. These are not based on the total execution time of the programs but merely on the time spent for the iterative solution of the system. Therefore, creating the right-hand side of the system and the CSR-format, among other things, do not affect the speedup and efficiency plots.

Focussing on the (modified) Jacobi CSR implementations first, Figures 5.2 and 5.3 exhibit the speedup behaviour, whereas Figures 5.4 and 5.5 portray the efficiency behaviour in single and double precision, respectively. As described in Section 2.4, the efficiency is simply the speedup normalized by the number of processors used, therefore they both display equivalent information. Each of these Figures reveal the results of both the modified and unmodified Jacobi CSR implementations for all grids (with $N = M = \{512, 1024, 2048, 4096, 8192\}$), using $p = \{1, 2, 4, 8, 16, 32, 64\}$ threads. Figures 5.2 and 5.4 use single precision floats, while Figures 5.3 and 5.5 use double precision floats. In the legend of these Figures we also added the amount of memory that will be needed to store the variables for the specific precision and grid size (counting the current and previous approximations, the CSR format, and the right-hand side of the linear system). In addition to the results obtained from the programs' performance, we added ideal linear scaling to the speedup Figures 5.2 – 5.3 for comparison purposes. The equivalent in Figures 5.4 – 5.5 for ideal scaling is simply $E \equiv 1$.

We see in Figures 5.2 – 5.5 that using OpenMP with one single thread is already notably slower than the "purely serial" counterpart. That is because even when using only one thread with OpenMP and therefore letting the code run serially, still there are overheads from the data management, the creation of the "parallel" region and the scheduling of the "parallel" loop. Moreover, it can be concluded that the modified Jacobi CSR implementation involves more overhead, since its efficiency behaviour using one thread is notably worse than the unmodified equivalent, with exceptions of only N=M=512 in single precision and N=M=1024 in double precision. However, since these cases involve only a small number of iterations and therefore finish comparatively fast, they are more prone to be affected by disturbances caused by background processes. The trend that the modified Jacobi CSR implementation has weaker scaling performance in general continues when increasing the number of threads used, with the exception of the modified CSR implementation for the grid with $N = M = 1024$. This implementation has a very similar scaling behaviour to its unmodified correspondent, especially when using double precision.

Looking at the efficiency plots in Figures 5.4 – 5.5, it can be observed that the efficiency does not vary mentionably when using eight threads or less (except when using the smallest grid with $N = M = 512$) and then starts to degrade quickly when using 16 or more threads. When using eight processor cores on the Silicon Graphics machine, every processor core has access to its own L2 cache in our work. From 16 processor cores on, each L2 cache will be shared among two processor cores. This may explain the steep decline in efficiency when using 16 threads or more, though further investigation may be needed for confirmation. The scaling performance of the cases with the smallest grid considered here, however, already starts degrading when using eight or more threads. This behaviour might be explained by not being able to provide each processor cores enough work to stay busy for a lower number of threads used with this small grid. In general the performance degradation of scaling (or equivalently efficiency) is "steeper" when using double precision.

An interesting difference between the single precision and double precision cases is that resolving the grid with 2048^2 grid points with the unmodified CSR implementation using single precision has the overall best scaling performance except when using all 64 processors provided by the Silicon Graphics machine. When using all 64 processors, resolving the largest grid with 8192^2 grid points with the unmodified CSR implementation in single precision offers the best scaling performance. Focussing on the double precision cases, on the other hand, one can observe a different scaling behaviour: Here, resolving the biggest grid (N=M=8192) with the unmodified CSR implementation offers the best efficiency when using up to 32 threads. For 64 threads, both the modified and unmodified CSR implementation for the grid with 1024^2 grid points surpass the efficiency of the unmodified implementation for $N = M = 1024$. Note that for both single and double precision, the scaling behaviour of the modified and unmodified implementations differ (with an exception for the grid using 1024^2 points). For example, the unmodified CSR implementation for the $N = M = 8192$ grid has the *best* efficiency in the lower thread regime, whereas the modified CSR implementation for this grid has the *worst* efficiency when using four threads or less. Broadly speaking, the scaling (or equivalently efficiency) sequence of resolved grids for a given number of threads is not the same for the modified/unmodified CSR implementation.

Note however that better scaling does not imply a faster execution time, as can be seen in Figures 5.6 and 5.7, for the results in single and double precision, respectively. Both figures portray the relative execution time difference

$$\tau_{rel} = \frac{(t_{mod.CSR} - t_{CSR})}{t_{CSR}}$$

based only on the execution times of the respective Jacobi Method (and not of the entire program). These graphs show that the modified CSR implementation is roughly 63–15% faster than its unmodified counterpart. With the exceptions of the cases $N = M = 512$ (single precision) and $N = M = 1024$ (single and double precision), the overall trend is that serial execution (without using OpenMP) has the biggest absolute $|\tau_{rel}|$ which then decreases with the number of threads used. When using double precision variables $|\tau_{rel}|$ decreases faster with the number of threads than when using single precision variables. The relative execution time difference τ_{rel} for the grid with 1024^2 grid points is fairly constant (both for single and double precision) when varying the number of threads and for the serial execution. As mentioned before, the other exception is the solution of the system with 512^2 unknowns when using single precision and 64 threads: Here τ_{rel} has a similar value as for the serial execution, whereas when using 8–32 threads decreases with the number of threads. All this translates directly to the results presented in the speedup and efficiency plots (Figures 5.2 – 5.5). As τ_{rel} for the linear systems using 1024^2 unknowns does not vary mentionably, the speedup and efficiency for the modified and unmodified Jacobi CSR implementations of these systems behave equivalently similar. In the same way, as τ_{rel} for the single precision version of the grid with $N = M = 512$ has virtually the same value when using one thread and 64 threads, the speedup

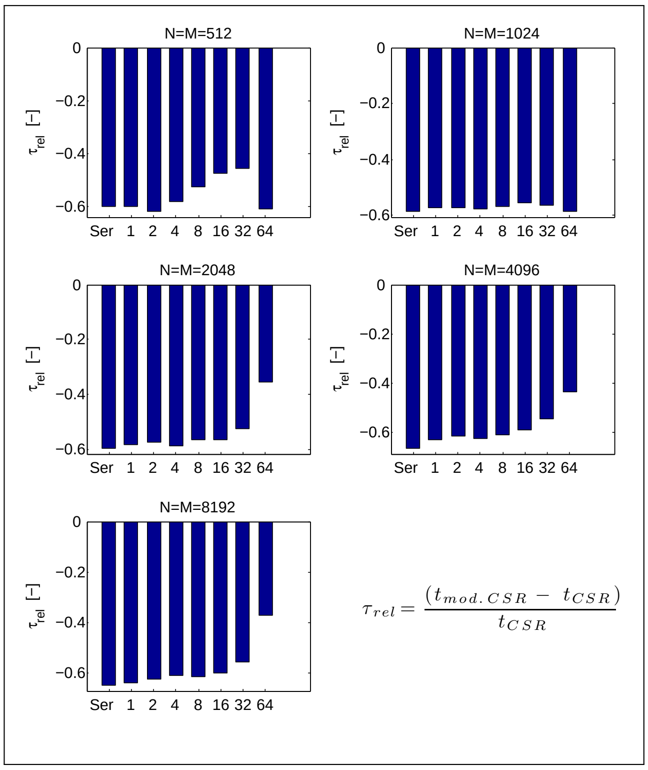


Figure 5.6: *Relative execution time difference between the modified and unmodified Jacobi CSR implementations using single precision.*

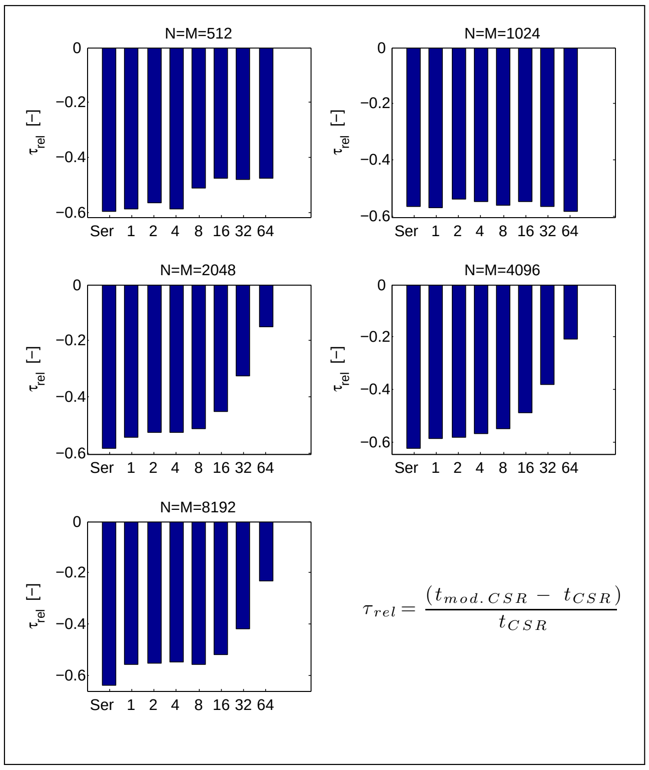


Figure 5.7: *Relative execution time difference between the modified and unmodified Jacobi CSR implementations using double precision.*

and efficiency for the (un)modified implementations at these points are almost identical, compare Figure 5.2 and 5.4. Moreover, the more τ_{rel} decreases in Figures 5.6 and 5.7, the more the scaling performance of the modified implementations degrades in relation to the unmodified implementations in Figures 5.2 – 5.5. This general behaviour of the relative time difference τ_{rel} underlines that the reduction operation becomes more efficient with more concurrent threads available. However, why the increase in threads does not affect τ_{rel} for $N = M = 1024$ in our programs still requires more investigation.

Next we will discuss the results of the OpenMP implementations of the five-point Jacobi, given in Figures 5.8 – 5.11. Analogously to the CSR implementations, we see that the systems with 512^2 unknowns are the outliers, performance wise. Overall, when comparing the speedup plots of the five-point implementation with the unmodified CSR implementation, one can instantly see that the graphs for the five-point implementation scatter less, when neglecting the $N = M = 512$ cases, and moreover the speedup does not flatten as much as the CSR implementations when using a higher number of threads. Studying the efficiency plots reveal even more details. Again, while neglecting the smallest grid used in our work, we see that the graphs for the five-point cases are overall smoother and the range of the efficiency is smaller. To be more precise it can be observed that the efficiency is flatter compared to the respective unmodified CSR analogue. The unmodified CSR single precision implementations stay above $E = 0.9$ for eight or less threads (except for the system with 512^2 unknowns), while all but the system with 1024^2 unknowns start below $E = 0.9$ for the five-point correspondents. This trend is then reversed for a greater number of threads used: Neglecting the $N = M = 512$ case, the five-point single precision implementation stays around or above 60% efficiency, though all except the biggest case ($N = M = 8192$) for the unmodified CSR implementations stay below 60% efficiency for 64 threads. The double precision cases behave similarly. As was the case for the CSR implementations, the grid with the best scaling behaviour in general is different for the single precision and double precision case. When considering single precision floating point numbers for the five-point implementation, the overall best scaling is given by the grid with $N = M = 1024$ (with exception to the case when using 64 threads), whereas the overall best scaling when using double precision floating point numbers is given by the biggest grid considered here ($N = M = 8192$), except when using 32 threads in the presented

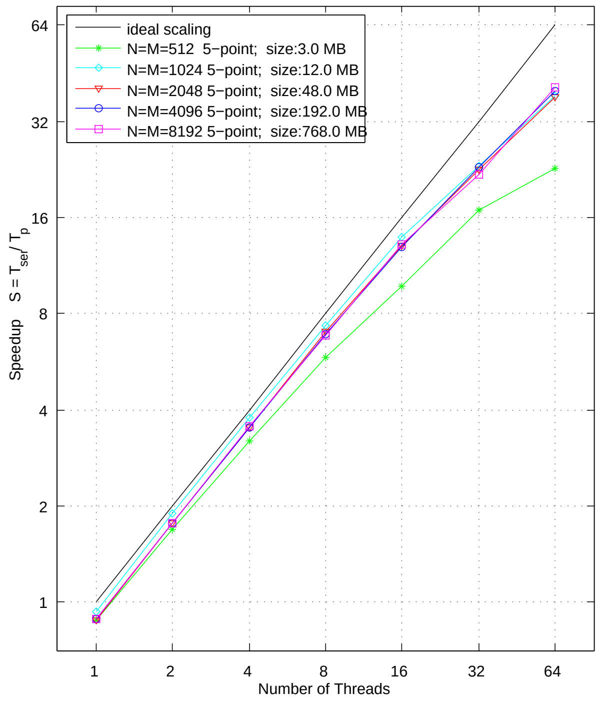


Figure 5.8: *OpenMP speedup plot for the five-star Jacobi implementation when using single precision. Based on the execution time of the Jacobi method only.*

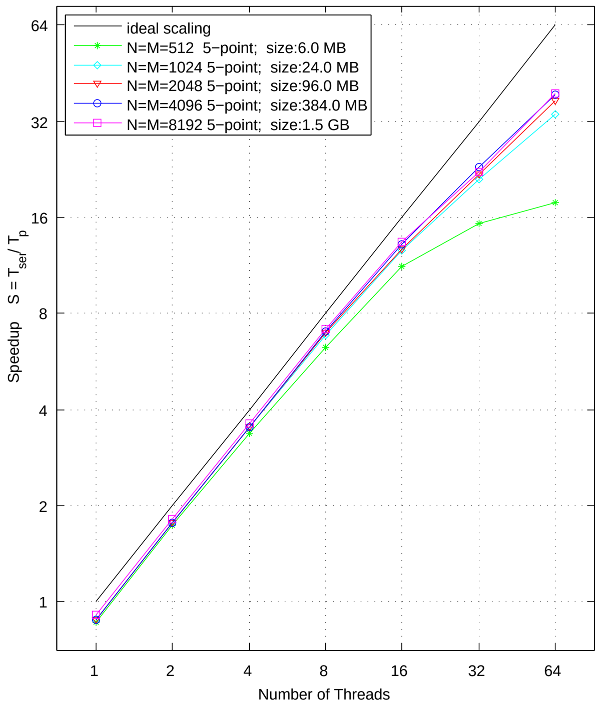


Figure 5.9: *OpenMP speedup plot for the five-star Jacobi implementation when using double precision. Based on the execution time of the Jacobi method only.*

work. A similar behaviour was shown for the double precision unmodified CSR implementation earlier, however for the single precision case it can be seen that the grid with 2048^2 grid points had mostly the best scaling for the unmodified CSR implementation.

It has been attempted to explain the scaling behaviour of the OpenMP codes with the `perf` profiling tool, however the profiling results are not conclusive. The results given from this profiling are given in the Appendix B.

Next we present and discuss the results given by the OpenACC and CUDA applications and compare them with the OpenMP results. Table 5.2 presents the absolute program execution times for all single precision implementations of this work. The OpenMP execution times given is based on the execution with 64 threads. Figure 5.12 is the graphical representation of Table 5.2. Analogously, Table 5.3 and Figure 5.13 show the double precision results. All these execution times are measured with the bash command `time` (using the `real` time given by this command), in order to also measure allocating memory on the GPU which is important for the CUDA programs. These results contain (among other things) the execution time for the creation of the CSR structure and of the right-hand side of the linear system. The right-hand side is created in parallel by all parallel methods. Though it was not possible to parallelize the creation of the CSR structure with OpenACC. Therefore we create the CSR format serially on the CPU and then transfer it to the GPU. The OpenMP and CUDA programs do parallelize the CSR creation, however. Note that in contrast to our OpenACC implementation, CUDA allocates the memory for both the right-hand side and CSR structure on the GPU's global memory and creates it directly there with no need to copy them from the host.

One clear distinction between the execution time of OpenMP programs and OpenACC/CUDA programs can be seen immediately: For the smaller grid sizes, OpenMP is substantially faster than the GPU codes. However in this grid size regime, the OpenMP execution time increases significantly more rapid with the grid size than the GPU execution times. This is due to the fact that in this regime the execution times of the GPU codes is dominated by the data transfer between the host and the device and/or memory allocation on the device. Once there is more computation involved, the effect of data transfer and device memory allocation is more and more negligible. For the bigger grids, the GPU codes run several times faster than the OpenMP codes. And even more so, Table 5.2 exhibits that for the grids with 4096^2 and 8192^2 grid points, the

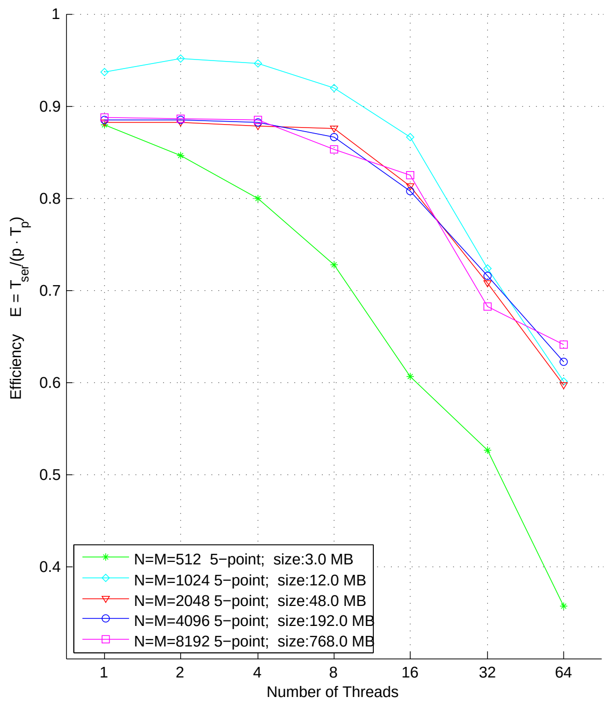


Figure 5.10: *OpenMP efficiency plot for the five-star Jacobi implementation when using single precision. Based on the execution time of the Jacobi method only.*

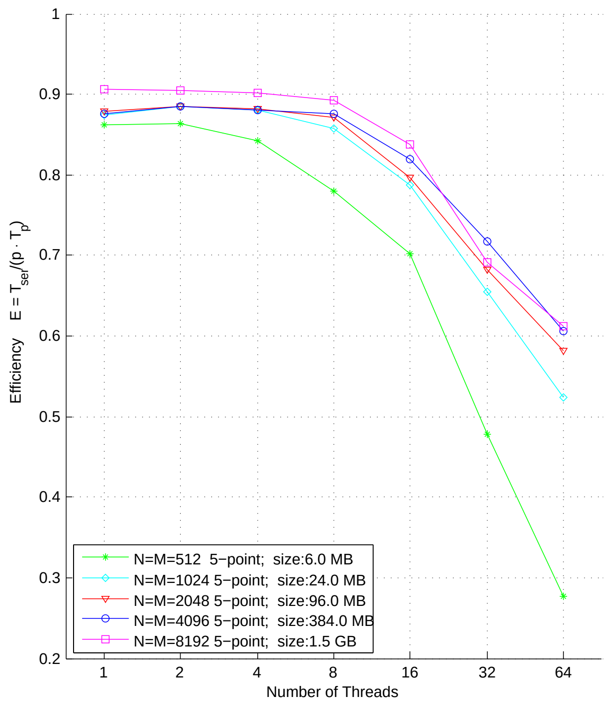


Figure 5.11: *OpenMP efficiency plot for the five-star Jacobi implementation when using double precision. Based on the execution time of the Jacobi method only.*

OpenACC version of the five-point Jacobi even is one order of magnitude faster than the OpenMP version. However this is not fully achieved using double precision, as the maximum total execution time difference factor achieved between the OpenMP and OpenACC implementations is only ~9.8 (for the five-point Jacobi with N=M=8192), whereas the maximum total execution time difference factor for the single precision case is ~17.3. Another aspect when comparing the performance of the OpenMP and GPU programs is that on the right side of the plots in Figures 5.12 and 5.13 the slopes of all three parallelization methods considered here are fairly similar with the only exception being the OpenACC five-point Jacobi implementation in single precision. It is observable that the impact of the modification to the CSR version of the Jacobi Method is fairly small for the GPU programs compared to the impact the modification has when using OpenMP. The total execution times of the CSR implementations using OpenACC and CUDA are very similar even though our OpenACC versions have the disadvantage that the creation of the CSR format is executed in serial[1].

