GPU-accelerated CFD Simulations for Turbomachinery Design Optimization

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op
maandag 2 oktober 2017 om 10:00 uur

door

Mohamed Hassanine AISSA
Diplom in Aerospace Engineering, University of Stuttgart, Germany

geboren te Nabeul,Tunesië

Dit proefschrift is goedgekeurd door de
Promotor: Prof.dr.ir. C. Vuik
Copromotor: Dr.ir. T. Verstraete

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. C. Vuik,	Technische Universiteit Delft, promotor
Dr.ir. T. Verstraete,	Von Karman Institute, Belgium, copromotor

Onafhankelijke leden:

Prof.dr. S. Hickel	Technische Universiteit Delft , The Netherlands
Prof.dr.ir. C.W. Oosterlee	CWI and Technische Universiteit Delft , The Netherlands
Prof.dr.ir. B. Koren	Technische Universiteit Eindhoven, The Netherlands
Prof.dr. S.F. Portegies Zwart	Universiteit Leiden , The Netherlands
Dr. A. Lani	Von Karman Institute, Belgium

GPU-accelerated CFD Simulations for Turbomachinery Design Optimization

Dissertation at Delft University of Technology.

The work described in this thesis has received funding from the European Union Seventh Framework Programme (FP7/2007-2013), Marie Curie Initial Training Networks (ITN) action, under grant agreement no. 316394, AMEDEO. This work was also partially funded by VLAIO in the framework of the SBO EUFORIA project (IWT-140068). The support is gratefully acknowledged.

ISBN 978-2-87516-123-9

Published by: The Von Karman Institute for Fluid Dynamics with permission

Contents

Samenvatting **vii**

Summary **ix**

Acknowledgment **xi**

1 Introduction **1**
1.1 CFD and automated optimization 1
1.2 High-Performance Computing 4
1.3 Research objectives 6
1.4 Context and outline of the thesis 8

2 Graphics Processing Units for High Performance Computing **11**
2.1 Introduction 11
2.2 From graphics pipelines to High Performance Computing 12
2.3 GPUs: a throughput-oriented latency-tolerant HPC device 14
2.4 CUDA: a programming language and an execution model 15
2.5 Memory hierarchy of the GPU 21
2.6 Profiler-Driven code optimization 24
2.7 Conclusion 27

3 Literature Review: Use of the GPU in Design Optimization **29**
3.1 Introduction 29
3.2 Topology optimization 30
3.2.1 Solid Isotropic Microstructure with Penalization method (SIMP) 31
3.2.2 Level-Set method 33
3.2.3 Underlying FEM 33
3.3 Shape optimization 36
3.4 Multidisciplinary Design Optimization (MDO) 39
3.5 GPU in meta-heuristics 39
3.6 Discussion 40
3.7 Conclusion 40

4 GPU-accelerated Simulations with Explicit Time-Stepping 43
4.1 Reynolds-Averaged Navier-Stokes Equations . . . 43
4.2 Implementation and discussion . . . 45
4.2.1 The convective flux evaluation . . . 45
4.2.2 Viscous flux . . . 58
4.2.3 Boundary conditions . . . 63
4.2.4 Interface update . . . 63
4.2.5 Runge-Kutta stages . . . 65
4.2.6 Convergence acceleration techniques . . . 67
4.3 Validation and benchmark . . . 72
4.4 Conclusion . . . 75

5 GPU-accelerated Simulations with Implicit Time-Stepping 77
5.1 Introduction . . . 78
5.2 Numerical scheme . . . 79
5.3 Preconditioned Krylov solvers . . . 80
5.4 Flow solver implementation . . . 85
5.4.1 System assembly . . . 85
5.4.2 Linear solver with *on-demand* factorization . . . 100
5.5 Results . . . 111
5.6 Discussion . . . 114
5.6.1 Effect of the CFL number on the GPU speedup . . . 115
5.6.2 Effect of the RK stages number on the GPU speedup . . . 115
5.6.3 Effect of the linear solver stop condition on the GPU speedup 116
5.7 Validation . . . 116
5.8 Summary . . . 118
5.9 Conclusion . . . 118

6 Explicit versus Implicit CFD Simulations, the GPU dimension 119
6.1 Introduction . . . 119
6.2 The classification . . . 122
6.3 Acceleration of the time integration method . . . 127
6.4 Numerical experiments: the subsonic turbine cascade T106C . . . 130
6.5 Discussion . . . 135
6.6 Conclusion . . . 136

7 Applications: GPU CFD solvers in Design Optimization 137
7.1 One-level optimization of a supersonic compressor cascade . . . 137
7.2 Metamodel-assisted optimization using Kriging . . . 140
7.2.1 Kriging . . . 140
7.2.2 The LS82 cascade . . . 143
7.3 Final remarks . . . 145

8 Conclusions 149
8.1 Future work . . . 151
8.1.1 Convergence acceleration of explicit solver . . . 151
8.1.2 Metamodeling on the GPU . . . 151

A Appendix: Used Test cases **153**
A.1 Turbine inlet guide vane: LS89 153
A.2 Compressor stator cascade: CC2D 155
A.3 Turbine stator cascade: T106C 155
A.4 3D Compressor stator blade: Turbolab 155

Curriculum vitae **157**

List of publications and presentations **159**

Samenvatting

Ontwerp optimalisatie is sterk afhankelijk van tijdrovende simulaties, vooral bij het gebruik van gradintvrije methodes. Deze methoden vereisen een groot aantal simulaties om een merkbare verbetering te krijgen op bestaande ontwerpen, die tegenwoordig op basis van de geaccumuleerde kennis over de jaren heen vaak reeds optimaal zijn.

High Performance Computing (HPC) is essentieel om de uitvoeringstijd van simulaties te verminderen. Terwijl parallelle programmering met behulp van de CPU gevestigd is sinds meer dan twee decennia, is het gebruik van andere technieken, zoals de Graphics Processing Unit (GPU), relatief recent in het domein van ontwerpoptimalisatie. De GPU heeft eigenlijk een enorme rekenkracht die vergelijkbaar is met een cluster van verschillende CPUs maar geconcentreerd in slechts n apparaat. Deze rekenkracht is evenwel niet gemakkelijk te gebruiken, aangezien volledige delen van de broncode moeten herschreven worden in een GPU-programmeertaal. Hoewel hoog-niveau programmeertalen (bijvoorbeeld openACC) een versnelling met een lage ontwikkelingskost kunnen realiseren, is het niet eenvoudig om met deze methoden grote snelheden te krijgen. Programma-talen op laag niveau zijn efficinter, maar er worden verschillende versnellingen gemeld en er is behoefte aan een diepere analyse om het GPU-potentieel transparanter te maken voor wetenschappers, vooral niet-experts in HPC.

Om de GPU-versnelling voor stationaire CFD-simulaties te bestuderen, zijn twee verschillende technieken binnen de GPU ingevoerd; n met expliciete en de tweede met impliciete tijdsintegratie. Na de overdracht en de validatie van de CPU-code naar de GPU, leidt de GPU-code optimalisatie tot het identificeren van een reeks sleutelparameters die de GPU-efficintie benvloeden. Tegelijkertijd zijn beide methoden vergeleken, wat resulteert in een prestatiemodel en een classificatie van de GPU-versnelling van sommige CFD-operaties. Het doel is om wetenschappers in staat te stellen een beslissing te nemen over de GPU-overdracht van hun CPU-applicaties door een GPU-versnelling te voorspellen.

Naast de twee GPU CFD-codes die nu gentegreerd zijn in het optimalisatie softwarepakket ontwikkeld in het VKI, verschafte dit onderzoek sleutelelementen om de dubbelzinnigheid over het GPU-potentieel te verminderen, namelijk een kwalitatieve analyse en een classificatie. Deze hulpmiddelen kunnen helpen bij het selecteren van

de beste kandidaat voor een doorbraak in CFD-acceleratie. Tegelijkertijd identificeerde dit werk ernstige beperkingen bij de preconditioning van een lineair systeem van vergelijkingen en de limiet van hedendaagse iteratieve matrixfactorisatiemethoden met betrekking tot stabiliteit en convergentie. Er is nood aan een paradigma verschuiving naar inherente parallelle preconditioners.

De ontwikkelde codes werden getest op het optimaliseren van een compressor en een turbine cascade, dewelke resulteerden in een sneller optimalisatieproces op de GPU.

Summary

Design optimization relies heavily on time-consuming simulations, especially when using gradient-free optimization methods. These methods require a large number of simulations in order to get a remarkable improvement over reference designs, which are nowadays based on the accumulated engineering knowledge already quite optimal.

High-Performance Computing (HPC) is essential to reduce the execution time of the simulations. While parallel programming using the CPU is established since more than two decades, the use of accelerators, such as the Graphics Processing Unit (GPU), is relatively recent in design optimization. The GPU has actually a huge computational power comparable to a many-core cluster but concentrated in one device. This raw power is not easy to utilize as entire code parts have to be rewritten using a GPU programming language. Even though high-level standards (e.g. openACC) are able to bring a basic acceleration with a low development effort, it is not simple to get large speedups with these methods. Low-level programming languages are more efficient but different speedups are reported and there is a need for a deep analysis to make the GPU potential more transparent to scientists especially non-experts in HPC.

In order to study the GPU acceleration for CFD steady simulations, two in-house CFD solvers have been ported to the GPU; one with explicit and the second with implicit time-stepping. After the porting and the validation of the GPU solvers, the GPU code optimization leads to the identification of a set of key parameters affecting the GPU efficiency. At the same time, both methods have been compared resulting into a performance model and a classification of the GPU acceleration of some CFD operations. The purpose is to enable scientists to take an educated decision concerning the GPU porting of their CPU applications by providing an expected GPU speedup.

In addition to the two GPU CFD solvers that are now integrated into the in-house design optimization software package, this research provided key elements to reduce the ambiguity about the GPU potential, namely a qualitative analysis and a classification. These tools can help selecting the best candidate for a breakthrough in CFD acceleration. At the same time, this work identified serious limitations in the preconditioning of a linear system of equations and the limit of today iterative

matrix factorization methods in terms of stability and convergence. There is a need for a paradigm shift toward inherently parallel preconditioners. The developed tools have been used for the optimization of a compressor and a turbine cascade resulting into a faster optimization process on the GPU.

Acknowledgment

My research work within my Ph.D. started with the project AMEDEO enabling me to meet and exchange with great people, visit many research institutes and access valuable training all over the journey. For that, I am very grateful to my supervisor Dr. Tom Verstraete who believed on my small but growing GPU expertise back on 2013. Tom has done a fantastic supervising work with me by first giving me the freedom to act in my research but then closely checking my results. Moreover, I would like to thank Juliet Jopson for successfully coordinating the AMEDEO project and making the administration part much easier for all research fellows within the project. I would like also to thank Prof. Harvey Thomson for offering me a secondment within his team at the University of Leeds. During the short stay, I cooperated with Nicolas Delbosc from whom I learned so much about GPU performance assessment. I am very grateful to you Nic. I am thankful to all of AMEDEO people and a special thank to Dr. Roeland de Breuker for putting me in contact with Prof. Kees Vuik who later become my Ph.D. promoter.

Prof. Vuik knows how to get a student fully motivated and prepared for the challenges within a Ph.D. project. You name a software library or a research facility and he would surely know someone to contact to make the stuff happen that you asked for (e.g. DAS-5, SURFsara, Paralution). For all the time and the effort you invested in my project I am really thankful.

I would like also to thank the graduate school at TU Delft for the enriching courses and the possibility they bring to meet other Ph.D. students and exchange about ideas, challenges, views, fears, and ambitions. I would like to thank Deborah Dongor for helping me through the process of being an external Ph.D. student.

My work would not have reached the maturity it has now without the constant support of the people of the TU department in VKI. My supervisor Tom and Prof. Tony Arts were always supportive when it comes to attending conferences and presenting my work. I owe a lot of what I learned during these last years to my colleague Lasse Müller. Besides the valuable discussions he provided me with the CPU version of the CFD simulation, the base of all my work. I am grateful to my colleague Christopher Chahine who taught me a lot of practical knowledge in design optimization. I would like to thank Roberto with whom I cooperated the last months and learned a lot. Eager to continue the collaboration. For the rest of the TU group I

hope we will renew our biweekly department meeting to continue exchanging about our latest results.

I would like to thank all the computer center staff here at VKI and also Christelle and Evelyne for supporting my numerous requests for papers and documentation. I am thankful to the VKI administration and especially to Dirk who made all administrative procedures so easy for me at the beginning of my Ph.D. and after him, Simone took over and facilitated all the paper work for me.

I would like to thank Dr. Andrea Lani and his team first, for the fruitful discussions over the best use of the GPU but also his patience for my long benchmark using his machine as a reference CPU for all my thesis and my publications.

I am also deeply grateful to all committee members for the time and effort invested on evaluating my manuscript.

I would like to thank my office mates: Guillaume, Miguel, and Zdenek. With those guys (and especially after Davide left :D), the office was well stuffed for a nice science journey.

There are some people without them I would not have reached the point to start a Ph.D. That is my father who taught me to be curious and enthusiastic about nature and engineering and my mother who taught me discipline and integrity. Being the youngest member in my family I learned a lot from my sisters: Nejla, Khaoula, Kaouther, and Amina. To all of them, I am very thankful.

Finally, thank you Meyssen for your unconditional support and the happy moments we spent together. You made it possible for me to face the challenges in the Ph.D. and beyond.

1 Introduction

Modern aircraft have a reduced CO_2 emission and fuel consumption compared to airplanes of last century. Figure 1.1 shows a threefold decrease of aircraft energy usage per Available Seat Kilometer (ASK) from the fifties to the end of the twentieth century. This is the result of a continuous optimization process of the whole aircraft including the topology, the shape, and materials. It is worth to note also that roughly 80% of the reduction in the specific fuel consumption is due to advances in gas turbines and propulsion in general. This effort is expected to increase further as a report by the European Commission [†] highlights the necessity of a stronger reduction of the emissions for future airplanes in order to reduce the environmental impact. The potential to further improve the efficiency of a design is decreasing with years (e.g. in 1960 it was much easier to reduce 1% of the specific fuel consumption (SFC)[‡] than it is now), while the optimization effort increases dramatically. In the last decades, two major developments are projected to become key enablers to further improve modern designs: automated high-fidelity optimization and High-Performance Computing (HPC). These two topics will be presented in this chapter in addition to the research objectives and the outline of the thesis.

1.1 CFD and automated optimization

Over the past 3 decades, the design process for turbomachinery applications has seen a large evolution. Nowadays, all components are evaluated through simulations using Computational Fluid Dynamics (CFD) and Computational Structural Mechanics (CSM) before being build. Unlike the traditional *trial-and-error* design approach, which relies on an extensive testing phase [Vassberg and Jameson, 2006], the process of virtual design allows for a large number of adaptations of the design leading to a higher performance. The next evolution currently enrolling within this process of virtual design is to rely on an automatic optimization algorithm to modify the design. These optimization algorithms can find a better-performing design in

[†] http://ec.europa.eu/transport/sites/transport/files/modes/air/doc/flightpath2050.pdf
[‡] SFC is a measure for the efficiency

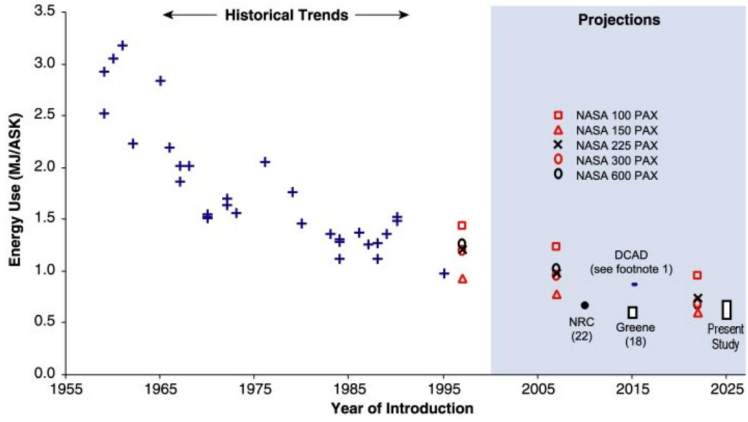


Figure 1.1: *Aircraft energy usage per Available Seat Kilometer (ASK): historical trends and projections (source:[Lee et al., 2001]).*

a reduced time with minimal intervention of the designer. The algorithm actually decides automatically on modifications to be made on the shape based on experience gradually build during the design process. The aim of the modifications is to maximize (or minimize) a certain objective related, for instance, to the efficiency (or to the losses).

While the existence of a better-performing design is case-dependent, reaching this design depends on how "*smart*" the optimization algorithm is and how much computational budget is available. First, more details are given of the optimization algorithm and later in the next section, the computational budget will be addressed.

Optimization methods can be roughly classified by the information required by the evaluation process. While zero-order methods require only the function evaluation of the objective, first-order methods require additionally the gradients of the objective function with respect to all design variables. First-order methods, such as steepest descent and conjugate gradient, have a better convergence behavior at the expense of computing the objective derivatives. The gradients can be computed with a repeated function evaluation within a finite difference scheme or through the adjoint state method. The gradient-based optimization improves, in general, a single design using the gradients and exposes, therefore, a reduced parallelism. Since this work studies mainly the effect of the massive parallelization on the design optimization, it focuses only on gradient-free optimization methods.

Most of zero-order optimization methods are nature-inspired and based on metaheuristics. Some gradient-free methods improve a single design by exploiting its neighborhood in the design space through a local search such as Tabu search [Glover, 1989] and simulated annealing [Talbi, 2009]. Other zero-order methods are population-based such as evolutionary algorithms [Fonseca and Fleming, 1995] and swarm intelligence [Talbi, 2009]. These methods explore the design space with a multitude of interacting and evolving designs. Explorative algorithms try to uncover within a few iterations a wide region of the design space, while exploitative algorithms tend to focus on a limited area of the design space (e.g. the neighborhood of a given

design).

The optimization of an airplane wing, for instance, involves external flow aerodynamics and in general targets a higher lift and a lower drag, which results in a better usage of the power delivered by the jet-engine. The delivered power itself can be increased through an optimization of the engine using internal flow aerodynamics. While more resistant materials and clever cooling methods brings the engine to operate at higher temperatures and deliver thus more power with a higher efficiency, the optimization of the shape of the turbine and compressor blades mainly reduces the losses for the flow through the engine. In aerodynamic optimization, a design is parameterized, in general, using some geometric parameters that control the shape of a blade such as the thickness, the blade chord and the blade angles. Some of these parameters can be chosen as design variables. All possible combinations for these design variables, which are allowed to vary within a fixed interval, form the design space. This space can be constrained by some conditions proper to the case such as a minimum thickness.

The final design will be validated with wind tunnel experiments to check the accuracy of the numerically delivered results. In order to reduce the gap between the experimental results and the simulations, the discretization of the flow governing equations should incorporate most of the relevant physical phenomena. Euler equations, for instance, are limited to inviscid flows [Anderson, 1995] neglecting all shear stresses and heat conduction terms [Hirsch, 2007], while Navier-Stokes equations cover the viscosity and the flow turbulence as well. CFD solvers need also to account for the turbulence, which is a complicated multiscale phenomenon. Some methods, such as the Reynolds-Averaged Navier-Stokes approach, rely on models to reproduce the effect of the turbulence. Other methods, such as the Large Eddy Simulation (LES), resolve the largest turbulent scales, while modeling the smaller scales [Pope, 2001] and the Direct Numerical Simulation (DNS) resolves all scales of the turbulence. It would be tempting to use DNS in order to reach the best possible numerical precision. For transonic conditions, this method remains out of reach of current high-performance systems as the execution time on conventional systems is proportional to Re^3 [Pope, 2001]. As a result of its lower computational cost and reasonable accuracy under attached flow conditions, RANS simulations are nowadays widely established in academia and industry.

In a context of a gradient-free optimization where the search for the optimum requires many evaluations, RANS simulations need to be accelerated to maintain feasible turnaround times. Three types of algorithm optimizations [Thevenin and Janiga, 2008, p.14] are possible to improve the convergence of gradient-free optimization methods: mathematically, physically and computer science based optimizations. The first two methods accelerate the convergence of the optimization toward better designs with a given computational budget, while the last adapts the optimization to a higher computational power. Physically-based optimization reduces the complexity of the objective function evaluation by replacing it with a less complicated model (metamodel) that generates a faster but less accurate design evaluation. A mathematical-based optimization takes advantage of preconditioner and multigrid techniques to accelerate the convergence while solving systems of linear equations. Finally, a computer-science motivated optimization consists of using

high-performance computing to accelerate the execution of numerical simulations. The latter type of optimization is the central focus of this work and will be further discussed in the next subsection.

1.2 High-Performance Computing

Accelerating simulations within a design optimization can bring two advantages: (1) more designs can be evaluated for the same time budget leading to a wider exploration of the design space or alternatively, (2) higher-fidelity simulations are accessible for the same time budget.

Getting an application to run faster can be done in different ways. Straightforward methods include, for instance, a hardware upgrade (e.g. a higher CPU rate) or the use of a more advanced compiler, which does more automatic code optimization. A more promising method consists of parallelizing the application. The latter is not trivial but ensures a higher acceleration when adequately used. The parallelization, in some cases, can require a simulation code to be partially or totally rewritten (e.g. CUDA[†], openCL[‡]). In other cases, few added lines could be enough to enable a parallel execution of a portion of a code, in general for/while-loops (e.g. OpenMP[§], OpenACC[¶]). Accessing high accelerations using any of the above introduced methods requires definitely a sound knowledge of the used hardware and the specificities of the application to port.

The high performance systems can be classified following the memory architecture as depicted in Figure 1.2. The two main architectures are the shared-memory and the distributed-memory systems. For the shared-memory architecture, a set of processors shares the same memory contingent. Current popular architectures such as dual-core and quad-core CPUs belong to this group. OpenMP [OpenMP, 2013], an Application Programming Interface (API), handles the parallelization in these systems. Few simple compiler directives (#pragmas) surrounding sequential for-loops divide automatically the work between available cores and every core processes a part of the loop. The communication among processors is very simple since they all share the same memory contingent. The maximum available number of cores and the memory capacities for this system are, however, too low to cover large-scale problems (maximum by Xeon Phi with 60 cores [Jeffers and Reinders, 2013]).

In the distributed-memory configuration, better known as clusters, every processor has its own memory. The communication between processors occurs through the Message Passing Interface (MPI[‖]). A decomposition of the computational domain is essential for the parallelization on distributed-memory systems. Every processor contributes to the solution of the simulation by solving a part of the computational domain. A high number of cores (e.g. cluster of CPUs) could speed up the whole process significantly. But the parallelization increases also the programming burden, since the designer has to distribute the computational work among the available

[†] `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`
[‡] `https://www.khronos.org/opencl/`
[§] `http://www.openmp.org/`
[¶] `https://www.openacc.org/`
[‖] `http://mpi-forum.org/`

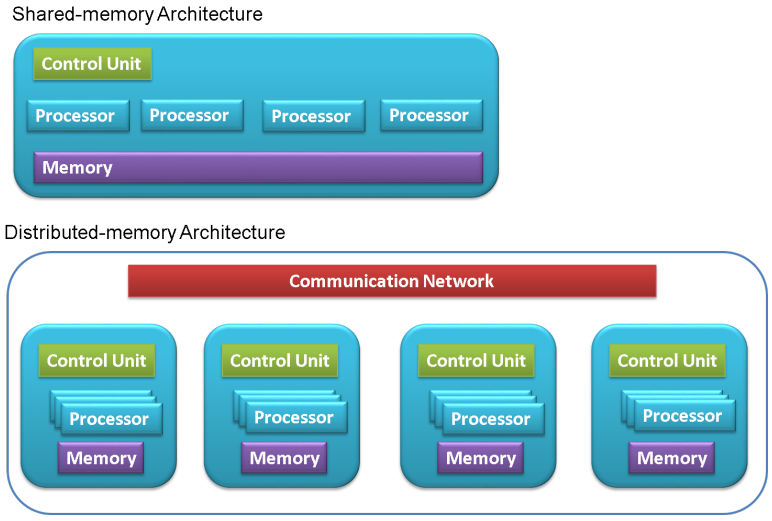


Figure 1.2: *Difference between shared and distributed memory architecture.*

CPU processors and regulate the communication. For a realistic application, a large number of cores is essential and MPI is the most implemented paradigm on today HPC systems. Hybrid systems, consisting of a cluster of shared-memory systems, are getting more and more popular. The most powerful HPC systems nowadays are hybrid [Strohmaier et al., 2016] while using in every cluster node not only standard CPUs but also accelerators such as Field Programmable Gate Arrays (FPGAs) and Graphics-Processing Units (GPUs).

The GPUs can be considered as a shared-memory system since a number of multi-processors are connected to the same device memory as depicted in Figure 1.3. In total, a GPU contains hundreds of cores mostly dedicated to arithmetic operations, called Arithmetic Logic Units (ALUs). Compared to a CPU core, a GPU core is less powerful. A GPU includes, however, a large number of these cores, which combined result in a higher computational power. The second advantage is the specialization of the GPU cores. While CPUs are inherently responsible for a wide spectrum of

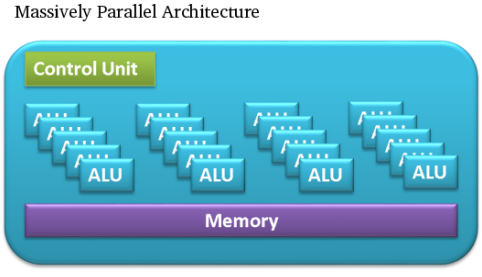


Figure 1.3: *Massively Parallel Architecture.*

tasks requiring a large cache capacity and an advanced flow control, a GPU is mostly dedicated to floating-point arithmetic calculations. This is reflected in the achieved high computation performance measured in float operations per second (FLOPs).

Today's GPUs evolved from graphics cards installed in most computers starting from the eighties. A graphics card is a complex electrical circuit that processes graphical data sent from the CPU to render and visualize it on the monitor with increasing quality and refreshing rates. A high pressure on graphics cards for fast refreshing of pictures (mainly for video-gaming) caused the spectacular increase of the computational power reflected by the large number of cores packed in one card.

The high computational power attracted scientists and engineers looking for low-cost high-performance alternatives to speed up their applications. To use these first graphics cards, scientists had to present their problems as graphical problem to the card, which implied a change to the data storage and the programming language. The term GPGPU, which stands for General Purpose Graphics Processing Unit, was established for this type of use of the graphics card. In response to this emerging demand, NVIDIA released in 2007 the first fully programmable open graphics processing units [Lindholm et al., 2008] in a C-based programming language called CUDA. At the same time, AMD released its programmable GPU with OpenCL. The programming model CUDA is specialized for NVIDIA GPUs whereas OpenCL can be run on AMD and NVIDIA GPUs. This portability has a price on the programming overhead and the peak performance gain [Fang et al., 2011]. Programming GPUs in a more generic way is possible with openACC, a similar approach to OpenMP. However, accessing high speedups on OpenACC is not trivial [Christgau et al., 2014] and the directive is open but the compiler for this directive is under commercial license, which reduces its impact on the scientific community compared to CUDA or OpenCL. In addition to that, GPU Libraries can assist developers to speed up there applications since many established CPU libraries have their equivalents for GPU such as CUFFT [†], CUBLAS [‡] and CUSPARSE [§].

1.3 Research objectives

Many algorithms are nowadays already intended to run in parallel when developed [Dongarra, 2016] but running an algorithm efficiently in a massively parallel device such as the GPU is still challenging. For CFD applications, speedups ranging from one to two orders of magnitude or even beyond are reported in the literature [Niemeyer and Sung, 2014b]. Different speedups are also reported for design optimization, which will be detailed in Chapter 3. At the same time, these accelerations are severely criticized by some researchers [Lee et al., 2010; Vuduc et al., 2010] highlighting a set of limitations for the GPU and judging the reported large speedups as artificially inflated by choosing an underoptimized CPU reference code.

Under these discordances of case-dependent and sometimes contradicting speedups, it is difficult to estimate a plausible GPU acceleration for a new application. The objective of this doctoral thesis is, therefore, to make the GPU potential more

[†] `http://docs.nvidia.com/cuda/cufft`,accessed 6/2017
[‡] `http://docs.nvidia.com/cuda/cublas`,accessed 6/2017
[§] `http://docs.nvidia.com/cuda/cusparse`,accessed 6/2017

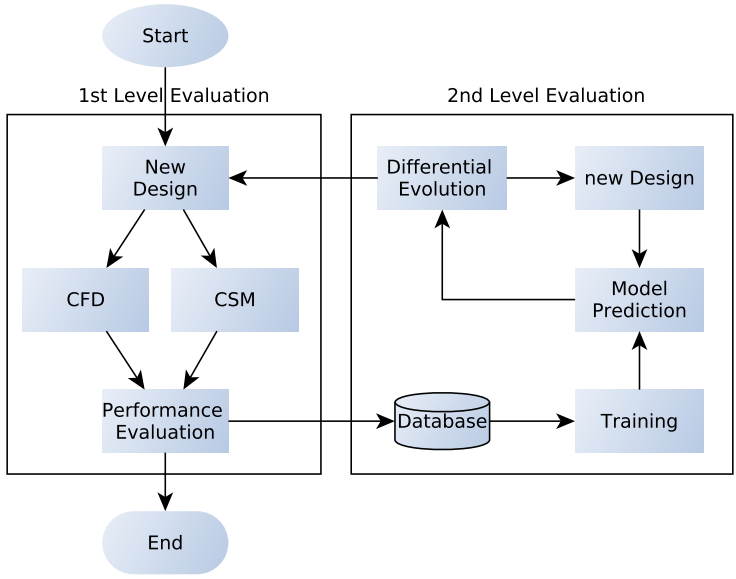


Figure 1.4: *Chart of the 2-level optimization in-house tool CADO [Verstraete, 2010].*

transparent to the design optimization community and CFD users in general. A clearer perception of the GPU potential can help to make an educated-decision about whether porting a simulation or not to the GPU. The scope of the study is limited to steady CFD simulations solving RANS equations on structured meshes within a population-based design optimization framework for turbomachinery applications.

Research activities in design optimization at the Von Karman Institute (VKI) have been started more than 15 years ago. Several innovative algorithms have been developed since, and are currently being used within the design of turbomachinery components in various applications grouped in a software package called CADO [Verstraete, 2010] with its algorithm sketched in Figure 1.4. A key feature of the developed multidisciplinary approach is the use of high fidelity CFD (fluid) and CSM (solid) computations, which constitutes the largest cost of the methodology. A fast interpolation method such as Kriging is extensively used in the 2nd level evaluation to help explore the design space.

The main objective of having a more tangible GPU potential is tackled within 4 steps:

- First, a literature survey reported on the main use of the GPU in multi-disciplinary design optimizations. The aim is to identify the type of operations delegated to the GPU and to answer the question whether the GPU is used as a workhorse for single-field simulations or as a coordinator of entire optimization algorithms.

- Second, a detailed performance assessment of GPU CFD solvers is performed with the aim of classifying operations based on exposed parallelism and reached acceleration. For that purpose, two RANS solvers have been ported to the GPU: the first with explicit time-stepping and the second with implicit time-stepping. The profiler-guided code optimization led to the identification of few speedup categories for some CFD operations differentiating between stencil-based operation and other operations such as linear system solver iterations and sparse matrix factorization.
- Motivated by the results of the second step, a third step consists of comparing the explicit to the implicit solver. First a comparison for one flow iteration has been delivered and then a generalization has been intended for a whole simulation by introducing a convergence ratio relating the number of flow iterations of both solvers (explicit and implicit) to reach a stationary solution resulting into a performance model.
- The last objective concerns the proof-of-concept for a design optimization in a high-end computer with a GPU accelerator. Two test cases are optimized, a compressor and turbine cascade, with the application of the new introduced performance model to identify the fastest alternative (explicit/implicit on GPU/CPU).