That the impact of the CSR modification is smaller for the GPU programs becomes even more evident with a look on Figure 5.14. Similar to Figures 5.6 and 5.7, Figure 5.14 exhibits the relative time difference between the unmodified and modified Jacobi CSR implementation:

$$\tau_{rel} = \frac{(t_{mod.CSR} - t_{CSR})}{t_{CSR}}.$$

From these bar plots and can be seen that the modification to the CSR implementation has only a neglectable impact for the OpenACC programs (this is especially true for the single precision cases). Overall, τ_{rel} stays relatively constant when varying the size of the linear system for the GPU programs. This may be counterintuitive when considering that the ratio of iterations performed of the modified to unmodified CSR

[1]As an example of this impact, using the `time` command we measured the `real` execution time to allocate and create the CSR structure on the GPU (CUDA) or to create the CSR structure serially and copy it to the GPU (OpenACC) for the case using single precision and $N = M = 8192$:

- CUDA: 2.070 s
- OpenACC: 3.281 s

Table 5.2: *Summary of all execution times using single precision floating point numbers. For the OpenMP results 64 threads are used.*

	N=M=512	N=M=1024	N=M=2048	N=M=4096	N=M=8192
CSR					
OpenMP	0.051 s	0.317 s	4.369 s	74.118 s	1219.251 s
OpenACC	2.023 s	2.107 s	2.871 s	14.478 s	200.365 s
CUDA	2.019 s	2.077 s	2.875 s	16.030 s	231.274 s
Modified CSR					
OpenMP	0.049 s	0.189 s	2.246 s	56.407 s	845.732 s
OpenACC	2.022 s	2.087 s	2.946 s	14.381 s	199.863 s
CUDA	2.012 s	2.073 s	2.869 s	14.083 s	194.606 s
Five-Point Stencil					
OpenMP	0.035 s	0.245 s	3.613 s	55.711 s	864.825 s
OpenACC	2.019 s	2.041 s	2.257 s	5.417 s	48.916 s
CUDA	2.023 s	2.053 s	2.401 s	8.007 s	103.168 s

Table 5.3: *Summary of all execution times using double precision floating point numbers. For the OpenMP results 64 threads are used.*

	N=M=512	N=M=1024	N=M=2048	N=M=4096	N=M=8192
CSR					
OpenMP	0.060 s	0.365 s	5.771 s	94.853 s	1534.282 s
OpenACC	2.034 s	2.127 s	3.223 s	19.799 s	338.902 s
CUDA	2.028 s	2.103 s	3.316 s	22.965 s	338.503 s
Modified CSR					
OpenMP	0.057 s	0.198 s	4.552 s	79.046 s	1225.102 s
OpenACC	2.023 s	2.117 s	3.316 s	19.184 s	329.543 s
CUDA	2.026 s	2.088 s	3.185 s	18.818 s	269.225 s
Five-Point Stencil					
OpenMP	0.040 s	0.273 s	3.937 s	55.711 s	873.874 s
OpenACC	2.024 s	2.071 s	2.421 s	7.805 s	89.037 s
CUDA	2.024 s	2.064 s	2.638 s	11.813 s	158.706 s

versions vary with the size of the linear system, compare Figure 5.15. The CUDA double precision values for τ_{rel} are approximately equivalent to the 64 threads OpenMP double precision values for the three biggest linear systems. A difference in behaviour of τ_{rel} between the OpenMP and CUDA is however that for CUDA the relative time difference of the single precision case is smaller in absolute values than its double precision analogue, while for the 64 thread OpenMP the single precision case results in a bigger absolute value of τ_{rel} compared to its double precision equivalent. That the single precision case for GPU the values of $|\tau_{rel}|$ are smaller than the double precision analogue can be connected to the fact that there are three times as many single precision cores on the Tesla K40 GPU than double precision cores. For this reason the reduction operations can be performed more efficiently. It is still needed to investigate why the values of τ_{rel} differ mentionably between the OpenACC and CUDA programs, especially since for both versions the reduction process was left to the compiler, as the intrinsic CUDA Fortran (CUF) directive was used for the reduction operation in the CUDA programs.

Having dealt with the general difference between CPU and GPU execution of our programs, we will now compare the performance of OpenACC and CUDA. As can be easily observed, the execution times of CUDA and OpenACC for the CSR codes are rather similar here, even though the CSR format is created serially in the OpenACC version. However for the five-point stencil Jacobi Method, the OpenACC version clearly outperforms the CUDA version for the two biggest grids considered in this work. To understand why

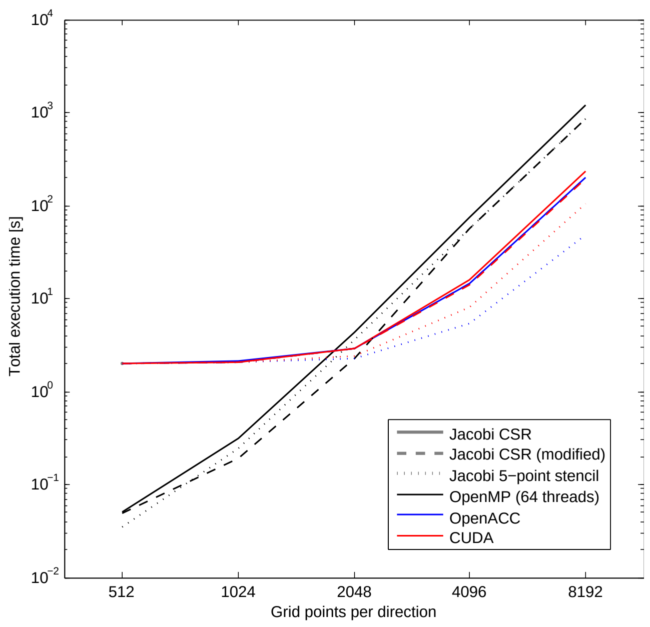


Figure 5.12: *Graphical representation of Table 5.2.*

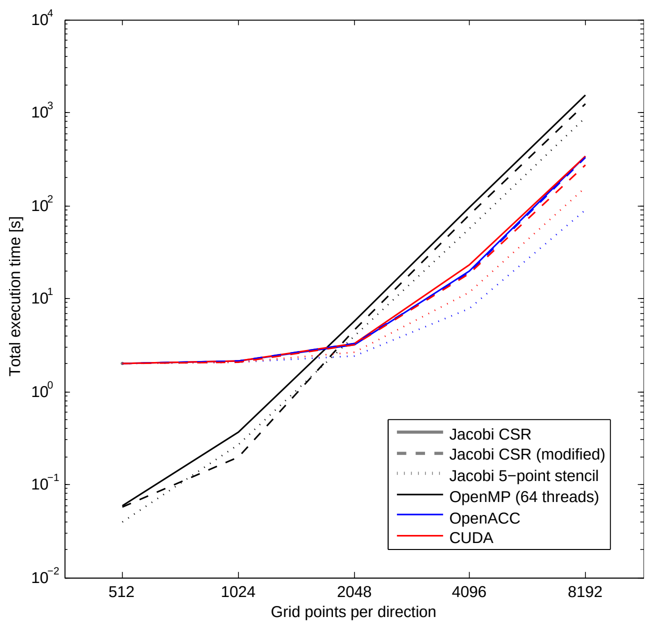


Figure 5.13: *Graphical representation of Table 5.3.*

this is the case it is necessary to look at some profiling results. The data in the following figures and tables were received using the NVIDIA Visual Profiler tool. There is actually only one Jacobi Jacobi CSR kernel to be considered here as the modification is done on the host. This means that even though we still have a modified and an unmodified CSR Jacobi implementation, the actual Jacobi kernel is identical in both cases.

Note that for all GPU programs we use a block size of [128,1,1]. The CUDA grid sizes are listed in Tables 5.4 and 5.5 for the (modified) CSR and five-point Jacobi versions, respectively. The CUDA grid sizes for the OpenACC programs were left to the compiler, whereas the CUDA kernels were programmed in a manner that would use one thread for every unknown or grid point. For reference, the maximum grid dimension achievable on the Tesla K40 is [2147483647, 65535, 65535]. As can be seen in Table 5.4, the OpenACC Jacobi CSR kernel is invoked with the same grid size as the respective CUDA kernel for the first three computational grids, $N = M = \{512, 1024, 2048\}$. But from the computational grid with $N = M = 4096$ on, the CUDA grid size acquires the maximum number of blocks achievable in the x- or y-coordinate for its x-value. In contrast to that, the CUDA grids continue to have the x-coordinate of the last smaller grid multiplied by four in order to guarantee that every thread calculates exactly one unknown. Looking at the grid sizes for the five-point implementation the difference in number of threads is even more prominent. For the CUDA versions, the CUDA grid sizes are equivalent to the ones used for the CSR cases, with the difference that now a two-dimensional CUDA grid is used: The y-coordinate of the CUDA grid equals M, while the x-coordinate equals $N/128$. The OpenACC version of the five-point implementation on the other hand remains to use a one-dimensional CUDA grid with the x-coordinate being $N = M$, there using 4 – 128 times less threads than the CUDA counterpart for the considered computational grids. This means that every thread in the OpenACC version calculates up to 128 unknowns. This keeps each thread busy and less warp scheduling is needed compared to the five-point CUDA kernel.

Tables 5.6 – 5.9 list the limiters for the Jacobi Kernels on the GPU. One limitation that every of our implementations suffer is the instruction latency, however only in some cases is it the primary limiter of the kernels. Due to memory dependence, instruction execution is stalling excessively resulting in the GPU not having enough work. In the analysis report created by NVIDIA's Visual Profiler it is explained that in this case

> "[a] load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns."

Computation and function unit utilization, on the other hand, never limit the kernels' performance. Another

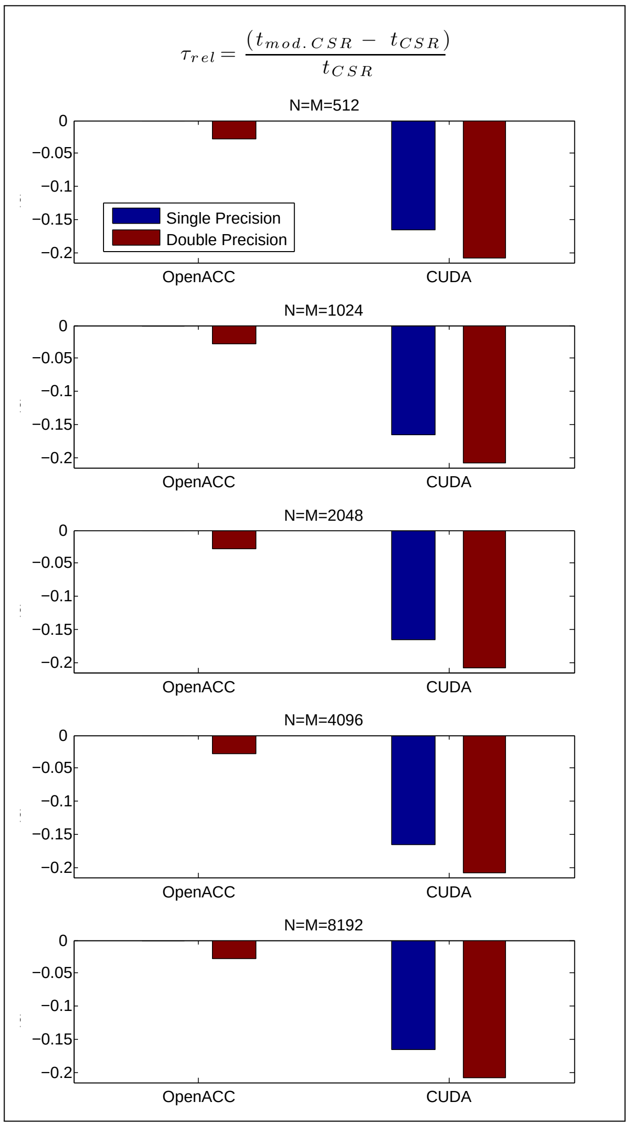


Figure 5.14: *Relative execution time difference between the modified and unmodified Jacobi CSR implementations for the OpenACC and CUDA programs for both single and double precision. All times based on the Jacobi kernel execution times only.*

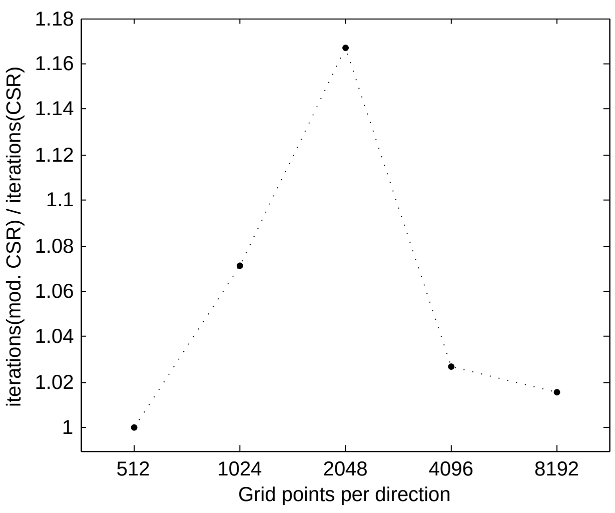


Figure 5.15: *Plot of the iteration fraction of the modified to unmodified CSR implementation.*

Table 5.4: *Grid Sizes for the (Modified) Jacobi CSR Implementations*

	Grid Size OpenACC	Grid Size CUDA
512	[2048,1,1]	[2048,1,1]
1024	[8192,1,1]	[8192,1,1]
2048	[32768,1,1]	[32768,1,1]
4096	[65535,1,1]	[131072,1,1]
8192	[65535,1,1]	[524288,1,1]

Table 5.5: *Grid Sizes for the Five-Point Stencil Jacobi Implementations*

	Grid Size OpenACC	Grid Size CUDA
512	[512,1,1]	[4,512,1]
1024	[1024,1,1]	[8,1024,1]
2048	[2048,1,1]	[16,2048,1]
4096	[4096,1,1]	[32,4096,1]
8192	[8192,1,1]	[64,8192,1]

limitation to the performance of the kernels on the GPU is memory bandwidth. In most cases presented here the bandwidth of the L2 cache is the limiter, however the device memory bandwidth is a limiter for the OpenACC implementations on the grid with 8192^2 grid points, as well as the double precisions versions of the five-point implementations (OpenACC and CUDA). The bandwidth usage of the L1 cache, L2 cache, texture memory, and device memory for all kernels is portrayed in Figures 5.16 – 5.19. Observe that the L1 cache is hardly used, whereas the texture memory is used a lot (even though it was not specifically declared in the source code) and offers a significantly higher bandwidth than the other memories. The number of registers needed for each thread is a limiting factor for the CSR implementations using double precision and all five-point implementations. Tables 5.10 – 5.17 show how much registers per thread are used for every case and how many blocks and warps can therefore execute simultaneously on each SM. Note that the maximum number of blocks per SM on the Tesla K40 is 16 and the maximum number of warps is 64. As can be seen in Tables 5.10 and 5.12, the SM is optimally occupied with blocks and warps, increasing or decreasing the block size would therefore decrease GPU performance. Another detail of these implementations observable in Tables 5.10 – 5.17 is that the CUDA versions generally use less registers than the OpenACC versions. This can be linked to the fact that each thread in the CUDA implementation performs less computations and therefore needs less data. Additionally, Figures 5.20 – 5.21 display the theoretical and achieved number of active warps per SM, and Figures 5.22 – 5.23 show the theoretical and achieved occupancy of the GPU for each implementation. Occupancy is the percentage of active warps on the GPU relative to the maximum number of warps supported by the GPU. The theoretical values are upper bounds, while the achieved values are indications for the actual values on the GPU.

Table 5.6: *Kernel Limiters: (Modified) Jacobi CSR, Single Precision. The ^{1} superscript marks the primary limiter.*

	OpenACC					CUDA				
	512	1024	2048	4096	8192	512	1024	2048	4096	8192
Instruction Latency	●^{1}	●	●	●	●	●^{1}	●	●	●	●
Memory Bandwidth	● (L2)	●^{1} (L2)	●^{1} (L2)	●^{1} (L2)	●^{1} (L2 + Device)	● (L2)	●^{1} (L2)	●^{1} (L2)	●^{1} (L2)	●^{1} (L2)
Computation										
Registers										
Function Unit Utilization										
Intra-Warp Divergence	●					●				

Finally, we present a comparison of the performance between the single precision and double precision cases in Figure 5.24. Again the OpenMP values are based on the execution with 64 threads and therefore using all processor cores available on the Silicon Graphics machine. It must be noted again that all these results are achieved after a single execution of the respective program and case, and therefore do not give a statistical representation of the general behaviour.

The bars in Figure 5.24 indicate how much faster the single precision cases are compared and normalized to their double precision equivalent. For the OpenMP versions it can be observed that the unmodified CSR

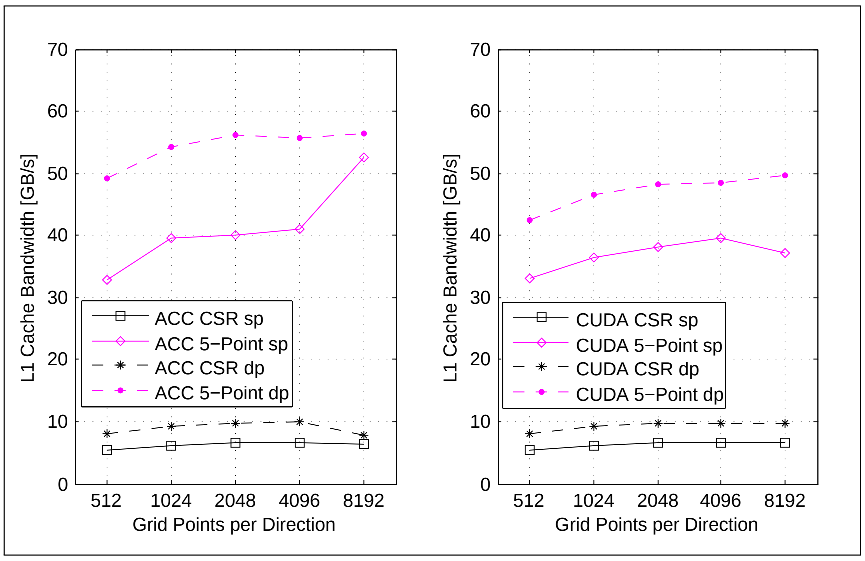


Figure 5.16: *Bandwidth utilization of the GPU L1 cache for all implemented Jacobi kernels.*

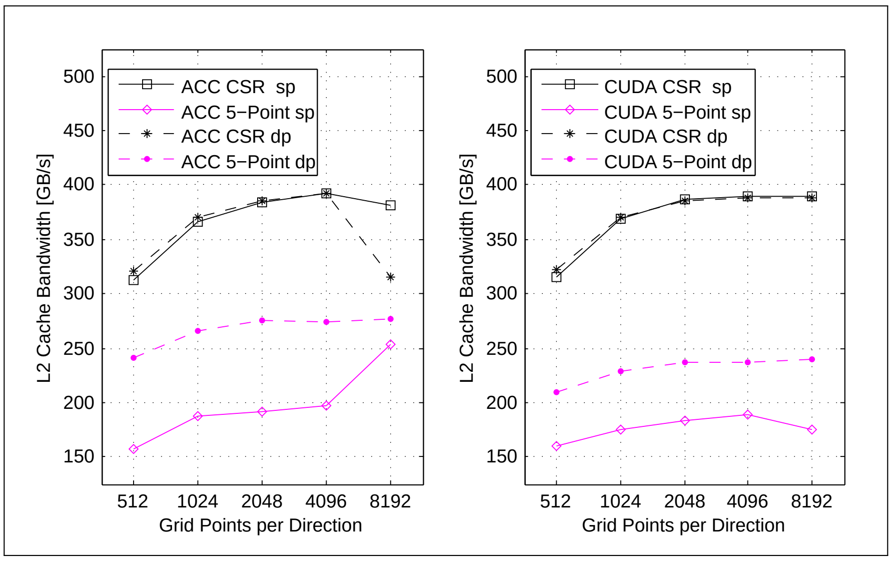


Figure 5.17: *Bandwidth utilization of the GPU L2 cache for all implemented Jacobi kernels.*

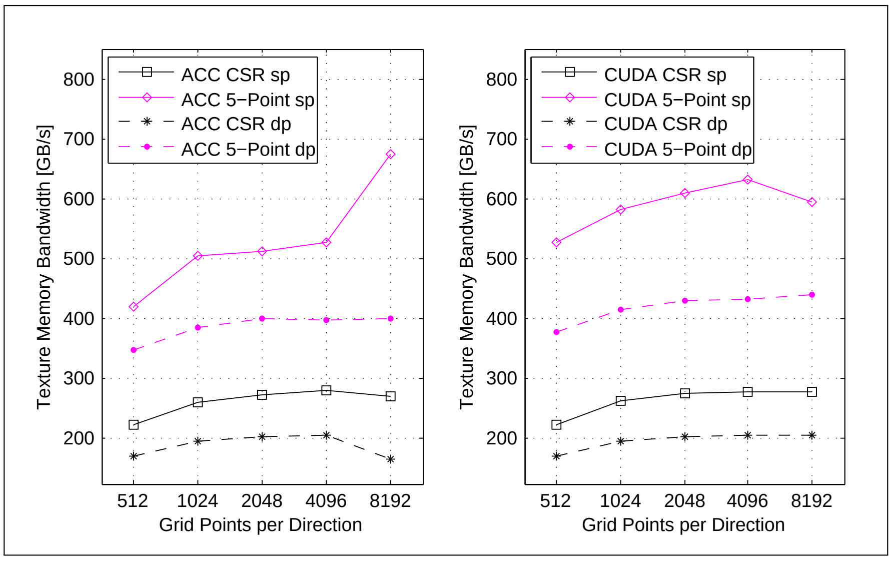


Figure 5.18: *Bandwidth utilization of the texture memory for all implemented Jacobi kernels.*

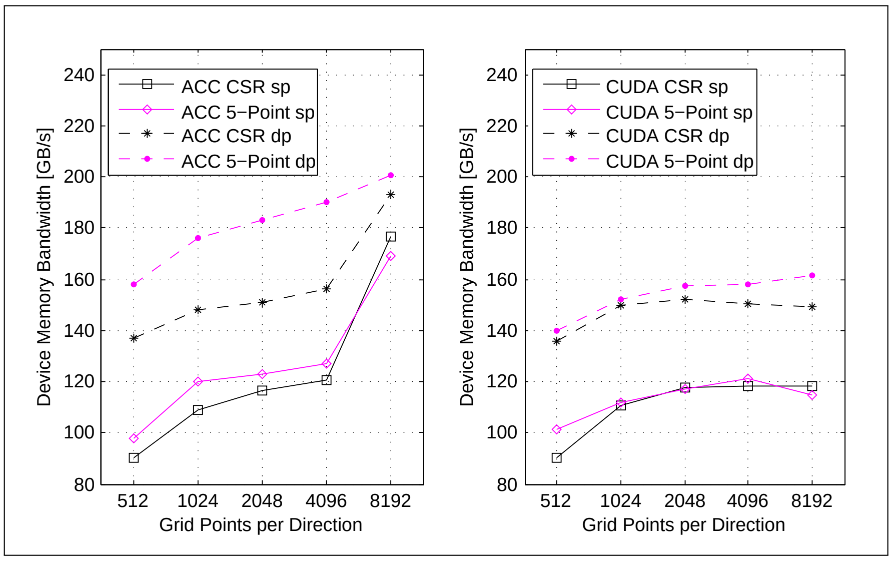


Figure 5.19: *Bandwidth utilization of the device memory for all implemented Jacobi kernels.*

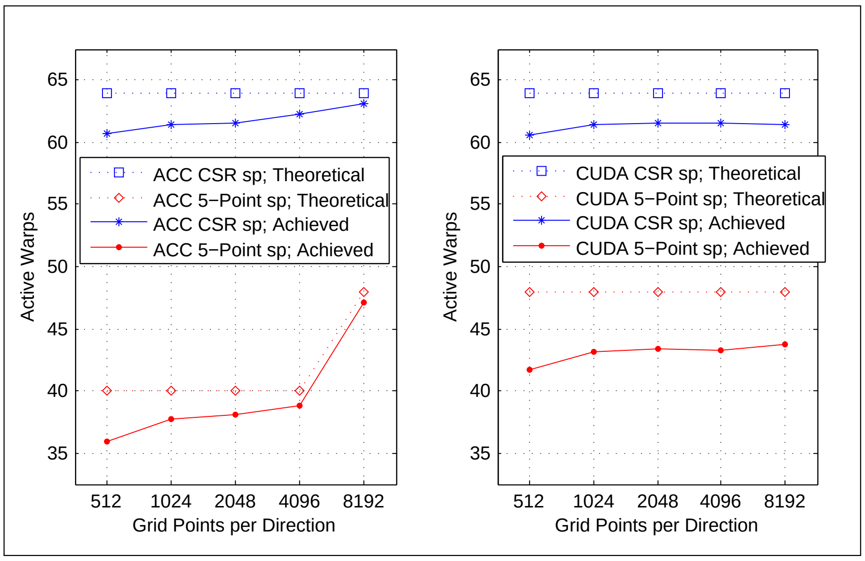


Figure 5.20: *Plot of active warps for all implemented Jacobi kernels, using single precision.*

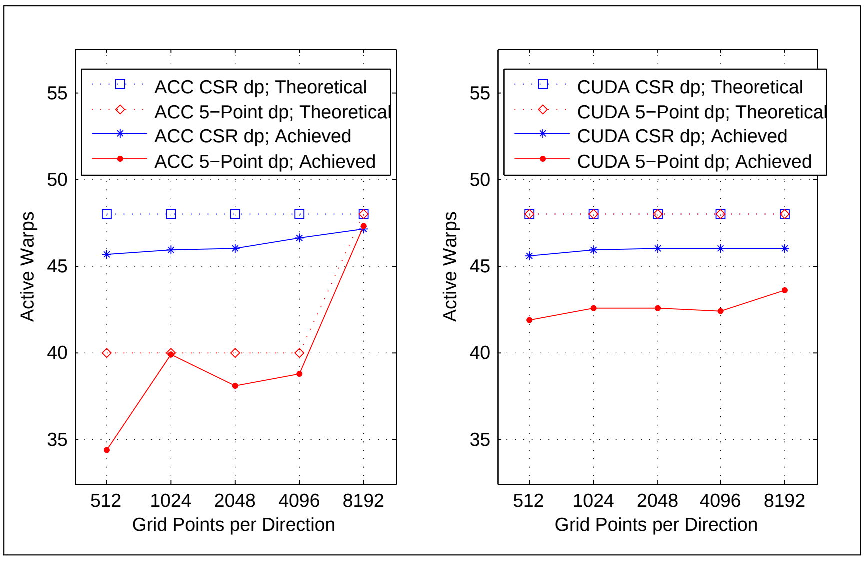


Figure 5.21: *Plot of active warps for all implemented Jacobi kernels, using double precision.*

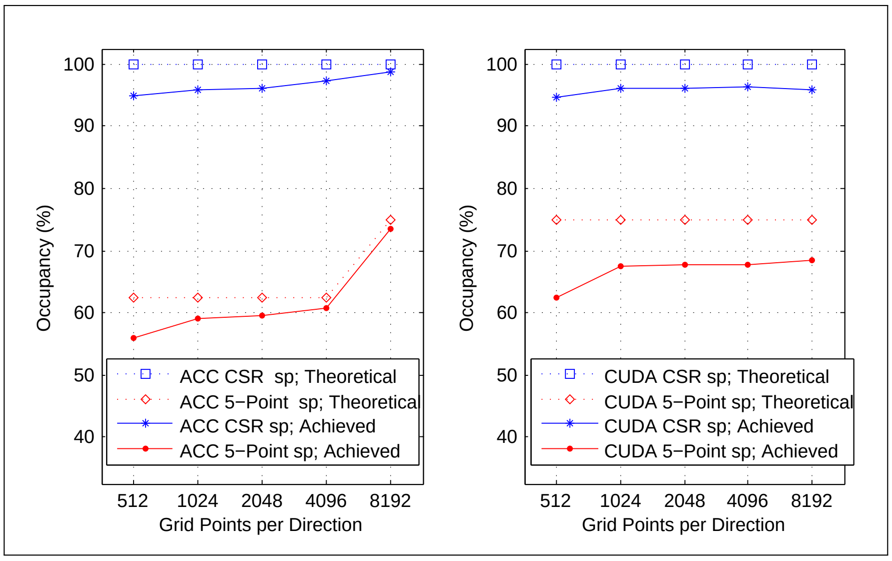


Figure 5.22: *Occupancy plot for all implemented Jacobi kernels, using single precision.*

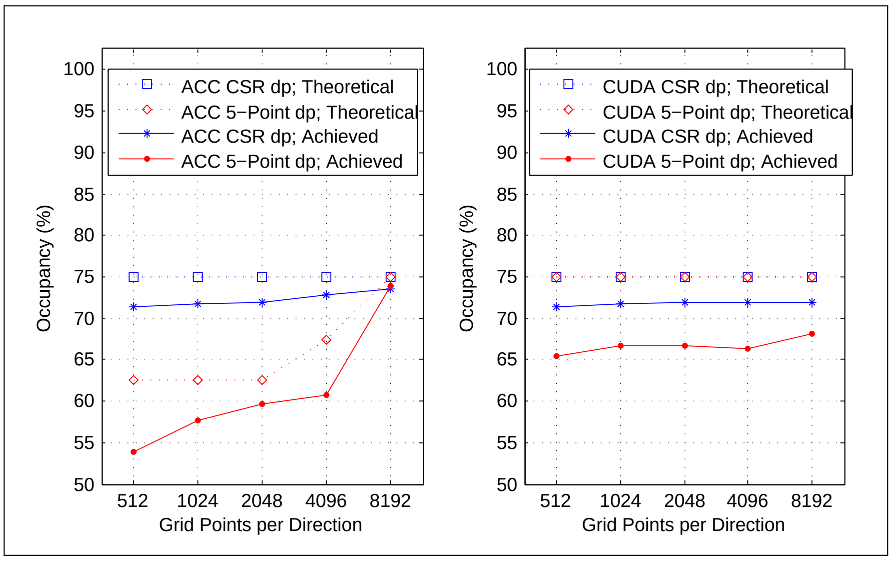


Figure 5.23: *Occupancy plot for all implemented Jacobi kernels, using double precision.*

Table 5.7: *Kernel Limiters: (Modified) Jacobi CSR, Double Precision. The ^{1} superscript marks the primary limiter.*

	OpenACC					CUDA				
	512	1024	2048	4096	8192	512	1024	2048	4096	8192
Instruction Latency	●^{1}	●	●	●	●	●^{1}	●	●	●	●
Memory Bandwidth	●	●^{1}	●^{1}	●^{1}	●^{1}	●	●^{1}	●^{1}	●^{1}	●^{1}
	(L2)	(L2)	(L2)	(L2)	(Device)	(L2)	(L2)	(L2)	(L2)	(L2)
Computation										
Registers	●	●	●	●	●	●	●	●	●	●
Function Unit Utilization										
Intra-Warp Divergence	●					●				

Table 5.8: *Kernel Limiters: Five-Point Stencil Jacobi, Single Precision. The ^{1} superscript marks the primary limiter.*

	OpenACC					CUDA				
	512	1024	2048	4096	8192	512	1024	2048	4096	8192
Instruction Latency	●^{1}	●^{1}	●^{1}	●^{1}	●^{1}	●^{1}	●^{1}	●^{1}	●^{1}	●^{1}
Memory Bandwidth					●					
					(Device)					
Computation										
Registers	●	●	●	●	●	●	●	●	●	●
Function Unit Utilization										
Intra-Warp Divergence										

Table 5.9: *Kernel Limiters: Five-Point Stencil Jacobi, double precision. The ^{1} superscript marks the primary limiter.*

	OpenACC					CUDA				
	512	1024	2048	4096	8192	512	1024	2048	4096	8192
Instruction Latency	●^{1}	●	●	●	●	●^{1}	●^{1}	●^{1}	●^{1}	●^{1}
Memory Bandwidth	●	●^{1}	●^{1}	●^{1}	●^{1}		●	●	●	●
	(Device)	(Device)	(Device)	(Device)	(Device)		(Device)	(Device)	(Device)	(Device)
Computation										
Registers	●	●	●	●	●	●	●	●	●	●
Function Unit Utilization										
Intra-Warp Divergence										

implementation's relative total execution time difference between single and double precision

$$\tau_{rel} = \frac{(t_{singleprec.} - t_{doubleprec.})}{t_{doubleprec.}}$$

varies very little, $\tau_{rel} \approx 1.5$ for the two smallest grids considered here and $\tau_{rel} \approx 2 - 2.3$ for the rest of the grids considered. For the two other OpenMP implementations (modified CSR and five-point) no clear pattern can be observed. The GPU codes have a similar order of magnitude for τ_{rel} for any given computational grid. For the two smallest grids there is no mentionable relative time difference due to the impact of allocating and copying data to the GPU. With bigger computational grid sizes, however, the computation dominates the total execution time and $|\tau_{rel}|$ increases. As was mentioned in Section 2.3, the Tesla K40 has three times as many single precision cores as double precision cores, nonetheless the single precision versions are not three times as fast as the double precision pendant. The reason for this is that the (single precision) integer operations prevail for all the Jacobi implementations (CSR and five-point).

As a final remark, it shall be mentioned that the two-dimensional Poisson Equation exhibits relatively little work per grid point. A greater speedup can be expected for parallelizations of systems that require

Table 5.10: *Register restriction: OpenACC, (Modified) Jacobi CSR, Single Precision*

	512	1024	2048	4096	8192
Registers/Thread	23	23	23	23	20
Blocks/SM (simultaneously)	16	16	16	16	16
Warps/SM (simultaneously)	64	64	64	64	64

Table 5.11: *Register restriction: OpenACC, (Modified) Jacobi CSR, Double Precision*

	512	1024	2048	4096	8192
Registers/Thread	36	36	36	36	36
Blocks/SM (simultaneously)	12	12	12	12	12
Warps/SM (simultaneously)	48	48	48	48	48

Table 5.12: *Register restriction: CUDA, (Modified) Jacobi CSR, Single Precision*

	512	1024	2048	4096	8192
Registers/Thread	17	17	17	17	17
Blocks/SM (simultaneously)	16	16	16	16	16
Warps/SM (simultaneously)	64	64	64	64	64

Table 5.13: *Register restriction: CUDA, (Modified) Jacobi CSR, Double Precision*

	512	1024	2048	4096	8192
Registers/Thread	36	36	36	36	36
Blocks/SM (simultaneously)	12	12	12	12	12
Warps/SM (simultaneously)	48	48	48	48	48

Table 5.14: *Register restriction: OpenACC, Five-point Jacobi, Single Precision*

	512	1024	2048	4096	8192
Registers/Thread	45	45	45	45	39
Blocks/SM (simultaneously)	10	10	10	10	12
Warps/SM (simultaneously)	40	40	40	40	48

Table 5.15: *Register restriction: OpenACC, Five-point Jacobi, Double Precision*

	512	1024	2048	4096	8192
Registers/Thread	45	45	45	45	39
Blocks/SM (simultaneously)	10	10	10	10	12
Warps/SM (simultaneously)	40	40	40	40	48

Table 5.16: *Register restriction: CUDA, Five-point Jacobi, Single Precision*

	512	1024	2048	4096	8192
Registers/Thread	35	35	35	35	35
Blocks/SM (simultaneously)	12	12	12	12	12
Warps/SM (simultaneously)	48	48	48	48	48

Table 5.17: *Register restriction: CUDA, Five-point Jacobi, Double Precision*

	512	1024	2048	4096	8192
Registers/Thread	36	36	36	36	36
Blocks/SM (simultaneously)	12	12	12	12	12
Warps/SM (simultaneously)	48	48	48	48	48

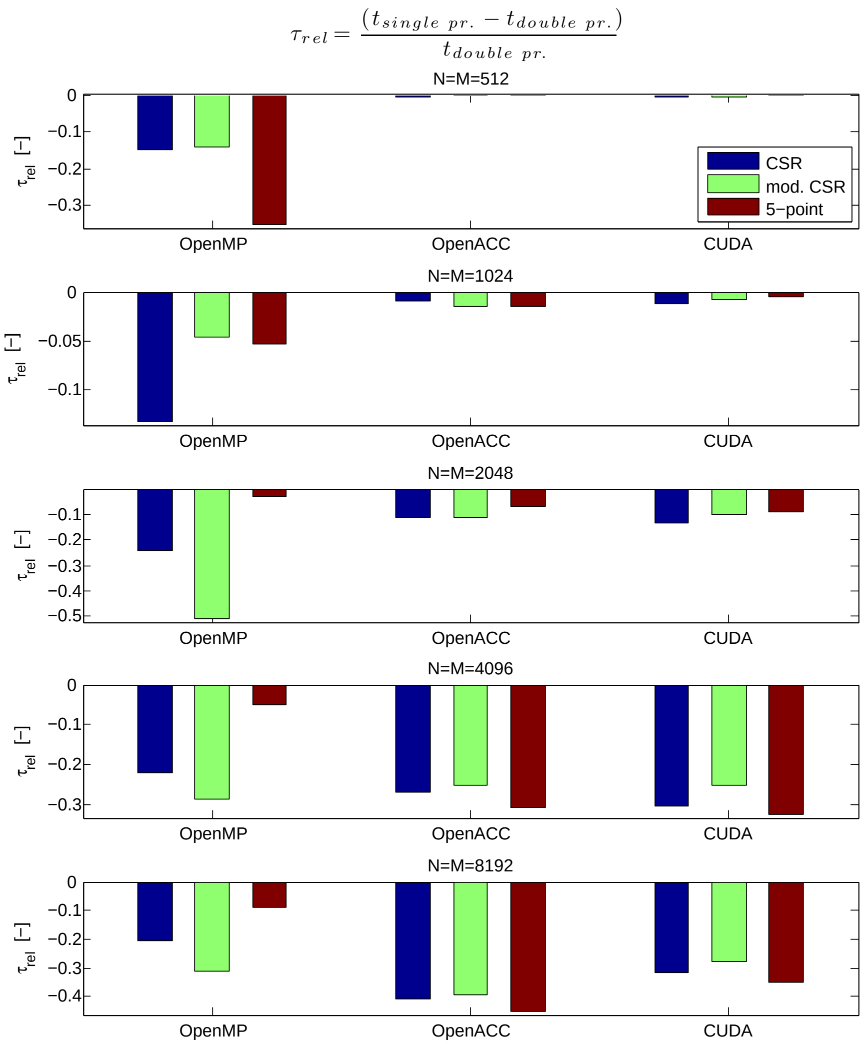


Figure 5.24: *Relative total execution time differences between single and double precision cases for all implementations used.*

more computational work per grid point, like the three-dimensional Poisson Equation or the Navier-Stokes Equations in two or more dimensions, to name only a few. The CSR implementations require more work per thread than the five-point analogue, however this extra work is not parallelizable. In this work, especially the grid with 512^2 grid points do not require sufficiently much computation to keep the processors busy. Moreover, keep in mind that in Tables 5.2 and 5.3, as well as in Figures 5.12, 5.13, and 5.24, we use all 64 processor cores of the Silicon Graphics machine for the OpenMP programs as it yields the biggest speedup. This way, though, it can be expected that the performance is more affected by background processes, as there are no free processors run daemons or other operating system tasks. More context switches will occur as a result, and threads that have run on one processor may continue to run on another which makes it necessary to copy the data needed for the computations in the respective caches. Due to this sensitivity to background processes performance suffers and the reproducibility will depend more on the state of the system.

Chapter 6

Conclusion

Parallelism will become more and more important, if not to say a necessity, in order to make use of modern hardware. In general the parallelization of a program is a iterative process which may require multiple profiling and optimization stages to make the most use of a given hardware. Even though there are parallelization methodologies like OpenMP and OpenACC that offer to parallelize am already existing program in the same source code by simply adding directives and library routines, however in the optimization stages it may be necessary to rewrite the serial code in order to be able to exploit more parallelism when adding the respective directives and library routines. It should be evident that one is able to make more use of the optimization stages the more one knows how the parallelization process is performed in the respective hardware and what the bottlenecks and potential factors that could boost the performance of a specific application on a given hardware are.

In this work we consider two implementations of a Jacobi solver for general sparse linear systems using the CSR format which are parallelized using OpenMP, OpenACC and CUDA. We apply this solver to the two-dimensional Poisson Equation which is discretized using finite differences and compared it to a direct Poisson equation solver based on the finite difference scheme and compared the performances of different configurations for different computational grid sizes and when using single precision or double precision, respectively. It was attempted to explain the performance behaviour of the different cases based on profilings of the respective programs.

Unfortunately, the profiling for the OpenMP programs using the `perf` profiling tool is not conclusive[1]. We observe different scaling behaviours of the different Jacobi implementations using OpenMP, also when comparing the single precision and double precision versions of the respective implementation. Note that even though the Jacobi method is embarrassingly parallel an ideal linear speedup was not achieved due to the necessity of synchronization. The only behaviour we notice to be shared among all configurations is a rapid scaling degradation when using more than eight threads or processor cores (except for the smallest grid considered in our work which already exhibits this behaviour for more than four threads). When using eight processor cores on our Silicon Graphics machine, there will be two active cores per processor each being able to use an entire L2 cache only for themselves. Whether the need to share the L2 cache with other cores when increasing the number of threads is the reason for the loss in efficiency may require further investigation. Moreover it is observed that the modified CSR Jacobi version in general exhibits worse scaling than the unmodified equivalent, however still execute faster. Why there is a loss in scaling with the modification is still left to be determined. Similarly we are not able to explain why the scaling of CSR implementations differ from the five-point implementations for different grid sizes.

Comparing the execution times of the GPU codes we find that the overall execution time for the CSR implementations are quite similar for the CUDA and OpenACC programs. This however is only the case because the creation of the CSR format has not been successfully parallelized with OpenACC and therefore run serially in this case which overall results in a slightly faster execution time of the Jacobi Method for OpenACC. This execution time difference between OpenACC and CUDA is even more obvious when looking at the five-point Jacobi solver. Here the the OpenACC version is up to two times faster than the CUDA equivalent for the biggest computational grid considered. We link these execution time differences

[1]The results of `perf` profilings are given in Appendix B

to optimizations performed by the compiler for the OpenACC code. These optimization result in more work per thread, while we only solve one unknown per thread with our CUDA program. Furthermore, we suspect compiler optimizations to result in an only insignificant relative execution time difference between the modified and unmodified CSR implementations for OpenACC.

As SMPs have comparatively few but fast processors and no expensive data transfer to their memory is needed, whereas GPUs comprise a high number of comparatively slow processing units and data first needs to be copied from the host, OpenMP is more advantageous for smaller to medium sized problems. This is because for those situations the performance of the GPU codes are dominated by the data transfer. For bigger problems, however, data transfer cost becomes less important and computation becomes the dominant factor. For problems that exhibit a sufficient amount of parallelism the GPU codes will perform drastically faster than an SMP code with up to 64 threads. OpenACC offers a good option for novice GPU developers and in situation in which it is desired to parallelize an already existing big serial code for the GPU. The compiler can already provide effective optimizations for the OpenACC application. However with OpenACC there is a lack of control since the entire parallelization is performed by the compiler, it is therefore not possible for the programmer to explicitly optimize the code for the GPU. CUDA provides such control and it can therefore be used to explicitly optimize program execution in the source code. It is possible to make use of all OpenMP, OpenACC and CUDA in one single code. This way these parallelization paradigms can be used in a way that combines the best properties of each.

When comparing the CSR and five-point Jacobi implementations it is observable that the price one has to pay for generality is a higher execution time and less optimizabilty. Because the CSR Jacobi is designed to solve a general sparse system it is not possible modify the code in a way that guarantees that the data needed to solve the linear system will be closer together in the cache. For the five-point Poisson Equation solver, though, it is possible to divide the computational domain into tiles that can be completely stored in the cache of a processor core and let every processor cache solve a different tile. Moreover, in order to make better use of the GPU and increase the compute to global memory access (CGMA) ratio it would be beneficial iterate several times on a given tile before synchronization occurs. In a future work it can be investigated how such a domain decomposition affects the GPU performance and what the impact on the iteration number needed to achieve a specific tolerance is. Another part of the five-point implementation that can be optimized in a future work is the update copy `u_old = u`. A more efficient implementation would iterate twice while implicitly updating `u_old` in the second iteration:

$$1) \qquad \mathtt{u(i,j)} = \delta^2 \mathtt{f(i,j)} + \begin{bmatrix} & \theta_x & \\ \theta_y & & \theta_y \\ & \theta_x & \end{bmatrix} \mathtt{u\_old(i,j)}$$

$$2) \qquad \mathtt{u\_old(i,j)} = \delta^2 \mathtt{f(i,j)} + \begin{bmatrix} & \theta_x & \\ \theta_y & & \theta_y \\ & \theta_x & \end{bmatrix} \mathtt{u(i,j)}$$

Moreover, as discussed in Appendix B, in a future work it may be desired to perform various instances of profilings per OpenMP program, each measuring a different set of events, in order to overcome multiplexing effects.