1.4 Context and outline of the thesis

The work for the present PhD thesis has been started within the EU-funded Project: AMEDEO [†] (Aerospace Multidisciplinarity Enabling DEsign Optimisation). The overview of the research conducted in this work is summarized in Figure 1.5 with the problem formulation, the research items and the outcome being linked to the outline of the document.

The GPU has a large computational power which may be very useful for many researchers working in MDO as they rely on time-consuming simulations. Chapter 2 brings the necessary details to situate the GPU within the established HPC architectures. Moreover, it shows the profiler-driven code optimization approach followed in this work to increase the performance of the implemented GPU functions.

Chapter 3 presents a literature survey on the use of the GPU for design optimization. The results of many researchers are reported while observing that the GPU has been used mostly as an accelerator for a single-field simulation (e.g. CFD or CSM) and not used to incorporate the complexity of MDO algorithms. For a single discipline, the literature reports speedups of one to two orders of magnitude, while speedups as a key measure of the GPU performance face some criticism [Lee et al., 2010]. For structural analysis, the GPU has been used to solve the system of equations created by the coupled Partial Differential Equations (PDEs) governing the structural analysis. For the computational fluid analysis, the GPU is very

[†] `http://www.amedeo-itn.eu/`

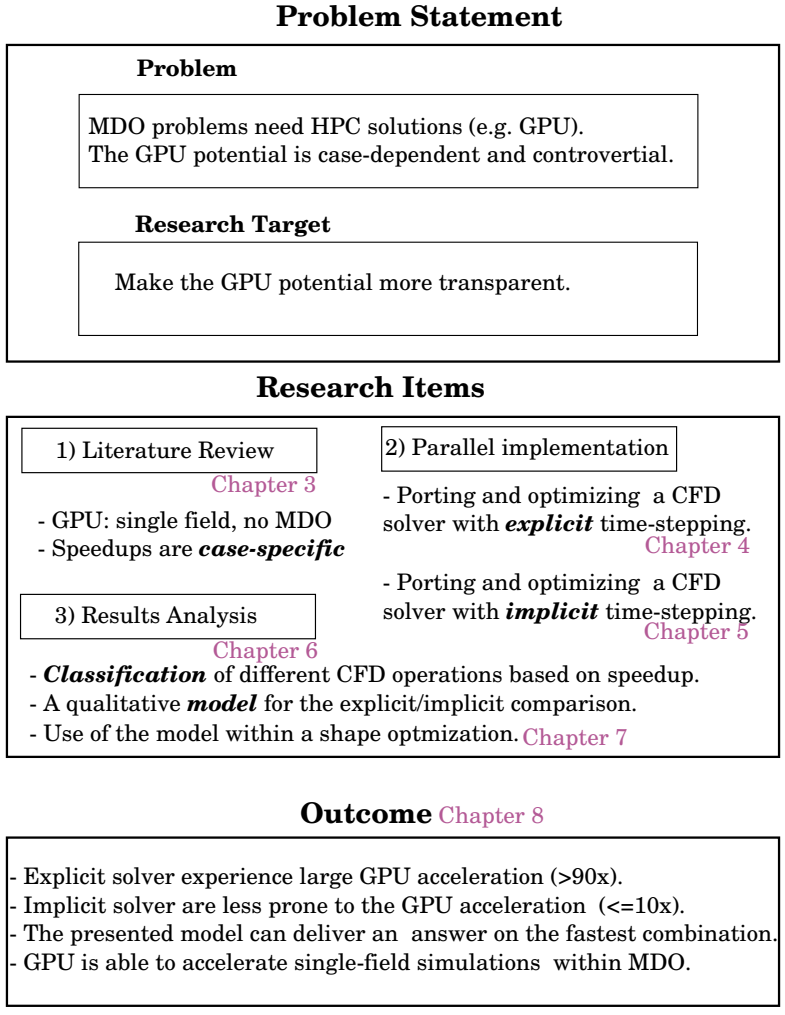


Figure 1.5: *Structure of the work through the thesis.*

efficient in running particle-based algorithms [Rinaldi et al., 2012; Zhao, 2008], less for structured meshes within the Finite Volume (FV) scheme and much less with unstructured meshes [Niemeyer and Sung, 2014b].

In order to assess more closely the effect of the GPU on single disciplines such as the CFD, an in-house CFD code is ported to run on the GPU. Chapter 4 and 5 present two GPU-based RANS solvers with different time-stepping methods. The first has an explicit time-stepping and experiences two orders of magnitude acceleration on the GPU. The second has an implicit time-stepping and a one order of magnitude acceleration is obtained. The numerical results have been verified against the experimental ones revealing a fair matching that guarantees the accuracy of the new solvers on subsonic and transonic conditions. The code optimization for both solvers led to some key observations on the profiler-driven code optimization for the GPU performance that are presented in Chapter 2. This latter chapter is intended to introduce the concepts used in the analysis of the different implementations for the functions used in both solvers.

Built in a modular way, the CADO optimization software allows to interact with different CFD solvers. The GPU acceleration benefited CADO simply by replacing the CPU-based solver by the new GPU-based one. Speed but also stability of CFD solvers are relevant for the MDO optimization, therefore a comparison of both solvers have been performed concerning speedup, stability, and convergence (see Chapter 6). First, the acceleration per iteration has been observed to be clearly advantaging explicit solvers. Secondly, the convergence rates, which are largely in favor of the implicit solver, have been incorporated in the comparison. Both elements are used to build a model, which selects the fastest alternative for a given convergence ratio.

In Chapter 7, the GPU-enhanced optimizer has been tested on a supersonic compressor cascade achieving an improvement of 25% in the entropy generation and the entire optimization process has been 90 times faster on the GPU than on a single core CPU. Moreover, the performance of a 1-level and a 2-level optimization algorithms has been compared for the shape optimization of a turbine cascade. The 2-level method consists of a metamodel assisted optimization using Kriging interpolation. The newly introduced performance model served on identifying the fastest CFD solver to be used in the optimization.

2

Graphics Processing Units for High Performance Computing

Graphics Processing Units (GPUs) originated from the performance-driven video gaming industry and were used as rendering pipelines with the main task of creating 2D representations of 3D-scenes. This implies a large number of floating-point operations per second (FLOPS) for a single video frame. The GPU evolved consequently to devote most of the device capacities to computing rather than sophisticated control logic and memory caching. This chapter treats first some key aspects of the GPU programming, which are essential to understanding the strong and weak points of the GPU and highlights then the performance-driven code optimization for the GPU.

2.1 Introduction

Current GPUs are multi-threaded processors which can be programmed with a C-like language called CUDA [†]. They are built over a large number of scalar processors provided by a large memory bandwidth. The common trend in the different NVIDIA modern GPU generations (Fermi, Kepler Maxwell and Pascal) is the gradual increase of the memory bandwidth and the computational power while keeping a similar power consumption. Motivated by the large performance advantage of the Kepler card compared to Fermi[‡], this work is mainly based on the Kepler architecture using the NVIDIA K40c and Geforce GTX 780.

A serial code can be parallelized to run on a GPU by rewriting it using a low-level language (CUDA or OpenCL), using GPU libraries (e.g. thrust, cuSolver etc[§]) or by using a compiler directive approach such as openACC. The openACC approach delivers easily small improvements but a substantial speedup requires a deep understanding of the low-level workflow of the ported code and at the same time an extensive knowledge of the openACC standard [Rueda et al., 2016]. Low-level

[†] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, retrieved March 2017

[‡] Newer architectures were not available at the start of the PhD, 2013

[§] `https://developer.nvidia.com/gpu-accelerated-libraries`, last accessed June 2017

changes are easier and more intuitive to implement with a low-level language rather than through a standardized compiler directive-based alternative. This work, therefore, combines CUDA with some GPU libraries such as `thrust` [Bell and Hoberock, 2011], Paralution[†] and CUSP [Bell and Garland, 2015].

This chapter focuses on CUDA C as it reveals the low-level code transformation essential to reach interesting speedups but the conclusions should also apply to OpenCL since both languages are fairly comparable [Karimi et al., 2010; Fang et al., 2011]. Section 2 presents some aspects of the history of GPUs in order to better understand the architectural design of modern GPUs, which is presented in Section 3. Then, the execution model of the programming language CUDA is shortly introduced in Section 4. After these prerequisites, the last two sections present the memory-related and the profiler-driven optimization of the GPU code.

2.2 From graphics pipelines to High Performance Computing

The predecessors of today GPUs have been developed in the early 1980's [Kirk and Wen-mei, 2013]. The primary idea was to offload the CPU from rendering images on the display and use a dedicated hardware instead. Image rendering is about creating surfaces defined by vertices, which belong to a polygon (e.g triangles), and use them to color the pixels of the final image that will be sent to the display. The first GPU was called fixed-function graphics pipeline since it had the hardware to rapidly compute vertices and pixel colors. Vertices have a color that is computed by the *vertex shader* involving neighbor vertices and rather intensive memory operations. The next stage, the *pixel shader*, creates an image by giving a color to every pixel based on its surrounding vertices. Features intended to improve the images and make them look realistic and sophisticated are performed at this stage, which performs thus many arithmetic computations using few vertices. Every stage has its specialized hardware optimized for fast memory access for the vertex shader and for fast arithmetic operations for the pixel shader. The graphics pipeline has many more stages, which are detailed in dedicated textbooks such by Kirk and Wen-mei [2013].

There has been a drive towards faster and more realistic rendering of 3D scenes in the gaming industry, which obviously led to a high demand for computational throughput. More sophisticated APIs [‡] were created to match the increased complexity of the pipeline such as directX and openGL. Following the developers demand for more flexibility required to cover a wider range of graphics algorithms, the vertices and then the pixel shader have been exposed for programmers. Many other stages (e.g rastering) from the graphics pipeline remained fixed functions.

A load-balancing problem occurs as small surfaces (e.g. triangles) require much more vertex treatment than pixel treatment, while large surfaces offer the opposite workload. The unification of both processors for vertex and pixels, first introduced

[†]PARALUTION Labs "PARALUTION v1.1.0", 2016, `http://www.paralution.com`

[‡]Application Programming Interfaces are a standardized set of routines that enable an application to use a hardware.

by AMD in the Xbox360 [Andrews and Baker, 2006], solved this load-balancing problem. The unification of different stages processors had also the benefit of reducing the development cost of 2 different processors.

General Purpose GPU computing (GPGPU) emerged from 2000 to 2006, when scientists started using the raw computational power of GPUs for numerical problems in different fields. However, they had to adapt their algorithms to the partially programmable GPU. The vertex shader accepts one input in form of a texture image containing vertices coordinates and the only output is a pixel color from the pixel shader. Some researchers managed to get a remarkable acceleration due to the GPGPU as reported in the survey of Owens et al. [2007]. The speedup required, however, an important programming effort as the developer had to use the graphics algorithms and terminology, which appeared to be too restrictive for numeric intensive algorithms.

The real start of GPU computing is related to the release of Tesla processors in 2006 [Lindholm et al., 2008]. Tesla has fully programmable unified processors in addition to the cache memory and load/store units enabling it to handle compiled code. NVIDIA provided also the compiler, the library and the API to make the Tesla card much easier to use for non-graphics applications. The following generation of GPUs (Fermi) has almost no remaining heritage from the original graphics pipeline.

The Fermi card is built around an array of highly threaded streaming multiprocessors (SMs). Figure 2.1 depicts a Fermi SM with 32 CUDA cores (with its arithmetic logic unit), 16 load/store units, 2 schedulers and dispatch units, 4 special function units (responsible for the computation of sine, cosine, etc.), register files and an L1 cache/shared memory. The number of load/store units determines the number of threads that can be served concurrently per clock cycle, here half a warp with a warp defined as a group of 32 parallel threads. With its 2 warp schedulers the SM can execute up to 2 warps at the same time. The Fermi GPU contains 16 SMs, as depicted in figure 2.2, reaching a total of 512 CUDA cores and can run up to 32 warps concurrently. It has also 6 DRAM memory interfaces, which can connect to at most 6 GB of global device memory. The GPU is connected to the host CPU through a PCI express bus. The large number of threads is managed by the *GigaThread* engine that acts as a global scheduler. The shared-memory and L1-cache are very important for a substantial performance gain for some applications.

The GPU Kepler generation, released in 2012, outperforms the Fermi architecture

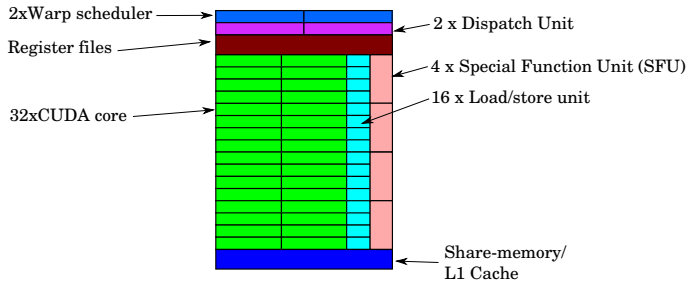


Figure 2.1: *Block diagram of one SM of a FERMI GPU card of NVIDIA.*

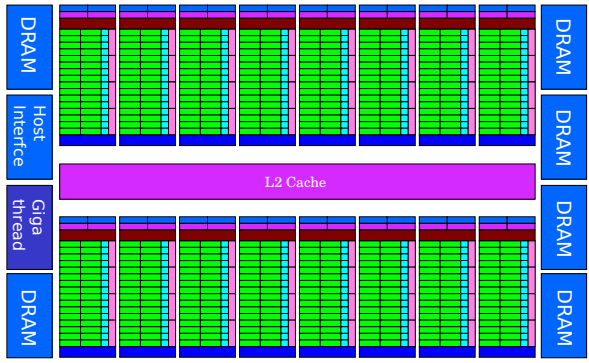


Figure 2.2: *Block diagram of a FERMI GPU card of NVIDIA.*

by increasing the number of CUDA cores up to 2880, doubling the precision units and cache size and introducing some new features such as dynamic parallelism and hyper-Q. Dynamic parallelism is about giving a thread the possibility to start a set of sub-threads and is adapted to a multi-level parallelism such as recursive functions. The Hyper-Q allows multiple kernels to run concurrently providing 32 hardware channels for the host to place kernels. As it is a hardware feature no programming is required to take advantage of Hyper-Q. The contribution of this feature can be observed by the profiler as more kernels can overlap with Kepler cards than with the Fermi cards. The card has 15 SMs (less than the Fermi card) but every streaming multiprocessor has been enhanced to host 192 CUDA cores and 64 double precision units. The register files have double size and Kepler has 4 warp schedulers.

2.3 GPUs: a throughput-oriented latency-tolerant HPC device

There is a performance gap between CPUs and GPUs in terms of memory bandwidth (amount of data transferred per second) and arithmetic throughput (FLOPS: FLoating OPerations per Second). This is related to the architectural differences between both devices (see Figure 2.3). The conventional multi-core CPU relies mainly on a large and fast memory cache, which serves a small number of cores, in order to reduce the instructions execution time (latency) to a minimum. A significant portion of the transistors is dedicated to the instruction flow control (instruction pipelining, branch prediction [†] and similar tasks). The CPU cache has a spacial and a temporal locality as data is more likely to be kept in the cache if it is often used (temporal) and if it is close to an often used data (spatial). Neither the large cache memory nor the sophisticated instruction control contributes to the computational power [Kirk and Wen-mei, 2013]. For a CPU, the computational power is provided by a high-speed processor optimized for serial operations and low latency. Moore's law predicted the doubling of the number of transistors in an integrated circuit every

[†]A digital circuit to predict the path that will be followed in a branching (e.g. if-else)

2 years [Moore, 1998]. This trend has been carried on until the energy consumption and the heat dissipation reached very high values, known as the power wall [Patterson and Hennessy, 2013, p. 40]. The increase of the clock frequency slowed down also as it is related to the energy consumption and the heat dissipation of the CPU too [Bergman et al., 2008, p.88]. These physical limits caused very powerful single cores to be abandoned in favor of multi-core technology, in which multiple cores are contributing to the computational power of a device.

The GPU, on the other hand, has been designed from the start on as a multi-core technology and disposes of a large number of lightweight processors able to perform single and double-precision computations. The GPU cache is, however, much smaller and disposes only of spatial locality. The gap between CPUs and GPUs concerns also the memory bandwidth. CPUs need in fact to "satisfy requirements from the operating system and the I/O devices which make the memory bandwidth more difficult to increase" [Kirk and Wen-mei, 2013, p.4].

2.4 CUDA: a programming language and an execution model

This section explains some concepts of the GPU programming related to the CUDA execution model. An extensive and detailed treatment of the topic can be found in the CUDA user manual[†] and some valuable textbooks [Kirk and Wen-mei, 2013; Cheng et al., 2014; Sanders and Kandrot, 2010; Farber, 2011].

CUDA C is based on the C programming language with a minimal set of extensions to handle the parallel execution and the memory organization. GPUs execute as a device (an accelerator) for a host application on a host CPU, therefore, a CUDA program contains host code and device code. The functions running on the GPU are called kernels and are executed by threads, which are organized in grids of blocks distributed among the streaming multiprocessors. CUDA kernels are launched as follows:

```
kernel_name<<<Nb,Nt>>>(var1,var2, ..),
```

[†] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, retrieved June 2017

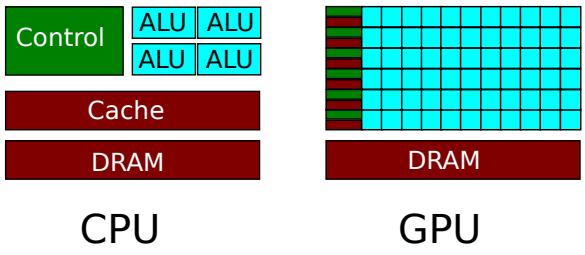


Figure 2.3: *The architectural design difference between the CPU and the GPU use of transistors.*

with `Nb` the number of thread blocks launched and `Nt` the number of threads per block. A kernel is usually executed by thousands of threads ($Nb \times Nt$); though a run with one block of one thread ($Nb = 1, Nt = 1$) is equivalent to a serial run on a CPU and can serve as a verification of a written kernel. The GPU starts threads always as a multiple of the warp size. Therefore, the number of requested threads per block should be a multiple of 32 otherwise, the last warp is not fully used and its extra threads are inactive but consume, nevertheless, registers and shared memory space.

Threads have a unique index accessed over `threadIdx`, a built-in† variable. Consecutive indices in `threadIdx` are handled in groups of 32 threads constituting warps. In a very common CUDA programming pattern, the grid of threads replaces a sequential loop (see Listing 2.1) over a large set of elements. For every thread, the index of its grid and its block are combined to create a unique global identifier as seen in Listing 2.2. Every thread treats a small subset of elements and the programmer should take care of a proper use of the index inside a kernel. Kernel calls are asynchronous since the control is returned to the host CPU immediately after the call, which can then proceed to the next call. A synchronization or a memory copy can force the CPU to wait for the completion of the last called kernel.

CUDA uses the SIMT (single instruction multiple threads) paradigm to execute instructions within a warp. The same instruction is broadcasted to all threads of a warp for execution but every thread has its own registers and instruction counters. A data dependent conditional branching (e.g. IF-ELSE) can cause threads in the same warp to face different execution paths. The execution is consequently serialized by running first the instruction of the threads of the first path then the instruction of the second path as depicted in Figure 2.4. The possibility of divergence within a warp, called *thread divergence*, differentiates SIMT from SIMD (Single Instruction Multiple data) processors, which impose strictly the same instruction on a vector of data. Even though it is tolerated, *thread divergence* is very damaging to the performance since GPUs do not offer the same level of complex branch prediction as CPUs. A fully diverged warp can run up to 32 times slower than a divergence-free warp. Therefore, it is crucial to measure the degree of warp divergence expressed as Branch Efficiency (E_B), which is defined as the ratio of the number of non-divergent branches to total number of branches (N_B^{Total}) [Cheng et al., 2014]:

†pre-initialized variable by the runtime system

Listing 2.1: *C/C++ code for the loop of the matrix addition.*

```
int i,j,idx;
for (i=0; i<N;i++){
   for (j=0; j<N;j++){
     idx=i+ j*N;
     C[idx]=A[idx]+B[idx];
   }
}
```

Listing 2.2: *CUDA code equivalent of the loop for the matrix addition.*

```
int i=blockIdx.x*blockDim.x + threadIdx.x;
int j=blockIdx.y*blockDim.y + threadIdx.y;
int idx= i+j*N;
if(i<N && j<N){
   C[idx]=A[idx]+B[idx];
}
```

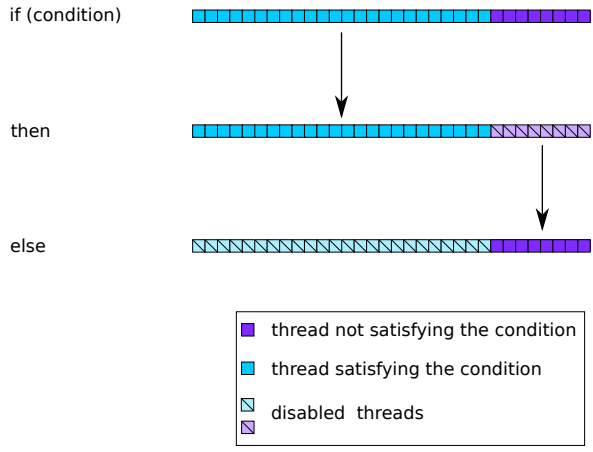


Figure 2.4: *Serial execution of diverged paths within a warp caused by a conditional statement (if-else).*

$$E_B = 100 \cdot \frac{N_B^{\text{Total}} - N_B^{\text{diverged}}}{N_B^{\text{Total}}} \,. \tag{2.1}$$

Multiple threads can access the same memory position in a non-controlled fashion which causes an unpredictable program behavior known as *race condition*. The threads execution order can be controlled by a threads synchronization within a block (`__syncthreads()`). The synchronization is about inserting a barrier that every thread has to reach before the kernel executes the next instruction. The synchronization of all threads of all blocks requires a call from the host (`cudaThreadSynchronize()`), which is not possible within a kernel.

The fact that thread blocks are independent reduces the ability for inter-block communication but it is essential to keep a transparent scalability between thread blocks. Transparent scalability means, in this context, that doubling the number of streaming processors will half the execution time as shown in the example depicted in Figure 2.5.

The organization of threads in warps of 32 threads enables to dispatch an idling warp (e.g. waiting for memory access) and schedule a new warp residing in the same SM (see Figure 2.6). This thread swap, called *context switch*, increases the instruction throughput and hides the waiting time for memory loads (memory latency). Unlike the CPU context switch, the GPU context switch is not expensive. The primary difference is that GPU warps have their resources already assigned during the kernel launch and the switch is purely done by the hardware. A CPU needs to copy the state of one thread before loading the state of the new one when switching context. For the GPU, the gain of the context switch is done at the cost of limiting the maximum number of active warps per SM, which cannot exceed the SM resources divided by the single thread consumption. When launching a kernel the threads in a block are sharing the registers (16k/SM for architecture 3.5) and the shared-memory (up to 48kB/SM for architecture 3.5). These are, in general, the limits dictating the number of warps active in the different SMs and is called the theoretical occupancy, which is a measure for the level of thread parallelism.

Warps with secured resources from the SM are called active. But physically active are only the selected warps by the warp scheduler. The non-selected warps could be either stalled, waiting for an argument[†], or eligible if they are ready for execution but the hardware is busy. From the active warps, a maximum of 4 can be selected concurrently per SM (for architecture 3.5) and if a warp is stalled the scheduler picks up one of the eligible warps. If a kernel is very demanding of shared-memory such that not even one single block of threads can be assigned to an SM, the kernel execution fails without terminating the host program. CUDA provides thus a function, `cudaGetLastError()`, to check the status of the last executed call. In the development phase, it is recommended to check the status of every CUDA command by calling the error check function or providing a macro as a safe call wrapper (see CUDA SDK[‡]). A kernel can fail because of a lack of memory resources or an inappropriate run configuration in terms of number of blocks and grids, while a memory allocation can fail because of a lack of global memory.

[†] a memory transaction or a computation

[‡] `https://developer.nvidia.com/cuda-downloads`, retrieved June 2017

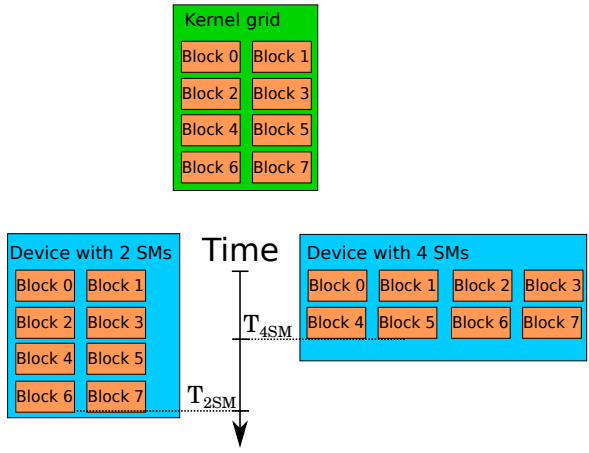


Figure 2.5: *A comparison of the execution time of 8 thread blocks in a GPU with 2 SMs and a GPU with 4 SMs.*

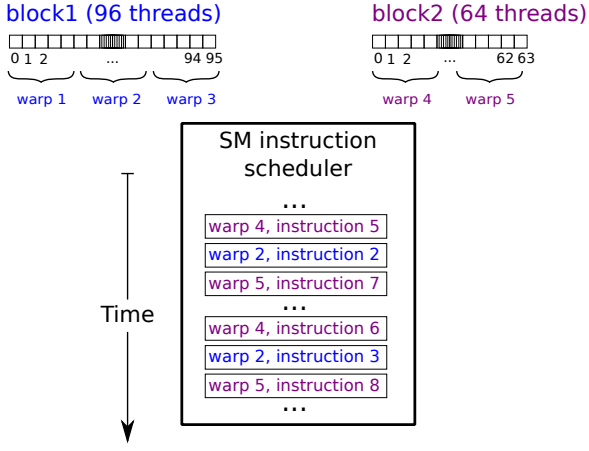


Figure 2.6: *Example of the work done by a scheduler while executing warp instructions.*

Warp switching is used to hide the instruction latency, which is defined as the required amount of time from dispatching an instruction until its completion. The arithmetic latency lasting between 10 to 20 cycles is smaller than the memory latency which can reach up to 800 cycles [Cheng et al., 2014]. The estimated number of independent instructions N to hide the latency λ is defined according to Little's law as follows:

$$N = \lambda \cdot \text{throughput}. \tag{2.2}$$

In order to reach the required number of independent instructions N, it is possible to increase the number of warps or/and increase the number of instructions per thread. The first technique is called Thread-Level Parallelism (TLP) and the second is known under Instruction-Level Parallelism (ILP). For memory operations, the throughput is defined based on the clock rate and the memory bandwidth as follows:

$$\text{bandwidth/Clock} = \text{throughput}. \tag{2.3}$$

The throughput, in general in kB, refers to the in-flight I/O memory operations required to saturate the GPU. For a small N, only few memory operations are started, which deteriorates the reached bandwidth (more in Section 4.2.1).

Until now only the fine-grained data parallelism through a grid of threads has been addressed. The GPU offers also a coarse-grained kernel (or task) parallelism by running multiple grids of threads concurrently using multistreams (see Listing 2.3) increasing the number of active warps and consequently the GPU utilization. All kernels and CUDA API calls, seen above, run in fact in a default stream with a stream defined as a sequence of operations executing in the order as issued from the host CPU. At the same time, multistreaming can be used to overlap data transfer and computation.

If a stream synchronization occurs or a kernel is executed by the default stream, the kernels overlapping is interrupted. Indeed, when the default stream executes a kernel or a memory call, all other kernels are in a halt. The hardware resources can also limit the number of concurrent kernels as depicted in Figure 2.7. The GPU keeps running more streams until all resources are assigned to kernels from already launched streams. No additional kernel is able to run from the rest of the streams until the first set of kernels finishes executing.

Fermi cards have only one hardware work queue, through which all launched kernels on all streams need to transit. Filling one stream after the other with kernels and API calls (*depth-first* approach [Cheng et al., 2014, p. 282]) creates a false dependency in Fermi processors. Only the last kernel of a stream and the first kernel of the following stream are independent and thus can overlap (see Figure 2.8). A software approach to solve the false-dependency problem is to make a *breadth-first* filling of streaming: every stream should have received a command to execute

Listing 2.3: *Example of a use of streams with kernels.*

```
for(i=0;i<Nstreams;i++){
    cudaStreamCreate(streamArray[i]);
    kernel_name<<<threads,blocks,0,streamArray[i]>>>(varibales)
}
```

before any stream gets a second command. This increases the kernels overlapping on Fermi cards to its maximum. NVIDIA proposed a hardware fix for this issue on the Kepler cards by a mechanism called *HYPER-Q*. Hyper-Q allows up to 32 hardware work queues[†] and thus any case with false-dependency on Fermi cards runs with maximum overlapping on Kepler GPUs. However, breadth-first approach is always advisable. The multistreaming feature is found to be crucial in order to reach interesting performance gain on the GPU (Cf. Section 4.2)

2.5 Memory hierarchy of the GPU

The memory hierarchy for the CPU-GPU system proposes different levels of speed and size for every type of memory (see Figure 2.9). The capacity increases from register and cache memory to global and system memory but so does the latency. An efficient use of the available memory levels can leverage the performance of a program to reach a maximum, which is dictated by the hardware limitations. No instruction can be executed, for instance, if a warp is stalled waiting for a variable to be loaded while all load/store units are busy. Therefore, the instruction-level parallelism can not be very beneficial if a kernel is not making good use of the possible memory types.