Appendices

Appendix A

Essential Definitions and Theoretical Results

Here we give the basic mathematical background for the discussion in Chapter 3.

A.1 Vector Norms

Whenever a linear system is solved numerically, whether it is by means of direct or iterative methods, in general it is only possible to achieve an approximate solution $\mathbf{x}$ that differs from the exact solution $\hat{\mathbf{x}}$ of a linear system

$$A\hat{\mathbf{x}} = \mathbf{f}.$$

Therefore, in order to be able to judge the accuracy of a numerical (approximate) solution it is necessary to measure the distance between two n-dimensional vectors. With such a measure it will also be possible to answer the question of whether an n-dimensional vector sequence will converge.

In the following we will only consider the vector space $\mathbb{C}^n$ which reduces to $\mathbb{R}^n$ as a special case.

Definition 4 (Vector norm Stoer and Bulirsch [1980]). *A vector norm is a real-valued function $\|\cdot\| : \mathbb{C}^n \to \mathbb{R}$ with the following properties:*

(i) $\|\mathbf{x}\| > 0 \;\; \forall \mathbf{x} \in \mathbb{C}^n, \mathbf{x} \neq \mathbf{0}$; $\;\|\mathbf{x}\| = 0$ *only if* $\mathbf{x} = \mathbf{0}$ *(non-negativity)*

(ii) $\|\alpha\mathbf{x}\| = |\alpha|\,\|\mathbf{x}\| \;\; \forall \alpha \in \mathbb{C}, \mathbf{x} \in \mathbb{C}^n$ *(homogeneity)*

(iii) $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ *(triangle inequality)*

The most commonly used norms in numeric analysis are:

l_1 **norm** $\|\mathbf{x}\|_1 = \sum_{i=1}^{n} x_i$;

l_2 **norm** $\|\mathbf{x}\|_2 = \left(\mathbf{x}^H\mathbf{x}\right)^{1/2}$;

l_∞ **norm** $\|\mathbf{x}\|_\infty = \max_i |x_i|$.

The superscript ^{H} stands for the hermitian operator:

$$\mathbf{x}^H = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)^T,$$

where the superscript ^{T} denotes the transpose operator and $\bar{x}_i$ is the complex conjugate of x_i. All of the above vector norms belong to the class of *p-norms*

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}.$$

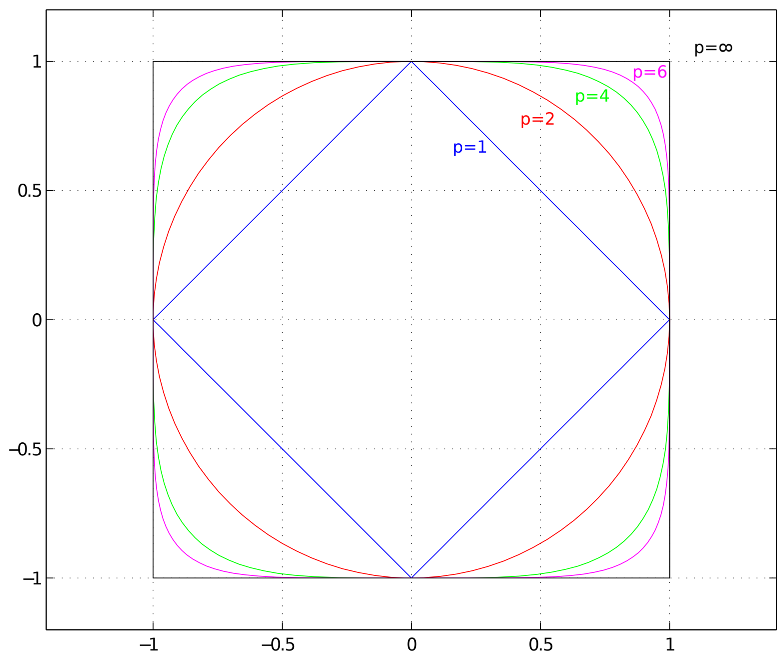


Figure A.1: *Plots of* $\|\mathbf{x}\|_p = 1$ *on* $\mathbb{R}^2$ *for different values of* p.

For the l_1 and l_2 norm this is obvious, for the l_∞ norm consider the following: Obviously

$$|x_k| \leq \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p} = \|\mathbf{x}\|_p$$

holds for any $p \geq 1$ and $1 \leq k \leq n$. Therefore, we have

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_p, \quad \forall p \geq 1,$$

and thus specifically

$$\|\mathbf{x}\|_\infty \leq \lim_{p\to\infty} \|\mathbf{x}\|_p.$$

On the other hand,

$$\lim_{p\to\infty} \|\mathbf{x}\|_p = \lim_{p\to\infty} \max_k |x_k| \left(\sum_{i=1}^{n} \left(\frac{|x_i|}{|x_k|}\right)^p\right)^{1/p} \leq \lim_{p\to\infty} \max_k |x_k| \cdot n^{1/p} = \|\mathbf{x}\|_\infty.$$

Which shows that $\|\mathbf{x}\|_\infty = \lim_{p\to\infty} \|\mathbf{x}\|_p$, in other words the l_∞ norm belongs to the class of p-norms. Figure A.1 portrays the graphs of $\|\mathbf{x}\|_p = 1$ on $\mathbb{R}^2$ for different values of p.

Note, however, that there are also different kind of vector norms. For an example see

Theorem 6. *Let* $\|\cdot\|$ *be an arbitrary norm on* $\mathbb{C}^n$ *and* P *an arbitrary non-singular* $n \times n$ *complex matrix. Then* $\|\mathbf{x}\|' = \|P\mathbf{x}\|$ *defines a norm on* $\mathbb{C}^n$. *Ortega [1972]*

Proof. Define $\mathbf{y} := P\mathbf{x}$ and note that since P is non-singular, it is a bijective mapping. It then follows that $\|\mathbf{x}\|' = \|\mathbf{y}\|$ is a norm since $\|\cdot\|$ is a norm for all $\mathbf{y} \in \mathbb{C}^n$. □

The distance between two n-dimensional vectors $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$ is defined as $\|\mathbf{x} - \mathbf{y}\|$.

Lemma 1. *For each norm the inequality*

$$\|\mathbf{x} - \mathbf{y}\| \geq \big|\, \|x\| - \|y\| \,\big|, \quad \forall \mathbf{x}, \mathbf{y} \in \mathbb{C}^n \tag{A.1}$$

holds.

Proof. From the triangle inequality in Definition 4 we have

$$\|\mathbf{x}\| = \|(\mathbf{x}-\mathbf{y})+\mathbf{y}\| \leq \|\mathbf{x}-\mathbf{y}\| + \|\mathbf{y}\|,$$

and therefore $\|\mathbf{x}-\mathbf{y}\| \geq \|\mathbf{x}\| - \|\mathbf{y}\|$. Furthermore, using the homogeneity property of Definition 4 we see that

$$\|\mathbf{x}-\mathbf{y}\| = \|\mathbf{y}-\mathbf{x}\| \geq \|\mathbf{y}\| - \|\mathbf{x}\|,$$

which proves Equation (A.1). Stoer and Bulirsch [1980] □

Theorem 7. *Each norm* $\|\cdot\|$ *is a uniformly continuous function with respect to the metric*

$$m(\mathbf{x},\mathbf{y}) = \max_i |x_i - y_i|, \quad \mathbf{x},\mathbf{y} \in \mathbb{C}^n.$$

Proof. From the inequality (A.1) we have

$$|\,\|\mathbf{x} = \mathbf{h}\| - \|\mathbf{x}\|\,| \leq \|\mathbf{h}\|.$$

Since we can write $\mathbf{h}$ as $\mathbf{h} = \sum_{i=1}^n h_i e_i$, where $\mathbf{h} = (h_1, h_2, \ldots, h_n)^T$, and e_i being the canonical basis vectors of $\mathbb{C}^n$. It follows that

$$\|\mathbf{h}\| \leq \sum_{i=1}^n |h_i|\,\|e_i\| \leq \max_i |h_i| \sum_{j=1}^n \|e_j\| + M \max_i |h_i|,$$

where we defined $M := \sum_{j=1}^n \|e_j\|$. Thus, for each $\epsilon > 0$ and all $\mathbf{h}$ satisfying $max_i|h_i| \leq \epsilon/M$, the inaquality

$$|\,\|\mathbf{x} + bfh\| - \|\mathbf{x}\|\,| \leq \epsilon$$

holds, which proves that $\|\cdot\|$ is uniformly continuous. Stoer and Bulirsch [1980] □

This result can be used for the following

Theorem 8 (Norm Equivalence Theorem)**.** *Let* $\|\cdot\|$ *and* $\|\cdot\|'$ *be any two norms on* $\mathbb{C}^n$*. Then there are constants* $c_2 \geq c_1 > 0$ *such that*

$$c_1\|\mathbf{x}\| \leq \|\mathbf{x}\|' \leq c_2\|\mathbf{x}\|, \quad \forall \mathbf{x} \in \mathbb{C}^n.$$

Proof. It suffices to prove this only for the special case that $\|\mathbf{x}\| = \|\mathbf{x}\|_\infty = \max_i |x_i|$, as the general result will then follow from this one. We consider the compact set

$$S = \left\{\mathbf{x} \in \mathbb{C}^n |\, \max_i |x_i| = 1\right\}.$$

From Theorem 7 we know that $\|\mathbf{x}\|'$ is continuous, which means that it will attain its minimum and maximum on S:

$$c_1 = \min_{\mathbf{x}\in S} \|\mathbf{x}\|', \qquad c_2 = \max_{\mathbf{x}\in S} \|\mathbf{x}\|'.$$

Therefore, it follows for all $\mathbf{y} \neq \mathbf{0}$, since $\frac{\mathbf{y}}{\|\mathbf{y}\|} \in S$, that

$$c_1 \leq \left\|\frac{\mathbf{y}}{\|\mathbf{y}\|}\right\|' = \frac{1}{\|\mathbf{y}\|}\|\mathbf{y}\|' \leq c_2,$$

and moreover that $c_1\|\mathbf{x}\| \leq \|\mathbf{x}\|' \leq c_2\|\mathbf{x}\|$. Stoer and Bulirsch [1980] □

A sketch portraying a specific case described in Theorem 8 is shown in Figure A.2.

Definition 5. *A sequence* $\left\{\mathbf{x}^{(k)}\right\}_{k=1}^{\infty}$ *of vectors is said to* converge *to* $\mathbf{x}$ *with respect to norm* $\|\cdot\|$ *if, given any* $\epsilon > 0$*, there exists an integer* $N(\epsilon)$ *such that*

$$\|\mathbf{x}^{(k)} - \mathbf{x}\| < \epsilon, \quad \forall k \geq N(\epsilon).$$ *Burden and Faires* [1989]

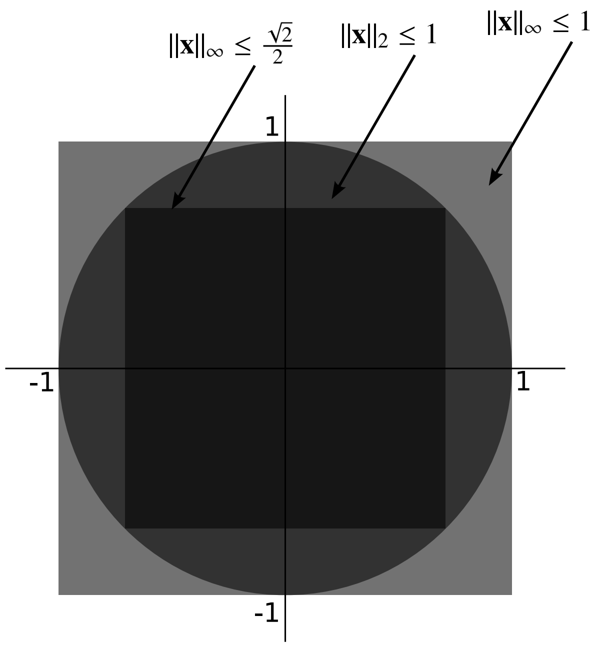

$\|\mathbf{x}\|_\infty \leq \frac{\sqrt{2}}{2}$
$\|\mathbf{x}\|_2 \leq 1$
$\|\mathbf{x}\|_\infty \leq 1$
1
-1
1
-1

Figure A.2: *Geometric presentation of the Norm Equivalence Theorem for the l_∞ and l_2 norms on $\mathbb{R}^2$: $\frac{\sqrt{2}}{2}\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_\infty$*

Theorem 9. *The sequence of vectors $\left\{\mathbf{x}^{(k)}\right\}_{k=1}^{\infty}$ converges to $\mathbf{x}$ with respect to $\|.\|_\infty$ if and only if $\lim_{k\to\infty} x_i^{(k)} = x_i$ for each $i = 1, 2, \ldots, n$.*

Proof. Assuming that $\left\{\mathbf{x}^{(k)}\right\}$ converges to $\mathbf{x}$ with respect to the l_∞ norm, that is for any arbitrarily small $\epsilon > 0$ we can find a $k \geq N(\epsilon)$ such that

$$\|\mathbf{x}^{(k)} - \mathbf{x}\|_\infty = \max_i |x_i^{(k)} - x_i| < \epsilon,$$

which implies $\lim_{k\to\infty} x_i^{(k)} = x_i$ for each $i = 1, 2, \ldots, n$.
Conversely, if $\lim_{k\to\infty} x_i^{(k)} = x_i$ is true for every $1 \leq i \leq n$, then, given any $\epsilon > 0$, there exist integers $N_i(\epsilon)$ such that

$$|x_i^{(k_i)} - x_i| < \epsilon, \quad \forall k_i \geq N_i(\epsilon).$$

Defining $N(\epsilon) := \max_i N_i(\epsilon)$ leads to

$$\|\mathbf{x}^{(k)} - \mathbf{x}\|_\infty < \epsilon, \quad \forall k \geq N(\epsilon).$$

□

But from the Norm Equivalence Theorem 8 we know that if the vector sequence $\left\{\mathbf{x}^{(k)}\right\}_{k=1}^{\infty}$ converges in one norm (like the l_∞ norm as in the Theorem above) then it also converges in *any other* vector norm.

A.2 Matrix Norms

Analogous to vector norms it is possible to define matrix norms to express a "distance" between matrices. Unless where indicated otherwise, this subsection is based on Stoer and Bulirsch [1980].

Since we can interpret a matrix $A \in \mathbb{C}^{n\times m}$ as a vector in a nm-dimensional space, analogously to vector norms we can introduce

Definition 6 (Matrix norm)**.** *A matrix norm is a real-valued function $\|\cdot\| : \mathbb{C}^{n\times m} \to \mathbb{R}$ with the following properties for all $A, B \in \mathbb{C}^{n\times m}$ and all $\alpha \in \mathbb{C}$:*

(i) $\|A\| > 0, \ \forall A \neq 0 \quad (\|A\| = 0 \Leftrightarrow A = 0)$*;*
(ii) $\|\alpha A\| = |\alpha| \, \|A\|$*;*
(iii) $\|A + B\| \leq \|A\| + \|B\|$.

However, one may want to be even more restrictive regarding the properties of matrix norms. For example, one may desire that a matrix norm $\|\cdot\|$ is *consistent* with the vector norms $\|\cdot\|_a$ on $\mathbb{C}^n$ and $\|\cdot\|_b$ on $\mathbb{C}^m$,

$$\|A\mathbf{x}\|_a \leq \|A\| \, \|\mathbf{x}\|_b, \quad \mathbf{x} \in \mathbb{C}^m, A \in \mathbb{C}^{n\times m},$$

or, that a matrix norm $\|\cdot\|$ is *submultiplicative*,

$$\|AB\| \leq \|A\| \, \|B\|, \quad \forall A, B \in \mathbb{C}^{n\times n}.$$

Examples of matrix norms are

$$\|A\| = \max_i \sum_{k=1}^n |a_{ik}| \qquad \text{(row-sum norm);} \qquad \text{(A.2a)}$$

$$\|A\| = \left(\sum_{i,k=1}^n |a_{ik}|^2 \right)^{1/2} \qquad \text{(Schur norm);} \qquad \text{(A.2b)}$$

$$\|A\| = \max_{i,k} |a_{ik}|, \qquad \text{(A.2c)}$$

where (a) and (b) are submultiplicative (in contrast to (c)), and (b) is consistent with the l_2 vector norm.

Furthermore, a special class of matrix norms for square matrices is the *subordinate matrix norm* (also called *induced matrix norm*) for any given vector norm $\|\mathbf{x}\|$:

$$\operatorname{lub}(A) := \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} = \max_{\|\mathbf{x}\|=1} \|A\mathbf{x}\|,$$

see Figure A.3. It is easy to see that this matrix norm is consistent with the vector matrix norm $\|\cdot\|$ that defined it:

$$\|A\mathbf{x}\| \leq \operatorname{lub}(A) \, \|\mathbf{x}\|. \qquad \text{(A.3)}$$

And even more so, $\operatorname{lub}(A)$ is the smallest of all norms $\|A\|$ that are consistent with the vector norm $\|\mathbf{x}\|$:

$$\|A\mathbf{x}\| \leq \|A\| \, \|\mathbf{x}\|, \ \forall \mathbf{x} \in \mathbb{C}^n \quad \Rightarrow \quad \operatorname{lub}(A) \leq \|A\|.$$

Equation (A.3) expresses that $\operatorname{lub}(A)$ is the maximum magnification that a vector $\mathbf{x}$ can attain under the mapping $A\mathbf{x}$, as it is the factor by which the source point $\|\mathbf{x}\|$ is magnified in the image point $\|A\mathbf{x}\|$.
Moreover, one can show that each subordinate matrix norm $\operatorname{lub}(\cdot)$ is submultiplicative:

$$\begin{aligned}\operatorname{lub}(AB) &= \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|AB\mathbf{x}\|}{\|\mathbf{x}\|} = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|A(B\mathbf{x})\|}{|B\mathbf{x}|} \frac{\|B\mathbf{x}\|}{\|\mathbf{x}\|} \\ &\leq \max_{\mathbf{y}\neq\mathbf{0}} \frac{\|A\mathbf{y}\|}{\|\mathbf{y}\|} \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|B\mathbf{x}\|}{\|\mathbf{x}\|} = \operatorname{lub}(A) \operatorname{lub}(B).\end{aligned}$$

Also note that

$$\operatorname{lub}(I) = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|I\mathbf{x}\|}{\|\mathbf{x}\|} = 1.$$

From now on we will only consider subordinate matrix norms and will write $\|A\|$ instead of $\operatorname{lub}(A)$.

Until indicated otherwise, the following is based on Ortega [1972]. In general it is quite difficult to compute $\|A\|$ explicitly for an arbitrary norm (as we shall see that is even true for the l_2 norm), however

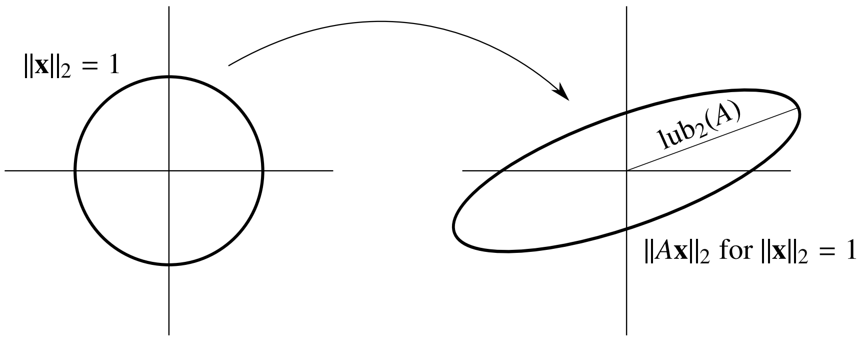


Figure A.3: *Geometric representation of* $lub(A) = \max_{\|\mathbf{x}\|=1} \|A\mathbf{x}\|$ *(for the* l_2 *norm).*

Theorem 10. *Let* $A \in \mathbb{C}^{n\times n}$. *Then*

$$\|A\|_1 \equiv \max_{\|\mathbf{x}\|_1=1} \|A\mathbf{x}\|_1 = \max_{1\le j\le n} \sum_{i=1}^{n} |a_{ij}| \tag{A.4}$$

and

$$\|A\|_\infty \equiv \max_{\|\mathbf{x}\|_\infty=1} \|A\mathbf{x}\|_\infty = \max_{1\le i\le n} \sum_{j=1}^{n} |a_{ij}| \tag{A.5}$$

Proof. First, consider the l_1 norm: For any $\mathbf{x} \in \mathbb{C}^n$ we have

$$\|A\mathbf{x}\|_1 = \sum_{i=1}^{n} \left| \sum_{j=1}^{n} a_{ij}x_j \right| \le \sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|\,|x_j| = \sum_{i=1}^{n} |a_{ij}| \sum_{j=1}^{n} |x_j| \qquad \le \max_j \sum_{i=1}^{n} |a_{ij}|\,\|\mathbf{x}\|_1.$$

Now, to show that there exists some $\mathbf{x}$ with $\|\mathbf{x}\|_1 = 1$ for which Equation (A.4) attains equality, define k such that

$$\sum_{i=1}^{n} |a_{ik}| = \max_j \sum_{i=1}^{n} |a_{ij}|.$$

Then with the kth coordintae vector, $\mathbf{e}_k$, and the kth column vector of A, $\mathbf{a}_k$, we see that

$$\|A\mathbf{e}_k\|_1 = \|\mathbf{a}_k\|_1 = \sum_{i=1}^{n} |a_{ik}|.$$

The proof for $\|A\|_\infty$ is analogous, using the maximum on the vector defined by

$$x_i = \begin{cases} a_{ki}/|a_{ki}|, & a_{ki} \ne 0 \\ 1, & a_{ki} = 1 \end{cases}$$

where k is defined is chosen such that the maximum in Equation (A.5) is attained. □

Calculating the l_2 norm of a matrix is unfortunately mentionably more cumbersome:

Theorem 11. *Let* $A \in \mathbb{C}^{n\times n}$. *Then*

$$\|A\|_2 \equiv \max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2 = \left[\rho(A^H A)\right]^{1/2}. \tag{A.6}$$

In order to prove Theorem 11 we will first introduce

Lemma 2. *Let $B \in \mathbb{C}^{n\times n}$ be hermitian with eigenvalues $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. Then*

$$\lambda_1 \mathbf{x}^H\mathbf{x} \leq x^H B\mathbf{x} \leq \lambda_n \mathbf{x}^H\mathbf{x}, \quad \forall \mathbf{x} \in \mathbb{C}^n.$$

Proof. Since B is hermitian it is normal and there exist a unitary matrix P such that

$$P^H BP = diag(\lambda_1, \lambda_2, \ldots, \lambda_n).$$

For this reason, with $\mathbf{y} = P^H\mathbf{x}$, we have

$$\mathbf{x}^H B\mathbf{x} = \mathbf{y}^H P^H BP\mathbf{y} = \sum_{i=1}^{n} \lambda_i y_i^2 \leq \lambda_n \mathbf{y}^H\mathbf{y} = \lambda_n \mathbf{x}^H\mathbf{x}.$$

The other inequality is proved analogously. □

Proof of Theorem 11. Set $\mu = \left[\rho(A^H A)\right]^{1/2}$, then for any $\mathbf{x} \in \mathbb{C}^n$ the previous Lemma yields

$$\|A\mathbf{x}\|_2^2 = \mathbf{x}^H A^H A\mathbf{x} \leq \mu^2 \mathbf{x}^H\mathbf{x},$$

which implies that

$$\|A\|_2 = \max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2 \leq \mu.$$

However, if $\mathbf{u}$ is an eigenvector of $A^H A$ corresponding the μ^2, then

$$\mathbf{u}^H A^H A\mathbf{u} = \mu^2 \mathbf{u}^H\mathbf{u},$$

which shows that equality holds in Equation (A.6). □

For a real symmetric $A \in \mathbb{R}^{n\times n}$, Theorem 11 obviously reduces to

$$\|A\|_2 = \rho(A).$$

However, this can be considered a special case indeed, as generally we know that

$$\|A\| \geq \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} = |\lambda|$$

which then specifically implies that furthermore

$$\|A\| \geq \rho(A).$$

As a matter of fact we have

Theorem 12. *Let $A \in \mathbb{C}^{n\times n}$. Then given any $\epsilon > 0$ there is a matrix norm $\|\cdot\| : \mathbb{C}^{n\times m} \to \mathbb{R}$ such that*

$$\|A\| \leq \rho(A) + \epsilon.$$

Proof. Let $A = PJP^{-1}$, J being the Jordan form of A, and let

$$D = \text{diag}(1, \epsilon, \epsilon^2, \ldots, \epsilon^{n-1}).$$

Then note that $\hat{J} = D^{-1}JD$ is J with every off-diagonal 1 in J substituted with ϵ. Therefore

$$\|\hat{J}\|_\infty \leq \rho(A) + \epsilon.$$

Setting $Q = PD$, we can define $\|\mathbf{x}\| = \|Q^{-1}\mathbf{x}\|_{infty}$ which is a vector norm as discussed in Section A.1. It then follows that

$$\begin{aligned}||A|| &= \max_{||\mathbf{x}||=1} ||A\mathbf{x}|| = \max_{||Q^{-1}\mathbf{x}||_\infty=1} ||Q^{-1}A\mathbf{x}||_\infty = \max_{||\mathbf{y}||_\infty=1} ||Q^{-1}AQ\mathbf{y}||_\infty \\ &= \max_{||\mathbf{y}||_\infty=1} ||\hat{J}\mathbf{y}||_\infty = ||\hat{J}||_\infty \\ &\leq \rho(A) + \epsilon.\end{aligned}$$

□

Nonetheless, there exists a special class of matrices for which we can obtain $||A|| = \rho(A)$, namely:

Definition 7 (Matrices of class M)**.** *A matrix $A \in \mathbb{C}^{n\times n}$ is of* class M *if and only if A is similar to a matrix of the form*

$$\begin{bmatrix} D & 0 \\ 0 & B \end{bmatrix}$$

where D is diagonal with $\rho(D) = \rho(A)$, and $\rho(B) < \rho(A)$.

This means, however, that for every eigenvalue λ with $|\lambda| = \rho(A)$ the associated Jordan block is 1×1, which is an equivalent form of defining matrices of class M.

We can now state

Theorem 13. *Let $A \in \mathbb{C}^{n\times n}$. Then there is a matrix norm $||\cdot|| : \mathbb{C}^{n\times m} \to \mathbb{R}$ such that $||A|| = \rho(A)$ holds if and only if A is of class M.*

Proof. Assume there there is an $\epsilon > 0$ so small that $|\lambda| + \epsilon < \rho(A)$, where λ denotes any eigenvalue of A that satisfies $|\lambda| < \rho(A)$. Then following the nomenclature from the proof of Theorem 12, certainly $||\hat{J}||_\infty = \rho(A)$ holds.

Conversely, assuming that we have $||A|| = \rho(A)$ for some norm, and that there is a an $m\times m$ Jordan block ($m \geq 2$) associated with an eigenvalue λ which satisfies $|\lambda| = \rho(A)$, which means that A is not of class M. We only consider $\lambda \neq 0$, as otherwise we get $||A|| = 0$ and thus $A = \mathbf{0}$. Therefore, considering the Jordan block

$$J = \begin{bmatrix} \lambda & 1 & & \\ & \ddots & \ddots & \\ & & \ddots & 1 \\ & & & \lambda \end{bmatrix}, \quad \lambda \neq 0$$

we need to show that $||J|| = |\lambda|$ is not possible in any norm. If we assume that $||J|| = |\lambda|$ and set $\hat{J} = \lambda^{-1}J$ then we would obviously have $||\hat{J}|| = 1$. However, a direct computation shows that $\hat{J}^k\mathbf{e}_2 = (k/\lambda, 1, 0, \dots, 0)^T$ (where $\mathbf{e}_2$ denotes the second standard unit vector $(0, 1, 0, \dots, 0)^T$), hence $\lim_{k\to\infty} ||\hat{J}^k\mathbf{e}_2|| = \infty$, which contradicts the assumption $||\hat{J}^k|| = 1$. □

This theorem shows that real symmetric matrices are of class M, which will be even more obvious in Section A.4.

As for the question when a matrix A converges, meaning when it is true that $\lim_{k=\infty} A^k = 0$, where 0 is the zero matrix with only zero matrix elements we state

Theorem 14. *Let $A \in \mathbb{C}^{n\times n}$. Then $\lim_{k=\infty} A^k = 0$ if and only if $\rho(A) < 1$. Moreover, $||A^k||$ is bounded as $k \to \infty$ if and only if $\rho(A) < 1$, or $\rho(A) = 1$ and A is of class M.*

Proof. If $\rho(A) < 1$, then by Theorem 12 we can choose a norm such that

$$||A|| < 1.$$

Thus

$$||A^k||_2 = ||A||_2^k \to 0 \quad \text{as } k \to \infty.$$

Conversely, suppose that $\rho(A) \geq 1$ and let some eigenvalue λ be such that $|\lambda| \geq 1$. Let $\mathbf{x}$ be the corresponding eigenvector, then

$$\|A^k\mathbf{x}\| = \|\lambda^k\mathbf{x}\| \geq \|\mathbf{x}\|,$$

which implies that $\|A^k\| \geq 1$ for all k. □

As stated in the beginning of this subsection, an $n\times m$ matrix can be considered a nm-dimensional vector and we can give analogous results:

Theorem 15. *Each matrix norm $\|\cdot\|$ is a uniformly continuous function with respect to the metric*

$$m(A,B) = \max_{i,j} |a_{ij} - b_{ij}|, \quad A, B \in \mathbb{C}^{n\times m}$$

Theorem 16. *Let $\|\cdot\|$ and $\|\cdot\|'$ be any two matrix norms on $\mathbb{C}^{n\times m}$. Then there are constants $c_2 \geq c_1 \geq 0$ such that*

$$c_1\|A\| \leq \|A\|' \leq c_2\|A\|, \quad \forall A \in \mathbb{C}^{n\times m}.$$

Finally, connected to matrix norms is the notion of the *condition number* or *condition of A*: $\kappa(A) = \|A\|\,\|A^{-1}\|$. The condition number is "a measure of the sensitivity of the relative error in the solution to changes in the right-hand side" of a linear system $A\mathbf{x} = \mathbf{b}$. Stoer and Bulirsch [1980] Moreover, it can be used for bounding the relative error in the solution to changes in the matrix A itself. The remainder of this Section is based on Stoer and Bulirsch [1980].

Let $\mathbf{x}$ be the solution of

$$A\mathbf{x} = \mathbf{b}$$

and $\mathbf{x} + \Delta\mathbf{x}$ be the solution of

$$A(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b} + \Delta\mathbf{b}.$$

Subtraction and solving for $\Delta\mathbf{x}$ then yields

$$\Delta\mathbf{x} = A^{-1}\Delta\mathbf{b}$$

and

$$\|\Delta\mathbf{x}\| \leq \|A^{-1}\|\,\|\Delta\mathbf{b}\|.$$

Now, it follows from $\|\mathbf{b}\| = \|A\mathbf{x}\| \leq \|A\|\,\|\mathbf{x}\|$ that

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A\|\,\|A^{-1}\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} = \kappa(A)\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|},$$

which shows that indeed $\kappa(A)$ is a measure of the influence on the relative error of the solution to relative errors in the right-hand side of a linear system.

Theorem 17. *Let $A \in \mathbb{C}^{n\times n}$ be nonsingular, $B = A(I + F)$, $\|F\| < 1$, and $\mathbf{x}$ and $\Delta\mathbf{x}$ defined by $A\mathbf{x} = \mathbf{b}$ and $B(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b}$. Then*

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\|F\|}{1 - \|F\|}$$

and

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\kappa(A)}{1 - \kappa(A)\frac{\|B-A\|}{\|A\|}}\,\frac{\|B - A\|}{\|A\|}.$$

In order to prove this Theorem let us first introduce

Lemma 3. *If $F \in \mathbb{C}^{n\times n}$ is such that $\|F\| < 1$ then $(I+F)^{-1}$ exists and respects*

$$\|(I+F)^{-1}\| \le \frac{1}{1-\|F\|}.$$

Proof. As shown in the previously,

$$\|\mathbf{x}-\mathbf{y}\| \ge |\,\|\mathbf{x}\| - \|\mathbf{y}\|\,|$$

hold for all $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$ and each vector norm. Hence it follows for all $\mathbf{x}$ that

$$\|(I+F)\mathbf{x}\| = \|\mathbf{x} + F\mathbf{x}\| \ge \|\mathbf{x}\| - \|F\mathbf{x}\| \ge (1-\|F\|)\|\mathbf{x}\|.$$

From $1-\|F\| > 0$ it is obvious that $\|(I+F)\mathbf{x}\| >)\|$ if $\mathbf{x} \neq \mathbf{0}$. This implies, however, that $(I+F)\mathbf{x} = \mathbf{0}$ has only the trivial solution $\mathbf{x}$ and that $(I+F)$ is nonsingular.

For the second part we introduce the abbreviation $C := (I+F)^{-1}$. Note that

$$\begin{aligned} 1 = \|I\| &= \|(I+F)C\| = \|C + FC\| \\ &\ge \|C\| - \|C\|\,\|F\| \\ &= \|C\|(1-\|F\|) > 0, \end{aligned}$$

from which follows the desired inequality. □

Proof of Theorem 17. From Lemma 3 it follows that B^{-1} exists, and we can write

$$\Delta\mathbf{x} = B^{-1}\mathbf{b} - A^{-1}\mathbf{b} = B^{-1}(A-B)A^{-1}\mathbf{b},$$

since $\mathbf{x} = A^{-1}\mathbf{b}$. Therefore,

$$\begin{aligned} \frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} &\le \|B^{-1}(A-B)\| = \|-(I+F)^{-1}A^{-1}AF\| \\ &\le \|(I+F)^{-1}\|\,\|F\| \\ &\le \frac{\|F\|}{1-\|F\|}. \end{aligned}$$

To conclude the proof, note that $F + A^{-1}(B-A)$ and $\|F\| \le \kappa(A)\,\|B-A\|/\|A\|$. □

Note that for any matrix A and any induced matrix norm

$$\kappa(A) = \|A\|\,\|A^{-1}\| \ge \|AA^{-1}\| = \|I\| = 1$$

is valid.

A.3 Similarity and Diagonalizability of Matrices

This section is based on Noble and Daniel [1998] and introduces the concepts of similarity and diagonalization of matrices.

Definition 8. *If there exists a nonsingular matrix P such that $P^{-1}AP = B$ then B is said to be* similar *to A and to be obtained from A by means of a* similarity transformation.

Theorem 18.
(i) Similar matrices have the same characteristic polynomial and the same eigenvalues;
(ii) Suppose that B is similar to A with $B = P^{-1}AP$. Then $\mathbf{x}$ is an eigenvector of A associated with the eigenvalue λ if and only if $P^{-1}\mathbf{x}$ is an eigenvector of B associated with the eigenvalue λ.

Proof.

(i) The characteristic polynomials are identical because

$$\begin{aligned}\det(B-\lambda I) &= \det[P^{-1}(A-\lambda I)P]\\ &= \det(P^{-1})\det(A-\lambda I)\det(P)\\ &= \det(P^{-1}P)\det(A-\lambda I)\\ &= \det(A-\lambda I).\end{aligned}$$

Furthermore the eigenvalues are identical since they are merely the roots of the characteristic polynomial.

(ii) From the definition of eigenvectors and eigenvalues we know that $\mathbf{x}$ is an eigenvector of A corresponding to the eigenvalue λ if and only if

$$A\mathbf{x} = \lambda\mathbf{x},$$

but since B is similar to A we have

$$(PBP^{-1})\mathbf{x} = \lambda\mathbf{x}$$

or equivalently

$$B(P^{-1}\mathbf{x}) = \lambda(P^{-1}\mathbf{x}).$$

□

Theorem 19. *Suppose that B is similar to A with $B = P^{-1}AP$. Then:*

(i) For each positive integer k, B^k is similar to A^k with $B^k = P^{-1}A^kP$;

(ii) $\det(B) = \det(A)$*;*

(iii) B is nonsingular if and only if A is nonsingular;

(iv) If A and B are nonsingular, then B^k is similar to A^k with $B^k = P^{-1}A^kP$ for negative integers k as well, so that in particular $B^{-1} = P^{-1}A^{-1}P$;

(v) If f is a polynomial with $f(x) = a_mx^m + \cdots + a_1x + a_0$ and if $f(X)$ for a square matrix X denotes $f(X) = a_mX^m + \cdots + a_1X + a_0I$, then $f(B)$ is similar to $f(A)$ with $f(B) = P^{-1}f(A)P$.

Proof.

(i) $B^k = \prod_{i=1}^{k}(P^{-1}AP) = (P^{-1}AP)(P^{-1}AP)\cdots(P^{-1}AP)$ but due to the associativity of matrix multiplication the parenthesis can be removed and the product collapses to $P^{-1}A^kP$ because PP^{-1} can be repeatedly evaluated to I.

(ii) $\det B = \det(P^{-1}AP) = \det(P^{-1})\det(A)\det(P) = \det(P^{-1}P)\det(A) = \det(A)$.

(iii) is a consequence of (ii) and the fact that a matrix is nonsingular if and only if its determinant in nonzero.

(iv) $B^{-1} = (P^{-1}AP)^{-1} = P^{-1}A^{-1}(P^{-1})^{-1} = P^{-1}A^{-1}P$. For general negative k, the result is shown by applying (i) to B^{-1} and A^{-1} with $|k|$.

(v) Simple calculation verifies that

$$\begin{aligned}f(B) &= a_mP^{-1}A^mP + \cdots + a_1P^{-1}A^1P + a_0P^{-1}P\\ &= P^{-1}(a_mA^m + \cdots + a_1A + a_0I)P\\ &= P^{-1}f(A)P.\end{aligned}$$

□

We can use the concept of similarity to introduce a special class of matrices:

Definition 9. *The matrix A is* diagonalizable *if it is similar to diagonal matrix, which means that there is an invertible matrix P and a diagonal matrix D such that $P^{-1}AP = D$. In this case we say that P is a* diogonalizing matrix *for A or that P* diagonalizes *A.*

Theorem 20. *$A \in \mathbb{C}^{n\times n}$ has a linearly independent set of n eigenvectors if and only if there exists a non-singular matrix P and a diagonal matrix Λ for which $A = P\Lambda P^{-1}$ (and equivalently $\Lambda = P^{-1}AP$. These compositions hold if and only if the columns $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ of $P = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]$ form a linear independent set of eigenvectors associated with the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ which are the diagonal entries of $\Lambda = diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$.*

Proof. Assume that $P = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$ is formed from a linearly independent set of eigenvectors and $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ is formed from the associated eigenvalues. Then from $A\mathbf{x}_i = \lambda_i\mathbf{x}_i$ we have $AP = P\Lambda$. But since P has full rank it is invertible and we have $A = P\Lambda P^{-1}$ or equivalently $\Lambda = P^{-1}AP$.
Conversely, if $A = P\Lambda P^{-1}$ with Λ being a diagonal matrix. Applying the rules of partitioned-matrix multiplication result in $A\mathbf{x}_i = \lambda_i\mathbf{x}_i$, $\mathbf{x}_i$ being the columns of P and λ_i the diagonal element of Λ (taken from the some column number). Since P is invertible, the $\mathbf{x}_i$ form a linearly independent set (and are nonzero), hence they form a linearly independent set of n eigenvectors. □

A.4 Unitary and Normal Matrices

As in the previous section, this section is based on Noble and Daniel [1998]. Here we introduce the concepts of some special classes of matrices and their properties which will be used in the discussion of Chapter 3.

Definition 10. *A matrix $P \in \mathbb{C}^{n\times n}$ for which $P^{-1} = P^H$, so that $PP^H = P^HP = I$, is said to be* unitary. *An* orthogonal *matrix is a real unitary matrix $P \in \mathbb{R}^{n\times n}$, so that $P^{-1} = P^T$ and $PP^T = P^TP = I$.*

Note that a unitary (orthogonal) matrix is a matrix whose columns form an *orthonormal* set of vectors with the standard inner product:

$$\mathbf{x}_i^H\mathbf{x}_i = 1 \;\; (\mathbf{x}_i^T\mathbf{x}_i = 1), \;\; \forall i$$

and

$$\mathbf{x}_i^H\mathbf{x}_j = 0 \;\; (\mathbf{x}_i^T\mathbf{x}_j = 0), \;\; \forall i \neq j.$$

A matrix A is unitarily (orthogonally) similar to B if there exists a unitary (orthogonal) matrix P such that

$$A = P^HBP \quad (A = P^TBP).$$

In the following only (complex) unitary matrices are considered and orthogonal matrices are to be taken as the real case of unitary matrices.

Theorem 21.

(i) $P \in \mathbb{C}^{n\times n}$ is unitary if and only if its columns form an orthonormal set;
(ii) $P \in \mathbb{C}^{n\times n}$ is unitary if and only if its rows form an orthonormal set;
(iii) If P is unitary, then $|\det(P)| = 1$;
(iv) If P and Q are both unitary, then so is PQ;
(v) If P is unitary and $\langle\cdot,\cdot\rangle$ is the standard inner product, then
  1. *$\langle P\mathbf{x}, P\mathbf{y}\rangle = \langle\mathbf{x}, \mathbf{y}\rangle$, $\forall \mathbf{x}, \mathbf{y}$ (the angle between $\mathbf{x}$ and $\mathbf{y}$ is preserved by P);*
  2. *$\|P\mathbf{x}\|_2 = \|\mathbf{x}\|_2$, $\forall \mathbf{x}$ (the length of $\mathbf{x}$ is preserved by P);*
  3. *$\|P\|_2 = 1$;*

(vi) If λ is an eigenvalue of the unitary matrix P, then $|\lambda| = 1$;
(vii) If $P \in$ is $n \times n$ and unitary while A is $n \times m$ and B is $m \times n$, then

$$\|PA\|_2 = \|A\|_2 \quad and \quad \|BP\|_2 = \|B\|_2.$$

Proof.

(i) First, suppose that $\mathbf{x}_1, \dots, \mathbf{x}_n$ is a orthonormal set on $\mathbb{C}^n$ with the standard inner product, that is

$$\begin{cases} \mathbf{x}_i^H \mathbf{x}_i = 1 & \forall i, \\ \mathbf{x}_i^H \mathbf{x}_j = 0 & \forall i \neq j. \end{cases}$$

Then obviously for the matrix $P = [\mathbf{x}_1, \dots, \mathbf{x}_n]$ we have

$$P^H P = I$$
$$\Leftrightarrow P^H = P^{-1}.$$

Conversely, assume that P is unitary, then by definition $P^H = P^{-1}$. However this implies that $P^H P = I$ and the columns then form an orthonormal set.

(ii) This follows by applying (i) to P^H.

(iii) Follows from $1 = \det(I) = \det(P^H P) = \det(P^H)\det(P) = |\det(P)|^2$.

(iv) $(PQ)^H(PQ) = Q^H P^H P Q = Q^H I Q = I$.