In general, the system memory residing in the CPU contains the entire data of the simulation. The data that will be needed by the GPU should then be moved to the device global memory which is of the order of few GBs (e.g. Tesla K40 has 12 GB) for a single GPU. This implies that the GPU could solve a problem slice by slice with every slice not exceeding the global memory capacities. Since the PCI bus is relatively slow, the communication between the host CPU and the device GPU has to be kept minimal and small data transfers are faster when grouped together and moved in one shot. Moreover new technology such as NVlink[‡] can further leverage this issue. The data residing in the global memory has higher access latency than the data residing in a cache or a register. Moreover, the access itself to the global memory

[†]The number of hardware queues can be accessed/changed in the environment variable `CUDA_DEVICE_MAX_CONNECTIONS`

[‡]more than 80Gb/s memory bandwidth `http://www.nvidia.com/object/nvlink.html`

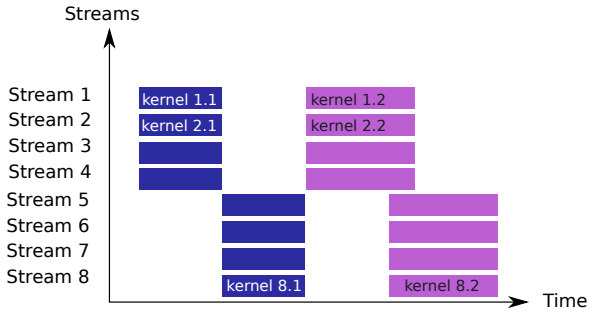


Figure 2.7: *Illustrative time line for the execution of 16 kernels distributed over 8 streams with the SM resources limiting the concurrency to 4 kernels.*

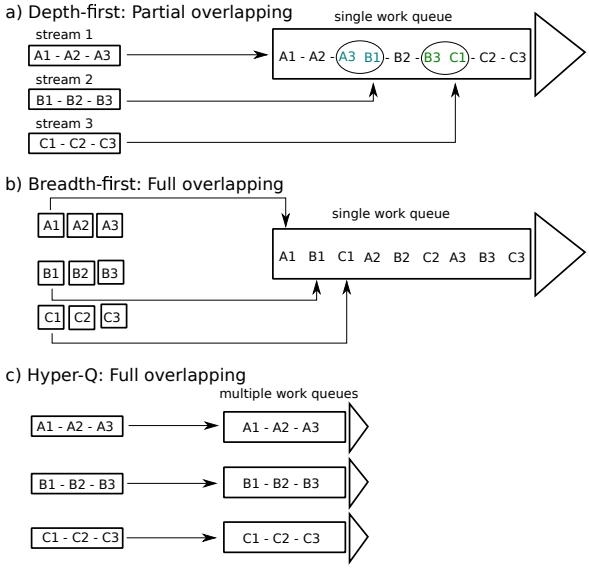


Figure 2.8: *Illustration of the multistream execution of a set of kernels highlighting the amount of overlapping with (a) depth first approach, (b) breadth-first and (c) Hyper-Q.*

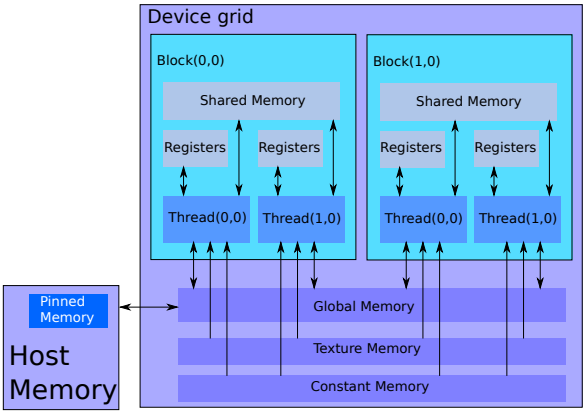


Figure 2.9: *Block diagram of the host and device software memory organization.*

should follow certain patterns to guarantee the best memory bandwidth. The GPU loads an entire word[†], when a thread accesses a memory position, and the loaded word is broadcasted to all threads of the warp. If the next thread within a warp accesses the next memory position, it is very likely that it is in the already loaded word. The memory transactions of the two threads are grouped or *coalesced* into one transaction. The line size of the L2-cache, by which all global memory accesses are migrating, is 128 bytes, which is optimally used if every thread in a warp requests consecutive 4 bytes. A second key technique is the alignment, which is guaranteed for requested addresses being a multiple of the cache line size. Figure 2.11 depicts first a case of an ideal access then two cases of misaligned and uncoalesced accesses. The alignment makes sure that the minimum number of words is loaded per warp. The misalignment can cause two words to be loaded for one access, while the uncoalesced access can load up to 32 words for one access depending on the scattering of the memory accesses within a warp.

The number of transactions required for a single memory load depends on the coalescence of the access and the data alignment of the transactions [Cheng et al., 2014]. The profiler can compute the average number of global memory load transactions performed for each global memory load. A practical case of storage in global memory is to transform a multi-dimensional array into one-dimensional long array. In a case of a 3D array accessed by the indices i, j and k with maximum values set to IMAX, JMAX and KMAX respectively, the index is as follows:

```
array[i+j*iMAX+k*iMAX*jMAX]=array[i][j][k];
```

The case of mapping the GPU threads to i, j and k is considered (see Listing 2.4). In order to keep the coalesced access, the thread mapping should follow the 1D array indexing with iMax, jMax and kMax set equal to IMAX, JMAX and KMAX respectively. This means a kernel should alter an uninterrupted set of data. In case a kernel is updating only selected parts of the data with some interruptions (see Figure 2.10 for an example), the thread partitioning does not map fully the storage layout and some warps will have uncoalesced access as a jump occurs in the memory accesses. The fact that some data is not used can affect also the alignment since the first accessed memory position is not guaranteed to be a multiple of the L2 cache line size. In that case, the global memory storage/load efficiency decreases from the ideal 100% (for loading/storing a double precision variable). The number of instructions per thread can improve the throughput, even if the added instruction provides uncoalesced accesses. The weight of one code optimization technique against another

[†]A word is a piece of data with a fixed-size managed as a unit by processor

Listing 2.4: *An example [Delbosc, 2014] of mapping of threads to 3 iterators ij and k.*

```
index1D=blockIdx.y * (gridDim.x * blockDim.x) + blockDim.x * blockIdx.x
     + threadIdx.x;
k= index1D /(IMAX*JMAX)    //integer devision
j= (index1D-k(IMAX*JMAX))/IMAX
i= (index1D-k(IMAX*JMAX))-j*IMAX
```

is thus case-dependent.

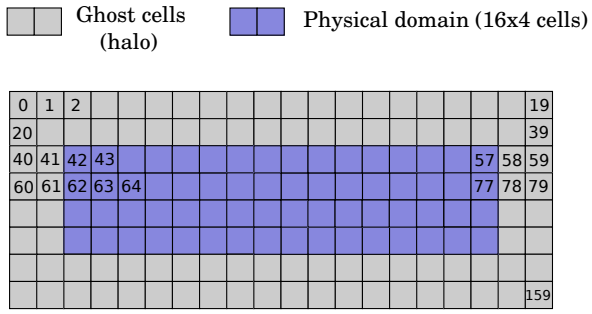
The use of registers is largely controlled by the compiler and the user can put a limit on the maximum amount of registers in general or for a certain kernel. The rule of thump is that all variables are stored in registers except large arrays. Registers are a fast-access memory but they are also limited for an SM and their overuse by a kernel limits its theoretical occupancy. If a kernel uses more registers than allowed by the hardware (63 for Fermi and 255 for Kepler GPUs) the excess is moved to the local memory, which is local to the thread but resides in the GPU slow global memory. This is called *spill-over* and should be avoided in general. The data in local memory is cached in L1 per SM and L2 per device. The L1 and L2 caches are rather hidden from the user and the accessible caches are texture and constant memory. Constant memory, for instance, is adapted to store read-only variables that are required by all threads in a warp. From practical experience, it has been observed in this work that the L1-cache combined with compiler optimization makes the effect of the use of constant memory negligible for CFD applications.

The shared memory can be used to reduce the use of the global memory by loading a subset of the data. The threads in a block can work on this sub-data and load it back to the global memory at the end of the procedure. For more details to shared memory, consider the CUDA user manual and some textbooks ([Cheng et al., 2014, p. 203],[Kirk and Wen-mei, 2013, p. 95])

2.6 Profiler-Driven code optimization

Occupancy is a measure of the performance defined as the portion of active warps from the maximum number of supported warps. The occupancy is linked to the number of registers used per kernel, the amount of shared-memory and the run configuration (number of threads per block). The more is a kernel demanding the lower its theoretical occupancy. The achieved occupancy, on the other hand, depends on the number of started threads. The number of threads per block can be very important in some cases, as the maximum number of allowed blocks can keep the SM resources underutilized if these blocks of threads are very small. On the other hand, large blocks consume much more resources reducing the available resources per thread [Cheng et al., 2014]. In case enough threads are started, the kernel should reach its maximum occupancy unless a thread divergence is occurring or the workload is not balanced between threads.

Occupancy is not the unique performance metric and in some cases maximizing the occupancy is not improving or even damaging the GPU performance as it is the case for loop unrolling [Volkov, 2010]. Loop unrolling is about changing a loop content to increase the work done at every iteration and reduce the total number of iterations (see Listing 2.5). The register consumption of an unrolled loop increases affecting negatively the theoretical occupancy and inevitably the achieved occupancy also. Surprisingly the performance of unrolled loops is improved. For a serial execution, loop unrolling has no effect on the performance since the total number of instructions is the same. Whereas for a vectorized or a massively parallel execution, it improves the performance by creating more independent instructions. Consequently, the instruction-level parallelism is improved leading to a better uti-



Kernel A updates only Physical domain:

Warp	Accessed Memory	Memory Transactions	Efficiency
warp 1	42 to 77	2x: 33 to 64 and 65 to 96	50%
warp 2	82 to 97	2x: 65 to 96 and 97 to 128	50%

Kernel B update the entire domain (halo included):

Warp	Accessed Memory	Memory Transactions	Efficiency
Warp 1	0 to 31	1x: 0 to 31	100%
Warp 2	32 to 63	1x: 32 to 63	100%
...	...	....	...
Warp 5	128 to 159	1x: 128 to 159	100%

Figure 2.10: *An example of the memory access of 2 kernels on a physical domain of 16x4 cells with two layers of ghost cells: kernel A accesses only physical cells and kernel B accesses all the domain.*

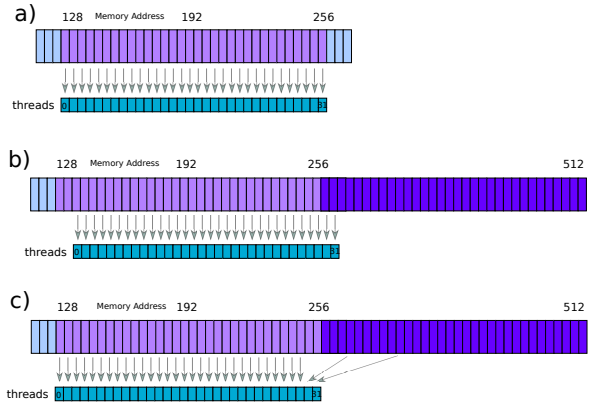


Figure 2.11: *An example of a) aligned coalesced access, b) misaligned coalesced and c) aligned uncoalesced.*

Listing 2.5: *Example of loop unrolling.*

```
for (i=0; i<N; i+=2){
    A[i] = ...;
    A[i+1]= ..;
}
```

lization of the compute and memory resources of the GPU by providing more eligible warps for the scheduler. In many cases, the `nvcc` compiler can figure out automatically how to unroll loops, if the instructions in every iteration are independent. Loop unrolling as defined above is useful only if the instruction independence is not explicit on compile time.

Rather than considering only the occupancy, the code optimization should take into account the reached memory bandwidth and the instruction throughput. This is done by using the profiler to check the memory access pattern, the thread divergence and the compute and L2-cache utilization. Profiling is mainly about measuring the execution time of functions. The NVIDIA profiler has the basic features of a common profiler but can also compute a large set of performance parameters related to the memory use, number of executed instructions and achieved occupancy in addition to many other performance related measures defined in the profiler manual[†]. It is an essential tool for understanding the time complexity of a program and rapidly reach better performance by tackling directly expensive kernels related to a larger overall effect. The profiler can help diagnose the source of a poor performance by measuring the quality of a memory access, the thread divergence and the reached memory bandwidth.

When targeting an improvement in a GPU program, the profiler should be used to rank the kernels following their execution time. Time-consuming kernels are identified as hotspots and an improvement on a hotspot has a more pronounced impact on the global performance. A time-consuming kernel should be first analyzed in order to identify the performance limiting factors. It could be limited by compute utilization, memory bandwidth or latency. The profiler can measure the utilization of the L2 cache (by which all load/stores are migrating) and the compute unit utilization. If both are similarly high the kernel is quite optimized, nevertheless, it is always recommended to further check thread divergence, global and shared-memory use. If both memory and compute units are similarly low the kernel is latency-bound and probably not launching enough warps to keep the SMs busy. In that case, the whole algorithm needs to be rethought in order to expose more parallelism and thus start more threads. It is always worthy to check if the run configuration is appropriate by measuring the level of activity of the different SMs. A small number of large blocks, for instance, could be not enough to have at least one block of threads per SM which will bring some of the SMs to be inactive.

For a memory-bound kernel, few profiler metrics are essential such as the throughput in [GB/s] for both the load and the store, which can be compared to the theoretical peak. The comparison to the theoretical peak is reflected by the L2 usage. The number of global transactions per load informs about the coalescence and the alignment of the memory access. The same information can be retrieved as a percentage of efficiency in both load and store. The efficiency reflects directly the access pattern as coalesced pattern for double precision load yields a 2 operations per load and an efficiency of 100%. For a coalesced integer loading, the efficiency could exceed 100%. The computational power is, on the other hand, important for compute-bound kernels. For those kernels, the throughput can be measured in terms of FLOPS and

[†]`http://docs.nvidia.com/cuda/profiler-users-guide/#metrics-reference-3x`,retrieved march 2017

compared to the theoretical value.

2.7 Conclusion

Parallel computing is about using a large number of processors to run concurrent computations. The GPU offers two types of parallelism: data and task parallelism. The data parallelism is about partitioning the data among threads that will execute ideally the same code. Task parallelism decomposes a large task in independent subtasks and executes them concurrently over multistreams.

GPU code optimization is about increasing the number of eligible warps and in-flight instructions and improving the coalescence of the memory access. A key step to reaching a good performance for a kernel on the GPU is to understand the mapping of threads into the GPU cores. Moreover, a GPU kernel should make good use of the instruction throughput and memory throughput both possible only when the low-level organization of the GPU is understood and assessed with a profiler.

In general, the data access starts in global memory with most HPC applications being memory-bound. Consequently improving the use of the global memory bandwidth is essential for performance tuning otherwise other optimization techniques will have a negligible impact. Registers, shared, and constant memory are fast access memories capable of accelerating an algorithm more effectively than global memory. However, an overuse of their reduced capacities can be a limiting factor for the occupancy in terms of the number of active threads.

This chapter made it clear that the GPU would not be able to accelerate every algorithm. Some algorithms will be adapted to its architecture and experience a large speedup while others can run even slower on the GPU. The next chapter, a review, addresses this issue for the algorithms used in the design optimization.

3

Literature Review: Use of the GPU in Design Optimization

Graphics Processing Units (GPUs) are a promising hardware for the acceleration of computationally demanding applications. These applications span from data mining to machine learning and life sciences. The field of design optimization benefited also from this new hardware. After introducing the GPU in the previous chapter and the way it works efficiently, this chapter reviews its use in design optimization.

The chapter provides an overview of the progress made in design optimization using GPUs and the challenging limitations mainly in topology optimization, shape optimization and multidisciplinary design optimization (MDO). It also identifies the role of the GPU as an accelerator for single-field simulations.

3.1 Introduction

Design engineers are interested in identifying rapidly a design with an optimal performance under specified constraints. This problem is easily formulated with the help of one or more objective functions that depend on typically a large number of design variables. The optimization process then requires finding a design from the design space, which minimizes the objective function respecting the set of constraints. For engineering problems, the design space is large and the objective function is relatively complicated and this leads to a computationally intensive problem.

This chapter is based on the article:

M.H. Aissa, T. Verstraete, and C. Vuik. Use of modern GPUs in design optimization. In *The 10th ASMO UK / ISSMO conference on Engineering Design Optimization Product and Process Improvement*, pages 1–11, 2014 .

The objective function itself is problem-specific. In topology optimization, approaches such as the general Solid Isotropic Microstructure with Penalization (SIMP) [Bendsøe and Sigmund, 2013] help to formulate the objective function. In shape optimization, the design can be changed within a fixed topology (e.g. a fixed number of holes). The objective function can be derived, for instance, from aerodynamics, structural or heat transfer considerations. In MDO the objective functions are originating from the interaction of different disciplines (e.g. structure mechanics, aerodynamics), where different levels of interactions exist. The definition of the objective function often depends on the solution of some partial differential state equations (PDE).

For topology optimization, the function evaluation is in many cases delegated to a method of Computational Structural Mechanics (CSM) for which Finite Element discretization is the most popular. The evaluation of the objective function consists then in solving a system of linear equations of the form $Ax = b$ with x the result from which the objective function depends, matrix A the problem specific system matrix depending on the design variables and vector b a problem specific right-hand side potentially depending on the design variables. In aerodynamic shape optimization, a CFD method performs the evaluation solving the non-linear governing equations (Navier-Stokes, Euler). In multidisciplinary design optimization (MDO), CFD, CSM and eventually other methods can work in a segregated or interactive manner to perform the function evaluation. The complexity of the optimization problem, as described above, leads to complex algorithms with large demand on computational resources. Thus high computational power and large memory resources are required to solve repeatedly the CSM and CFD models responsible for the objective evaluation.

The appearance of programmable graphics processing units (GPU) enabled at relatively low price to access a new high computational power system. GPUs are indeed a shared-memory system with a larger number of specialized lightweight cores than CPU shared-memory systems. The work of the design engineer is then to successfully divide the global optimization problem into small work packages that can be handled by a GPU core in a massively parallel manner. The problems that are easily divided into small and independent work packages are called *embarrassingly parallel* (e.g. simple image processing functions). If the work packages are not independent and need intercommunication, the problem is called coarse-grained parallel (e.g. low-order PDE solvers). Some problems do not show any simple form of parallelism and are called inherently serial (e.g. scanning and sorting).

The aim of this chapter is to review the use of GPU highlighting its advantages and limitations. The remainder of this chapter focuses on the use of GPUs to accelerate optimization methods in topology optimization, shape optimization and multidisciplinary design optimization.

3.2 Topology optimization

Many approaches have been developed to capture and guide the evolution of the topology during the optimization process (see Figure 3.1). Two of the main methods are level-set [Allaire, 2004] and Solid Isotropic Microstructure with Penalization

(SIMP) [Bendsøe and Kikuchi, 1988; Bendsøe and Sigmund, 2013]. While these two methods have been widely ported to the GPU (as it will be shown in the coming subsection), the GPU potential of other methods such as the bubble method [Eschenauer et al., 1994] and Evolutionary Structural Optimization (ESO) [Xie and Steven, 1993] are still to be explored. This section introduces briefly the two first methods and focuses on their adaptation to the GPU.

3.2.1 Solid Isotropic Microstructure with Penalization method (SIMP)

The SIMP method is based on a homogenization approach, which describes a structure as a combination of solid-void micro-elements. A pseudo-density variable (ρ) characterizes the material (solid $\rho = 1$, void $\rho = 0$) and is assumed to be constant within each element of the structure. The optimization process is about finding the optimal material distribution satisfying predefined constraints.

This type of integer-programming is highly computationally expensive and one way to avoid it for large-scale problems is to undertake a relaxation. The relaxation performs, indeed, an extension of the design space from two values 0 and 1 to the entire range of values $[0, 1]$. The problem becomes thus, convex and can be consequently optimized using a gradient-based method. Through the relaxation, the methods tolerates, however, non-interpretable intermediate values for the pseudo density. Penalization techniques are used to solve this issue by promoting the integer values 0 and 1 for the pseudo-density (e.g. the power law ρ^p [Bendsøe and Sigmund, 2013]). The contribution of elements with intermediate density is increasingly discarded with a higher penalization factor p. The problem formulation with penalization is, however, not convex and requires the application of a local filter on the density distribution averaging neighbor elements inside a predefined radius. An optimizer, such as the Optimality Criteria (OC) [Yin and Yang, 2001] or the Method of Moving Asymptotes (MMA) [Svanberg, 1987], is then responsible for the update of the design variables to locate the optimum solution. The linear elasticity state equations are solved with the Finite Element Method (FEM) and the Finite Element Analysis (FEA) is the central component of the optimization process in topology optimization. It generates the structure answer to loads (e.g. displacements), which is essential to the evaluation of the objective function.

The application of SIMP methods for large-scale problems with millions of design variables is computationally demanding and therefore requires a high performance system. The work of Mahdavi et al. [2006] is an example of a CPU parallelization for topology optimization. GPUs have been also tested for solving topology optimization problems with the SIMP method. The implementation from Schmidt and Schulz [2011] of SIMP on structured meshes with a matrix-free conjugate gradient solver is faster on the GPU than on a 48-core CPU. The GPU implementation from Wadbro and Berggren [2009] of the SIMP method using the Preconditioned Conjugate Gradient solver applied to a 2D plate with heat source yield a speedup of 20x against a single CPU and 3x against multi-threaded CPU. Zegard and Paulino [2013] implemented the SIMP approach for unstructured meshes in GPU focusing on the assembly.

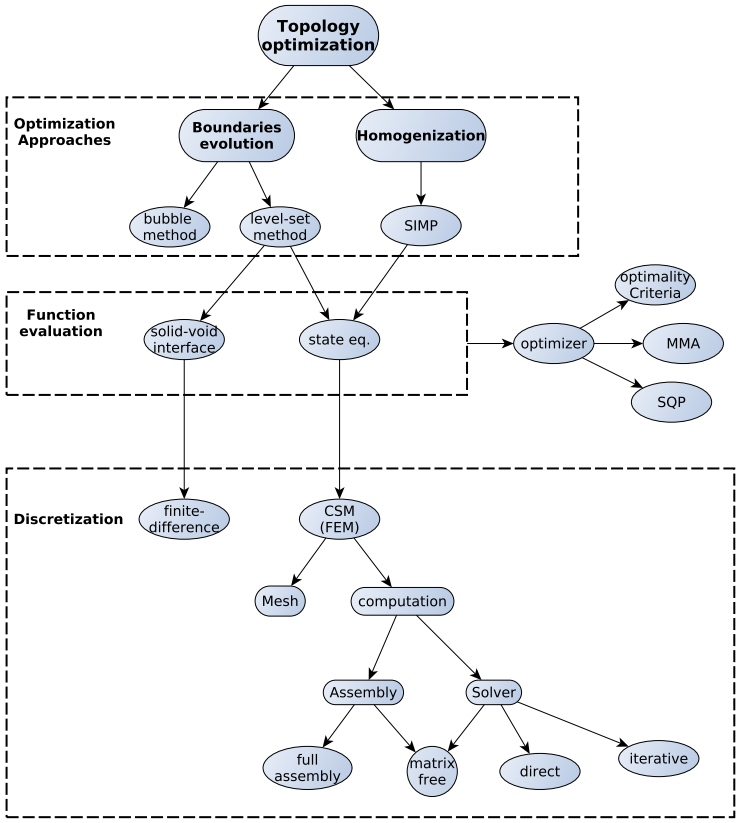


Figure 3.1: *Chart of different topology optimization approaches and common techniques for function evaluation such as CSM computation with FEM.*

3.2.2 Level-Set method

The level-set method (LSM) has been originally developed by Osher and Sethian [1988] as a scheme to advance the motion of an interface and was applied later to topology optimization. In topology optimization, the level-set method optimizes a given structure by a sequence of guided motions of the structure boundaries. The guided evolution converges to an optimum solution by minimizing the objective function [Allaire, 2004]. The flexible boundaries can represent complex shapes with the ability to create new topologies through inserting holes or through a structure splitting and merging. The review of the founder author [Osher and Fedkiw, 2001] presents a detailed insight into the level-set method and the work of Allaire [2004] shows a set of CPU applications.

The method is built on two fundamental equations: the boundary evolution and the state equations. The state equations are often discretized with FE methods, while the boundary evolution is simpler and can be solved with a Finite Difference method (FD). FD acts on a reduced group of elements of the mesh, which makes it suitable for efficient GPU processing [Micikevicius, 2009]. A GPU interpretation of this method is in the work of Herrero et al. [2013]. Challis et al. [2014] solved an inverse homogenization problem with a GPU implementation of a level-set method targeting high-resolution topology optimization. An increasing speedup with the problem size is reported reaching 13x for a 3D problem with over 4 million design variables [Challis et al., 2014]. For the parallelization of a parameterized LSM with Isogeometric Analysis Xia et al. [2017] reported speedups ranging between one to two orders of magnitude for the different steps of the optimization algorithm (e.g. assembly, sensitivity analysis). The reported speedups for the assembly on the GPU against the serial Matlab CPU implementation exceed 600x while the same GPU implementation is 65x times faster than a serial C implementation. Matlab is recognized to have performance issues when dealing with large-scale problems because of the inefficient memory allocation [Duarte et al., 2015]. The linear system of equations has been solved, however, on the CPU while it is in general the most time-consuming step in topology optimization [Duarte et al., 2015].

3.2.3 Underlying FEM

The methods used to solve the topology optimization problem present, in general, few common steps: (1) the area of application dictates the objective function, (2) a design evaluation is performed mostly by solving a linear system of equations $Ax = f$ and (3) an optimization scheme updates the design variable based on the evaluated objective. The level-set method, however, advances also a boundary equation in time. The solver of the FEM discretized state equations is the most time-consuming part of the optimization and should be the focus during the adaptation to the GPU architecture. The rest of the procedures such as the optimizer (e.g. OC, MMA) and boundary evolution (for level-set method) are not time-consuming and thus, not directly relevant for a high acceleration through the GPU. Georgescu et al. [2013] provide a review on the use of the GPU in all FEM steps from preprocessing to linear system solving and post-processing. This subsection focuses on the assembly, the solver and the mesh.

Linear solver

The system of equations discretized with FEM can be solved using a direct solver [Davis, 2006] such as the LU factorization or using an iterative solver. Direct solvers are more appropriate for small and averagely sized problems since the memory consumption scales thus with the number of variables [Wadbro and Berggren, 2009] following $\mathcal{O}(3/2N)$. Moreover, the large amount of inter-processors communication makes the direct solver challenging for parallelization [Mahdavi et al., 2006]. Sparse direct solvers, on the other hand, can process large-scale problems as they handle sparsely filled matrices. For some applications an approximation of the solution can be enough such as in steady CFD simulations, in that case iterative solvers are more appropriate. They require less memory ($\mathcal{O}(N)$) and are faster than sparse direct methods for large scale problems which explain their extended use on the GPU [Challis et al., 2014; Bolz et al., 2003]. If the system matrix is symmetric and positive definite, a preconditioned Conjugate Gradient solver (PCG) is favorable [Wadbro and Berggren, 2009]. Multigrid has been also implemented on the GPU to accelerate the convergence of linear system solvers [Geveler et al., 2011, 2013]. Cevahir et al. [2010] accelerated a Conjugate Gradient solver for unstructured meshes in a multi-GPU cluster. He used hypergraph partitioning [Catalyurek and Aykanat, 1999] to reach a fair load balancing and reduced the CPU-GPU communication, which resulted in an implementation running on 32 GPUs to be faster than 256 CPUs divided into 16 nodes of 16 CPUs per node. Duarte et al. [2015] reached a speedup of 13x using an element-by-element PCG solver [Augarde et al., 2006] on NVIDIA GTX Titan against Intel Core i7 CPU. The author used *greedy coloring* algorithm [Gebremedhin et al., 2005] to avoid race-condition within the used matrix-free iterative solver.

The system matrix is sparse and the sparse matrix-vector multiplication (SpMV) is a key feature. A first attempt to take advantage of the GPU regarding SpMV is to write kernels for the GPU responsible for the multiplication. This approach offers a large flexibility and the kernel can be adapted to specific needs of the problem to be solved in regard of the data storage layout and assembly. Good performance of SpMV kernels requires, however, an important development effort [Bolz et al., 2003; Geveler et al., 2011]. Geveler et al. [2013] based all the solver computation on one kernel for SpMV, which simplified the optimization. All the optimization effort was focused on one kernel with advantages for the whole solver. This approach brings, however, a programming overhead to turn all solver stages to SpMV functions. Many efficient SpMV GPU implementations exist already provided for instance by CUDA libraries (CUSPARSE[†]) or by PETSc [Minden et al., 2013]. Wadbro and Berggren [2009] used the CUBLAS[‡] library, the CUDA version of the linear algebra library BLAS, for inner products in the PCG. The libraries OP2 [Mudalige et al., 2012] and LISZT [DeVito et al., 2011] provide a high-order abstraction for matrix-vector multiplication. The GPU global memory is relatively small for one GPU card (e.g. 12GB for the K40) and therefore an explicit building of the global system matrix limits the maximum size of the problems that could be treated. FEM problems in topology optimization are inherently local, thus a matrix-free solver of state equation

[†] `http://docs.nvidia.com/cuda/cusparse/`,retrieved March 2017

[‡] `http://docs.nvidia.com/cuda/cublas/`,retrieved March 2017

is feasible [Schmidt and Schulz, 2011] and can be even one order of magnitude faster than a matrix storing solver [Reguly and Giles, 2015].

Assembly

Standard solvers build the system matrix by first computing node contributions and then summing them up to central elements for a global assembly. The solver is, in general, the most time-consuming part of a structure optimization method but adapting the assembly part to the GPU reduces the need for a time-consuming CPU-GPU transfer in every optimization iteration. During the assembly, a problem occurs if nodes are contributing in a parallel manner to build the element stiffness matrix. Two or more nodes could add their values to the same element at the same time, which causes a race condition (Cf. section 2.4) associated with an information loss. Zegard and Paulino [2013] used a graph coloring technique to avoid this problem coloring a set of non-racing nodes with the same color. Different colors cover the entire computational domain and all nodes of the same color can be run safely in parallel. Another approach is to assemble the stiffness matrix element-wise, which does not cause a race condition but results in a redundant calculation of same node contribution for different adjacent elements. Cecka et al. [2011] focus more on FEM assembly on GPU presenting low-level code optimization on CUDA targeting a speedup of 30x with single precision GPU code against double precision CPU version. The problem of race condition is peculiar to any parallel programming also using the CPU (e.g. with OpenMP [Jarzebski et al., 2015]).

Mesh

The mesh has a major impact on the reached GPU acceleration in structure analysis and optimization. Both size (number of cells) and type (structured/unstructured, multi-block) of the mesh are important. An unstructured mesh provides certainly more flexibility to mesh complex domains surrounding complex geometries but it is more challenging to reach a high GPU speedup with such a mesh. The absence of ordered indexing and regular neighboring (see Figure 3.2) make the memory access for the node or the cell data irregular and thus uncoalesced. For unstructured meshes, the cell-based approach presents at least more regularity compared to vertex-based meshes since the number of neighbor cells is constant unlike neighbor nodes. A vertex can have a different number of related vertices depending on its place in the mesh. A cell with m edges, on the other hand, has either m neighbors when situated in the interior of the computational domain or less when situated in the boundaries (e.g. $m-1$ or $m-2$). For unstructured meshes and especially the vertex-based scheme, node renumbering and an index list can help keep a partly coalesced memory access. It is more difficult to implement an efficient application for an unstructured mesh on the GPU than on the CPU. The CPU offers, indeed, much more cache memory per processor, which can handle the irregularity of the memory accesses for unstructured meshes.

It is obvious that structured meshes are better fitted to the GPU architecture. In turbomachinery for instance multi-block structured meshes are widely used. A multi-block mesh layout is intended to improve the mesh quality towards accurate

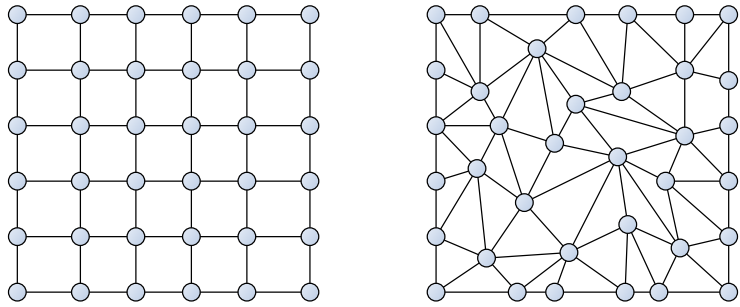


Figure 3.2: *On the left a structured mesh with a fixed number of neighbors for cells and vertices. On the right an unstructured mesh with the same number of nodes. The number of cell neighbors for the unstructured mesh is the same for all interior elements ($m = 3$) but for interior vertices the number of neighbors varies from 5 to 7.*

CFD results [Hirsch, 2007]. The layout for a body-fitted mesh around complicated geometries presents blocks of different sizes and many interfaces between the mesh blocks. Large blocks provide the GPU kernels with a high amount of independent operations for processing at the same time, which maximizes the number of active threads. The limiting factor, in this case, is the register usage. Since the kernels are starting a large number of threads and computing large algorithms with many lines of code, the total number of used registers is very high. The register consumption limits the achieved occupancy (Cf. section 2.6). A way to improve the occupancy for these memory-demanding kernels is to divide them when possible into small, less memory demanding, sub-kernels. Blocks with few cells are, on the other hand, in fact not limited by register usage but by the small number of started threads. Small mesh blocks do not provide the GPU with enough active threads to hide the memory latency. In this case, the multistream technique can improve the occupancy by starting more than one kernel at the same time. The mesh block interfaces require a cell update between blocks and this procedure involves few cells proportional to the *surface-to-volume* ratio ($r_{\mathrm{StoV}} * N_{\mathrm{Cells}}$). A solution is to use a mesh generator that takes into account the reduction of the number of blocks and neighboring blocks along with the increase in the block size in terms of cells (e.g. `hMETIS` [Karypis and Kumar, 1998]).