(v)

1. $\langle P\mathbf{x}, P\mathbf{y}\rangle = (P\mathbf{x})^H(P\mathbf{y}) = \mathbf{x}^H P^H P \mathbf{y} = \mathbf{x}^H \mathbf{y} = \langle \mathbf{x}, \mathbf{y}\rangle$.
2. This is a direct result of (v)1. for $\mathbf{x} = \mathbf{y}$.
3. Applying the definition of matrix norms and unitary matrices yields:

$$\begin{aligned} \|P\|_2 &= \max_{\|\mathbf{x}\|_2=1} \|P\mathbf{x}\|_2 = \max_{\|\mathbf{x}\|_2=1} \left[(P\mathbf{x})^H(P\mathbf{x})\right]^{1/2} \\ &= \max_{\|\mathbf{x}\|_2=1} \left[\mathbf{x}^H P^H P \mathbf{x}\right]^{1/2} = \max_{\|\mathbf{x}\|_2=1} \left[\mathbf{x}^H \mathbf{x}\right]^{1/2} \qquad = \max_{\|\mathbf{x}\|_2=1} \|\mathbf{x}\|_2 \\ &= 1. \end{aligned}$$

(vi) If $P\mathbf{x} = \lambda\mathbf{x}$ then applying (v)2. we have

$$\|\mathbf{x}\|_2 = \|P\mathbf{x}\|_2 = \|\lambda\mathbf{x}\|_2 = |\lambda|\,\|\mathbf{x}\|,$$

and since $\mathbf{x} \neq \mathbf{0}$ this means that $|\lambda| = 1$.

(vii) We have

$$\|PA\|_2 = \max_{\|\mathbf{x}\|_2=1} \|PA\mathbf{x}\|_2 \overset{(v)2.}{=} \max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2 = \|A\|_2$$

and

$$\begin{aligned} \|BP\|_2 &= \max_{\|\mathbf{x}\|_2\neq 0} \frac{\|BP\mathbf{x}\|_2}{\|\mathbf{x}\|_2} \overset{(v)2.}{=} \max_{\|\mathbf{x}\|_2\neq 0} \frac{\|BP\mathbf{x}\|_2}{\|P\mathbf{x}\|_2} \\ &= \max_{\|\mathbf{y}\|_2\neq 0} \frac{\|B\mathbf{y}\|_2}{\|\mathbf{y}\|_2} = \|B\|_2. \end{aligned}$$

□

From Theorem 21 (v) 3. it directly follows that if P is unitary then $\kappa_2(P) = 1$, since if P is unitary then certainly so is P^H.

Definition 11. *A matrix $A \in \mathbb{C}^{n\times n}$ is said to be* normal *if $A^H A = AA^H$.*

From this definition it follows that

- hermitian matrices ($A^H = A$) are normal;
- real symmetric matrices ($A^T = A$) are normal;
- unitary matrices ($A^H A = AA^H = I$) are normal;

- orthogonal matrices ($A^T A = AA^T = I$) are normal.

In Theorem 22 we will introduce the *Schur form* and will use this concept and its attributes later to prove Theorem 23 which states the first special property of normal matrices.

Theorem 22. *Let $A \in \mathbb{C}^{n\times n}$.*

(i) A is unitarily similar to an upper-triangular matrix $T = P^H AP$ with P unitary and with the eigenvalues of A (repeated according to their algebraic multiplicities) on the main diagonal of T. T is called a Schur form *of A and the decomposition $A = PTP^H$ a* Schur decomposition *of A.*

(ii) If A and its eigenvalues are real, then P may be taken real and thus orthogonal.

Proof. We will prove this theorem via induction. Both parts are trivially satisfied for $n = 1$. Thus, assuming that the Theorem is valid for $n = k$ we will prove that it stays valid for $n = k+1$. For this reason we consider the $(n+1)\times(n+1)$ matrix A with eigenvalue λ_1 and associated normalized eigenvector $\mathbf{x}_1$, i.e. $\|\mathbf{x}_1\|_2 = 1$.

Note that if A and λ_1 are real then we can take $\mathbf{x}_1$ to be real as well, since for a general associated complex eigenvector $\hat{\mathbf{x}}_1 = \mathbf{u} + i\mathbf{v}$, with real $\mathbf{u}, \mathbf{v}$ we have

$$A(\mathbf{u} + i\mathbf{v}) = \lambda_1(\mathbf{u} + i\mathbf{v})$$

and can choose $\mathbf{x}_1$ to be either $\mathbf{u}$ or $\mathbf{v}$, for example.

We can extend $\{\mathbf{x}_1\}$ to form a basis for $\mathbb{C}^{k+1}$ (or, if $\mathbf{x}_1$ is real, $\mathbb{R}^{k+1}$) and then use the Gram-Schmidt process to create an orthonormal basis from it. Then there is a set of vectors $\mathbf{y}_1, \dots, \mathbf{y}_k$ such that $\{\mathbf{x}_1, \mathbf{y}_1, \dots \mathbf{y}_k\}$ is orthonormal and the matrix

$$U = [\mathbf{x}_1, \mathbf{y}_1, \dots \mathbf{y}_k] = [\mathbf{x}_1, Y]$$

is unitary (or orthogonal if $\mathbf{x}_1$ is real). We can now compute

$$\begin{aligned} B = U^H AU &= [\mathbf{x}_1, Y]^H A[\mathbf{x}_1, Y] = [\mathbf{x}_1, Y]^H[A\mathbf{x}_1, AY] \\ &= [\mathbf{x}_1, Y]^H[\lambda_1\mathbf{x}_1, AY] \\ &= \begin{bmatrix} \lambda_1 & \mathbf{x}_1^H AY \\ \mathbf{0} & Y^H AY \end{bmatrix} = \begin{bmatrix} \lambda_1 & \mathbf{b}^H \\ \mathbf{0} & C \end{bmatrix}, \end{aligned}$$

since $\|\mathbf{x}_1\|_2 = 1$ and since $Y^H\mathbf{x}_1 = \mathbf{0}$ because U is unitary. The eigenvalues of B and A are identical since A and B are similar. Now, by expanding $\det(B - \lambda I)$ by its first column, it is evident that the characteristic polynomial of B is $(\lambda_1 - \lambda)\det(C - \lambda I)$, this means that the eigenvalues of B besides λ_1 are the eigenvalues of C. But C is $k\times k$ and our inductive hypothesis holds for it and we can find a unitary matrix V such that $V^H CV = \hat{T}$, where $\hat{T}$ is an upper-triangular matrix with the eigenvalues of C (and therefore A) on its main diagonal. Finally, defining P as

$$P = U\begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0} & V \end{bmatrix}, \quad \text{then} \quad P^H AP = \begin{bmatrix} \lambda_1 & \mathbf{b}^H V \\ \mathbf{0} & \hat{T} \end{bmatrix},$$

which has the desired proper form and hence proves that since the inductive hypothesis is also valid for $n = k+1$, it is actually true for all n. □

Theorem 23. *$A \in \mathbb{C}^{n\times n}$ is normal if and only if A is unitarily similar to a diagonal matrix $\Lambda = P^H AP$, where P is unitary and Λ is diagonal with the eigenvalues of A (repeated according to their algebraic multiplicity) as the diagonal elements.*
If A and its eigenvalues are real, then P can be taken real and therefore orthogonal.

Proof. Suppose there exists a unitary matrix P with $P^H AP = \Lambda$ diagonal. A is then normal as diagonal matrices commute:

$$\begin{aligned} A^H A &= (P\lambda P^H)^H(P\lambda P^H) = P\Lambda^H P^H P\Lambda P^H \\ &= P\Lambda^H\Lambda P^H = P\Lambda P^H P\Lambda^H P^H = (P\Lambda P^H)(P\Lambda P)^H \\ &= AA^H. \end{aligned}$$

Conversely, assume that A is normal, then let T be a Schur form with $T = P^H AP$. T is normal since

$$\begin{aligned}T^H T &= (P^H AP)^H (P^H AP) = P^H A^H AP = P^H AA^H P = (P^H AP)(P^H AP)^H \\ &= TT^H.\end{aligned}$$

Now, letting t_{ij} denote $[T]_{ij}$, we know that $t_{ij} = 0$ for $i > j$. Since $TT^H = T^H T$ their diagonal entries must be equal. We have

$$[TT^H]_{ii} = |t_{ii}|^2 + |t_{i,i+1}|^2 + \cdots + |t_{in}|^2$$

as well as

$$[T^H T]_{ii} = |t_{1i}|^2 + |t_{2i}|^2 + \cdots + |t_{ii}|^2.$$

Equating both expressions while subtracting the common term $|t_{ii}|^2$ yields

$$|t_{i,i+1}|^2 + \cdots + |t_{in}|^2 = |t_{1i}|^2 + \cdots + |t_{i-1,i}|^2$$

for all i. Then, for $i = 1$, we find that the right-hand side is zero (since there are no terms) which results for the left-hand side in $t_{12} = t_{13} = \cdots = t_{1n} = 0$. Next, for $i = 2$, we find on the right-hand side t_{12} which we know is zero. This leaves for the left-hand side $t_{23} = t_{24=\cdots=t_{2n}=0}$. Continuing in this manner reveals that indeed T is diagonal, and we know from Theorem 22 that the diagonal of T is formed by the eigenvalues of A. Therefore we can write $T = \Lambda$. The second part of this Theorem regarding the reality of P follows from Theorem 22 (ii). □

Theorem 24. *$A \in \mathbb{C}^{n\times n}$ is normal if and only if A has a linearly independent set of n eigenvectors that may be chosen in so as to form an orthonormal set.*

Proof. This follows immediately from Theorems 20 and 23. □

Corollary 2. *The eigenvalues of an $n \times n$ hermitian matrix A are real, and the associated eigenvectors may be chosen so as to form an orthonormal set of n vectors.*

Proof. For the first part note that we can write $A = P\Lambda P^H$ with $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$. Since $A^H = A$,

$$\Lambda^H = (P^H AP)^H = P^H A^H P = P^H AP = \Lambda$$

shows that Λ indeed is real.

The second part follows from the normality of A by Theorem 24. □

Appendix B

Profiling of OpenMP Implementations using `perf`

`perf` is a profiling tool for Linux based systems which is used as a command line interface.

"The perf tool supports a list of measurable events. The tool and underlying kernel interface can measure events coming from different sources. For instance, some event are pure kernel counters, in this case they are called software events. Examples include: context-switches, minor-faults.

Another source of events is the processor itself and its Performance Monitoring Unit (PMU). It provides a list of events to measure micro-architectural events such as the number of cycles, instructions retired, L1 cache misses and so on. Those events are called PMU hardware events or hardware events for short. They vary with each processor type and model", https://perf.wiki.kernel.org/index.php/Tutorial.

For our work we measured the following events:

- task clock,
- cycles,
- instructions,
- cache references,
- cache misses,
- L1 data cache loads,
- L1 data cache load misses,
- branches,
- branch misses,

among some other events that seemed to have no relevancy as they were rarely or not detected. Additionally `perf` also measure the the time that elapsed while running the (profiling for the) program.

Based on these events, `perf` automatically determines such values like instructions per cycle, the average CPU frequency, the "number of CPUs utilized", as well as the percentages of the total cache misses, L1 cache misses, and branch misses. The number of CPUs utilized and the averages CPU frequency is calculated as follows

$$\begin{aligned}\texttt{\# CPUs utilized} &= \frac{\texttt{task clock [ms]}\cdot 10^6}{\texttt{elapsed time [s]}},\\ \texttt{Frequency [GHz]} &= \frac{\texttt{cycles}}{\texttt{task clock [ms]}\cdot 10^6}.\end{aligned}$$

Furthermore, we found in https://perf.wiki.kernel.org/index.php/Tutorial that "[i]f there are more events than counters, the kernel uses time multiplexing (switch frequency = HZ, generally 100 or 1000) to give

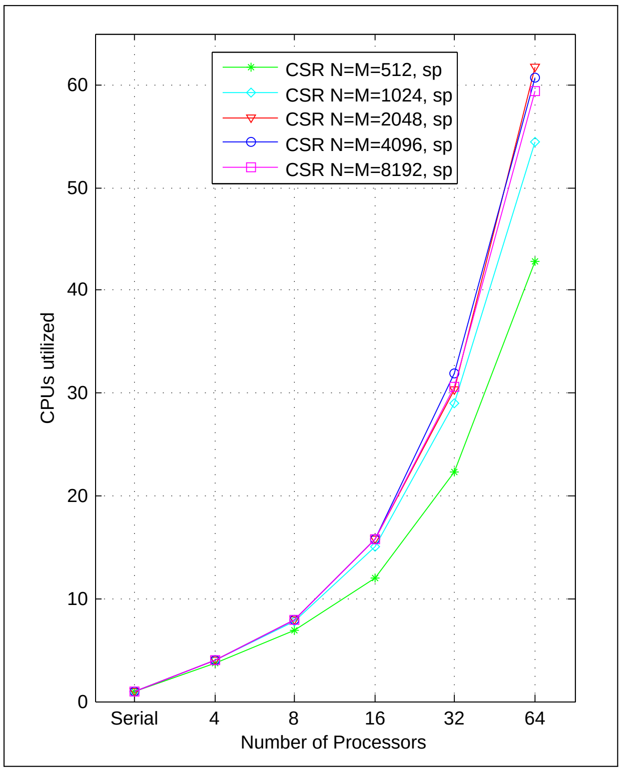


Figure B.1: *"Number of CPUs utilized" for the unmodified single precision Jacobi CSR implementation.*

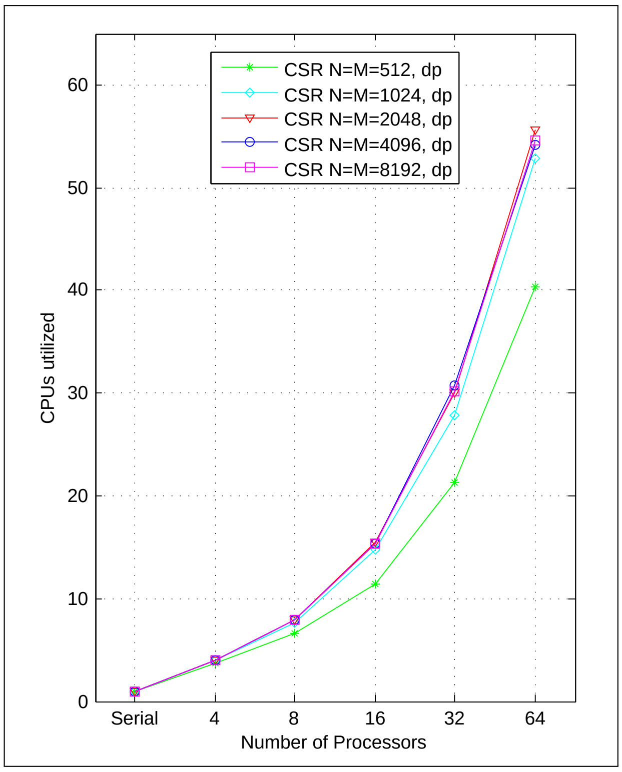


Figure B.2: *"Number of CPUs utilized" for the unmodified double precision Jacobi CSR implementation.*

each event a chance to access the monitoring hardware. Multiplexing only applies to PMU events. With multiplexing, an event is not measured all the time. At the end of the run, the tool scales the count based on total time enabled vs time running. The actual formula is:

```
final_count = raw_count * time_enabled/time_running
```

This provides an estimate of what the count would have been, had the event been measured during the entire run. It is very important to understand this is an estimate not an actual count. Depending on the workload, there will be blind spots which can introduce errors during scaling."

Since we performed the profiling with many events to keep track of, the results presented in the following are scaled from as little as 20% at times. Additionally, those scaling values (and therefore the event values) have been observed to vary a lot in a certain amount of instances. All this makes the `perf` profiling results presented here inconclusive. A way to overcome multiplexing is to perform multiple profilings per program, each measuring different events.

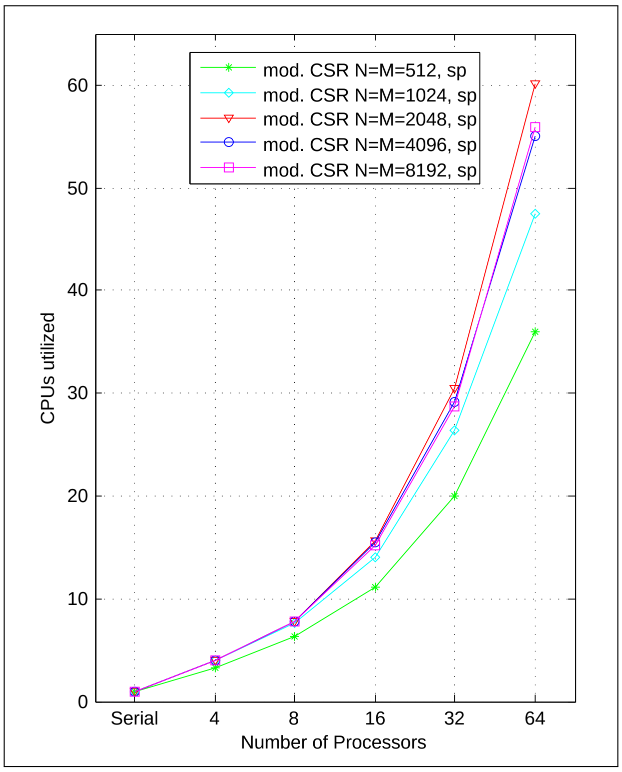


Figure B.3: *"Number of CPUs utilized" for the modified single precision Jacobi CSR implementation.*

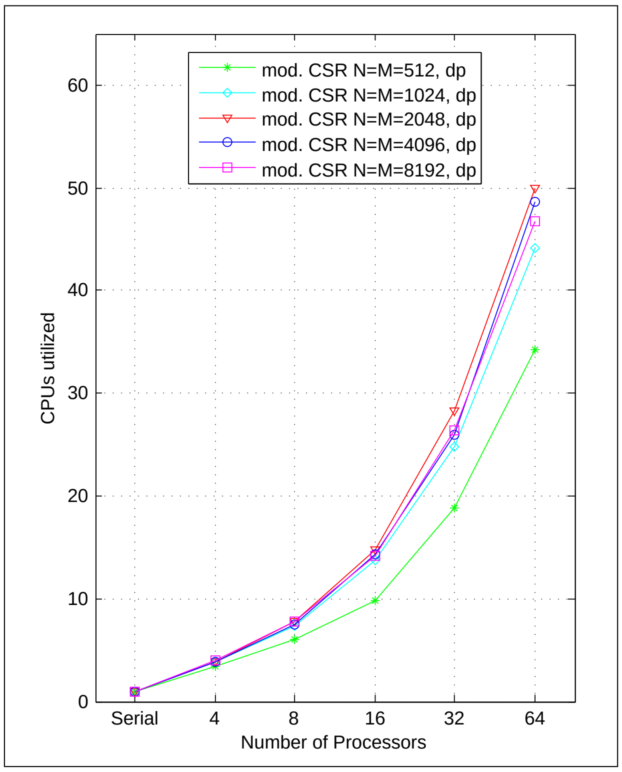


Figure B.4: *"Number of CPUs utilized" for the modified double precision Jacobi CSR implementation.*

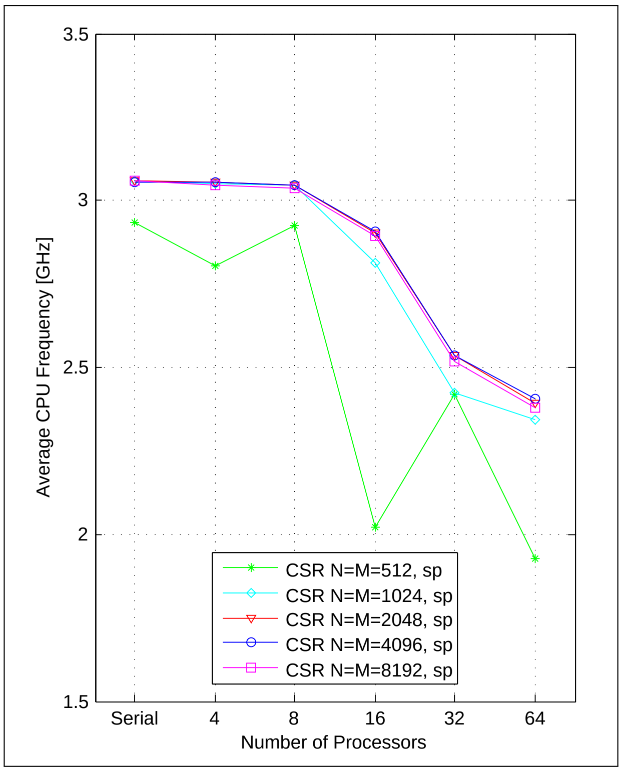


Figure B.5: *Average CPU frequency achieved for the unmodified single precision Jacobi CSR implementation.*

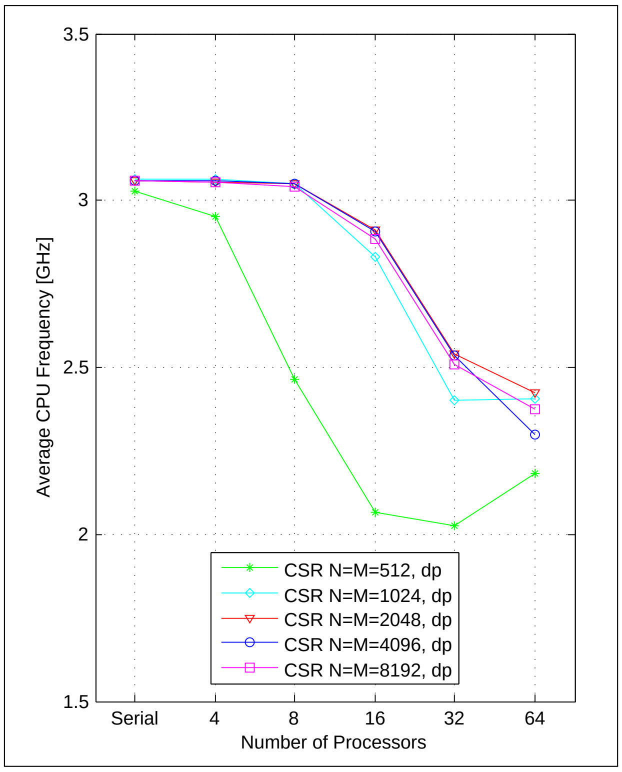
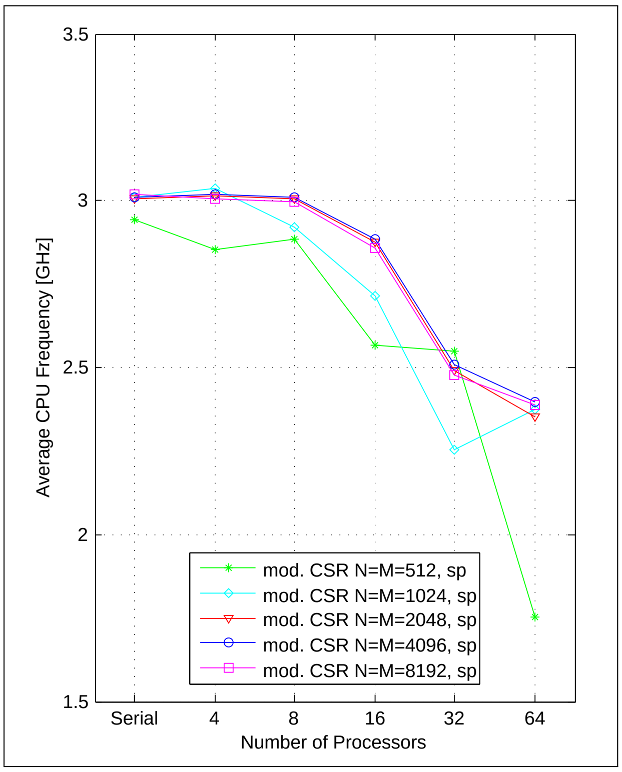


Figure B.6: *Average CPU frequency achieved for the unmodified double precision Jacobi CSR implementation.*



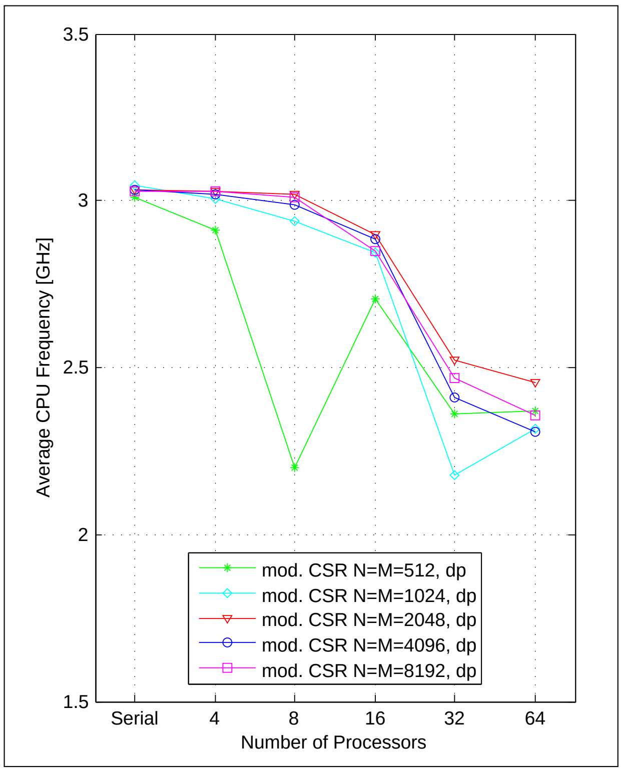
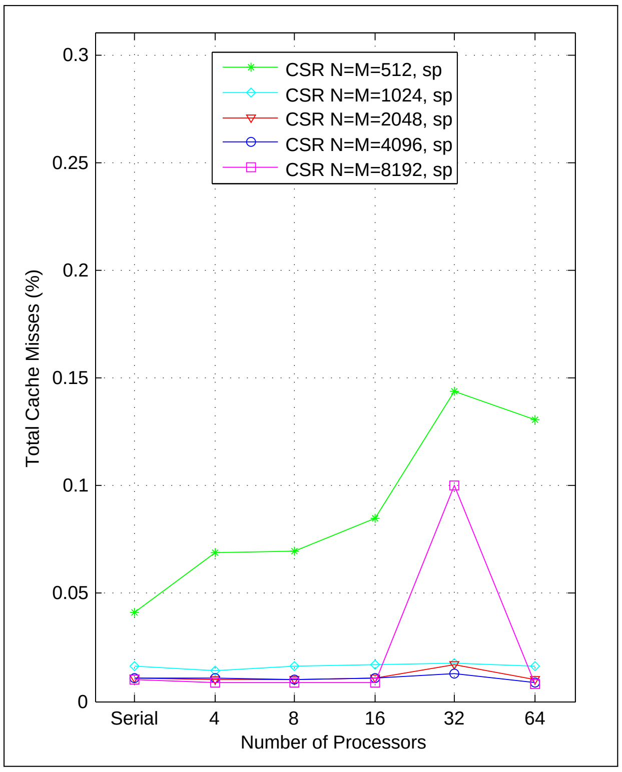


Figure B.7: *Average CPU frequency achieved for the modified single precision Jacobi CSR implementation.*

Figure B.8: *Average CPU frequency achieved for the modified double precision Jacobi CSR implementation.*



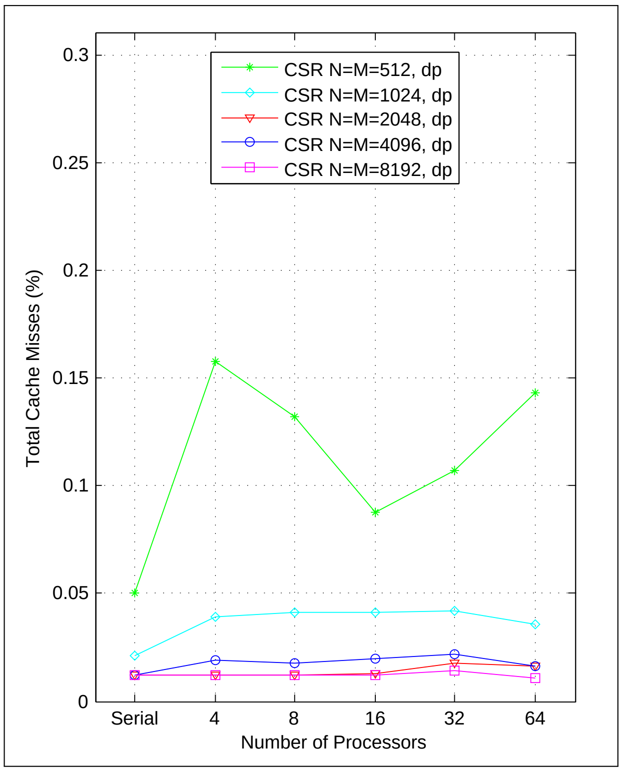


Figure B.9: *Percentage of the total cache misses (based on all cache references) for the unmodified single precision Jacobi CSR implementation.*

Figure B.10: *Percentage of the total cache misses (based on all cache references) for the unmodified double precision Jacobi CSR implementation.*

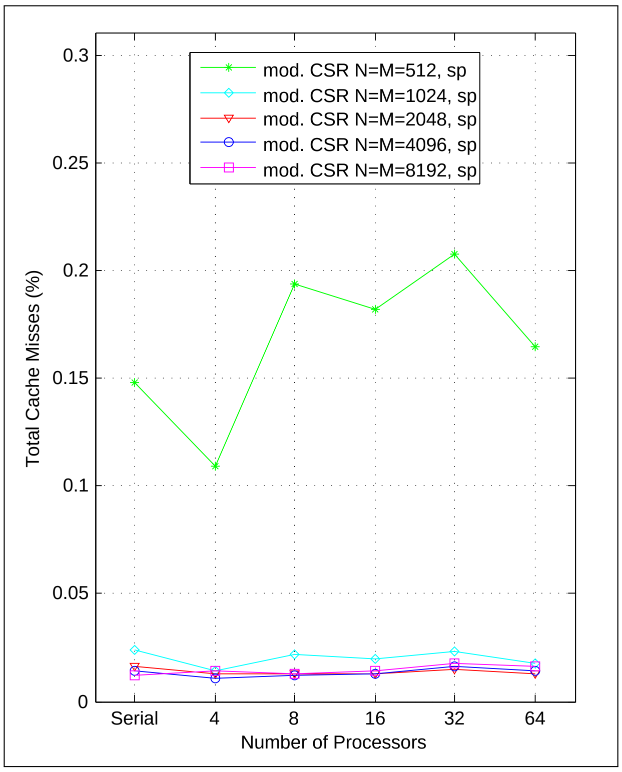


Figure B.11: *Percentage of the total cache misses (based on all cache references) for the modified single precision Jacobi CSR implementation.*

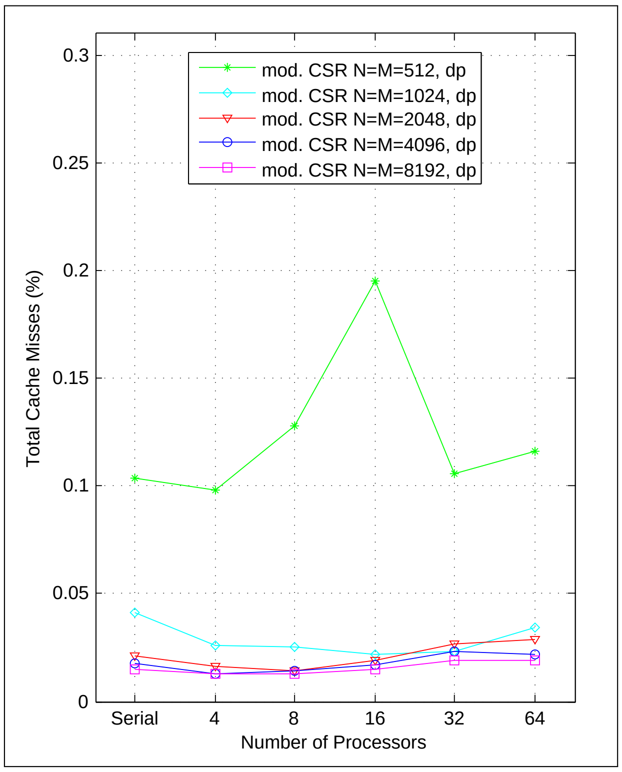


Figure B.12: *Percentage of the total cache misses (based on all cache references) for the modified double precision Jacobi CSR implementation.*

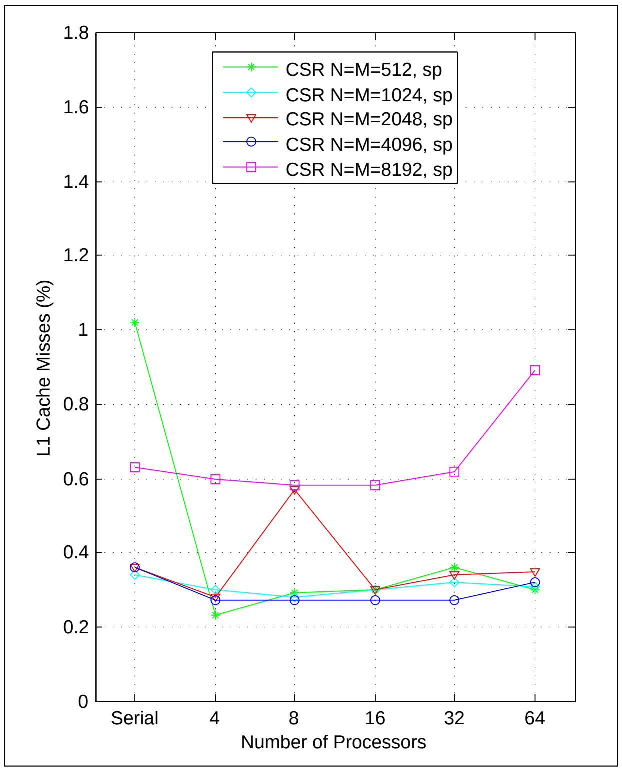


Figure B.13: *Percentage of the L1 cache misses (based on all L1 cache references) for the unmodified single precision Jacobi CSR implementation.*

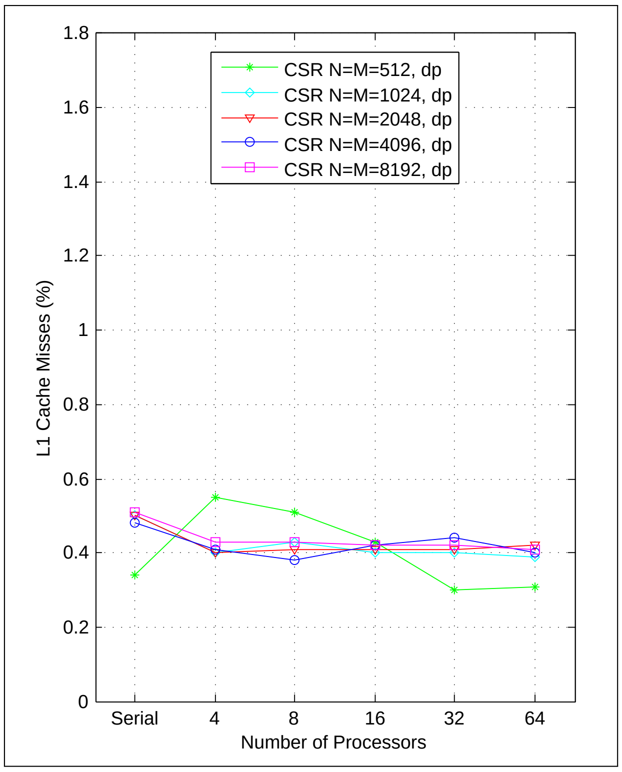


Figure B.14: *Percentage of the L1 cache misses (based on all L1 cache references) for the unmodified double precision Jacobi CSR implementation.*

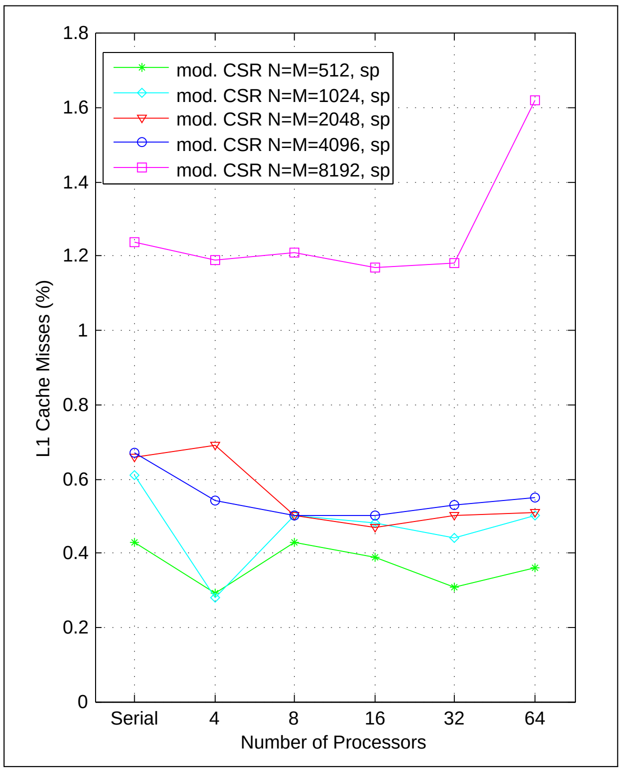


Figure B.15: *Percentage of the L1 cache misses (based on all L1 cache references) for the modified single precision Jacobi CSR implementation.*

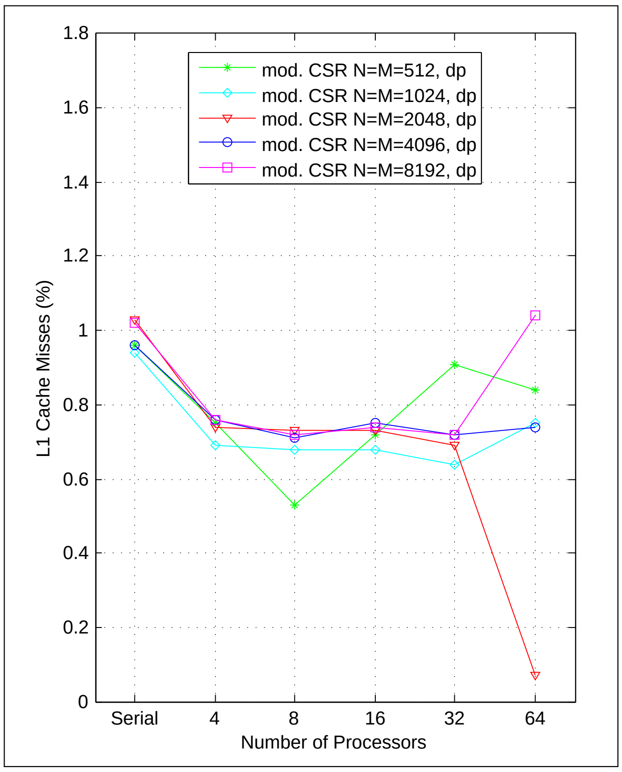


Figure B.16: *Percentage of the L1 cache misses (based on all L1 cache references) for the modified double precision Jacobi CSR implementation.*

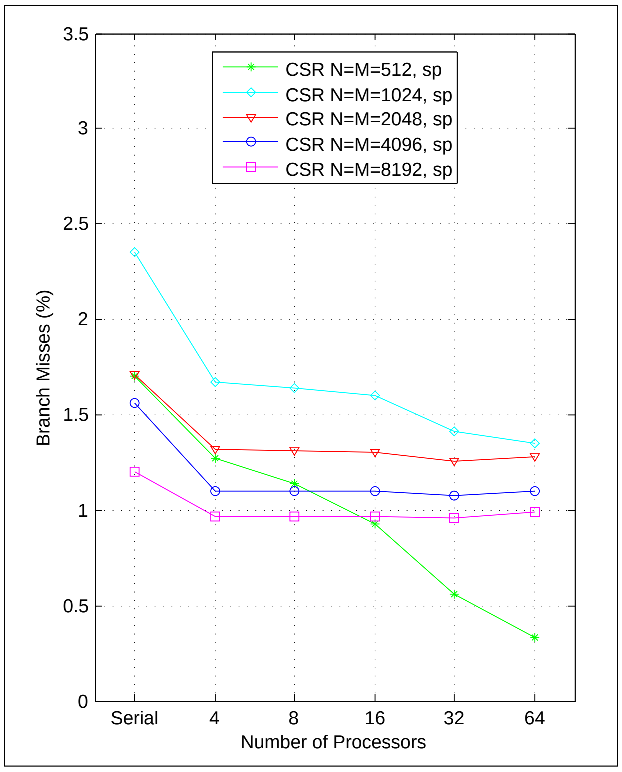


Figure B.17: *Percentage of the branch misses for the unmodified single precision Jacobi CSR implementation.*

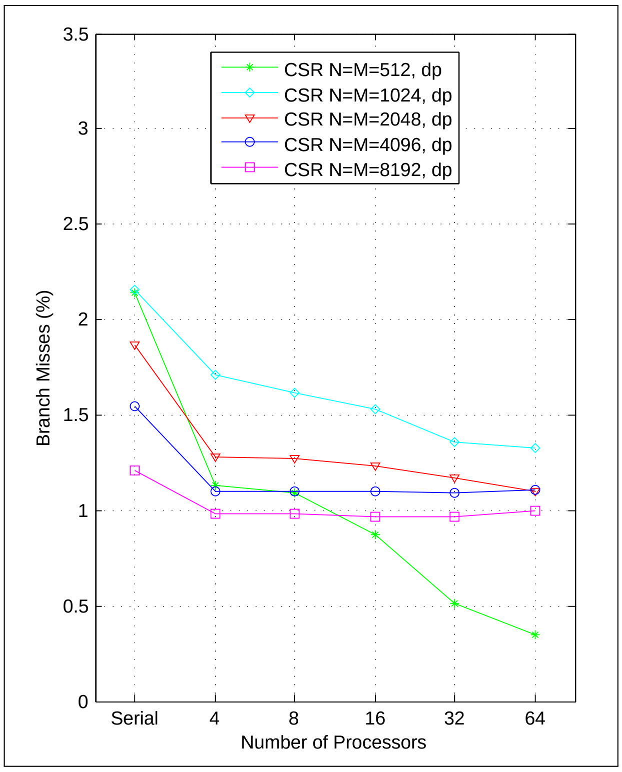


Figure B.18: *Percentage of the branch misses for the unmodified double precision Jacobi CSR implementation.*

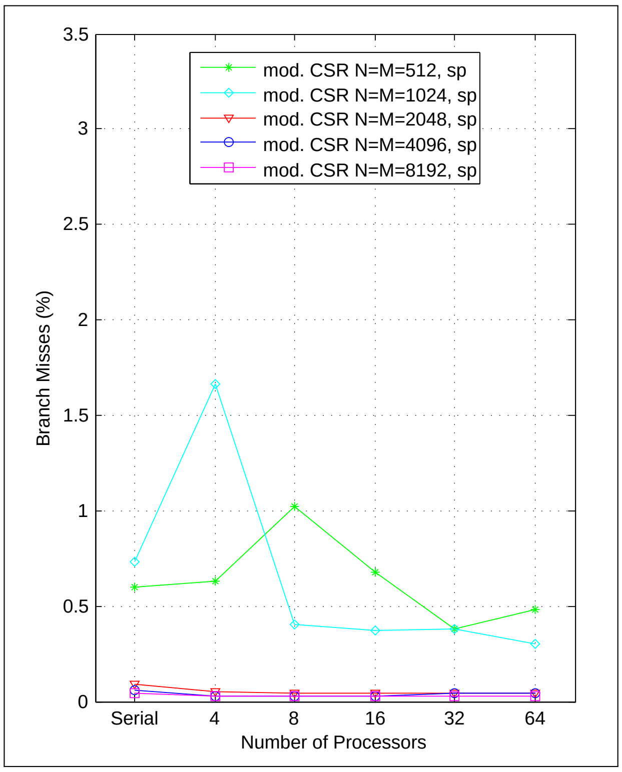


Figure B.19: *Percentage of the branch misses for the modified single precision Jacobi CSR implementation.*