Finally, the process of meshing has been traditionally the responsibility of a CPU with a large set of mesh optimization available on the CPU. D'Amato and Vénere [2013] used however GPU for meshing.

3.3 Shape optimization

The shape optimization could be considered as a fine tuning for the topology delivered by the topology optimizer. The topology remains indeed untouched throughout the shape optimization process (e.g the number of structure parts, the number of

holes) and only the shape is changing to meet the problem specific constraints and objectives.

The shape is, in general, parameterized through mathematical function (e.g. Bezier splines). The design variables of the shape optimization problem are taken from the set of parameters controlling the geometrical construct, such as the spline control points or the curvature. For many shape optimization problems, the flow is the driving factor through the optimization. For external flows over whole planes or wings, the optimization reduces the drag and increase the lift, while for internal flows through engines or channels, it increases mainly the engine efficiency by reducing the losses.

Depending on the nature of the flow and the leading phenomena different equations have to be solved (see Figure 3.3). Lefebvre et al. [2012] optimized a 2D/3D Euler solver for the GPU. Brandvik and Pullan [2011] implemented a source-to-source compiler to execute a CPU Navier-Stokes solver on the GPU reaching a speedup of one order of magnitude for turbomachinery applications. Kampolis et al. [2010] used a Navier-Stokes solver with an evolutionary optimization algorithm for both external (airfoil) and internal applications (compressor cascade airfoil) achieving a speedup of 27x. Shape optimization on the GPU raises similar issues as in topology optimization regarding race condition and mesh-dependent performance.

In the CFD area, particle-based methods (e.g. Solvers for Lattice Boltzmann equations) are reported to profit the most from GPUs [Rinaldi et al., 2012]. These types of solvers provide a large number of independent simple arithmetic instructions that are proportional to the number of particles used. The Lattice-Boltzmann method is, however, not adapted to transonic and supersonic compressible flows [Chen and Doolen, 1998], which are standard flows in turbomachinery. He and Luo [1997] evoked indeed a practical limit for the Mach number of $M < 0.15$ for Lattice-Boltzmann methods. Finite Volume (FV) solvers, on the other hand, compute fluxes on cell faces involving a number of neighbors that increases with the order of the solver. Consequently, FV solvers run more arithmetic operations per cell and present more data dependency achieving smaller speedups on the GPU than particle based solvers. Nevertheless, a good management of the workflow and the GPU features, such as multistreaming, can lead to speedups of one to two orders of magnitude [Niemeyer and Sung, 2014b].

In some GPU-accelerated applications [Georgescu et al., 2013; Bell and Garland, 2008] with a major part of the execution time devoted to the linear solver, the CPU is used for the calculation of the right-hand side and the system matrix, for which the high porting effort is not worth the reduced performance gain. A linear solver is in general implemented on a GPU using a low-level programming language. The flexibility of the low-level approach makes it possible to adapt the data storage and the algorithm to the sparsity pattern (non-zero elements distribution) of the system matrix in order to enhance the performance. In this context, the effort is concentrated on accelerating the sparse matrix-vector product (SpMV), which constitutes the core of many linear solvers. Bell and Garland [2008] examined the optimization possibilities for SpMV on GPU without reordering the system matrix. He identified the diagonal format (DIA) as suitable for structured meshes and the Hybrid matrix format (HYB) for unstructured ones. The optimization is part of

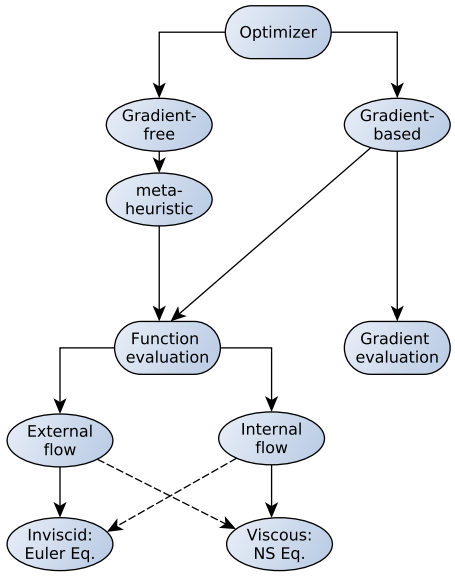


Figure 3.3: *Relation between optimizer and function evaluation in shape optimization.*

the CUSP library. Cecka et al. [2011] did similar work for problems based on Finite Element Methods (FEM). They examined the effect of the memory optimization on the overall performance comparing local, global and shared memory. Reguly and Giles [2012] reviewed relevant research for SpMV on GPU and concentrated on GPU tuning of SpMV operations for the Compressed Sparse Row (CSR) matrix format making use of the L1-cache locality, shared memory, and thread cooperation. The author presents a speedup of 1.4x over cuSparse and suggests that cache hit maximization is the key method behind the observed performance gain. The SpMV method was used, however, for a relatively simple Poisson problem which is not representative for global performance gain in realistic FEM or CFD problems. The work is also heavily based on the Fermi architecture with the hardware characteristic hard-coded in the performance fine tuning algorithm.

The performance of iterative solvers on the GPU has been gradually increasing but the bottleneck remains the inherently serial preconditioners such as the Incomplete LU factorization (ILU). These functions have been the subject of extended research [Saad and Van Der Vorst, 2000]. In order to improve the performance, the system matrix has to be reordered. This expose more fine-grained parallelism and thus provide the GPU with more independent instructions. Level-scheduling is one established alternative to elevate the parallelism of the factorization, where independent rows of the system matrix are implicitly grouped in the same level. Graph-coloring is another method where an explicit reordering is performed giving independent matrix elements the same color, then every color is thread-safe for a massively parallel linear solver. Naumov et al. [2015] showed a parallel graph col-

oring method reaching a higher parallelism than in level-scheduling. His work is included in the cuSparse NVIDIA library. Dutto [1993] examined the effect of many reordering techniques on the GMRES solver convergence behavior. Another method to extract more parallelism, introduced by Chow and Patel [2015], is to transform the ILU factorization in a minimization problem of a set of equations that could be computed in groups independently. Groups can be so small to contain only one equation making it possible for every non-zero element of the incomplete L and U matrices to be computed asynchronously and in parallel. This ILU version can be found in ViennaCL[†] and MAGMA sparse [Anzt et al., 2014].

3.4 Multidisciplinary Design Optimization (MDO)

MDO methods consider a multitude of disciplines when optimizing a design. An example of an MDO method is the bi-disciplinary optimization making use of structure analysis and flow analysis [Verstraete, 2010]. The general pattern seen in the literature is that the CPU governs the MDO method and starts the GPU as an accelerator to perform the bulk of the arithmetic operations. In general, a single-discipline simulation is ported (e.g. CFD or CSM) to the GPU as seen above for topology and shape optimization but not the whole MDO algorithm. The high computational power of the hardware added to its limited memory make it more adapted to accelerate single-field simulations rather than combining multiple simulations that could possibly not fit in the device memory for realistic test cases. Consequently, GPUs have no tangible impact on the MDO methodology beside the fact that they accelerate single-field simulations in a similar way traditional clusters do and, to the knowledge of the author, no MDO technique has been created based on the specificities of the GPU.

3.5 GPU in meta-heuristics

Metaheuristic methods can be classified as population-based or trajectory-based[‡]. Population-based methods, such as Evolutionary algorithms; swarm intelligence and particle swarm, manage a set of interacting individuum (solutions), that are evolving in every optimization iteration. These methods focus more on the exploration of the solution space rather than in the exploitation of the neighborhood of one solution. Single solution-based methods such as advanced local search, simulated annealing or Tabu search update continuously only one solution toward finding an optimum. Some of the single solution-based methods keep track of all intermediate solutions (e.g. Tabu search) others just replace the old with the new solution. All single solution-based methods focus more on the refinement of a solution through the exploitation of local neighborhood than a general global exploration of the solution domain.

Some other methods are hybrid combining both exploration and exploitation features. These methods start with a wide exploration of the solution space which

[†]Rupp, K. "ViennaCL." `http://viennacl.sourceforge.net`

[‡]also referred to as single solution methods

is progressively refined. As seen in shape optimization some metaheuristic methods are applied for design optimization in GPU [Asouti et al., 2011]. Many other metaheuristic methods have been already used in shape and topology optimization but only implemented for CPU (e.g. Differential evolution [Verstraete, 2010]). The GPU implementations of many meta-heuristic methods in other areas [Krömer et al., 2014; Talbi, 2014] are encouraging for applying them in design optimization. Ant colony optimization (ACO) is widely used [Cecilia et al., 2013] along with Genetic Algorithm [Langdon, 2011], local search [Van Luong et al., 2013; Czapiński, 2013] and Particle Swarm Optimization [Mussi et al., 2011]. Taillard et al. [2012] explored the GPU potential for hybrid metaheuristic methods.

3.6 Discussion

The CUDA developer community is increasing and useful tools help designing well-performing codes (debugger, profiler and memory checker). The prompt change in the GPU hardware scene from generation to generation requires, however, a continuous updating of developed applications. Whereas a set of optimization techniques coming from tremendous work can be made insignificant with the next hardware generation as for the computing precision for instance. First GPUs of 2007-2009 did not support double precision calculation and an important development effort has been invested to mix GPU single-precision and double-precision to achieve fast results without an important accuracy loss [Göddeke, 2010]. This work is of less interest when using recent GPUs which support double precision[†]. Moreover, some changes in the GPU memory layouts, such as a larger shared memory or a larger L1 cache, can require a code to be re-edited to keep an optimal performance.

CUDA itself was a relief from the graphics programming burden of early GPUs and a further abstraction seems unavoidable. The hardware specific optimization should be separated from the algorithm itself and the hardware-oriented code tuning should be rather a responsibility of a low-level system. One solution is to use a directive-based tool such as openACC or to apply toolbox libraries such as CUSPARSE and Thrust. A trade-off is inevitable between the higher performance of the hardware specific tuned code and the portability of the easy maintained code. But a high-level abstraction should not make from a GPU a black box for users. Learning the hardware in use helps always to design adapted algorithms and recognize algorithms with high parallelism potential.

3.7 Conclusion

This chapter covered the use of the GPU in the acceleration of design optimization problems focusing on topology optimization, shape optimization and multi-disciplinary design optimization. Interesting speedups of one to two orders of magnitude have been reported for topology optimization problems and aerodynamics shape optimization. The core of the optimization process is often the simulation of the governing equations in single fields (CFD, CSM). Porting an optimization

[†]Mixed-precision is still a hot topic for gaming GPUs

application is then more about porting the CSM/CFD solver. In all reviewed work the computational tasks are distributed among CPUs and GPUs taking advantage of both systems.

The review of the literature suggested using the GPU to accelerate single-discipline simulations instead of running an entire MDO algorithm on it. While the GPU has been successful on accelerating data collection, assembly and linear system solving, no change has been observed in the multidisciplinary optimization methods. In that sense the GPU is not able to change massively the existing paradigms in MDO[†]. The impact of the GPU is mainly the acceleration of single-discipline simulations in the same way a cluster of CPUs does. Since the architecture of the GPU is different from common CPU clusters, it requires some changes in the algorithms and the implementation of single-discipline simulations.

In this work, the focus is on aerodynamic optimization, especially using the Finite Volume discretization, as an example of single-field optimization. For that purpose, the explicit then the implicit time-stepping are implemented and studied in details and a comparison of both methods follows out of the use of this two tools.

[†]MDO paradigms as for example defined in the work of Martins [Martins and Lambe, 2013]

4

GPU-accelerated CFD Simulations with Explicit Time-Stepping

CFD simulations are nowadays the cornerstone of design evaluations for a large variety of components (aircraft, car, turbomachines, etc.). While the accuracy of these simulations is proven and still improving, they are still very time consuming which leads to a long design cycle. Modern High-Performance Computing systems, especially Graphic Processing Units (GPUs), are able to alleviate this inconvenience by accelerating the design evaluation itself.

This chapter presents a validated steady CFD solver with explicit time-stepping running on the GPU. First, the Reynolds-averaged Navier-Stokes equations are introduced then the software implementation of the GPU solver. The solver is validated on a turbine nozzle guide vane and a large benchmark on a set of 2D and 3D test cases has been performed. An achieved speedup of about two orders of magnitude makes it possible for the design optimization algorithm to run on a high-end computer instead of a costly large cluster.

4.1 Reynolds-Averaged Navier-Stokes Equations

The integral form of the Reynolds Averaged Navier-Stokes equations (RANS) using conservative variables is listed below [Blazek, 2005, p.16]:

This chapter is based on the article:

M. H. Aissa, T. Verstraete, and C. Vuik. Aerodynamic optimization of supersonic compressor cascade using differential evolution on GPU. In *AIP Conference Proceedings. Eds. Theodore Simos, and Charalambos Tsitouras.*, volume 1738, page 480077. AIP Publishing, 2016 .

$$\frac{\partial}{\partial t}\int_{\Omega}\vec{W}d\Omega + \oint_{\partial\Omega}(\vec{F_c}-\vec{F_v})dS = \int_{\Omega}\vec{Q}d\Omega\ , \tag{4.1}$$

where Ω is the cell volume, $\delta\Omega$ is the cell surface and $\vec{W}$ is the conservative variable vector: $\vec{W} = [\rho, \rho u, \rho v, \rho w, \rho E]$. Convective and viscous fluxes along with the source term are defined as follows:

$$\vec{F_c} = \begin{bmatrix} \rho V \\ \rho u V + n_x p \\ \rho v V + n_y p \\ \rho w V + n_z p \\ \rho H V \end{bmatrix} \tag{4.2}$$

$$\vec{F_v} = \begin{bmatrix} 0 \\ n_x\tau_{xx} + n_y\tau_{xy} + n_z\tau_{xz} \\ n_x\tau_{yx} + n_y\tau_{yy} + n_z\tau_{yz} \\ n_x\tau_{zx} + n_y\tau_{zy} + n_z\tau_{zz} \\ n_x\Theta_x + n_y\Theta_y + n_z\Theta_z \end{bmatrix} \tag{4.3}$$

$$\vec{Q} = \begin{bmatrix} 0 \\ \rho f_{e,x} \\ \rho f_{e,y} \\ \rho f_{e,z} \\ \rho\vec{f_e}\cdot\vec{v} + \dot{q}_h \end{bmatrix} \tag{4.4}$$

$$\begin{aligned} \Theta_x &= u\tau_{xx} + v\tau_{xy} + w\tau_{xz} + k\frac{\partial T}{\partial x} \\ \Theta_y &= u\tau_{yx} + v\tau_{yy} + w\tau_{yz} + k\frac{\partial T}{\partial y} \\ \Theta_z &= u\tau_{zx} + v\tau_{zy} + w\tau_{zz} + k\frac{\partial T}{\partial z} \end{aligned} \tag{4.5}$$

and the viscous stress tensor τ after application of the Stokes' hypothesis ($\lambda + \frac{2}{3}\mu = 0$) reads in tensor notation:

$$\begin{aligned} \tau_{ii} &= 2\mu(\frac{\partial v_i}{\partial x_i} - \frac{1}{3}div\vec{v}) \\ \tau_{ij} = \tau_{ji} &= \mu(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}). \end{aligned} \tag{4.6}$$

The Roe scheme [Roe, 1981] is used to discretize the convective flux and the central scheme is used to discretize the viscous flux. The turbulence is modeled with Spalart-Allmaras (SA) model [Allmaras and Johnson, 2012] in the Finite-Volume framework using a first-order upwind scheme. The "method of lines" is used and thus space and time integration are treated separately. After the space integration, which consists of summing the fluxes and source term in a residual R, a time integration advances the flow toward a stationary state. The equation (4.1) can be reformulated to highlight the time integration as follows:

$$\frac{(\Omega\bar{I})_I}{\Delta t_I}\Delta\vec{W}_I^n = -\beta\vec{R}_I^{(n+1)} - (1-\beta)\vec{R}_I^n\ , \tag{4.7}$$

with $\Delta\vec{W}_I^n = \vec{W}_I^{n+1} - \vec{W}_I^n$ the solution update, which depends on a combination of the two residuals at time point n and $n+1$. While the residual R_I^n at time point n is available right after the space integration, the residual R_I^{n+1} depends on the solution at time point $n+1$, which is not available before the time integration. For $\beta = 0$, the right-hand side (RHS) of equation (4.7) contains only available quantities and thus the solution update can be computed *explicitly*. In this work the explicit time integration implements a Runge-Kutta scheme following Jameson et al. [1981] for the time integration:

$$\begin{aligned}
\vec{W}_I^{(0)} &= \vec{W}_I^n \\
\vec{W}_I^{(1)} &= \vec{W}_I^{(0)} - \alpha_1 \tfrac{\Delta t_I}{\Omega_I} \vec{R}(\vec{W}_I^{(0)}) \\
\vec{W}_I^{(2)} &= \vec{W}_I^{(0)} - \alpha_2 \tfrac{\Delta t_I}{\Omega_I} \vec{R}(\vec{W}_I^{(1)}) \\
&\vdots \\
\vec{W}_I^{n+1} &= \vec{W}_I^{(m)} = \vec{W}_I^{(0)} - \alpha_m \tfrac{\Delta t_I}{\Omega_I} \vec{R}(\vec{W}_I^{(m-1)}).
\end{aligned} \tag{4.8}$$

While the classical Runge-Kutta scheme stores the intermediate stage values $W_I^{(k)}$, this formulation stores only the initial solution $W_I^{(0)}$ and the current solution $W_I^{(k)}$ along with the residual. The memory footprint is consequently independent of the number of stages m leading to a constant memory consumption of three arrays (of a length $N = N_{\text{cells}} N_{\text{var}}$) instead of $2 + m$ arrays.

4.2 Implementation and discussion

The GPU version of the RANS solver is based on a CPU in-house code. The porting procedure has been performed function by function guaranteeing the exact same output between both versions after every porting step. At the end of the porting, only the initialization and the post-processing remained on the host side and are performed by the CPU. The entire flow computation has been delegated to the GPU as depicted in Figure 4.1.

The explicit time-stepping is based on a set of *embarrassingly parallel* stencil-based operations. A kernel computes the convective fluxes, a second computes the viscous fluxes and a third computes the source terms. The three kernels add, in turn, their contributions to the same residual. Another kernel performs the time integration based on the old solution and the new computed residual. These kernels proceed on all mesh cells (or faces) and have similar run configurations. Boundary and interface updates are performed at the end of every Runge-Kutta stage. These updates affect only a portion of the cells depending on the surface-area-to-volume ratio, which depends on the number of mesh blocks within a multi-block body-fitted mesh layout as depicted in Figure 4.2. The implementation of the main kernels will be presented and different implementation strategies will be discussed focusing on their relative performance on the GPU.

4.2.1 The convective flux evaluation

The convective flux evaluation will be treated in detail as an example of performance analysis for a typical CFD kernel. First, the scheme is introduced in order to clarify the algorithm behind the implementation. Then, a set of implementation possibilities will be analyzed using some performance parameters such as the device utilization and the achieved memory bandwidth. Moreover, the influence of the scheme order on the performance is assessed along with the effect of the GPU multistreaming.

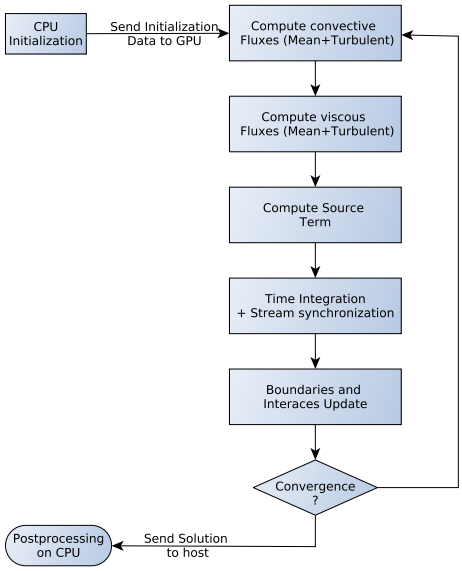


Figure 4.1: *The solver computes first the inviscid and viscous Residuals along with the source term. Secondly the time integration takes place and finally boundaries and mesh interfaces are updated.*

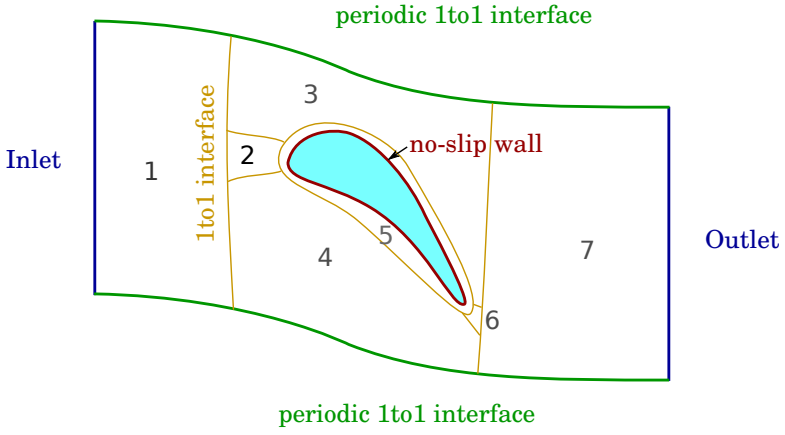


Figure 4.2: *Plot of the multi-block mesh layout, which is used for all meshes in this work, highlighting the boundaries and interfaces.*

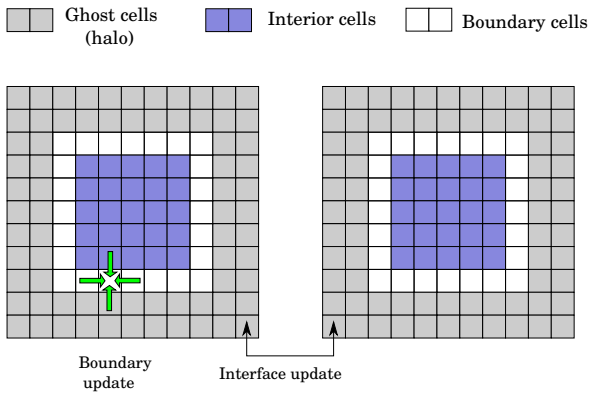


Figure 4.3: *Two mesh blocks showing the different type of cells in a computational domain and the interface update (ghost cells swap).*

Scheme

The convective flux is discretized by a *flux-difference splitting* scheme, which evaluates the flux at a cell face by solving a Riemann problem. The time-consuming Godunov solution [Godunov, 1959] of the shock tube problem is avoided by solving the linearized problem introduced by Roe [1981], which is known for solving shocks accurately [Blazek, 2005] and reads as follows:

$$(\vec{F}_c)_{I+1/2} = \frac{1}{2}[\vec{F}_c(\vec{W}_R) + \vec{F}_c(\vec{W}_L) - |\bar{A}_{Roe}|_{I+1/2}(\vec{W}_R - \vec{W}_L)] \,, \tag{4.9}$$

with $\bar{A}_{Roe}$ the *Roe matrix* defined as a combination of the conservative variables $\vec{W}_R$ and $\vec{W}_L$ [Roe and Pike, 1985]. $\vec{F}_c(\vec{W}_R)$ and $\vec{F}_c(\vec{W}_L)$ are evaluated following equation (4.2). The subscripts L and R refer to the left and right state of the considered face as depicted in Figure 4.4. The span of involved grid cells in order to compute $\vec{W}_L$ and $\vec{W}_R$ determines the degree of the space integration. When these states are simply set respectively equal to the content of the right and left cells of the active face, the scheme is first-order accurate. Second-order accuracy is achieved through the combination of more than one neighbor to define the right and left states. In this work, second-order accuracy is reached by using the Monotone Upstream-Centered Schemes for Conservation Law (MUSCL [Van Leer, 1979]). Left and right state are then defined as follows:

$$\begin{aligned} U_R &= U_{i+1} - \frac{1}{2}\Psi_R(U_{i+2} - U_{i+1}) \\ U_L &= U_i - \frac{1}{2}\Psi_L(U_i - U_{i-1}) \end{aligned} \tag{4.10}$$

with Ψ a limiter function preventing the solution from spurious oscillations near strong discontinuities [Blazek, 2005]:

$$\Psi_{L/R} = \frac{1}{2}\left[(1+\hat{\kappa})r_{L/R} + (1-\hat{\kappa})\right]\Phi_{L/R} \tag{4.11}$$

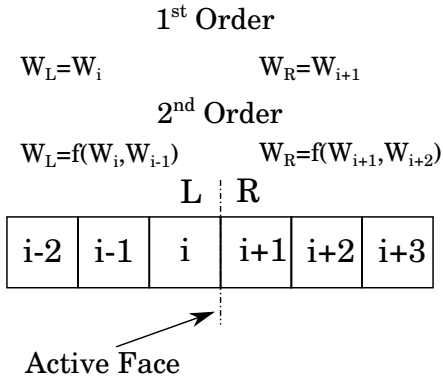


Figure 4.4: *Definition of the right and left states at a cell face for cell-centered Finite Volume approach.*

where $r_{L/R}$ is the ratio for consecutive solution variations ($\Delta W_L/\Delta W_R$), Φ a slope limiter and $\hat{\kappa}$ a parameter to determine the spatial accuracy of the interpolation. The second-order upwind approximation is implemented in the MUSCL scheme by setting $\hat{\kappa} = 0$ and using the *Venkatakrishnan* slope limiter function [Venkatakrishnan, 1991]. The final formulation of the left and the right states for this second-order accurate scheme is the following:

$$\begin{aligned} U_R &= U_{i+1} - \frac{1}{2}\delta_R \\ U_L &= U_i - \frac{1}{2}\delta_L \end{aligned} \tag{4.12}$$

with

$$\begin{aligned} \delta_R &= \frac{\Delta W_{i+2}(\Delta W_{i+1}^2 + \epsilon) + \Delta W_{i+1}(\Delta W_{i+2}^2 + \epsilon)}{\Delta W_{i+2}^2 + \Delta W_{i+1}^2 + 2\epsilon} \\ \delta_L &= \frac{\Delta W_{i+1}(\Delta W_i^2 + \epsilon) + \Delta W_i(\Delta W_{i+1}^2 + \epsilon)}{\Delta W_{i+1}^2 + \Delta W_i^2 + 2\epsilon} \end{aligned} \tag{4.13}$$

with $\Delta W_i = W_i - W_{i-1}$ enlarging the requested stencil from 2 to 4 cells in every space dimension. The above formulation is valid for interior cells as defined in Figure 4.3. For a face on a boundary cell and belonging to a non-slip wall or a symmetry boundary, the flux is directly evaluated using equation (4.2) with an interpolated value of the pressure. Consequently, every face on the boundary has to be checked for belonging to a wall or a symmetry boundary.

The CPU has a powerful flow control unit able to optimize the cost of an extra conditional statement. Unlike the CPU, an extra conditional statement creates possibly a thread divergence inside warps for the GPU kernel. The divergence can be solved by making sure an entire warp computes the same flux whether the wall flux (cf. Equation (4.2)) or the Roe flux (cf. Equation (4.9)). The different execution paths, however, deteriorate the load balancing between the different streaming multiprocessors (MPs) on the GPU. In fact, some MPs will be performing the lengthily standard flux calculation using the Roe scheme while others will be calculating the

simplified version for wall cells. Therefore a separate kernel has been devoted only for boundary faces. The latter kernel suffers, however, from the small number of faces to treat which decreases its occupancy. Multistreaming is used at this level to run concurrently many boundary flux updates.

Another formulation of the convective flux from Yee [1987] makes it possible to treat all faces at the same time separating the flux to a central part and a dissipation part. This scheme is used by commercial solvers and goes, however, beyond the scope of this work.

Different implementation strategies

The convective flux computation exposes a relatively fine-grained parallelism enhanced by a good data locality, since every face flux depends only on two neighbors for the first-order evaluation or four neighbors for the second-order evaluation. As a throughput-oriented device, the GPU delivers better performance when kernels start a large number of lightweight independent threads. The number of started threads depends on the decomposition of the algorithm into small pieces of work. The overall performance of the kernel is the product of an optimized distribution of the work among the threads in a way that maximizes the number of started threads while at the same time preserves the data locality and the threads independence. Before detailing the different implementation strategies analyzed in this work, the following paragraph addresses the question of why not to map threads to design variables even if this mapping increases the amount of independent work pieces.

By mapping threads to flow variables the number of started threads is equal to the number of cells multiplied by the number of variables. Three-dimensional flows provide 5 flow variables and few turbulence-related variables. The one-equation Spalart-Allmaras turbulence model adds only one variable namely the turbulent eddy viscosity. Consequently, mapping the threads to these flow variables increases the number of used threads by 6 times, which improves the thread level parallelism and the achieved occupancy. At the same time, the dependency between the coupled flow variables requires inter-threads communication, which is only possible within the shared-memory and with a costly thread synchronization. The communication harms the performance and increases the risk of memory misusage (e.g. race condition). Moreover, a load balancing problem can occur as the threads are performing different computations for different flow variables. The small gain in thread-level-parallelism through the mapping to flow variables is rather not able to outbalance the reduction of the instruction-level-parallelism engendered by the coupled variables. Therefore, it has been chosen to make every thread responsible for updating all flow variables for the rest of the analysis. The workload is the same among threads and no synchronization is needed during the flux evaluation.

The basic mappings treated in this work cover mapping threads to (1) cells, (2) space directions or (3) cell faces. Figure 4.5 depicts the three cases and specifies for each case the number of calls to the kernel in order to cover the entire mesh and the number of updated faces per thread. Figure 4.6(a) shows the procedure to follow to switch from a cell-based approach to a space direction-based approach and finally to a face-based approach. For these three alternatives, the focus is on the amount of redundant work and the number of started threads, which is depicted in

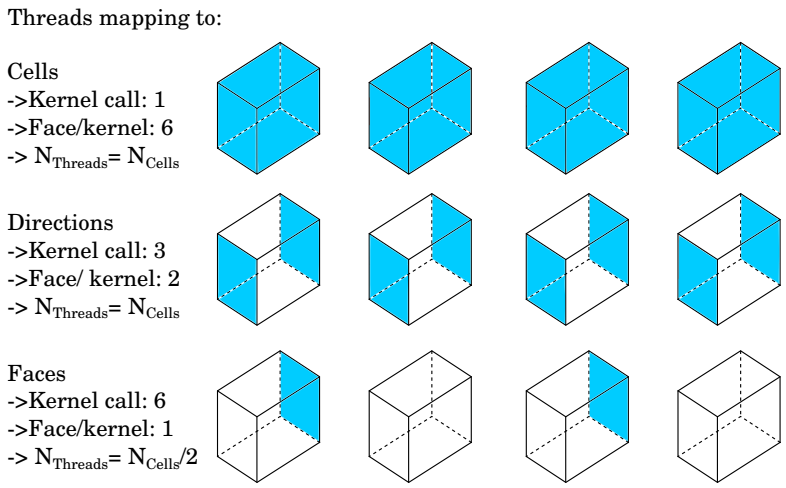


Figure 4.5: *Plot of some different threads mapping strategies.*

Figure 4.6(b).

When mapping one thread to one cell, every thread is responsible for 6 faces. One call to this kernel, which starts $N_{Threads} = N_{Cells}$, is enough to update all cells. Such a kernel is, however, very demanding in registers which lowers its theoretical occupancy. Moreover, the contribution of faces common to two cells is computed twice, once by every cell, which is a redundant work. Since a cell has six common faces with neighbor cells [†], the cell-based approach computes the same contribution in total six times.

In order to limit the register consumption of a kernel and reduce the redundant work, it is usual to split it into a set of sub-kernels. Splitting the computation into a set of kernels enables also special code optimization for every kernel call fitting its specific range of data. In the current case the threads can be mapped to space directions and every kernel call is responsible only for one direction. In total, three calls to such a kernel are needed to update the whole mesh and in every call two faces are computed.

Even though the same amount of work is done for both approaches (cell and direction-based mappings) and the same number of threads is started, splitting a large kernel reduces the register consumption which boosts the occupancy of the split kernel and provides a larger number of concurrently running threads. On the other hand, running one large kernel as for the cell-based mapping decreases the total calls to the local memory as variables are more often reused and stored in registers.

Both kernels suffer from a redundant calculation of the contribution of common faces, which are shared between two cells. Especially with compute-bound kernels, such as the treated case of the inviscid flux evaluation, a redundant calculation could be very harmful to the overall performance of the kernel.

In order to remove the redundancy, a third alternative maps threads to a faces.

[†] six common faces in case it is not a boundary cell

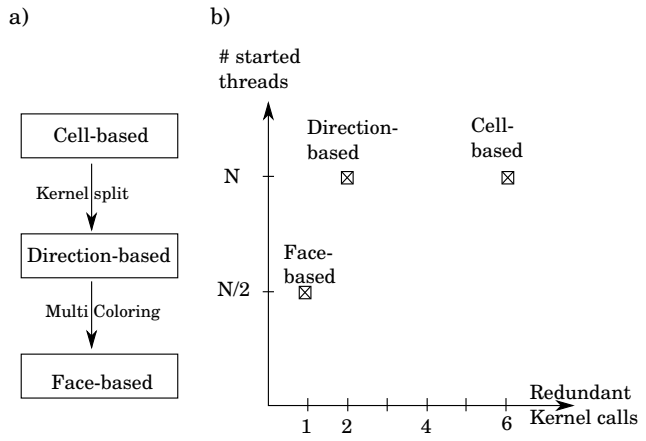


Figure 4.6: *(a) Relation between cell, direction and face-based approaches for the convective flux computation, (b) the number of started threads and redundant calls for the different approaches.*

For 3D structured meshes, six faces contribute to the residual of the same cell. In the CPU serial implementation, the residual computation is done face-wise as the value of the flux generated by a face is added to one neighbor cell and subtracted from the other cell. The flux calculation is based on a summation of the face contributions to the central cell, which is however not thread-safe for parallel execution. Multiple faces could add face contributions at the same time to the same cell and consequently, some contributions could be discarded corrupting the flow solution and the deterministic character of the simulation. Multi-coloring, which creates groups (colors) of faces not sharing cells in common, is used to solve this issue. For every color the computation is thread-safe but the number of started threads for every kernel is reduced to the number of faces per color. For a structured mesh, the colors have an equal number of faces and the number of threads is $N_{\text{Faces}}/N_{\text{Colors}}$. For a 3D flow, 6 colors are enough to run thread-safe computations with two colors by flow direction, one color for odd-indexed faces and the other for even-indexed faces. On the other hand, threads of the same color are not accessing consecutive memory positions anymore, which reduces the overall memory coalescence to striped access. The average number of memory transactions per memory request increases thus from 3 for the redundant computation kernel (coalesced access) to 5[†] for the multi-coloring kernel (striped access).

In the following, the direction-based mapping kernel with redundant calculation is compared to the face-based mapping kernel with multi-coloring. The comparison is not trivial as both have competing advantages. The memory coalescence is very important for memory-bound kernels while a redundant computation is very harmful to compute-bound kernels. Figure 4.7 shows an example of unit utilization for the face-based mapping (Multi-coloring) and the direction-based mapping (No Multi-coloring). Both kernels reach a higher utilization for the compute unit (up to 90%)

[†]The number of memory transactions has been retrieved using the NVIDIA profiler.

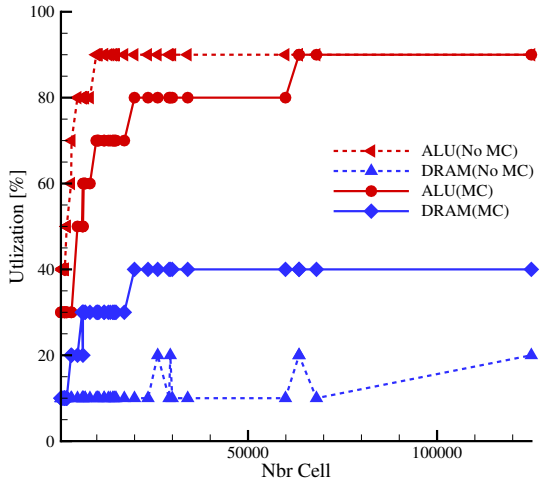


Figure 4.7: *The effect of the Multi-Coloring (MC) on the first-order convective flux evaluation on the GeForce 780 for different mesh block sizes.*

than for the memory unit (up to 40% for MC face-based mapping and up to 20% for direction-based mapping). With the redundant calculation of the direction-based mapping, the compute unit saturates for a smaller number of cells as the computations per thread are doubled. Consequently, the reached memory utilization level is very low leading to a lower memory bandwidth (see Figure 4.8) and thus a lower overall performance as depicted in Figure 4.9.

In summary, it is possible to switch from a cell-based to a direction-based approach through kernel splitting and from a direction-based to a face-based approach through multi-coloring. The redundant work is reduced respectively but also the number of started threads per kernel which is very important for the performance of the kernel. Figure 4.9 compares the performance of the three versions (cell, direction and face-based mappings) measured in updated cells per second for different mesh sizes. The convective flux evaluation on the GPU using the multi-coloring version of the code outperforms both the cell and the direction-based mapping. In essence, the register consumption and the saturation of the compute unit govern the performance of the convective flux evaluation.

Influence of the spatial accuracy on the GPU performance

In order to analyze the effect of the order of the convective flux discretization on the performance of the kernels on the GPU, both evaluations the first and the second-order have been compared. The key performance parameter for the GPU kernels, in this work, is the number of updated cells per second as shown in Figure 4.9. Moreover, a study of some other performance parameters, such as the device utilization (memory and compute units) and the global memory bandwidth, is performed in

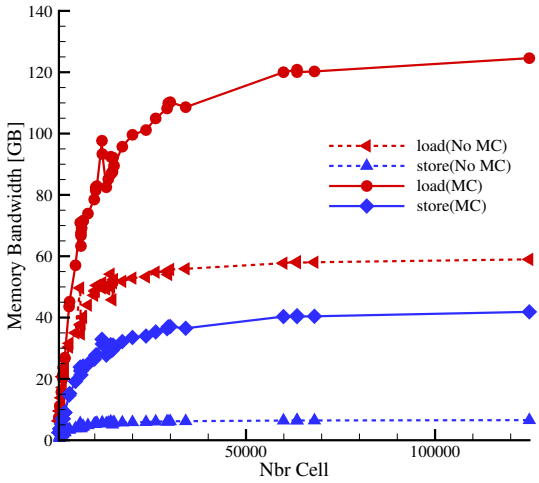


Figure 4.8: *Effect of the Multi-Coloring (MC) compared to the redundant calculation on the reached peak memory bandwidth both in reading and writing to the global memory for the first-order accuracy evaluation of the convective flux.*

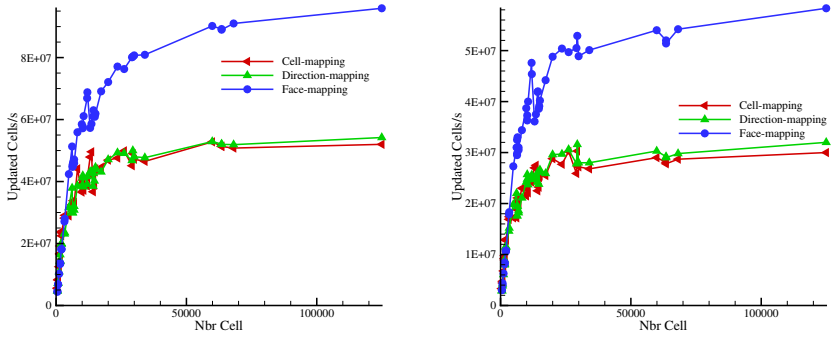


Figure 4.9: *Performance comparison of the three treated thread mappings: cell, direction and face-based mapping of the convective flux evaluation both for first (left) and second-order (right).*

order to understand the cause of the performance difference.

Figure 4.10 shows the occupancy[†] and the SM efficiency[‡] of the second-order evaluation of the convective scheme. The GPU multistreaming feature has been deactivated during the profiling to be able to measure the execution time of every mesh block when having access to the full GPU resources. Small mesh blocks counting 512 cells can keep the SMs busy only for 20% of the time. This is caused by the lack of warps to hide the important memory latency on the GPU. Large blocks of thousands of cells, on the other hand, approach asymptotically 100% of active time for the SMs. The implemented Roe scheme performs also an entropy correction to better capture the flow shocks as the original Roe formulation does not recognize the sonic point [Harten et al., 1997]. This correction creates possibly two different execution paths within one warp, which explains the remaining percentage of inactive warps even for very large mesh blocks.

The theoretical occupancy, which can be calculated by the profiler (`nvprof`) or the GPU Occupancy Calculator [NVIDIA, 2017] as a function of used registers and run configuration, is equal to 25%. The achieved occupancy starts by 5% for the smallest mesh block and increases with the block size approaching the theoretical value. Consequently, the performance increases with the mesh block size for the first and the second-order accuracy. Both kernels achieve a very similar level of occupancy. The number of eligible warps per cycle (2 to 3 warp/cycle[§]) for both kernels confirms that these two kernels reach the same level of thread parallelism (TLP). In fact, this is mainly regulated by the number of registers consumed by every kernel and in this case, the register consumption is marginally different.

Figure 4.12 presents the performance related characteristics of the convective flux kernel for the interior cells. Small mesh blocks are latency-bound as they have low utilization ($< 60\%$) of both memory and compute units of the GPU for the first-order accuracy. The big mesh blocks starting a large number of threads are compute-bound[¶] which is related to the lengthy algorithm behind the Roe scheme.

The first-order kernel reaches a higher peak memory bandwidth both for reading and writing to the global memory as depicted in Figure 4.11. However, the profiler measured very similar values for the number of memory transactions per memory request in both kernels as this parameter reflects solely the degree of coalescence of the memory access. On the other hand, the level of utilization of the DRAM memory and the compute unit (ALU) are significantly different for both kernels as shown in Figure 4.12. Even though both kernels are compute-bound[‖] reaching gradually a level of utilization of 90%, the second-order accuracy kernel reaches saturation levels of utilization for smaller mesh blocks than the first-order kernel. The second-order kernel performs indeed a larger number of floating point operations originating from the evaluation of the limiter function (see Equation (4.13)). The profiler measured indeed 50% more floating point operations for the second-order kernel. Because of the earlier saturation of the compute unit for the second-order kernel, it reaches also a lower level of memory utilization. The latter improves gradually with the increase

[†]Occupancy is defined as the portion of active warps (Cf. Section 2.4)

[‡]portion of time being active performing computations at least for one warp

[§]4 warp per cycle ensure full activity of the SMs

[¶]Kernel compute-bound on the GeforceGTX780. It may differ on other cards.

[‖]The kernel is compute-bound on a Geforce 780. Other GPUs could show other behavior.

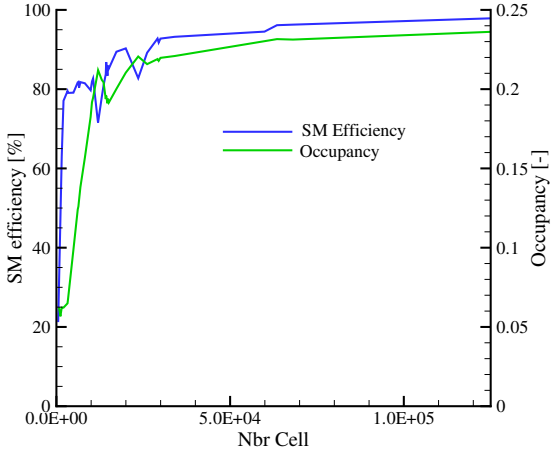


Figure 4.10: *Evolution of the achieved occupancy and the average streaming processors efficiency for the second-order convective flux evaluation as a function of the processed mesh block size.*

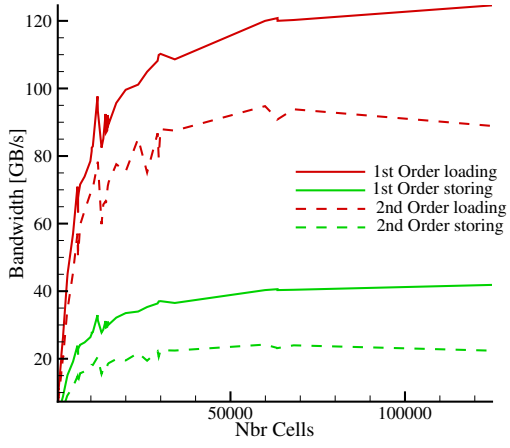


Figure 4.11: *Comparison of the reached peak memory bandwidth both in reading and writing to the global memory for the first and second-order accuracy of the convective flux kernel.*

of the number of cells until the compute saturation occurs. To conclude, the second-order kernel has a higher compute utilization and a lower memory utilization.

Multistreaming

For the above-introduced performance analysis of the convective flux evaluation, the kernels were all running in a default stream with thousands of threads but with no possibility of kernels concurrency. It has been observed that the GPU performance increases with the number of started threads. The mesh layout for multi-blocks body-fitted meshes (see Figure 4.2) requires sometimes the use of small mesh blocks in order to improve the mesh quality around complex bodies in terms of cell aspect ratio and skewness. The mesh quality enhances, in general, the accuracy of the produced CFD results. Therefore, the GPU should be able to efficiently run small blocks.

Multistreaming, which consists of launching many kernels simultaneously, can boost the achieved acceleration for small multi-block meshes. In this work, the kernels processing different mesh blocks are sent to different streams. These streams are independent and run concurrently. Small mesh blocks are consequently not the only contributor to the GPU occupancy but other kernels of larger mesh blocks are running at the same time. The streams overlapping during the execution is, nevertheless, influenced by the number of streams and the available GPU resources. If a stream treating a large mesh block is saturating the GPU capacity, other mesh blocks on other streams will not be able to run until the resources are being freed.

The performance increase through multistreaming depends on the degree of overlapping of different kernels and the mesh size. In order to assess the effect of multistreaming on the GPU performance, a set of meshes is used with a different number of cells but the same multi-block layout. The multistreaming gain, depicted in Figure 4.13, is more important for small meshes, as multiple mesh blocks are overlapping, and it fades with the increase of the number of cells. Multistreaming does not offer a linear scaling of the performance with the number of used streams mainly because of two reasons: the limited GPU resources and the unpredictable cache behavior. A kernel can access the GPU only when enough resources are available. Even if multiple streams are started only as many kernels will run in parallel as the GPU resources can afford. Moreover, the behavior of a set of thread grids (even of the same kernel) is different from the behavior of one thread grid [Farber, 2011, p.169]. A set of grids of the same kernel is not guaranteed to have the same cache-hit success as a single grid of threads for instance.

Benchmark

The fastest kernel for the convective flux evaluation, namely the face-based mapping with multi-coloring, is used to run a set of relevant test cases in turbomachinery. The test cases includes two cascades of turbine stators (t106c [Michálek et al., 2012] and LS89 [Arts et al., 1990]) and two cascades of compressor stators (CC2D [Aissa et al., 2016] and Turbolab[†]). All test cases follow the same mesh layout of 7 blocks of cells depicted in Figure 4.2 and the multistreaming has been deactivated to assure

[†] `http://aboutflow.sems.qmul.ac.uk/events/munich2016/benchmark/testcase3/`

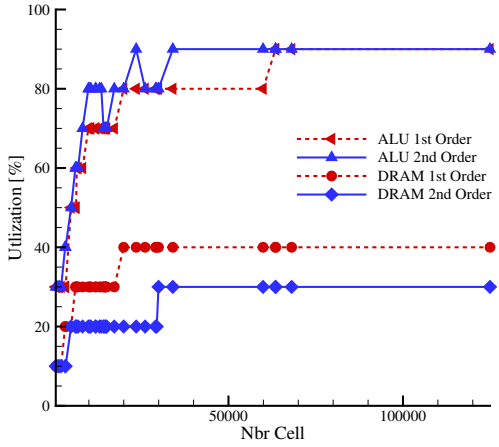


Figure 4.12: *The level of utilization of the DRAM memory unit and the compute unit (ALU) for both first and second-order convective flux evaluation on a Geforce 780 card for different mesh block sizes.*

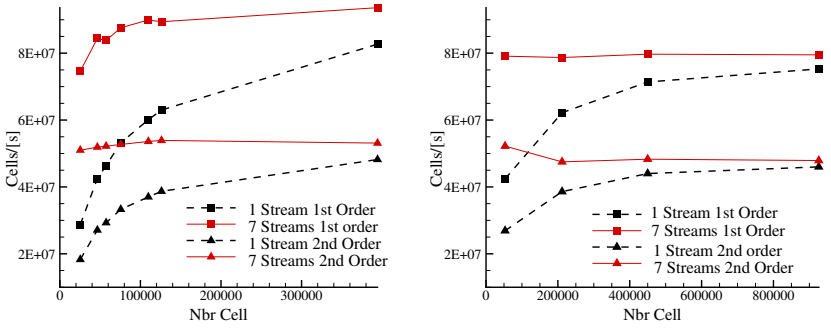


Figure 4.13: *Effect of multistreaming on the convective flux on a pseudo-3D compressor cascade (right) and a 3D turbine cascade (left).*

full resources for every mesh block. The performance of every mesh block from these test cases is plotted in the double logarithmic Figures 4.14 for both first and second-order accuracy.

The performance increase has two slopes: a large one for mesh blocks with $N_{\text{Cells}} < 400k$ and a smaller slope for mesh blocks with $N_{\text{Cells}} > 400k$. The same tendency is observed for both first and second-order accuracy. The part until the inflection point (change of the slope in the double logarithmic figure) can be labeled the *exponential growth* part of the performance curve as depicted in Figure 4.15. The second part until the next change in the slope can be called *pre-saturation* part and the last part is then the *saturation* part. The saturation itself could be caused, in general, by a full utilization of the compute power (for integer, single precision or double precision operations) or by a full utilization of the memory bandwidth. For this benchmarked kernel the saturation has been linked to the high compute-unit utilization (see subsection 4.2.1).

4.2.2 Viscous flux

The performance analysis of the viscous flux evaluation covers the same three approaches for the thread mapping as the above analyzed convective flux evaluation; namely the cell, direction and face-based mappings (see Figure 4.5). Unlike the convective flux evaluation, the viscous flux evaluation does not require a flux reconstruction. On the other hand, it uses velocities and temperature gradients, which will be treated in this subsection.

Scheme

The viscous flux is discretized using the central scheme, for which the state at a cell face is the average of both neighbor cells. The evaluation of the velocity derivative and the temperature derivative, which are needed in equation (4.5) and (4.6), constitute the bulk of the computation for the viscous flux evaluation. In general, Finite Differences (FD) or Green's theorem are used to evaluate these derivatives. The second approach conforms with the Finite Volume (FV) method and is implemented in both the CPU and the GPU versions of the code. Green's theorem relates the volume integral of a variable derivative to its surface integral. For that purpose, an auxiliary control volume $\Omega^{'}$ has to be created with the center at $i+1/2$ for the direction i and the same applies for the other directions as depicted in Figure 4.16. When applied within the cell-centered finite volume scheme, the derivative in x direction (equivalent to i index) for instance reads as follows [Blazek, 2005, p.120]:

$$\frac{\partial U}{\partial x} = \frac{1}{\Omega'} \int_{\delta\Omega'} U dS'_x \approx \frac{1}{\Omega'} \sum_{m=1}^{N_F} U_m S'_{x,m}, \tag{4.14}$$

with N_F the number of faces (6 for 3D flows) and U_m an averaged value on the auxiliary faces.

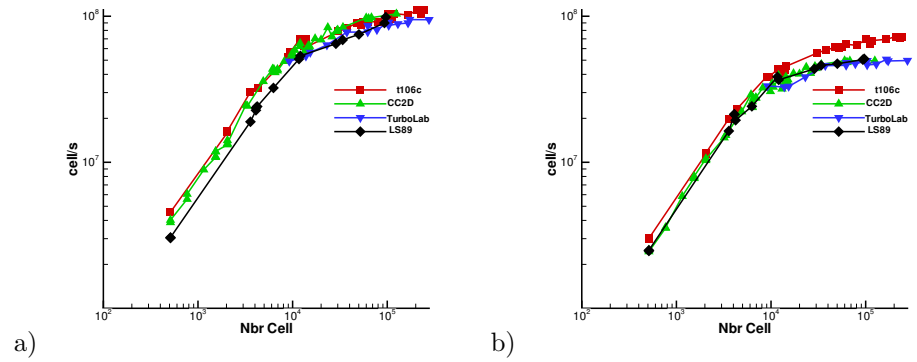


Figure 4.14: *Performance measured in updated cells per second for the evaluation of the convective flux on a Geforce780 in four different test cases for both a) first-order and b) second-order accuracy.*

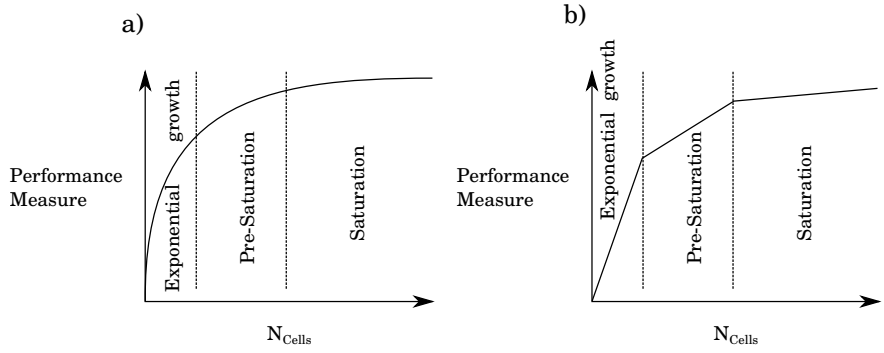


Figure 4.15: *Qualitative plot of the different phases of a GPU performance curve on a normal and a double logarithmic scales.*

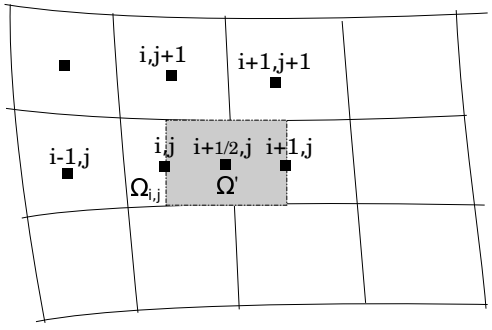


Figure 4.16: *An auxiliary face for a cell-centered scheme in a 2D structured mesh.*

Viscous gradients

Since the gradients are used many times in the viscous flux evaluation, a separate function should be computing and storing them for all cells at the beginning of every Runge-Kutta stage. In the following, two implementation approaches are compared: 1) one kernel for all space directions and 2) one kernel per space direction. Many memory requests are issued during the evaluation of Green's theorem (see equation (4.14)). The normal vectors of all faces of the auxiliary control volume have to be computed based on the averaging of the value of the two neighbor faces. The velocity at the active face is also averaged and in every averaging many variables are read from the global memory and stored in registers. As a result the register consumption is very high for those operations. If one kernel is responsible for all three space directions (*kernel fusion*), it consumes 238 registers of the 255 available for Kepler GPUs leading to a theoretical occupancy of 13%. Splitting the large kernel into three kernels, one for every space direction, reduces the register consumption to 125 and boosts the theoretical occupancy to 25%. Figure 4.17 compares the memory bandwidth of the fusion version and the split version both in reading and writing to the global memory. The reduction of the register consumption explains the observed performance gap in favor of the split version.

The reached peak memory bandwidth for the split version, shown in Figure 4.18, accumulates to large values asymptotically approaching 250 GB/s. This memory bandwidth is equivalent to 86% of the theoretical peak of 288 GB/s[†].

Viscous flux evaluation

The same three approaches for thread mapping, used with the convective flux evaluation, have been benchmarked for the viscous flux evaluation. The face-based mapping with multi-coloring uses fewer registers as the flux is computed only once. Therefore, the theoretical occupancy increases from 18.6% (130 registers) for the cell and direction-based mappings to 37.5% (78 registers) for the face-based mapping. Figure 4.19(a) shows the level of memory and compute utilizations of the face-based mapping kernel while Figure 4.19(b) shows the reached peak memory bandwidth both in reading and writing to the global memory. The viscous flux evaluation does not require a flux reconstruction, a very compute intensive operation compared to the convective flux evaluation, and shows therefore well balanced high utilization for both units memory and computation with a more pronounced compute utilization.

The face-based mapping version of the viscous flux evaluation reaches high values of accumulated memory bandwidth reflecting the high instruction level parallelism of the kernels (ILP), as many independent memory transactions are issued per thread. The face-based mapping version outperforms the cell-based mapping version by more than one order of magnitude as depicted in Figure 4.20. The cell and direction-based mapping versions have similar performances since the occupancy has not been significantly improved by the splitting (from 19% to 25%).

[†]`http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780/specifications`

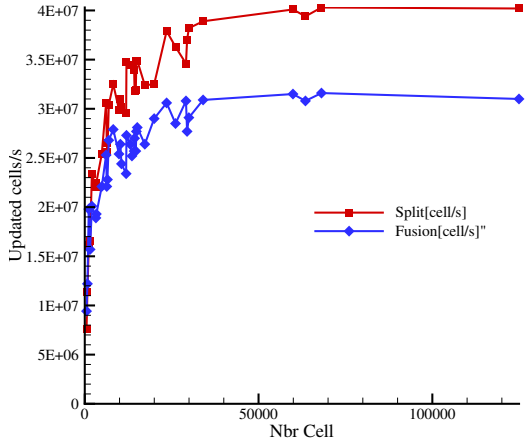


Figure 4.17: *Performance comparison between fusion and split versions of the kernel for gradient calculation.*

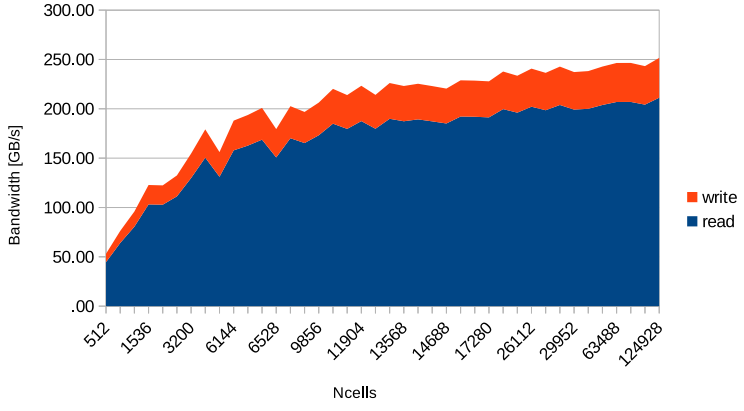


Figure 4.18: *Gradient kernel: Achieved peak memory bandwidth both on reading and writing to global memory for different mesh block sizes.*

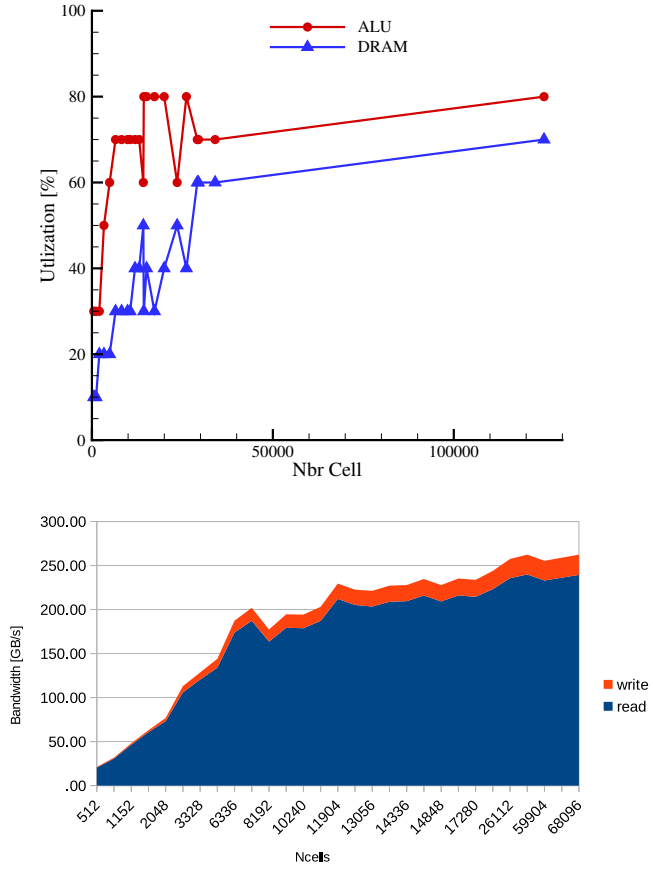


Figure 4.19: *Characteristics of the face-based mapping of the viscous flux evaluation kernel: (top) utilization and (down) memory bandwidth.*

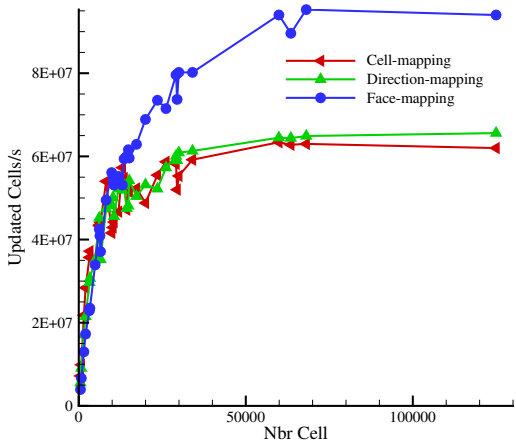


Figure 4.20: *Performance of the viscous flux evaluation measured in updated cells per second for both fusion and split versions using redundant calculation and a multi-coloring version for different mesh block sizes.*

4.2.3 Boundary conditions

The reference CPU code implements both subsonic and supersonic conditions in the inlet and the outlet boundaries (see Figure 4.2). It provides also inviscid and no-slip walls in addition to symmetry. The periodicity is treated in the interface update in section 4.2.4. All boundaries act on a small set of cells and perform few arithmetic and memory operations. These updates are fully independent and can run in different streams, which enables these kernels to reach a similar performance as the residual update kernels despite their latency-bound character. Figure 4.21 depicts the performance of the boundary update function summing all active conditions for both one stream and multiple streams. Surprisingly, the multistreaming does not bring an acceleration for small meshes and starts delivering a speedup only for the largest mesh. The explanation is related to the fact that the number of faces to be updated for the boundary conditions kernel is very small. Consequently, the different kernels on different streams are not overlapping since the one kernel in one stream is finished before the kernel in the next kernel is started due to the overhead of kernel launching[†]. The multistreaming causes a small overhead of launch and synchronization which explains the performance degradation for small meshes.