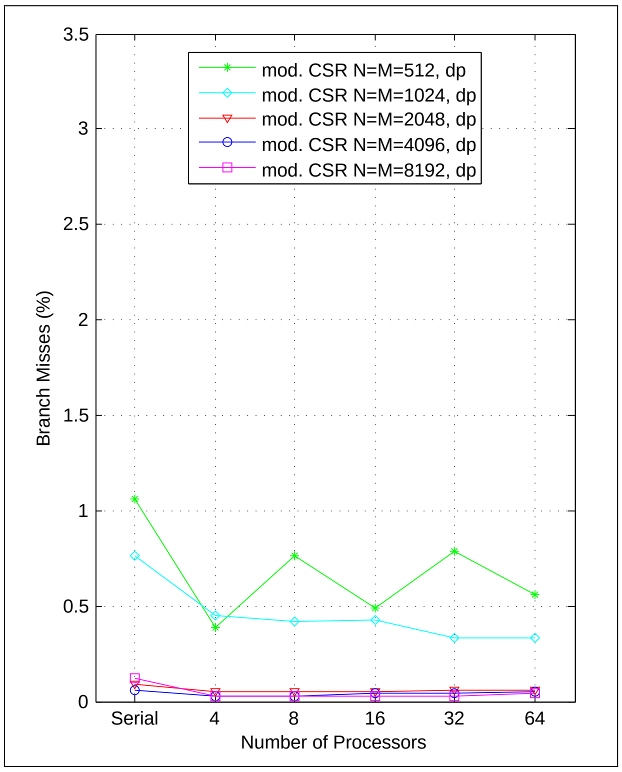


Figure B.20: *Percentage of the branch misses for the modified double precision Jacobi CSR implementation.*

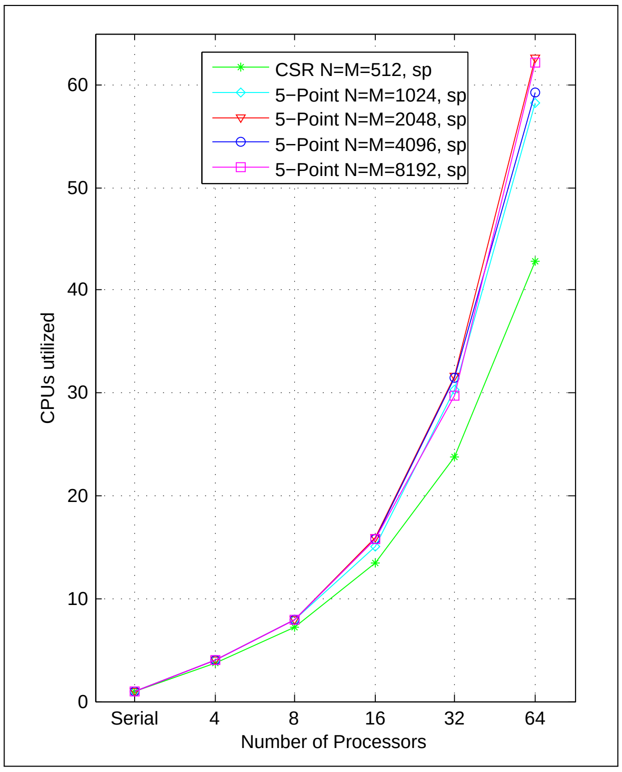


Figure B.21: *"Number of CPUs utilized" for the single precision five-point Jacobi implementation.*

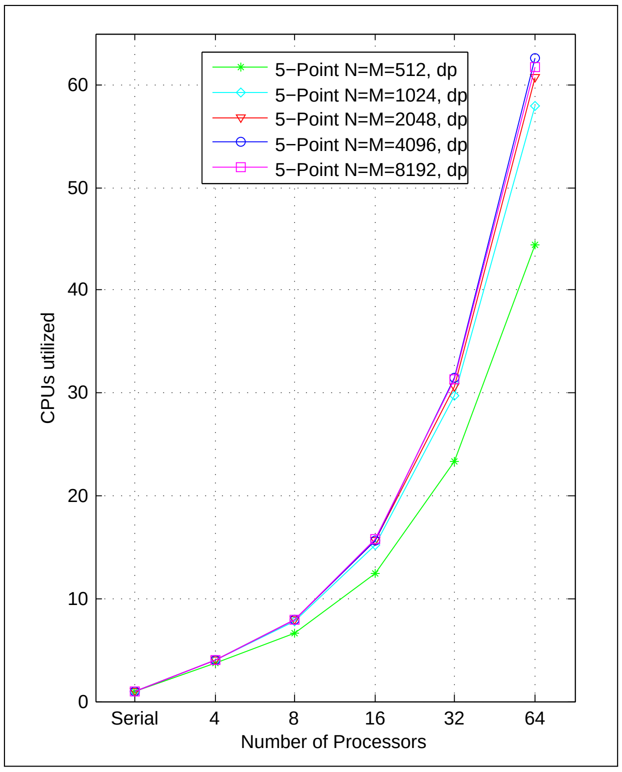


Figure B.22: *"Number of CPUs utilized" for the double precision five-point Jacobi implementation.*

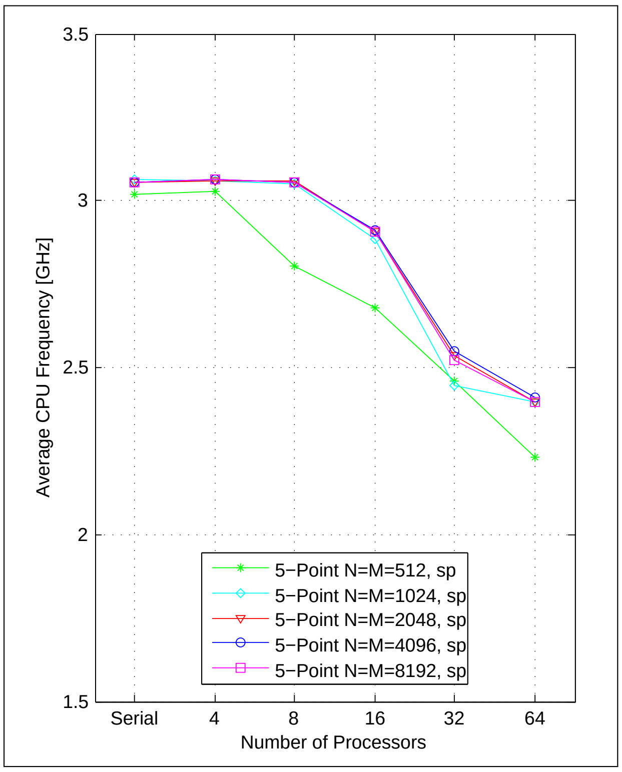


Figure B.23: *Average CPU frequency achieved for the single precision five-point Jacobi implementation.*

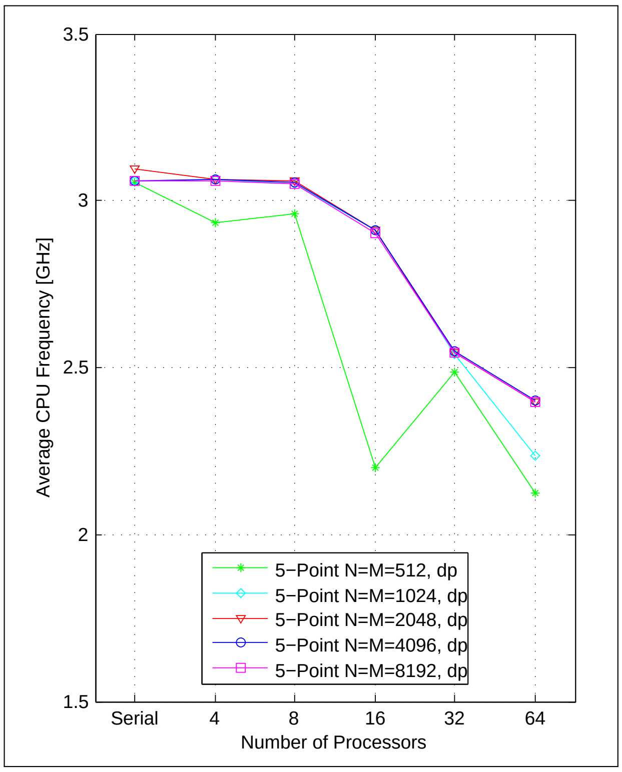


Figure B.24: *Average CPU frequency achieved for the double precision five-point Jacobi implementation.*

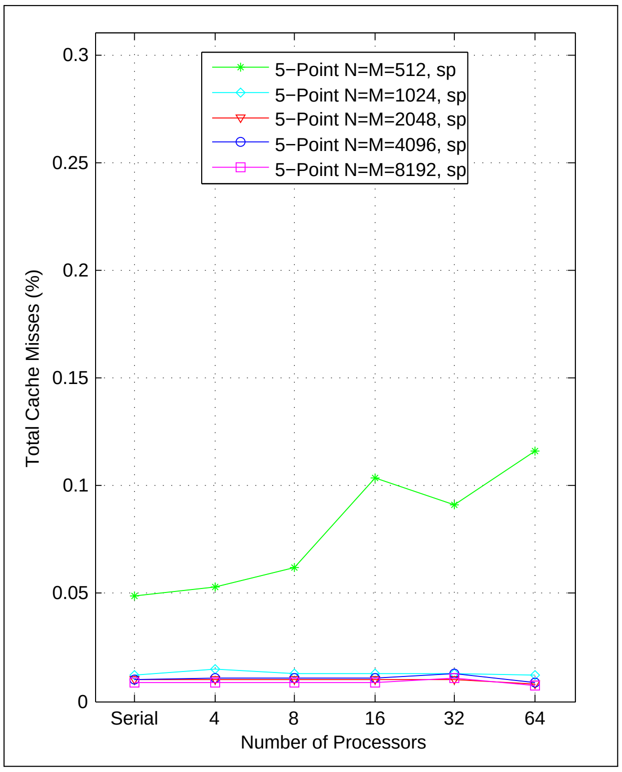


Figure B.25: *Percentage of the total cache misses (based on all cache references) for the single precision five-point Jacobi implementation.*

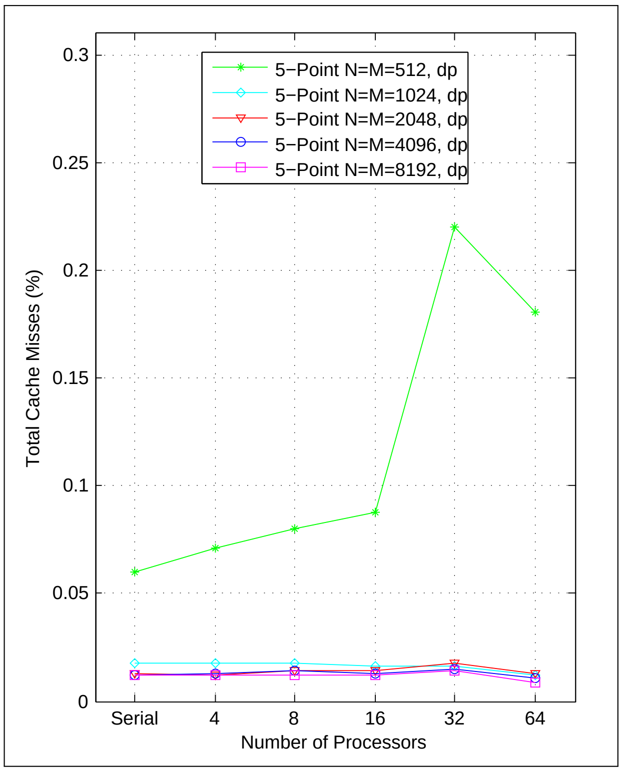


Figure B.26: *Percentage of the total cache misses (based on all cache references) for the double precision five-point Jacobi implementation.*

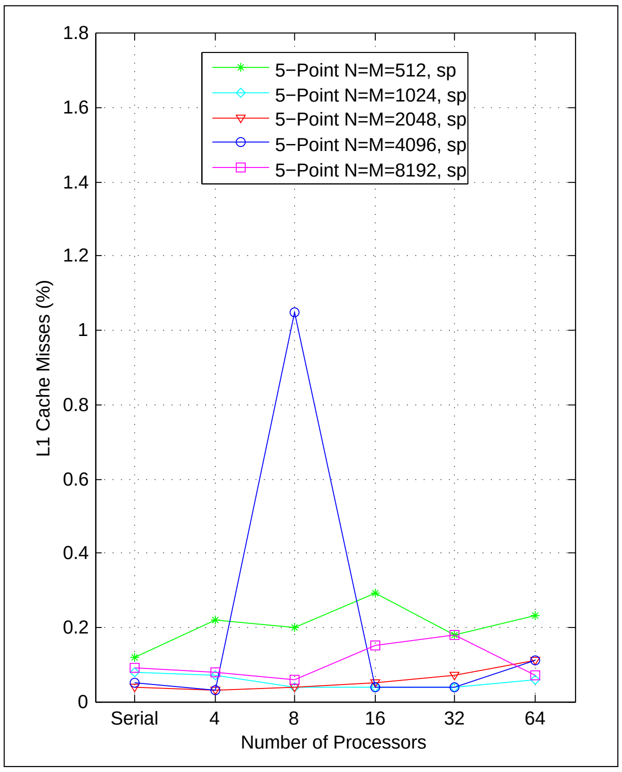


Figure B.27: *Percentage of the L1 cache misses (based on all L1 cache references) for the single precision five-point Jacobi implementation.*

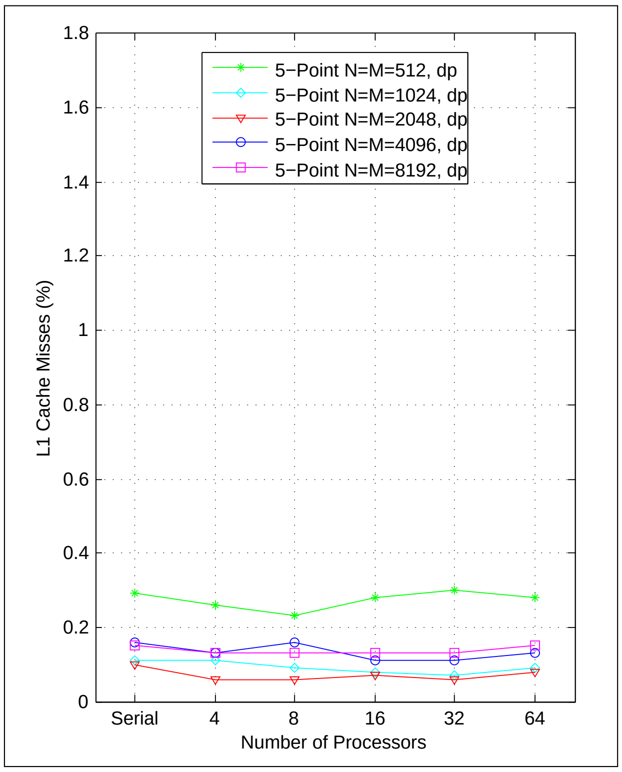


Figure B.28: *Percentage of the L1 cache misses (based on all L1 cache references) for the double precision five-point Jacobi implementation.*

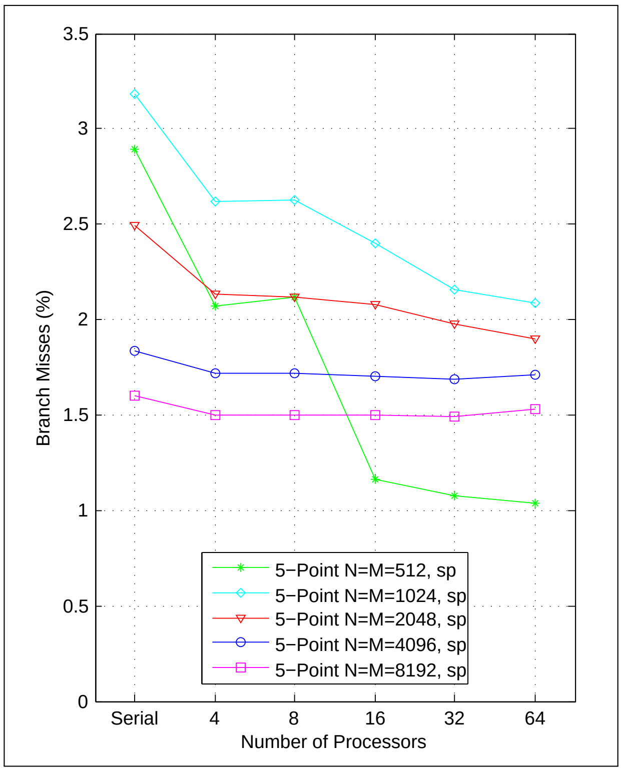


Figure B.29: *Percentage of the branch misses for the single precision five-point Jacobi implementation.*

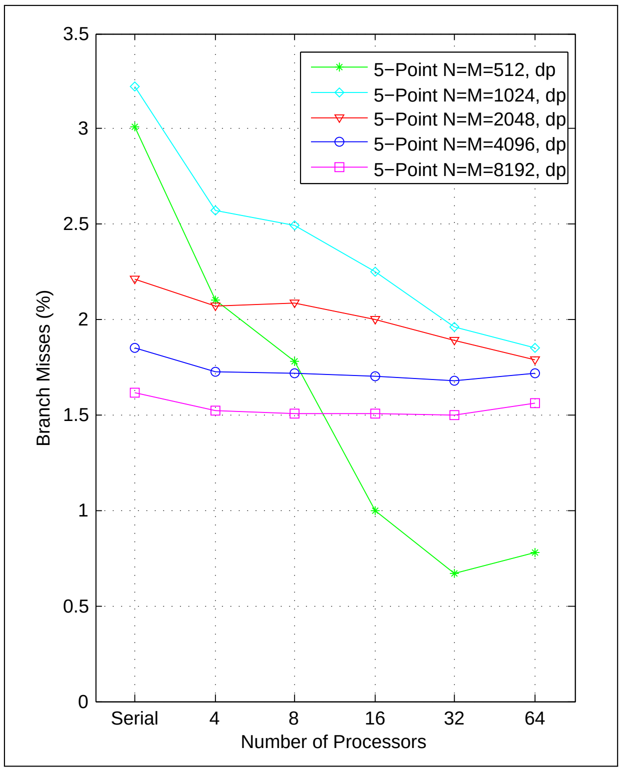


Figure B.30: *Percentage of the branch misses for the double precision five-point Jacobi implementation.*

Bibliography

John Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359579. URL http://doi.acm.org/10.1145/359576.359579. 6

Blaise Barney. OpenMP. https://computing.llnl.gov/tutorials/openMP/, 2016. [Online; accessed 01-September-2016]. ix, 5, 7, 10

OpenMP Architecture Review Board. OpenMP Application Programming Interface. http://www.openmp.org/mp-documents/openmp-4.5.pdf, 2015. [Online; accessed 05-September-2016]. 5

Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Cengage Learning, 4th edition, 1989. 61

Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. ix, 5, 6, 7, 8, 9, 10, 11, 12, 22

Ed Grochowski and Murali Annavaram. Energy per Instruction Trends in Intel® Microprocessors. http://www-cs.intel.com/pressroom/kits/core2duo/pdf/epi-trends-final2.pdf. [Online; accessed 27-October-2016]. ix, 1, 2

Nitin Gupta. Texture Memory in CUDA: What is Texture Memory in CUDA Programming. http://cuda-programming.blogspot.com.br/2013/02/texture-memory-in-cuda-what-is-texture.html, 2013. [Online; accessed 13-September-2016]. 17

E. Isaacson and H.B. Keller. *Analysis of Numerical Methods*. Dover Books on Mathematics. Dover Publications, 3rd edition, 1996. 25, 26, 27, 29, 30

David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 2nd edition, 2013. ix, xiii, 5, 6, 8, 12, 17, 19, 20

B. Noble and J.W. Daniel. *Applied linear algebra*. Prentice Hall, 3rd edition, 1998. 68, 70

NVIDIA. NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012. [Online; accessed 09-September-2016]. ix, 12, 14, 15, 18

OpenACC-Standard.org. The OpenACC® Application Programming Interface Version 2.5. http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf, 2015. [Online; accessed 16-September-2016]. 5, 20

J.M. Ortega. *Numerical Analysis: A Second Course*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 1972. 29, 31, 60, 63

Greg Ruetsch. An Easy Introduction to CUDA Fortran. https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-fortran/, 2012. [Online; accessed 11-September-2016]. 15

Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2000. 34

Sajjan G. Shiva. *Computer Organization, Design, and Architecture*. CRC Press, 4th edition, 2007. ix, 5, 6

J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Texts in Applied Mathematics. Springer New York, 1980. 59, 61, 62, 67

Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. http://www.gotw.ca/publications/concurrency-ddj.htm, 2005. [Online; accessed 22-September-2016]. ix, 1, 3

R.S. Varga. *Matrix iterative analysis*. Prentice-Hall series in automatic computation. Prentice-Hall, 1962. 30, 31

Michael Wolfe. Understanding the CUDA Data Parallel Threading Model – A Primer. https://www.pgroup.com/lit/articles/insider/v2n1a5.htm, 2012. [Online; accessed 11-September-2016]. ix, 12, 13, 17, 18