4.2.4 Interface update

The RANS solver implements two types of interfaces between mesh blocks: *1to1* interfaces and *periodic* interfaces. The first applies to a shared interface between two neighboring blocks and the second applies to distant interfaces used to implement

[†]kernel launching takes few milliseconds

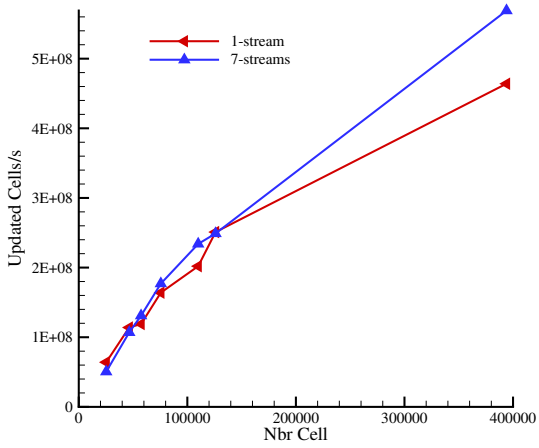


Figure 4.21: *Effect of multistreaming on the boundary update.*

a flow periodicity as depicted in Figure 4.2. When using the ghost cells approach, the interface update consists on swapping the content of two layers of ghost cells of a mesh block with the content of two layers of a neighboring mesh block (see Figures 4.3 and 4.22). The CPU implementation for the swapping uses a buffer strategy. It fills first a buffer with values read from one block without corner cells and then writes the buffer values to the ghost cells of the neighbor block. This implementation is purely serial as the buffer is filled in an order that has to be respected while writing. When naively ported to the GPU, this kernel was very slow as a CUDA core is much less powerful than a CPU core. The first attempt to improve the global performance of the GPU RANS solver was to copy the solution to the CPU, which performs the update then copy the solution back to the GPU. For the storage of the flow solution on the host, the page-locked[†] memory has been used which improved the performance of the data transfer [Aissa et al., 2016]. A substantial change in the method was, nevertheless, needed to adapt the interface update to the GPU SIMT architecture. For this reason, a lookup table has been implemented, which stores for every ghost cell the indices of the related physical cell in the neighbor block. The storage requirement in a 3D case attains 3 times the number of ghost cells (N_{ghost}) and could be very important for meshes with a large surface-area-to-volume coefficient. The use of the lookup table made the interface update *embarrassingly parallel* since the dependencies between the ghost cells have been completely decoupled. Consequently, all interface cells can be updated at the same time except the corner cells, which are treated separately. The only performance limitation of the new kernel is the small number of started threads that can not bring the GPU to run in a very efficient regime. CUDA streams are able to solve this issue since every interface update is independent of the other and can be done

[†]Page-locked memory is a faster type of host memory

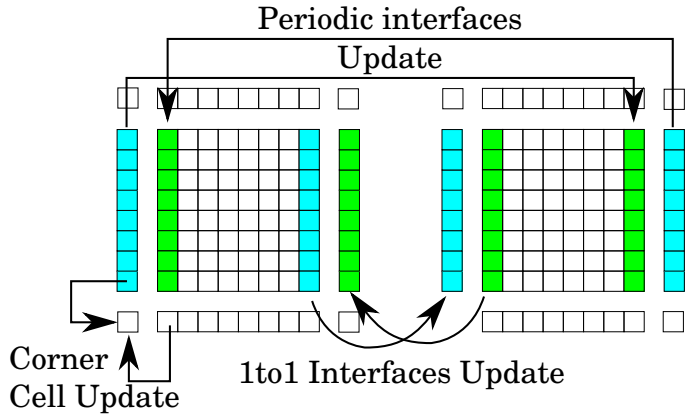


Figure 4.22: *Plot of three types of interface updates using the ghost-cell approach: Update of 1to1, periodic interfaces and corner cells.*

concurrently.

The interface update of the corner cells depends on neighboring blocks and has a different workflow as depicted in Figure 4.22. A separate kernel is thus implemented for these cells. When using the method of ghost cells averaging to compute the values of the corner cells as defined in [Blazek, 2005, p. 297], the ghost cells of the same block are used to compute the state in the corner cells. A more accurate corner cell update, involving an input from multiple neighboring blocks, would indeed improve the convergence rate of the flow simulation but at the same time, it prevents a concurrent update of the interfaces leading to a performance deterioration. A primary test on a supersonic compressor cascade (coarse mesh) converged after 19600 iterations taking 246 seconds when using the ghost cells averaging for the values of the corner cells. A more accurate corner cells update, combining four ghost cells layers of two different neighbor blocks, converges after only 16600 iterations but the simulation took 260 seconds. The effect is expected to be amplified with the increase of the mesh size.

4.2.5 Runge-Kutta stages

As shown in equation (4.8), the Runge-Kutta scheme computes in every stage the solution update based on the available residual and adds it to the existing solution. It performs 20 arithmetic operations and 10 memory calls per cell. Some operations are related to the transformation of the solution update from the conservative to the primitive form, since the residual has a conservative form while the solution should have the primitive.

The theoretical occupancy is 50.0% and the achieved occupancy depends on the solved mesh block size as it is shown in Figure 4.23(a). The achieved occupancy is approaching asymptotically the theoretical one as it is the case for other shown kernels. Both utilization levels are high on the GTX780 showing a balanced kernel with a high level of thread parallelism. The compute utilization is caused by the

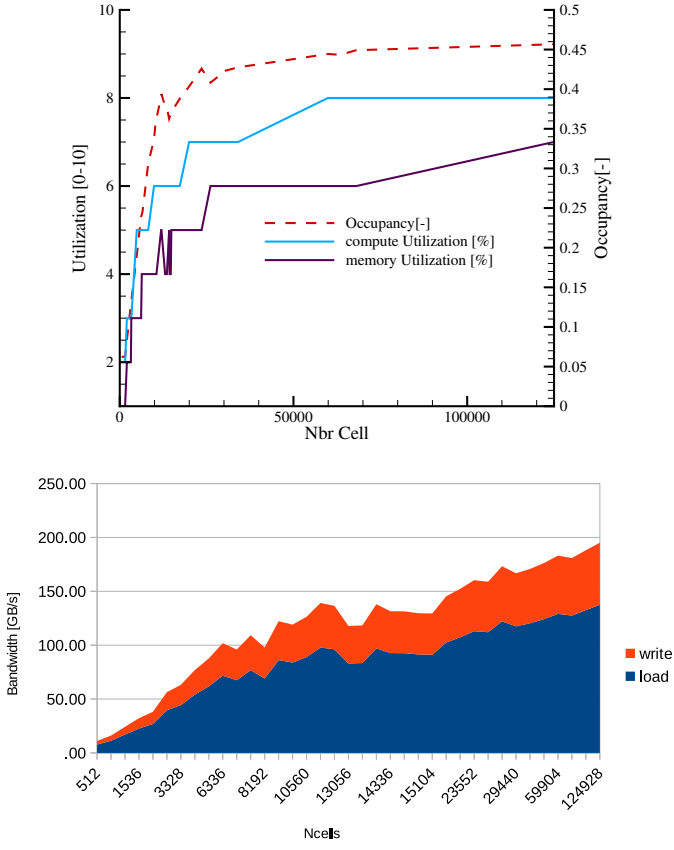


Figure 4.23: *Characteristic of the Runge-Kutta kernel: (top) utilization and occupancy and (down) memory bandwidth in GB/s.*

transformation of the solution from the conservative to the primitive form. The reached bandwidth of 200 GB/s, shown in Figure 4.23(b), constitutes 69% of the theoretical peak bandwidth. The relatively high reached bandwidth is still limited by the low number of memory instructions per thread, called instruction level parallelism (ILP), as every thread performs a small number of independent memory operations. Figure 4.24 depicts the performance of the Runge-Kutta stages as a function of the mesh size for the GPU compared to three different CPUs.

4.2.6 Convergence acceleration techniques

Convergence acceleration techniques such as Implicit Residual Smoothing (IRS) and multigrid are nowadays standard operations in commercial explicit CFD solvers (e.g. Numeca FINE^{TM} [NUMECA]). For a serial execution, the additional cost of these methods is rather negligible compared to the convergence acceleration they are able to achieve. In a parallel framework and especially massively parallel execution, the overall improvement of the solver performance using these convergence acceleration techniques is not obvious and rather case-dependent. Hence, these two techniques are first introduced and the performance of different GPU implementations is analyzed. Then, a benchmark is performed on different test cases to assess the performance impact.

Multigrid

The multigrid technique was adapted first for Euler equations then for Navier-Stokes equations by Jameson [1983]. Since then, the technique has been continuously extended and improved to tackle viscous and turbulent flows. The basic principle is to solve the governing equations on a set of coarse meshes and then interpolate the solution to the fine mesh in order to accelerate the solution convergence to steady-state on the finest mesh. In addition to the basic solving of the governing equations on the fine mesh, one extra run is counted per used coarse mesh along with two interpolation functions for the solution transfer from/to the fine mesh. While the additional memory cost is negligible as the coarse mesh is in general 1/8 of the size of the fine one, the additional execution cost on the GPU is not as easy to assess.

The performance of the GPU improves with the mesh size to reach asymptotically a saturation level from a certain mesh size as observed many times in the above-introduced kernel benchmarks. Figure 4.14 showed a linear relation between the number of cells and the performance with two different proportionality constants for meshes smaller and larger than 400k. The performance penalization δP when moving from a fine mesh with N^F_{cell} to a coarser mesh with $1/8N^F_{\text{cell}}$ is equal to $\delta P = 7/8\alpha N^F_{\text{cells}}$ with α the proportionality constant. The lower α the smaller the penalization. When the GPU is saturated ($\alpha \approx 0$) the coarse mesh takes 1/8 of the execution time of the fine one based on the number of cells to be updated. For small meshes, the proportionality is larger and the penalization is consequently more important. Moreover, both runs on the fine and the coarse meshes are interdependent and can thus not overlap.

A test case has been run to assess the cost of coarse meshes on a 3D flow around

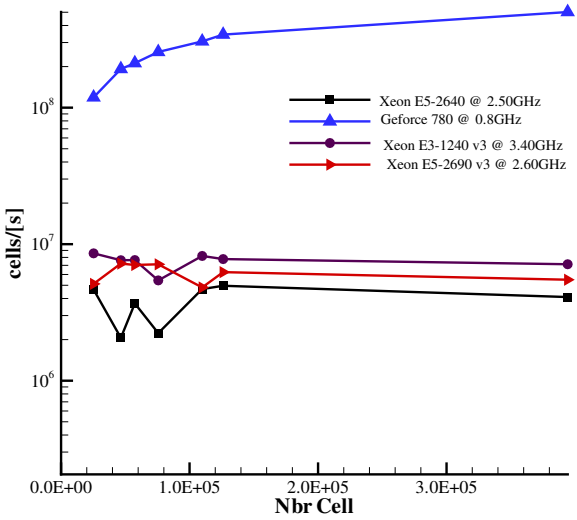


Figure 4.24: *Comparison of the performance, measured in updated cells per second for one Runge-Kutta stage, on a Geforce 780 to three other single CPUs for a RANS simulation of 2D flow on a supersonic compressor cascade.*

a compressor cascade (see Table 4.1). The cost of a two-grid-V-cycle[†] is between 120% and 180% more than a simple flow iteration without multigrid. The ideal case ($\alpha \approx 0$ and $T_c = 1/8T_F$) produces theoretically a ratio of 9/8 equivalent to 125% of the cost of a simple flow iteration without multigrid assuming the interpolation cost are negligible. The ideal case is asymptotically approached in the shown test case. The convergence acceleration itself is case dependent and ranges from 2 to 5 [Blazek, 2005]. Therefore, it is very promising to have a multigrid cycle running on the GPU compared to the CPU performance.

Implicit residual smoothing

Low CFL numbers are a characteristic of the explicit time-stepping. The residual smoothing, based on equation (4.15), improves the convergence by extending the envelope of safe CFL numbers to higher values. A factor of 3 to 5 is common in the literature for the increase of stable CFL numbers when using smoothed residuals compared to the CFL of non-smoothed residuals as reported by Blazek [2005, p.307]. A change in the CFL number has no effect on the execution time of a single flow iteration as it does not change the workflow of the algorithm. Consequently, the extra execution time of the smoothing is spent solely on the smoothing function.

[†]The cycle contains a run on a fine mesh and another in a coarse mesh.

Table 4.1: *Effect of the multigrid on the execution time of 1000 iterations on a pseudo 3D test case (cf. Appendix case 2).*

N_{Cells}	T_{1Grid} [s]	T_{2Grids} [s]	Ratio $T_{\text{2G}}/T_{\text{1G}}$
25 k	12.1	22	1.81
46 k	16.1	27.7	1.72
57 k	18.7	30	1.61
75 k	22.4	35	1.58
110 k	30.2	45	1.49
126 k	34.3	50	1.46
394 k	95.5	129	1.35

The smoothing reads in 3D:

$$\begin{aligned} -\epsilon^I R^*_{I-1,J,K} + (1+2\epsilon^I) R^*_{I,J,K} - \epsilon^I R^*_{I+1,J,K} &= \vec{R}_{I,J,K} \\ -\epsilon^J R^{**}_{I,J-1,K} + (1+2\epsilon^J) R^{**}_{I,J,K} - \epsilon^J R^{**}_{I,J+1,K} &= \vec{R}^*_{I,J,K} \\ -\epsilon^K R^{***}_{I,J,K-1} + (1+2\epsilon^K) R^{***}_{I,J,K} - \epsilon^K R^{***}_{I,J,K+1} &= \vec{R}^{**}_{I,J,K} \end{aligned} \tag{4.15}$$

where $\vec{R}^*$,$\vec{R}^{**}$ and $\vec{R}^{***}$ denote the smoothed residuals in the three flow directions and ϵ_I, ϵ_J and ϵ_K are the smoothing parameters. These smoothing parameters are defined based on the viscous spectral radii [Liu and Jameson, 1993] and they control the amount of smoothing (error damping) in every direction:

$$\begin{aligned} \epsilon^I_v &= max(0, C(\tfrac{\sigma^*}{\sigma}) \tfrac{\Lambda^I_v}{\Lambda^I_c + \Lambda^J_c + \Lambda^K_c}) \\ \epsilon^J_v &= max(0, C(\tfrac{\sigma^*}{\sigma}) \tfrac{\Lambda^J_v}{\Lambda^I_c + \Lambda^J_c + \Lambda^K_c}) \\ \epsilon^K_v &= max(0, C(\tfrac{\sigma^*}{\sigma}) \tfrac{\Lambda^K_v}{\Lambda^I_c + \Lambda^J_c + \Lambda^K_c}) \end{aligned} \tag{4.16}$$

with $C \approx 5/4$ and $\frac{\sigma^*}{\sigma}$ the maximum ratio of the CFL numbers before and after smoothing. The convective and viscous spectral radii Λ_c and Λ_v are defined in [Blazek, 2005, p.189] based on the work of Rizzi and Inouye [1973] and Müller and Rizzi [1986].

The three smoothing equations (4.15) are interdependent and need to be solved consecutively. First, the smoothing in the I direction has to be performed, which computes the I-smoothed residual $\vec{R}^*$ as a solution of the tridiagonal system of equations with the non-smoothed residual $\vec{R}$ as a right-hand side (RHS). Then the smoothing in the J direction uses the I-smoothed residual as a RHS and solves a tridiagonal system to compute the $\vec{R}^{**}$. Finally, the smoothing in the K direction uses the J-smoothed residual as a RHS and solves a tridiagonal system to compute the $\vec{R}^{***}$. The three diagonal system of equations has the form $Ax = B$ with:

$$A = \begin{pmatrix} (1+2\epsilon) & -\epsilon & & & & & \\ -\epsilon & (1+2\epsilon) & -\epsilon & & 0 & & \\ & -\epsilon & (1+2\epsilon) & -\epsilon & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & -\epsilon & \\ & & 0 & & -\epsilon & (1+2*\epsilon) & \end{pmatrix} \text{ and}$$

$$B = \begin{pmatrix} \rho_1 & \rho_1 u_1 & \rho_1 v_1 & \rho_1 w_1 & \rho_1 E_1 \\ \rho_2 & \rho_2 u_2 & \rho_2 v_2 & \rho_2 w_2 & \rho_2 E_2 \\ & & \vdots & & \\ \rho_N & \rho_N u_N & \rho_N v_N & \rho_N w_N & \rho_N E_N \end{pmatrix}.$$

The smoothing in one direction is done line by line in the mesh as depicted in Figure 4.25 for a 2D case. Every line of cells produces a subsystem $Ax = b$. For a 2D case, N_J subsystems have to be solved for the smoothing in the I direction with N_J equal to the number of cells in the J direction. For the smoothing in the J direction, N_I subsystems have to be solved. The 3D case used in this work builds on the same subsystems and makes use of the third direction. The smoothing in the I direction solves as many subsystems as the number of cells in the J direction multiplied by the number of cells in K directions ($N = N_i N_j$).

The smoothing itself is not *embarrassingly parallel* since it is based on solving three tridiagonal systems of equations, for which the baseline CPU implementation uses the Thomas algorithm [Thomas, 1949]. The Thomas algorithm is a simplified version of the Gaussian elimination when applied to tridiagonal matrices. Since the Gaussian elimination is inherently serial, other *GPU-friendlier* alternatives have been studied such as cyclic reduction (CR), parallel cyclic reduction (PCR) and Recursive Doubling [Chang and Wen-mei, 2014; Zhang et al., 2010; Stone et al., 2011; Chang et al., 2012].

New GPU kernels have been implemented to prepare the tridiagonal matrix and fill it following equation (4.16) and a tridiagonal solver is used to solve the system. The approach is validated by comparing the flow residuals with the ones computed by the reference CPU implementation that uses the Thomas algorithm also.

In the following, two different tridiagonal solvers from the CUSPARSE library and the self-implemented GPU-based Thomas algorithm are presented and benchmarked. First, the function `gtsv` of the `CUSPARSE` library [†], which consists of a combination of CR and PCR with pivoting, has been tested to solve the tridiagonal systems of equations. The CUSPARSE tridiagonal solver `gtsv` is derived from the work of Chang et al. [2012], who developed a high-performance tridiagonal solver for GPUs based on the SPIKE algorithm [Polizzi and Sameh, 2006]. This function is able to solve large systems of equations and not only a set of small systems. The SPIKE algorithm divides the large system of equations with banded system matrix into a set of smaller independent subsystems that can be solved simultaneously. In the treated case a number of small independent subsystems - one for every line of cells smoothed - is to be solved and not a large system of equations. These subsystems have been concatenated, as depicted in Figure 4.25, to form a global system

[†] `http://docs.nvidia.com/cuda/cusparse`,retrieved March 2017

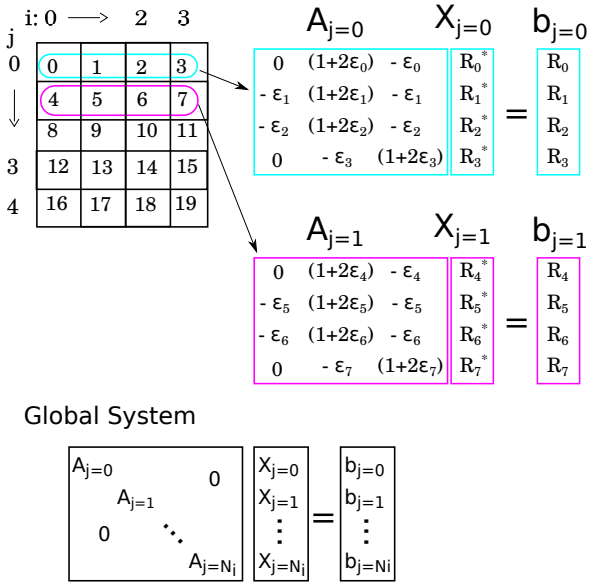


Figure 4.25: *Example of the creation of the tridiagonal linear system for 2D smoothing.*

matrix for the `gtsv` function.

The implicit residual smoothing reduced the number of flow iterations required to reach steady state by a factor of 2 to 3 but every flow iteration has been circa 4x times slower. Detailed profiling showed that the majority of the execution time devoted for the smoothing is caused by the call to the tridiagonal solver of CUSPARSE. Unlike all self-written kernels showed in this work, there is no overlapping and thus no positive effect of the multistreaming for CUSPARSE solvers, even though every mesh block is running in a different CUDA stream. Mesh blocks are thus smoothed one after the other. This implicit synchronization should not occur especially when solving small mesh blocks, which are unlikely to saturate the GPU resources. The profiler showed a frequent allocation of device memory. The CUSPARSE functions allocate, indeed, a memory space for temporary objects which is freed right after being used. While this is a proper use of the memory in order to limit the function memory consumption, it prevents the host CPU from launching other stream kernels. Memory allocation alters, in fact, the state of the memory in the device and the host CPU does not perform any further operation until the memory allocation is finished to prevent memory misusage and data corruption. Consequently, a single memory allocation blocks the flow of operation in all streams provoking a large loss of performance. Therefore all functions on other streams are blocked leading to the observed absence of overlapping.

Another function of CUSPARSE (`gtsv_nopivot`) has been therefore tested which is able to solve a set of small subsystems based on cyclic reduction. This function is faster than the SPIKE-based method especially for subsystem sizes equal to a power of 2.

Finally, the Thomas algorithm has been implemented on the GPU with every CUDA thread solving a subsystem. The register usage of the function, not exceeding 40 register files, is not prohibitive compared to other kernels written for the RANS solver. Nevertheless, there are two sequential loops for the forward sweep and the backward substitution, which can not be unrolled because of the data dependency. The parallelization granularity is at the subsystem level. The Thomas algorithm showed a good level of overlapping with multistreaming but the low number of launched threads makes the kernel latency-bound.

Figure 4.26(a) treats the case of running an entire mesh of 7 blocks with multistreaming activated without IRS and then with the three available methods. Figure 4.26(b) depicts the same setting with the multistreaming deactivated. The self implemented Thomas methods have the smallest slowdown for most of the treated meshes mainly due to a better use of multistreaming. The gain over CR and SPIKE algorithms decreases a lot when the size of the mesh increases due to the longer forward elimination and backward substitution loops in the Thomas kernel. Cyclic reduction and SPIKE algorithm are getting more efficient with the increase of the mesh size as the multistreaming effect is fading.

A suggested further optimization is to optimize the Thomas algorithm by using the shared-memory for small meshes. For large meshes, it is useful to implement a tridiagonal solver based on CR or parallel CR (PCR) while avoiding any memory action that cancels the overlapping over the streams.

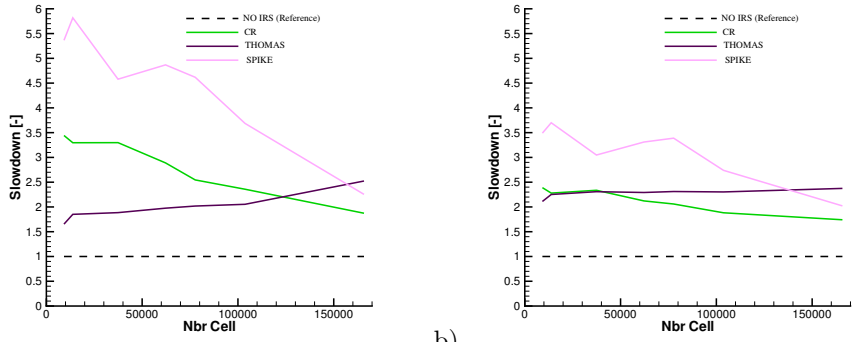
The residual smoothing is a very efficient tool for the convergence acceleration of the explicit solver. Its implementation on the CPU is straightforward with a negligible added execution cost. Implementing the same technique on the GPU is much more delicate and causes a slow-down with all tested library implementations. This confirms the idea that convergence acceleration techniques are hardware-specific and for the GPU the available convergence acceleration techniques are still to be optimized.

4.3 Validation and benchmark

For viscous turbulent flows, the nozzle guide vane LS89 [Arts et al., 1990] has been used for validation. The inlet flow is subsonic but experiences an acceleration on the suction side of the vane reaching supersonic conditions which leads to a normal shock (Figure 4.27(a)). Figure 4.27(b) shows the isentropic Mach number profile of the blade matching the experimental results. The shock occurring on $\frac{X}{X_0} \approx 0.9$ is well resolved and accurately predicted.

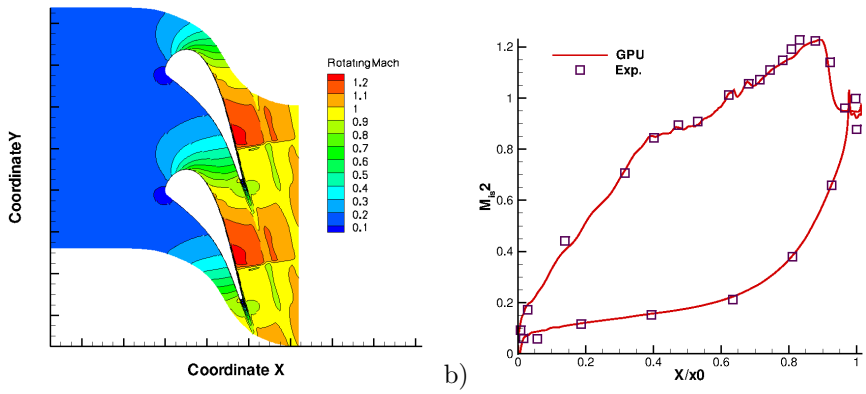
The execution time of different kernels has been measured by the CUDA profiler on a Kepler K40 card and plotted in Figure 4.28 for the full list of kernels and Figure 4.29 for a reduced list of main kernels. The same profiling has been done on different meshes with an increasing size. In general, GPU kernels are not very efficient for small meshes but the performance increases rapidly with the mesh size. In a second phase, the performance improvement is not as important and the kernel is asymptotically saturating.

When a kernel has a constant ratio of execution time for an increasing mesh size, it means the kernel has a linearly improving performance with the number of cells



a) b)

Figure 4.26: *Slowdown caused by the IRS measured over 100 flow iterations with the NO-IRS version as a reference for different meshes: a)multistreaming activated b) no Multistreaming.*



a) b)

Figure 4.27: *a)Mach contours of the transonic LS89 [Arts et al., 1990] turbine guide vane test case, b) Computed and experimental distributions of the isentropic Mach number on the LS89 ($M_{2_{is}} = 1.02$ $P_{01} = 1.605bar$).*

Table 4.2: *Comparison of the execution time for 1000 iterations of the second-order RANS solver on a pseudo 3D test case (cf. Appendix case 2).*

N_{Cells}	time on Xeon E5 [min]	time on K40 [min]	speedup
25 k	9.10	0.26	35.0
46 k	16.96	0.34	49.8
57 k	21.00	0.38	54.9
75 k	28.09	0.45	61.6
110 k	40.91	0.59	68.4
126 k	47.59	0.66	71.2
394 k	159.75	1.77	90.0

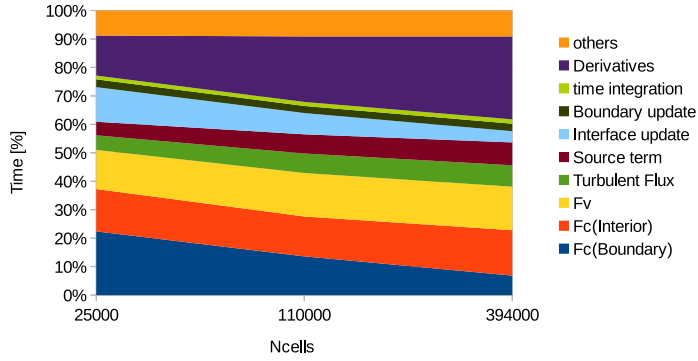


Figure 4.28: *Plot of the execution time of GPU kernels for different meshes with second-order accuracy for the convective flux evaluation (F_c).*

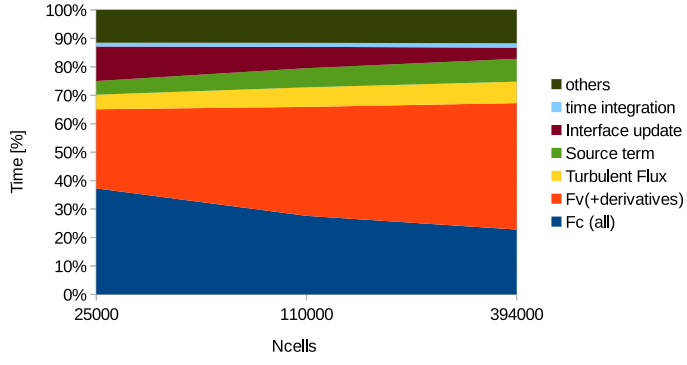


Figure 4.29: *Plot of the execution time of the main GPU kernels for different meshes with second-order accuracy for the convective flux evaluation (F_c).*

and probably near saturation. Kernels losing in importance for an increasing mesh size are those that are still improving their performance with a fast pace and far from saturating the GPU. Kernels gaining in importance with the increase of the mesh size are already saturating the GPU and the more cells they treat the more costly they are and earn in importance.

The viscous flux is developing into a hotspot for dense meshes driven by the cost of the evaluation of the velocities and temperature derivatives using the Green theorem. The part of the convective flux is, however, shrinking. This is induced by the improved performance of the boundary flux update for large meshes. Boundary flux evaluation and interface update are the hotspots for small meshes as the surface-area-to-volume ratio is more important. The turbulent flux and the source term are compute-bound kernels for averagely sized meshes. For large meshes they are also earning in importance and could develop into a hotspot for very large meshes. Other GPUs could give slightly different results as the memory bandwidth and the computing power varies.

For an ultimate benchmark, the most efficient kernel implementations have been combined in the final GPU RANS solver to run a set of meshes of a compressor cascade with an increasing mesh size (Cf. Appendix A.2). The execution times of the GPU solver are compared against the execution times of the serial CPU solver on one core in Table 4.2 reaching a large speedup of two orders of magnitude.

4.4 Conclusion

This chapter presents the GPU implementation of an in-house RANS solver with the explicit time-stepping. The GPU code had been gradually optimized by tuning the code to best occupancy and register consumption first, then by finding the right algorithm for the best memory bandwidth. A lookup table for the interface update made it possible to have a full GPU solver without the need for a frequent communication with the host CPU. The performance benchmark analyzed the major kernels and different implementation strategies such as mapping threads to cells, space directions and faces but also comparison between large kernels and equivalent split into sub-kernels. Moreover, the execution costs on a GPU of two convergence acceleration methods namely multigrid and implicit residual smoothing have been presented. For the explicit solver the GPU achieved a speedup approaching 100x making the GPU explicit solver, even though characterized by small CFL numbers, a serious competitor to implicit solvers.

5

GPU-accelerated CFD Simulations with Implicit Time-Stepping

The RANS solver with explicit time-stepping, presented in the previous chapter, required only stencil-based operations, which are easily implemented in parallel contexts. In some cases, however, explicit methods fail to converge or converge very slowly and are consequently not able to provide valid CFD results in a decent turnaround time. Therefore, this chapter studies the possibilities of accelerating a CFD RANS solver with implicit time-stepping on the GPU.

Compared to the explicit solver, the implicit solver considers the entire computational domain for the time-stepping reducing thus the dependence on the CFL number as a guarantee for stability. At the same time, it brings into focus another aspect of parallel programming which is solving efficiently a linear system of equations in parallel. First, the theory behind the implicit time integration is explained in detail and then the implementation approach is introduced. A benchmark of several libraries for linear system solvers and preconditioners is performed to identify the fastest solver on the GPU. Finally, the observations gained from the benchmark inspired a discussion on the effect of some parameters, such as the number of RK stages, the CFL number and the linear stop condition on the reached GPU speedup. A validation of the solver along with a summary close the chapter.

This chapter is based on the articles:

M.H. Aissa, L. Müller, T. Verstraete, and C. Vuik. Acceleration of turbomachinery steady simulations on GPU. In Desprez F. et al., editor, *Euro-Par 2016: Parallel Processing Workshops. Lecture Notes in Computer Science, vol 10104.*, pages 814–825. Springer, Cham, 2017a ,

M.H. Aissa, T. Verstraete, and C. Vuik. Toward a GPU-aware comparison of explicit and implicit CFD simulations on structured meshes. *Computers & Mathematics with Applications*, 74(1):201–2017, 2017b .

5.1 Introduction

Steady CFD simulations are widely used, amongst others, for design evaluation. These simulations advance an initial flow solution based on an explicit or implicit time integration scheme. Implicit schemes are more stable and faster to converge due to a larger allowed time step. This property comes, however, at a high cost of assembling and solving multiple linear systems of equations ($Ax = b$) at every flow iteration. The system assembly comprises the computation of the system matrix A and the right-hand side b. Since every mesh cell interacts only with few neighbor cells, the unknowns are loosely coupled and the generated system matrix is thus sparsely populated. Due to the sparsity character, iterative solvers are mostly used to solve the sparse linear system of equations.

With the growth of the problem size, the use of High-Performance Computing (HPC) becomes inevitable. In this field, Graphics Processing Units (GPUs) are gaining in importance through the reported speedups of many CFD applications [Brandvik and Pullan, 2007; Fu et al., 2014; Niemeyer and Sung, 2014b][†]. While dense matrix-vector operations are very efficiently solved on the GPU [Barrachina et al., 2008], solving a sparse linear system of equations is more challenging, since there are fewer independent operations for the large GPU computational power. Moreover, some linear systems require a factorization-based preconditioner to accelerate the otherwise very slow converge, which enhances the serial aspect of the algorithm and thus reduces drastically the GPU performance gain.

The reference CFD simulation is performed on the CPU using the library package PETSc [Balay et al., 2014]. The CPU simulation spends 70% of the execution time on the system assembly, while the rest is devoted for the linear solver. The same balance is also found in some FEM applications: Wong et al. [2015] ported, for instance, a CPU application based on PETSc with 80% of the execution time dedicated for the system assembly. This observation was the motivation to port the assembly part as well to the GPU in order to avoid any data transfer to the CPU during the simulation and thus enhance the speedup.

The GPU-based version of the CFD simulation uses the preconditioned flexible GMRES (the Generalized Minimal Residual Algorithm for Solving Non-symmetric Linear Systems (GMRES) [Saad and Schultz, 1986]) of the Paralution library[‡], which uses building blocks of the efficient cuSparse library. The Paralution library is reported to allow a speedup of factor 5x for a neutron diffusion problem [Trost et al., 2015]. Paralution performs, however, the assembly of the system matrix on the host, which implies a data transfer from the GPU to the host CPU. To address this issue, an interface is developed to connect the GPU-assembled system of equations to the linear solver.

The preference for a library over a self-implementation of the linear solver is motivated mainly by the maturity of today's GPU-based linear solvers. Moreover, the design of a fast linear solver is not part of the scope of this work but rather the optimization of its frequent use in steady simulations. For this purpose, an algorithm (*on-demand* LU factorization) is proposed, which is capable of reducing the number

[†]see literature review in chapter 3 for more details

[‡]PARALUTION Labs "PARALUTION v1.1.0", 2016, `http://www.paralution.com`

of times an ILU preconditioner matrix is built for the linear solver without altering the flow accuracy. This technique enables the linear solver to re-use the previously computed preconditioning matrix instead of computing a new one for every iteration. When combined with standard ILU, the technique improves drastically the reached speedups for coarse and fine meshes.

The rest of the chapter is structured as follow: section 2 introduces the numerical scheme used by the CFD solver while section 3 describes the implementation of the solver on the GPU. A discussion is presented in section 4 and the main findings are summarized in section 5.

5.2 Numerical scheme

The flow solver uses a cell-centered finite volume discretization on multi-block structured grids. It solves the Reynolds-Averaged Navier-Stokes (RANS) equations in the time-dependent integral form:

$$\frac{\partial}{\partial t}\int_{\Omega}\vec{W}d\Omega + \oint_{\partial\Omega}(\vec{F_c}-\vec{F_v})dS = \int_{\Omega}\vec{Q}d\Omega\ , \tag{5.1}$$

with $\vec{W} = \{\rho, \rho V_x, \rho V_y, \rho V_z, \rho E\}$ the vector of conservative variables, Ω the cell volume and $\partial\Omega$ the cell surface. The convective fluxes $\vec{F_c}$ are computed using a Roe upwind approximation [Roe, 1981] of a Riemann Solver while second-order accuracy is achieved through the Monotone Upstream-Centered Schemes for Conservation Law (MUSCL [Van Leer, 1979]). The viscous fluxes $\vec{F_v}$ are approximated using a central discretization scheme. The source term $\vec{Q}$ contains contributions from the Spalart-Allmaras (SA) one-equation turbulence model [Allmaras and Johnson, 2012] with an upwind first-order discretization within the Finite-Volume framework. The space discretization is detailed in the previous chapter (see section 4.1).

Analogously to the explicit solver, the implicit solvers uses also the "method of lines", for which the space and the time integration are treated separately. Equation (5.1) can be reformulated to highlight the time integration as follows:

$$\frac{\Omega I}{\Delta t}\Delta\vec{W}^n = -\beta\vec{R}^{(n+1)} - (1-\beta)\vec{R}^n\ , \tag{5.2}$$

with $\Delta\vec{W} = \vec{W}^{n+1} - \vec{W}^n$ the solution update, which depends on a combination of residuals of time point n and $n+1$. For $\beta = 0$, the right-hand side (RHS) of equation 5.2 is known and updates can be computed explicitly, as treated in the previous chapter. If however $\beta \neq 0$, the residual is linearized to allow to formulate R^{n+1} as a function of R^n and the Jacobian $\frac{\delta\vec{R}}{\delta\vec{W}}$ to first order accuracy:

$$\vec{R}^{n+1} \approx \vec{R}^n + \left(\frac{\delta\vec{R}}{\delta\vec{W}}\right)\Delta\vec{W}^n\ . \tag{5.3}$$

Substituting the linearization into the initial equation gives a linear system of equations of the form $Ax = b$:

$$\left[\frac{\Omega I}{\Delta t} + \left(\frac{\delta\vec{R}}{\delta\vec{W}}\right)\right]\Delta\vec{W}^n = \vec{R}^n\ . \tag{5.4}$$

with $\vec{R}$ the residual containing the fluxes and the source term, $\Delta\vec{W} = \vec{W}^{n+1} - \vec{W}^n$ the solution change, I the identity matrix and $\frac{\delta\vec{R}}{\delta\vec{W}}$ an approximate *Jacobian* matrix as formulated in [Blazek, 2005, p.450] including the most important feature of the space discretization scheme. When equation (5.4) is applied to the entire mesh a large linear system is built with the form $Ax = b$. Residuals and Jacobians are first evaluated on cell surfaces and then summed up in a local assembly procedure (see equation (5.2)). The global assembly concatenates the local items to a large global matrix and a right-hand-side containing all the problem unknowns. A multistage time-stepping method such as the Jacobian-Trained Krylov Implicit-Runge-Kutta scheme (JT-KIRK [Xu et al., 2015]) solves then multiple successive linear systems -one per stage- for every flow iteration, in which only the right-hand side is updated then multiplied by a different stage coefficient α:

$$\begin{aligned} \vec{W}^{(0)} &= \vec{W}^n \\ A^{(0)}[\vec{W}^{(1)} - \vec{W}^{(0)}] &= -\alpha_1\vec{R}(\vec{W}^{(0)}) \\ A^{(0)}[\vec{W}^{(2)} - \vec{W}^{(1)}] &= -\alpha_2\vec{R}(\vec{W}^{(1)}) \\ &\vdots \\ A^{(0)}[\vec{W}^{(m)} - \vec{W}^{(m-1)}] &= -\alpha_m\vec{R}(\vec{W}^{(m-1)}) \\ \vec{W}^{n+1} &= \vec{W}^{(m)}. \end{aligned} \tag{5.5}$$

The high CFL numbers reached by the implicit time-stepping lead to large time steps Δt, which decreases the diagonal dominance of the system matrix and increases the condition number resulting in an ill-conditioned linear system. Such an ill-conditioned system matrix requires further treatment to enhance the linear system convergence. One of the efficient techniques used with ill-conditioned systems is preconditioning, which is any form of modification to the original linear system able to accelerate the convergence of an iterative method [Saad, 2003]. The next section treats in detail the techniques used for solving the linear system.

5.3 Preconditioned Krylov solvers

This section presents few insights into the background of the GMRES method and the ILU preconditioning. Elements shown in this section are crucial to understanding the linear solver algorithm in the first place and the different tuning parameters associated with it.

In the context of solving a linear system $Ax = b$, many iterative methods are based on a projection, which is a way to extract an approximation $x_m \approx A^{-1}b$ of a solution from a subspace $\mathcal{K}_m$ [Saad, 2003, p.153]. The approximation has to satisfy a set of m constraints $b - Ax_m \perp \mathcal{L}_m$ from the subspace $\mathcal{L}_m$. A Krylov method extracts the approximation from a Krylov subspace $\mathcal{K}_m$, which is spanned by m independent vectors formed as a matrix-vector product of r_0 and a polynomial of A:

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2r_0, .., A^{m-1}r_0\} \text{ with } r_0 = b - Ax_0\ . \tag{5.6}$$

The different Krylov methods make use of different subspaces $\mathcal{L}_m$ for the constraints [Saad, 2003, p.177] and the GMRES uses the constraint space $\mathcal{L}_m = A\mathcal{K}_m$. The different Krylov vectors are generated using the modified Gram-Schmidt method within

the Arnoldi's process [Arnoldi, 1951]. The Arnoldi process, shown in Algorithm 1, starts with a vector $||v_1||_2 = 1$ (line 1). It generates for every iteration j (lines 3-12), a vector w_j by multiplying the previous vector v_j with the matrix A (line 3). The vector w_j is then orthonormalized against all previous base vectors following a modified Gram-Schmidt procedure (lines 4-7). Once the base vectors are generated,

Algorithm 1 Arnoldi-Modified Gram-Schmidt[Saad, 2003, p. 162].

1: Choose v_1 of norm 1
2: **for** j=1 **to** m **do**
3: $w_j := Av_j$
4: **for** i=1 **to** j **do**
5: $h_{ij} =< w_i, v_i >$
6: $w_j := w_j - h_{ij}v_i$
7: **end for**
8: $h_{j+1,j} = ||w_j||_2$
9: **if** $h_{j+1,j} = 0$ **then**
10: Stop
11: **end if**
12: $v_{j+1} = w_j/h_{j+1,j}$
13: **end for**

the approximate solution is then computed as follows [Barrett et al., 1994]:

$$x^{(j)} = x^0 + y_1x^1 + ... + y_jx^j, \tag{5.7}$$

with y_j the coefficients which resulted from the minimization of the residual norm $||b - Ax^{(k)}||_2$. The full algorithm of the basic GMRES is shown in Algorithm 2.

Algorithm 2 basic GMRES[Saad, 2003, p. 172].

1: $r_0 = b - Ax_0$, $\beta := ||r_0||_2$ and $v_1 := r_0/\beta$
2: **for** j=1 **to** m **do**
3: $w_j := Av_j$
4: **for** i=1 **to** j **do**
5: $h_{ij} = (w_i, v_i)$
6: $w_j := w_j - h_{ij}v_i$
7: **end for**
8: $h_{j+1,j} = ||w_j||_2$
9: **if** $h_{j+1,j} = 0$ **then**
10: $m := j$ goto 14
11: **end if**
12: $v_{j+1} = w_j/h_{j+1,j}$
13: **end for**
14: Define (m+1)xm Hessenberg matrix $\bar{H} = h_{ij}{}_{1\leq i<m+1,1\leq j<m}$
15: $y_m \leftarrow$ minimize($||\beta e_1 - \bar{H}_m y||$) {wiht e1 a base vector}
16: $x_m = x_0 + V_m y_m$

The approximate solution is explicitly formed only if the orthogonalization process is finished after m steps or w disappears (lines 9-10). This condition applies to exact arithmetic but computers have a finite precision and w can decrease without reaching zero. The condition is therefore replaced by a mathematically equivalent condition, which is adapted to the practical precision of a computer. This new stopping condition monitors the norm of the residual $r = b - Ax$ and the process is stopped when the residual is below a certain limit. Instead of forming the approximate solution in every iteration the residual is evaluated as follows ([Saad, 2003, p.177] and [Kelley, 1995, p.38]):

$$||r_k|| = ||V^{k+1}(\beta e_1 - H_k y^k)||_2 \ . \tag{5.8}$$

During every step of the algorithm, a new vector is generated which increases the computational and memory consumption. In order to limit the algorithm resources consumption, a modified version of the method restarts after generating a set of m vectors (see Algorithm 3). If the convergence, which is related to the stopping condition, is not reached the solution is constructed and a new cycle starts with $x_0 := x_m$. Keeping m low (few iterations) reduces effectively the resource consumption but at the same time can deteriorate the convergence property of the method [Barrett et al., 1994, Ch.2] and increases the number of times the solution is explicitly constructed. When associated with the incomplete LU factorization the number of GMRES iterations is low and the restart GMRES (30) or even GMRES (10) is converging well.

Algorithm 3 restart GMRES (modified notation from [Saad, 2003, p. 179]).

1: $r_0 = b - Ax_0$, $\beta := ||r_0||_2$ and $v_1 := r_0/\beta$
2: **while** $||r||_2 > \epsilon||b||_2$ **do**
3: **for** j=1 **to** m **do**
4: $w_j := Av_j$
5: **for** i=1 **to** j **do**
6: $h_{ij} = (w_i, v_i)$
7: $w_j := w_j - h_{ij}v_i$
8: **end for**
9: $h_{j+1,j} = ||w_j||_2$
10: **if** $h_{j+1,j} = 0$ **then**
11: $m := j$ goto 15
12: **end if**
13: $v_{j+1} = w_j/h_{j+1,j}$
14: **end for**
15: Define $(m+1)$xm Hessenberg matrix $\bar{H} = h_{ij}{}_{1\leq i<m+1, 1\leq j<m}$
16: $y_m \leftarrow$ minimize$(||\beta e_1 - \bar{H}_m y||)$
17: $x_m = x_0 + V_m y_m$
18: $x_0 := x_m$
19: **end while**

For the preconditioning, three versions are possible; namely the left, the right and the split versions; but they are all similar in terms of convergence acceleration

unless the preconditioning matrix M is ill-conditioned [Saad, 2003, p.287]. The less intrusive version is the right preconditioning as only the system matrix and the unknowns are concerned while the RHS is the same:

$$AM^{-1}u = b, x \equiv M^{-1}u \ . \tag{5.9}$$

Algorithm 4, showing the preconditioned GMRES(m), is slightly different from Algorithm 3 of the unpreconditioned GMRES(m). The inverse of the preconditioning matrix M is applied $m + 1$ times (see lines 4 and 13). The flexible version of GMRES [Saad, 2003, p 287] is slightly different from Algorithm 3 as it accepts a new matrix M in every iteration (line 4).

Algorithm 4 preconditioned GMRES (modified notation from [Saad, 2003, p. 284]).

1: $r_0 = b - Ax_0$, $\beta := ||r_0||_2$ and $v_1 := r_0/\beta$
2: **while** $||r||_2 > \epsilon||b||_2$ **do**
3: **for** j=1 **to** m **do**
4: $w_j := \boldsymbol{AM^{-1}}v_j$
5: **for** i=1 **to** j **do**
6: $h_{ij} = (w_i, v_i)$
7: $w_j := w_j - h_{ij}v_i$
8: **end for**
9: $h_{j+1,j} = ||w_j||_2$ and $v_{j+1} = w_j/h_{j+1,j}$
10: $V_m := [v1, ..., v_m]$, $\bar{H} = h_{ij}{}_{1\leq i<m+1, 1\leq j<m}$
11: **end for**
12: $y_m = \text{argmin}_y||\beta e_1 - \bar{H}_m y||)$
13: $x_m = x_0 + \boldsymbol{M^{-1}}V_m y_m$
14: $x_0 := x_m$
15: **end while**

The next point is about choosing the way of generating the matrix M. Many preconditioners of the Paralution library have been tested and only the incomplete LU factorization has shown a good convergence for all considered test cases (see Appendix). Therefore, a detailed analysis of the generation of the LU matrix is given below focusing on the parallelization strategies. The LU factorization is the Gaussian elimination for sparse matrices. The factorization generates however many fill-ins, which are non-zero values in positions where the system matrix had zeros. When considering only the L_{ij} and U_{ij} which have a counterpart in the system matrix, many values are discarded and the factorization is said to be *incomplete*. Such a factorization can not be used as a direct solver since $M^{-1}b$ is only a very crude approximation of x. However, the incomplete LU is very efficient when used as a preconditioner coupled with a Krylov iterative method. From a performance point of view for the GPU, the ILU(0) with no fill-in is the fastest among the family of ILU related methods. ILU(0) factorization forces the same sparsity pattern[†] of the system matrix on the preconditioner matrix.

[†]A sparsity pattern is a set of matrix positions, which accepts non-zero elements

Algorithm 5 ILU(0) with S the sparsity pattern of A [Saad, 2003, p. 307]).

```
1: for i=2 to n do
2:   for k=1 to i-1 do
3:     if (i,k) ∈ S then
4:       a_ik = a_ik/a_kk
5:       for j=k+1 to n do
6:         if (i,j) ∈ S then
7:           a_ij = a_ik/a_kj
8:         end if
9:       end for
10:    end if
11:  end for
12: end for
```

The ILU algorithm (see Algorithm 5) has many serial loops, which run much faster on the CPU than on the GPU. In order to expose more parallelism during the assembly of the ILU matrices, it is possible to identify independent rows that can be updated concurrently [Saad, 2003]. Few options are available to improve the performance of the ILU factorization on the GPU, among them multicoloring [Naumov et al., 2015; Lukarski, 2012; Li and Saad, 2013] and level-scheduling [Naumov, 2011].

While the methods cited above increase slightly the parallelism of the factorization, other methods such as the iterative ILU [Chow and Patel, 2015] propose a novel algorithm. The factorization is replaced by a minimization problem able to provide an approximate L and U with more fine-grained parallelism. The method relies on a fixed point iteration $x^{n+1} = G(x^n)$ which is guaranteed to converge [Chow and Patel, 2015; Frommer and Szyld, 2000] after a set of sweeps[†]. The GPU implementation of this method, presented in [Chow et al., 2015, p.5], evokes a trade-off between the convergence and the parallelism as the fixed point iteration tends to use less frequently updated values when more threads are used. While it could be a promising research direction to overcome the serial Gaussian elimination in ILU, it is still not stable for the cases treated in this work.

Another class of algorithms for preconditioning considers approximating the inverse of the matrix A by explicitly computing $M^{-1} \approx A$. The inverse of the matrix A can be approximated as the solution of the constrained minimization problem $min||I - AM||_F$, with $||.||_F$ referring to the Frobenius norm of a matrix. One of these methods, proposed by Grote and Huckle [1997], is among the most successful as reported by Benzi [2002] and is widely ported to the GPU [Lukash et al., 2012; Anzt et al., 2016a].

To sum up, a preconditioner should be cheap to construct and to apply but at the same time, the preconditioned system should be easier to solve [Benzi, 2002]. These two requirements are however conflicting. The more efficient is a preconditioner in reducing the number of linear solver iterations, the more likely it is to be time-consuming to compute. On the GPU, this effect is more pronounced es-

[†]a sweep is one full update of the L and U matrices

pecially in building preconditioners through factorization. The ILU preconditioner, for instance, has an important cost for the setup and a smaller extra cost for every GMRES iteration. These costs have to be amortized over the gain in GMRES iterations and more important over the use of the same preconditioner for multiple consecutive systems.

5.4 Flow solver implementation

The novelty in the implementation of an implicit solver compared to an explicit solver is the need to prepare and solve a linear system of equations. In the preparation of the linear system, a residual has to be first computed and stand for the right-hand side then, the Jacobians are evaluated in order to fill the system matrix.

The reference CPU-based implicit solver, written in C++, computes the residual and the flux Jacobians serially by looping over all mesh faces before it solves the linear system of equations using the PETSc package. For all coming benchmarks, the CPU version was run on the Intel double quad-core Xeon(R) CPU E5-2640 with a clock rate of 2.50 GHz and a 15MB cache size.

The GPU implicit solver, on the other hand, builds on the implementation of the explicit solver presented in the previous chapter for the residual evaluation. New kernels are written to compute the Jacobians and form the system matrix (see Figure 5.1). Once the data for the linear system of equations is available, a linear solver can be used to iteratively compute the solution. Solving a linear system of equations is a generic operation with applications in many fields, consequently a rich literature exists on the topic along with a multitude of libraries with few among them operating on the GPU. For all coming benchmarks, the GPU code was run on Tesla K40 GPU (Kepler architecture) with a memory bandwidth of 280GB/s, a theoretical peak performance of 1,682 Gflops in double precision and 12 GB of global memory. The GPU implementation is realized with CUDA 7.0.

The first subsection discusses some alternatives for the implementation of the system assembly regarding both the Jacobian and the residual calculation. The next subsection treats the solving of the linear system with a benchmark of multiple preconditioners.

5.4.1 System assembly

Within the finite volume scheme, the global assembly of the linear system of equations is made by looping over the cell faces in the mesh. On every cell face, a contribution to the cell local system matrix is computed. The global system matrix is then a concatenation of local block matrices, which are divided into diagonal and off-diagonal blocks. The dominance of the diagonal blocks, which contain a contribution proportional to the inverse of the time step, improves the convergence of the linear solver. The off-diagonal blocks contain solely the flux Jacobians defining the bandwidth of the matrix, which is the number of rows separating the non-zero elements from the matrix diagonal. Matrices arising from 3D flows, for instance, have a larger bandwidth than matrices arising from 2D CFD applications. A larger

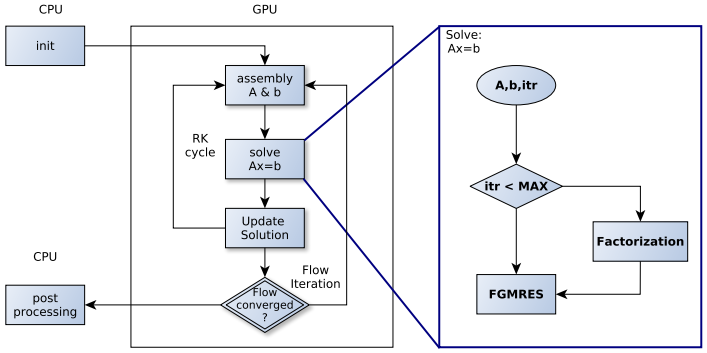


Figure 5.1: *Flow Solver algorithm showing the assembly and the linear solver with on-demand factorization.*

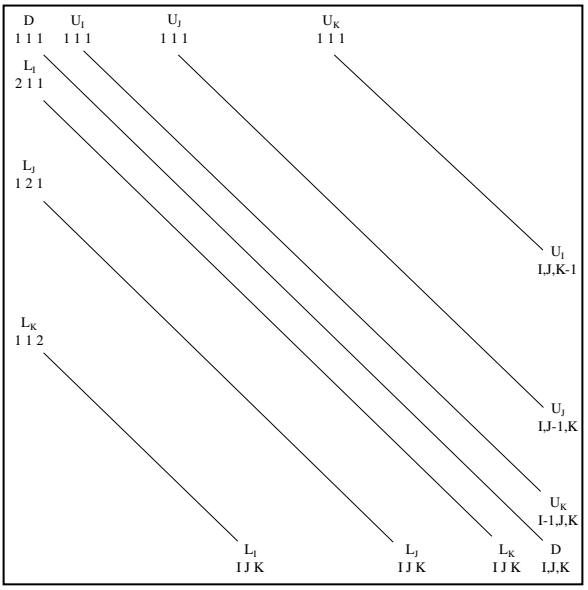


Figure 5.2: *Structure of the system matrix showing 7 diagonals filled with block matrices of $N_v * N_v$ with N_v the number of flow variables. I, J and K are the number of cells in the first, second and third directions, respectively.*

bandwidth increases, in general, the number of iterations to solve the related linear system of equations.

Figure 5.2 shows the structure of the system matrix arising from the discretization of a 3D flow on a structured mesh. The number of elements per row is not constant and varies depending on the number of diagonals crossing the row. As detailed in the previous chapter (see Section 4.2), each face computes a flux $F(U_L, U_R)$ which is then respectively added and subtracted from the residual of the left and right cell ($R_L = +F, R_R = -F$). These two face contributions to the residual lead to 4 contributions to the Jacobian matrix:

$$\begin{array}{rcl} \partial R_L/\partial U_L & = & \partial F/\partial U_L, \\ \partial R_L/\partial U_R & = & \partial F/\partial U_R, \\ \partial R_R/\partial U_L & = & -\partial F/\partial U_L, \\ \partial R_R/\partial U_R & = & -\partial F/\partial U_R, \end{array} \tag{5.10}$$

but only two values are computed: $\partial F/\partial U_L, \partial F/\partial U_R$. The 4 contributions are depicted in Figure 5.3 with two diagonal ($\partial R_L/\partial U_L, \partial R_R/\partial U_R$) and two off-diagonal ($\partial R_L/\partial U_R, \partial R_R/\partial U_L$) contributions.

The off-diagonal contributions are generated by faces common to two neighbors. These contributions are set to null if no neighbor exists in that direction. A cell in the interior of the computational domain, for instance, has neighbors in all directions and the corresponding row has thus 7 non-zero blocks [†] for 3D problems. On the other hand, a Jacobian of a face on the block boundary has fewer neighbors. Every missing neighbor decreases by one the number of non-zero blocks per row. The number of elements per row plays a role on the performance of sparse matrix-vector operations on the GPU as many library implementations assign a row to a CUDA thread block. It is thus more efficient to have a number of non-zero elements ($N_{\text{var}} \times N_{\text{blocks}}$) proportional to the warp size [‡] (actually 32).

Since every cell receives the contributions of six faces, a risk of *race condition* is eminent, in which up to 6 threads simultaneously update the same diagonal position in the system matrix. To avoid race conditions, *atomic add* or *graph-coloring* are generally used. *Atomic add* makes the hardware responsible for the serialization of all threads when they reach the atomic operation. Although it is the simplest method, it penalizes drastically the fine-grained parallelism. *Graph coloring* consists of giving a face a different color from its neighbors. Consequently, independent faces have the same color at the end of the coloring process and they can be processed concurrently for the flux and the Jacobian calculation without creating a race condition. This approach disturbs, however, the global memory coalesced access leading to performance losses.

For the convective flux evaluation in the explicit solver (see Section 4.2.1), three thread-mapping methods have been discussed: mapping to cells, to space directions and to faces. These mappings are not all as useful for the Jacobian evaluation, which is different from the flux evaluation. Every face generates only one flux which acts on two neighbor cells, while it generates two Jacobians, left and right, and acts on 4 positions in the system matrix, namely, the diagonal positions relative to the two

[†] a block here is a N_{var}xN_{var} dense matrix

[‡] Warp is the smallest unit of execution for the GPU

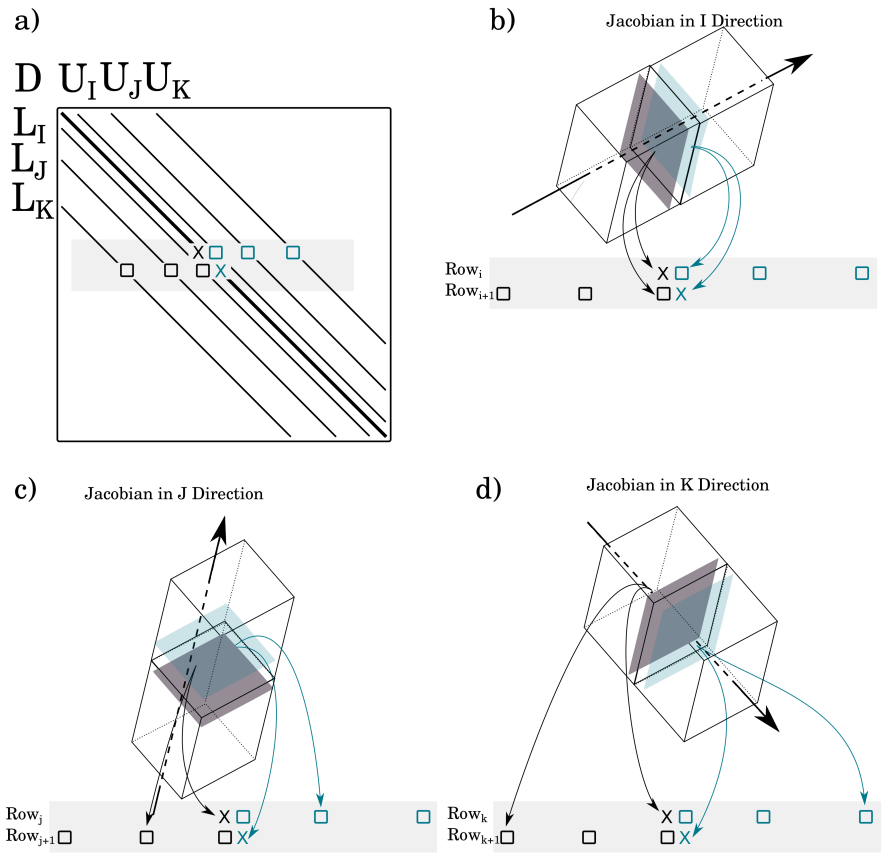


Figure 5.3: *Storage of the Jacobian contributions into the global system matrix.*

neighbor cells in addition to the two off-diagonal contributions (see Figure 5.5a). For a cell-based approach or a direction-based approach, which have inherently redundant computations of the Jacobian, the two opposed faces of a single cell contribute to the same diagonal position in the system matrix. Therefore, these contributions should be serialized to solve the race condition, which deteriorates heavily the performance by having every face Jacobian computed 4 times. For the Jacobian evaluation, only the face-based approach is thus considered in this work.

In the generation of the approximate Jacobian some data is related to a face and is reused for all 4 contributions. The face-based GPU kernel is thus responsible for the computation of both the left and right[†] Jacobian of a face and its storage. The approximate Jacobian involves a lengthy computation [Blazek, 2005, p.450] and the GPU kernel is consequently very consuming in terms of registers (204-215 registers) and its theoretical occupancy is thus very low (12.5%). The occupancy is, however, not the only contributor to the performance especially for an implementation that allows threads to perform multiple independent operations (Cf. [Volkov, 2010]).

This subsection discusses the race condition free evaluation of the Jacobian and the storage of the Jacobian contributions in the system matrix. As shown in Figure 5.4, the storage can be coalesced in arrays following the COO format (see Figure 5.6) then sorted into the CSR format. Contributions can be also stored uncoalesced directly in their final positions in the system matrix following the CSR format. In the following, first three approaches for the Jacobian evaluation are benchmarked namely *multicoloring*, *redundant* and *all-store* approaches. At this stage all three approaches write the contributions in coalesced COO format. Afterward, two of the three approaches - *multicoloring* and *redundant*- are benchmarked with uncoalesced direct insertion of the Jacobian contributions into the CSR format. For all benchmarks (see Table 5.1) two meshes are used (see Table 5.2) a coarse and a dense one of the LS89 case (Cf. Appendix A.1).

Coalesced Jacobian storage with sorting

Since a face generates a contribution for the diagonal elements corresponding to two neighbor cells, a serialization of the evaluation is needed to solve the race condition. A first attempt is to make a redundant call to the Jacobian evaluation and the second attempt is to use multicoloring as depicted in Figure 5.5(b) and 5.5(c), respectively. For the redundant computation, every face is processed twice by the Jacobian kernel to compute and store all contributions without race condition. In every run, the

[†]Left and right refer to cell, with respect to which the gradients of the flux are evaluated.

Table 5.1: *Different Jacobian evaluation and storage methods. Those considered for this work marked with (✓).*

Sorting	Multicoloring	Redundant	All-store
yes	✓	✓	✓
No (Direct insertion)	✓	✓	-

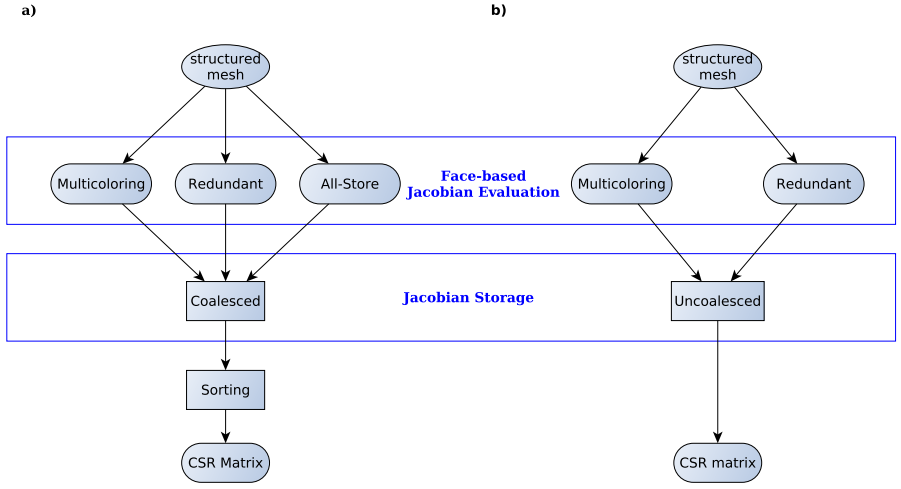


Figure 5.4: *Chart of the considered combinations of Jacobian evaluation and storage pattern: (a) with Sorting and (b) direct insertion without sorting.*

number of started threads is equal to N_{faces}. The number of faces depends on the considered computational domain. For every space direction, the number of faces is equal to the number of cells plus one. For the I direction, for instance, $N^I_{\text{faces}} = N^I_{\text{cell}} + 1$.

The multicoloring approach (see Figure 5.5c), on the other hand, colors differently odd-indexed faces and even-index faces for every space direction which makes a total of 6 colors for the entire 3D mesh. There is no redundancy, in the sense of processing a face twice, since Jacobian contributions of faces from the same color are stored in different positions in the system matrix but the number of started threads is halved compared to the number of threads the redundant version starts. The execution time of both methods is shown in table 5.3 averaged over 100 flow iterations for a coarse mesh and in Table 5.4 for a dense mesh. The number of cells for the coarse and dense meshes are given in Table 5.2. The version with redundant computations shows a higher performance by 3.69% to 5.9% compared to the multicoloring (MC) version.

A third alternative, labeled here *all-store*, stores all computed contributions in

Table 5.2: *Characteristics of the used meshes to solve the flow around the LS89 turbine stator blade (Cf. Appendix A.1).*

Name	# cells
Coarse Mesh	38848
Dense Mesh	310784

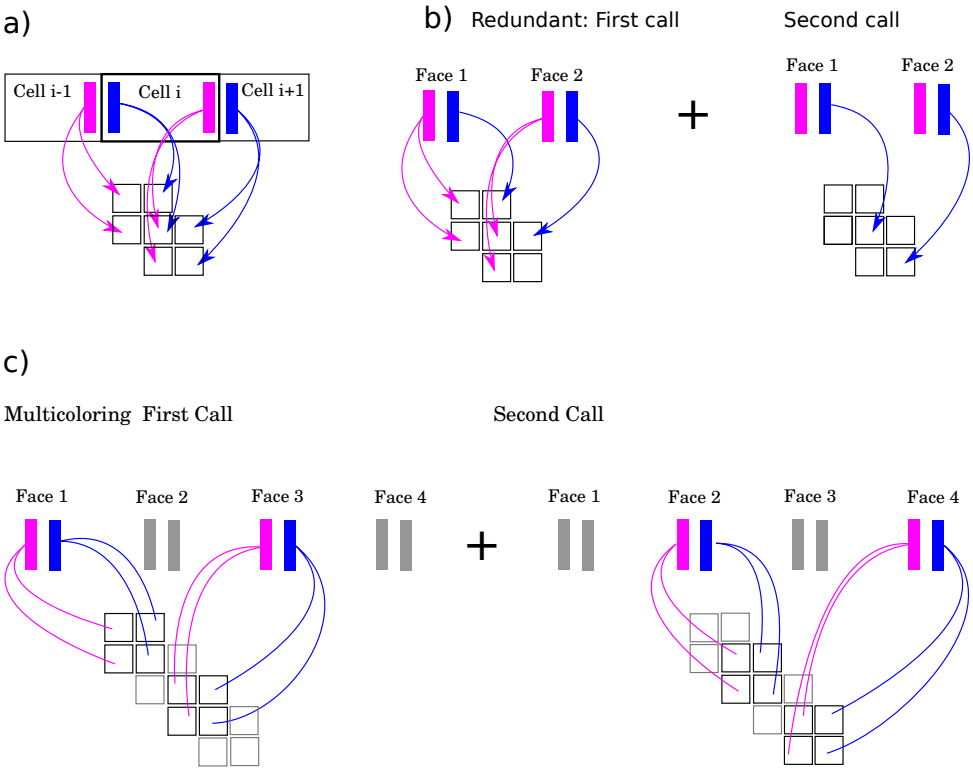


Figure 5.5: *(a) Face contributions to the system matrix, (b) redundant work approach and (c) multicoloring approach.*

Table 5.3: *Execution time on the Tesla-K40 of the three alternatives for a single evaluation of the convective Jacobian on a small mesh of 38848 cells (averaged over 100 flow iterations of the 2-stage Runge-Kutta).*

Convective Jacobian	Multicoloring kernels	Redundant computation	All-store kernel
I direction[ms]	2.29	1.92	1.66
J direction[ms]	1.12	1.28	1.09
K direction[ms]	1.90	1.80	1.54
Total[ms]	5.31	4.99	4.29

Table 5.4: *Execution time on the Tesla-K40 of the three alternatives for a single evaluation of the convective Jacobian on a dense mesh of 310787 cells (averaged over 100 flow iterations, 2-stage Runge-Kutta).*

Convective Jacobian	Multicoloring kernels	Redundant computation	All-store kernel
I direction[ms]	11.97	10.63	9.4
J direction[ms]	10.2	9.93	8.78
K direction[ms]	8.86	9.34	8.19
Total[ms]	31.03	29.90	26.37

different positions in a large array following the Coordinate Format (COO) shown in Figure 5.6. This alternative combines first, the advantage of the multicoloring version as it has has no redundancy and second, it starts as many threads as the redundant version. These two advantages come at the cost of sorting the data inside the large "values" array into the CSR format in addition to the memory footprint of these arrays.

Table 5.5 shows the profiling results of the 3 alternatives on a coarse mesh block. It is obvious that the difference in the reached occupancy is at the origin of the performance gap between the multicoloring version and the two other approaches. The lower occupancy of the multicoloring approach is caused by the halved number of started threads.

Even though the multicoloring kernel reaches comparable memory bandwidths with the other alternatives both for reading and writing (L2 read/write), it is rather an inflated bandwidth caused by the lack of coalescence. The effect of the data coalescence is measured as the ratio of required bandwidth and actual bandwidth. The closer the values of the two bandwidths the more coalesced is the access. Bad memory access results into a much larger memory traffic than required, which is counted as memory transactions per memory operation. Therefore, to satisfy a single memory request generated by the code the multicoloring kernel performs multiple memory transactions. The lack of coalescence is very visible for the evaluation of the Jacobian in the i space direction as the threads are mapped directly to the i index. Every variable is stored continuously for the whole mesh. The mesh is scanned in the i direction for all indices of $j \in [0..J_{\text{Max}}]$. The same occurs in all (i,j) planes for $k \in [0..K_{\text{Max}}]$. Therefore, two consecutive threads are mapped to i and $i+1$ respectively, while moving in the j direction is related to a strided access with a stride equal to the number of cells in the i direction. Moving in the k direction is related to larger stride equal to the number of cells in i direction multiplied by the number of cells in j direction. The coalesced access is almost guaranteed [†] when the stride is equal to one which corresponds to the above-described storage scheme.

Multicoloring in the i direction breaks the coalescence and causes thus a strided access with a step size equal to two. Such a strided access can easily half the memory

[†]Access should be also aligned, see Section 2.5

Table 5.5: *Profiling results on the Tesla-K40 of the three alternatives for the evaluation of the convective Jacobian on a small mesh block (3584 cells).*

Profiling variable	Multicoloring kernels	Redundant computation	All-store kernel
Requested store Throughput	22.17 GB/s	40.52 GB/s	44.04 GB/s
Global store Throughput	47.33 GB/s	49.46 GB/s	53.76 GB/s
Store efficiency	46.88%	81.91%	81.91%
Requested load Throughput	14.21 GB/s	27.41 GB/s	28.18 GB/s
Global load Throughput	28.95 GB/s	31.62 GB/s	32.24 GB/s
Load efficiency	49.07%	86.69%	87.42%
L2 read	40.77 GB/s	51.25 GB/s	53.81 GB/s
L2 write	53.25 GB/s	62.97 GB/s	64.78 GB/s
Occupancy	0.058	0.108	0.108
Multiprocessors activity	94.7	94.8	94.6
DRAM utilization	30%	40%	40%
Compute utilization	10%	10%	10%

bandwidth [†]. The evaluation of the convective Jacobian in j and k directions are not suffering any deterioration of the coalescence for the multicoloring version. The redundant version and the *all-store* versions request, consequently, a higher bandwidth as both reach a better occupancy and show a higher coalescence (load/store efficiency).

Table 5.6 shows the profiling results of the three alternatives on a dense mesh block. The multicoloring kernels reach an occupancy very close to the occupancy reached by the other two approaches, which can be explained referring to the general trend observed with the explicit solver (see Figure 4.15) relating the GPU performance to the number of started threads. First, an exponential growth phase occurs when the number of threads increases then a pre-saturation precedes a saturation phase [‡]. Consequently, decreasing the number of threads damages the performance only when it causes a kernel to start fewer threads than required to reach the saturation phase of the used GPU. For the dense mesh, all kernels -including the MC kernel- saturate the GPU. The coalescence is, on the other hand, related to the memory access pattern and is consequently not improved by the size of started threads for the dense mesh. The multicoloring kernel shows thus a poor store/load efficiency.

For the convective flux evaluation (Cf Section 4.2.1), the multicoloring alternative had a better performance than the redundant version. The difference with the Jacobian evaluation is that the flux kernel was compute-bound and the redundant computation caused the arithmetic activity to increase. This increase caused the kernel to be slower as the computation units of the GPU used for the benchmark [§]

[†]`https://www.karlrupp.net/2016/02/strided-memory-access-on-cpus-gpus-and-mic/`

[‡]Saturation means in this context that the GPU reached a maximum performance that can not be improved by starting more threads

[§]GTX780

Table 5.6: *Profiling results on the Tesla-K40 of the three alternatives for the evaluation of the convective Jacobian (in I direction) on a dense mesh block (28672 cells).*

Profiling variable	Multicoloring kernels	Redundant computation	All-store kernel
Requested store Throughput	37.30 GB/s	47.31 GB/s	49.98 GB/s
Global store Throughput	79.61 GB/s	57.75 GB/s	61.02 GB/s
Store efficiency	46.88%	81.91%	81.91%
Requested load Throughput	23.91 GB/s	32.01 GB/s	31.98 GB/s
Global load Throughput	48.71 GB/s	36.92 GB/s	36.59 GB/s
Load efficiency	49.07%	86.69%	87.42%
L2 read	67.84 GB/s	59.22 GB/s	60.52 GB/s
L2 write	89.56 GB/s	73.53 GB/s	73.51 GB/s
Occupancy	0.117	0.124	0.124
Multiprocessors activity	98.65	98.75	98.9
DRAM utilization	60%	50%	50%
Compute utilization	10%	10%	10%

were saturated reaching 90% of their compute utilization. The Jacobian kernel is conversely memory-bound and the redundant computations increase the compute utilization which is far from saturation.

From previous benchmarking, it has been observed that the *all-store* approach is the fastest alternative among the coalesced storage approaches (see Figure 5.4). Before opting for this alternative, which is greedy in terms of storage, the cost of the sorting of the system matrix has to be assessed.

In order to store the contribution into the system matrix, it is possible to write directly the Jacobian into the system matrix but following a scattered memory access pattern, or first write them unsorted into large arrays following a coalesced access pattern and then sort the arrays to get the global matrix. While the first approach does not guarantee coalesced memory access, the second approach has a costly data sorting from Coordinate format (possibly with redundant entries) into CSR format. Figure 5.6 illustrates the difference between the Coordinate format (COO) and the compressed sparse row format (CSR). Both methods will be analyzed and benchmarked.

The contributions of the flux Jacobian are stored along with their row and column indices into the COO format. Computing and storing all face contributions lead to three large arrays: two for indices (column array, row array) and a third array for the contribution's value. Contributions belonging to the same cell are identified over identical index in column and row arrays then summed up using *sort* and *reduce* functions of the `THRUST` library [Bell and Hoberock, 2011]. This library generates three arrays free of repetition hosting the positions and values of all non-zero elements (nnz) of the system matrix. This COO data arrangement stores nnz values in double precision and $2 * nnz$ integers. To reduce the storage size while keeping

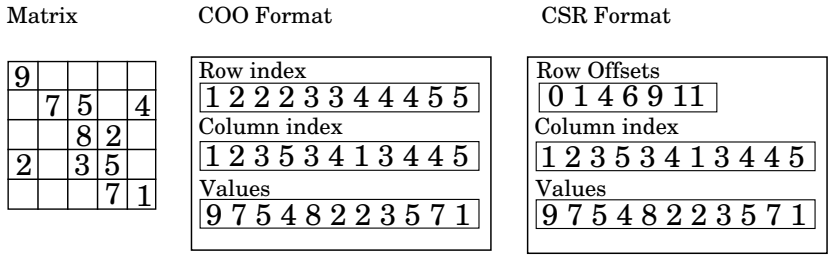


Figure 5.6: *illustrative example of the Coordinate (COO) and compressed spare row (CSR) formats for matrix storage.*

the same information content, the row array can be transformed into the row offset array, in which the column offset of the first non-zero element in every row is stored. This operation is performed by the CUSP library [Bell and Garland, 2015], which provides the CSR arrays that constitute the input for the iterative solver of Paralution. A similar but less complicated algorithm allows to sort and scan the right-hand side for duplicate entries.

For the case of the LS89 coarse mesh shown in table 5.7, the sorting costs in total 42.07 seconds for a converged simulation. The entire simulation takes 377.31 seconds with 317.8 seconds for the time integration. So the sorting costs almost 70% of the assembly time (Jacobian and residual calculation). This value reaches 80% for a dense mesh, which makes the cost of the sorting equal to a quarter of the cost of the time integration. Recalling that the time integration contains solving a large system of equations using a time-consuming ILU preconditioner, the sorting is really prohibitive for dense meshes and should be avoided.

Uncoalesced Jacobian storage without Sorting

Since the sorting is a time-consuming operation on the GPU with a cost that increases with the size of the array to sort, a sorting-free method labeled *direct-insertion* of the Jacobian contributions in the system matrix is implemented. As the method accesses directly the "values" array of the system matrix (see Figure 5.6), the *all-store* alternative can not be used for the evaluation of the Jacobian as it is not thread-safe. The remaining options are the multicoloring and the redundant computation.

Table 5.7: *Breakdown of the execution time on the Tesla-K40 of a converged simulation with Jacobian sorting of 534 flow iterations both for coarse and dense mesh.*

mesh	Jacobian and Residual [s]	Time integration [s]	Sorting kernel	Total
LS89 coarse	17.43	317.8	42.07	377.31
LS89 dense	54.3	999.7	251	1305

The direct-insertion method for the Jacobian storage computes the position of the contribution in the "values" array used in the CSR format. The same kernels are used to evaluate the Jacobian contributions (multi-coloring and redundant) but instead of storing them unsorted in a huge array, the new function inserts the contribution directly into its final position in the global system matrix. The writing function uses the row number, which is related to the cell indexes (i,j,k), and reads the offset of the row from the array of row offsets. With the read offset, the segment from the "values" array is known, which should host the entries of this row (see Figure 5.7). The difficulties rely on finding the exact offset in the row.

The algorithm takes then the cell position (i,j,k), the mesh block size (number of cells in I, J and K directions) and the nature of the contribution, which is a number `inputID` $\in$ [-3 .. 3]. A contribution arising from a common face with a neighbor in negative K direction has an `inputID` $= -3$. A contribution arising from a common face with a neighbor in positive K direction has an `inputID` $= 3$. Analogously, the J direction presents the `inputID`$= \pm 2$ and the I direction has an `inputID`$= \pm 1$. The diagonal contribution has the `inputID`$= 0$. The value of the `inputID` obey the convention on how the indices (i,j,k) are converted into a matrix index. The algorithm computing the offset of the column is shown in Listing 5.1

The method of direct insertion of the Jacobian in the system matrix cuts the costs of arranging the data at the expense of some added instructions to compute the right position of each contribution in the CSR format. Another advantage of the direct insertion is to reduce the memory cost by suppressing the need for very large arrays for the different Jacobian contributions. This is crucial for very large meshes as it increases the maximum size of accepted meshes for the limited GPU memory.

The direct insertion runs a short algorithm (see Listing 5.1) for every contribution to compute its column offset after reading the row offset from the row offset array. This added arithmetic and memory operations to the kernels increase the register usage which further decreases the occupancy. The performance of the multicoloring kernel and the "redundant" kernel are expected to decrease with the direct insertion method. A benchmark is essential to compare the effect of the performance decrease for realistic cases to the saved sorting cost. Therefore, the same test case benchmarked above for the versions using sorting (see Table 5.3) is benchmarked with direct insertion. The results, shown in Table 5.8, prove that the cost of the direct insertion of the Jacobian increased the execution time of the flux evaluation by 70 to 80% for the coarse mesh. The benchmark showed also that the redundant computation have a slightly higher performance.

In order to identify the cause of the performance deterioration, key parameters have been profiled and compared between the coalesced method with sorting and the uncoalesced sorting-free method for both multicoloring and Redundant computation (see Table 5.9). The number of memory transactions per request increases from around 3 to 20 for the load and reached 30 for the storage which reflects a scattered memory access. The store/load efficiency is almost halved causing the requested memory bandwidth to decrease severely for the direct insertion methods. This is expected since the direct method writes in a scattered way a dense block matrix of $N_{\text{var}} \times N_{\text{var}}$ elements per thread.

The ultimate benchmark includes the *all-store* alternative with sorting and the

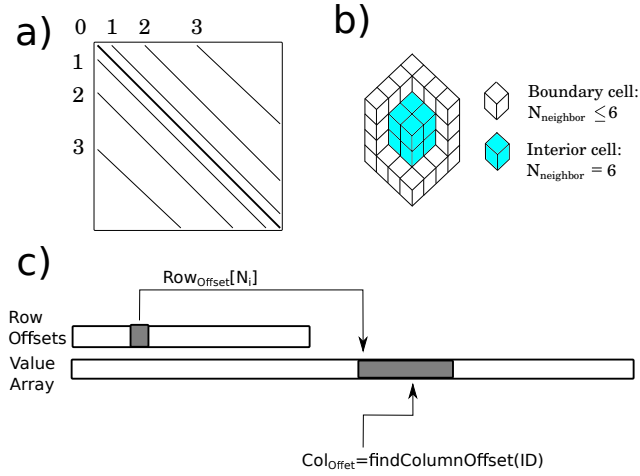


Figure 5.7: *Identification of the position of a value in the array of value within the CSR format.*

Table 5.8: *Execution time on the Tesla-K40 of the three alternatives for a single evaluation of the convective Jacobian on a small mesh of 38848 cells (averaged over 100 flow iterations, 2-stage Runge-Kutta).*

Convective Jacobian	Multicoloring kernels	Redundant computation
I direction[ms]	3.68	3.41
J direction[ms]	2.00	2.16
K direction[ms]	3.8	3.45
Total[ms]	9.48	9.02

Listing 5.1: *Code of the direct insertion of a contribution into the system matrix.*

```
int DirectEntryInsertion (int input_ID){
  columnOffset=0; // value valid for inputID=-3
  switch(input_ID){
     case -2:
       //check if neighbor exists in negative K direction.
       if(k!=KMIN) columnOffset+=5; break;
     case -1:
       if(k!=KMIN) columnOffset+=5;
       //check if neighbor exists in negative J direction.
       if(j!=JMIN) columnOffset+=5; break;
     case 0:
       if(k!=KMIN) columnOffset+=5;
       if(j!=JMIN) columnOffset+=5;
       //neighbor exists in negative I direction.
       if(i!=IMIN) columnOffset+=5; break;
     case 1:
       // offset for diagonal  contribution
       columnOffset=5;
       if(k!=KMIN) columnOffset+=5;
       if(j!=JMIN) columnOffset+=5;
       if(i!=IMIN) columnOffset+=5;break;
     case 2:
       columnOffset=5;
       if(k!=KMIN) columnOffset+=5;
       if(j!=JMIN) columnOffset+=5;
       if(i!=IMIN) columnOffset+=5;
       // in case a neighbor exist in positive I direction.
       if(i!=IMAX) columnOffset+=5; break;
     case 3:
       columnOffset=5;
       if(k!=KMIN) columnOffset+=5;
       if(j!=JMIN) columnOffset+=5;
       if(i!=IMIN) columnOffset+=5;
       if(i!=IMAX) columnOffset+=5;
       // in case a neighbor exist in positive J direction.
       if(i!=JMAX) columnOffset+=5; break;
  }
  return columnOffset;
}
```

Table 5.9: *Profiling results on the Tesla-K40 of the three alternatives for the evaluation of the convective Jacobian on a small mesh block (3584 cells).*

Profiling variable	Multicoloring kernels	Redundant computation
Requested store Throughput	7.61 GB/s	10.96 GB/s
Global store Throughput	30.47 GB/s	43.87 GB/s
Store efficiency	25%	25%
Requested load Throughput	10.87 GB/s	16.45 GB/s
Global load Throughput	40.38 GB/s	54.29 GB/s
Load efficiency	26.92%	30.3%
L2 read	48.42 GB/s	64.76 GB/s
L2 write	34.55 GB/s	51.18 GB/s
Occupancy	0.058	0.108
Multiprocessors activity	95.7	96.8
DRAM utilization	20%	30%
Compute utilization	10%	10%

Table 5.10: *Breakdown of the execution time on the Tesla-K40 of a converged simulation without Jacobian sorting of 534 flow iterations both for coarse and dense mesh.*

mesh	Jacobian and Residual [s]	Time integration [s]	Sorting kernel	Total
LS89 coarse	25.55	322.43	10.25	358.234
LS89 dense	103.15	1007.5	31.35	1142

redundant computation without sorting for a whole simulation. Table 5.10 shows the breakdown of the execution time for a converged simulation into basic operations including the time integration, the sorting and the assembly. The small value in the execution time of the sorting is not related to the Jacobian but to the residual sorting. The cost of the sorting shrunk, nevertheless, considerably especially for the dense mesh and overcompensated the extra execution time for the assembly. To conclude, the direct insertion with redundant computation is the fastest alternative for the system assembly.

Separate kernels are implemented for the residual and the Jacobian calculation and both need to store the results, the former in a vector, the latter in a sparse matrix format. For both kernels, sorting can be applied to ensure a coalesced storage access. Table 5.10 showed a small cost of sorting related to the sorting of the residual array.

Table 5.11 shows a benchmark of the version with direct insertion for the RHS. Even though the sorting time is basically zero the saved turnaround time is marginal. This is caused by the extra cost of the residual evaluation when using the direct insertion.

For the system matrix, the sorting was too slow as it follows two indices, namely the row and the column. In the case of the residual evaluation, the sorting is more simple for two reasons. The arrays are shorter (5 variables per cell compared to 25 as for the system matrix) and the sorting is one dimensional following one index. For the flux calculation, the cost of the sorting of the residual array is not prohibitive while it keeps the coalescence, assures more threads per kernel and solves the race condition. In contrast to the system assembly, the sorting for the RHS is better for the performance than the multicoloring and direct insertion.

To conclude it is worth to highlight that coalescence and sorting are more efficient for small data (residual or system matrix of small mesh). Redundant kernels and direct entry are, on the other hand, the best choice to generate the system matrix of (very) large meshes.

5.4.2 Linear solver with *on-demand* factorization

From the analysis of the execution time of the system assembly and the time integration introduced in the last subsection, it has been observed that the time integration is the most time-demanding part of the simulation. This subsection, therefore, analyses the reason behind the slow GPU time integration and proposes

Table 5.11: *Breakdown of the execution time on the Tesla-K40 of a converged simulation (534 flow iterations) with direct assembly (no sorting of the Jacobian and the residual) both for coarse and dense mesh.*

mesh	Jacobian and Residual [s]	Time integration [s]	Sorting kernel	Total
LS89 coarse	33.32	320.65	0.0	353.98
LS89 dense	128.3551	995.73	0	1124.09

some improvements.

First, a set of preconditioners will be studied toward finding the best one, recalling that sometimes the preconditioner speed can be traded against Krylov iterations. The next level includes the benchmarking of the whole simulation (run until flow convergence). Finally, the optimization of the sequence of consecutive application of the preconditioner is considered.

Table 5.12 shows the four libraries, which have been considered for solving the linear system of equations on the GPU: PETSc (GPU version [Balay et al., 2015]), ViennaCL [Rupp, 2017], MAGMA sparse [Anzt et al., 2014] and Paralution [PARALUTION Labs, 2016]. The first requirement for a GPU library to be efficient within an implicit RANS solver is to be able to process data already residing in the GPU. Many libraries provide wrappers to incorporate externally computed data but some libraries do not. While PETSc requires only a small change on the data type of the system matrix and the right-hand side to run the linear solver on the GPU, the library does not accept external data computed on the GPU which reduces the scope of the parallelization to the linear solver minimizing the expected global speedup. Moreover, it does not provide a GPU implementation of the incomplete LU factorization[†]. ViennaCL, an OpenCL-based library, can process data residing in the GPU but it performs a costly data copy from CUDA to OpenCL type. The third alternative is Paralution, which can process data residing in the GPU and has no overhead on incorporating external data. It uses also many functions from the cuSparse library. Like Paralution, the last alternative, MAGMA sparse, accepts external data and wraps them very fast. It also proposes other ILU algorithms such as iterative ILU [Chow and Patel, 2015] and Incomplete Sparse Approximate Inverse (ISAI [Grote and Huckle, 1997]) for ILU application inside GMRES.

Since MAGMA sparse and PARALUTION provide wrappers to incorporate GPU data with a negligible cost, a short interface is added to the implicit solver in order to finalize the system assembly, launch the linear solver and retrieve the solution update. Within the multistage Runge-Kutta scheme, the solution update is the required data to move to the next Runge-Kutta stage and start the computation of the next residual. The system assembly itself is done only on the fist Runge-Kutta stage.

In order to benchmark the different preconditioners of different libraries, the

[†]Iterative ILU are being integrated into PETSc[Rupp]

Table 5.12: *Four different GPU linear solver packages considered for this work.*

Library	Accept external Data	Fast incorporation of external data	benchmark in this wo
PETSc-GPU [Balay et al., 2015]	-	-	-
ViennaCL [Rupp, 2017]	✓	-	-
MAGMA sparse [Anzt et al., 2014]	✓	✓	✓
Paralution [PARALUTION Labs, 2016]	✓	✓	✓

ratio of the cost of the preconditioner setup to the cost of one Krylov iteration is compared for different mesh sizes both on the CPU and on the GPU. PETSc reports on the performance when run with the flag `-log_view`. Within the performance related output of PETSc, there is the total execution time of the Krylov solver, the execution time of the preconditioner setup and the Krylov iterations. For the GPU version with Paralution, timings have been collected using the CPU `std::clock` function.

The test case is the low-pressure turbine blade T106C [Michálek et al., 2012] depicted in Figure 5.8. For all coming benchmarks, the meshes presented in Table 5.13 have been used. Larger meshes for the test case have been created by keeping the number of cells in the xy-plane constant while increasing the number of cells on the z-direction. The T106C turbine stator experiences a 2D flow, therefore the same flow phenomena are solved for all generated meshes. This ensures, that only the amount of computational work is increasing which is the focus of the computational performance study.

For different mesh sizes of the test case T106C, table 5.14 and Table 5.15 report the execution time of the preconditioner setup and one single Krylov iteration on the Xeon E5 CPU and on the Tesla K40 GPU, respectively. The ILU factorization is an inherently serial operation as discussed in chapter 3 and is very slow on the GPU, 15 to 40 times slower than a GPU Krylov iteration. The factorization is also 5 to 7 times slower than a Krylov iteration on the CPU. Therefore, any optimization of the update rate for the ILU preconditioner should be benchmarked both on the CPU and on the GPU.

Table 5.16 compares the execution time of the preconditioner setup and a single Krylov iteration on the CPU and on the GPU. A Krylov iteration is performed faster on a CPU for a small mesh while it can reach a speedup of almost 4x on the K40 for a very large system matrix. A Krylov iteration is mainly based on matrix-vector operations and such a speedup for large matrices is expected. The preconditioner setup, on the other hand, is always faster on the CPU for conventional ILU preconditioners.

Another type of preconditioner is based on the approximation of the inverse of the system matrix. The most basic one is the Jacobi preconditioner which contains only the inverse of the diagonal elements. On the GPU, the Jacobi preconditioner has two advantages compared to the GPU ILU preconditioner. First, the preconditioner setup cost is much shorter and second, the speedup per linear iteration is larger even

Table 5.13: *Characteristics of used meshes and underlying linear systems to solve the flow around the T106C turbine stator blade.*

# cells	#rows	# non-zero	# *nnz/row*
52928	52928	264640	[20 .. 30]
211712	211712	1058560	[20 .. 35]
449888	449888	2249440	[20 .. 35]
926240	926240	4631200	[20 .. 35]

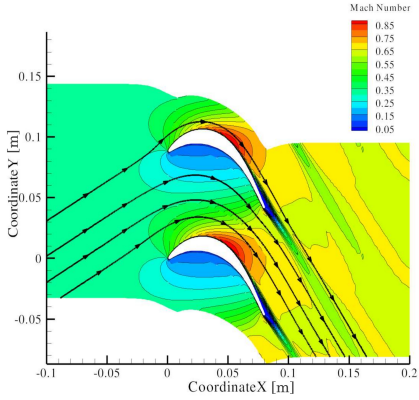


Figure 5.8: *Mach number contours around the T106C nozzle guide vane.*

Table 5.14: *Execution time on the Xeon E5 CPU for the ILU setup and a single Krylov FGMRES iteration using PETSc for different mesh sizes of the case T106C.*

System Matrix N_{row}	1 Krylov iteration T_{itr} [s]	preconditioner setup T_{ILU}[s]	Ratio $T_{\text{ILU}}/T_{\text{itr}}$
264640	0.020	0.114	5.832
1058560	0.088	0.590	6.740
2249440	0.197	1.373	6.985
4631200	0.424	3.076	7.356

Table 5.15: *Execution time on the Tesla K40 for the ILU setup and a single Krylov FGMRES iteration using Paralution for different mesh sizes of the case T106C.*

System Matrix N_{row}	1 Krylov iteration T_{itr} [s]	preconditioner setup T_{ILU}[s]	Ratio $T_{\text{ILU}}/T_{\text{itr}}$
264640	0.034	0.495	15.094
1058560	0.047	1.076	23.386
2249440	0.067	2.016	30.862
4631200	0.106	3.954	38.046

Table 5.16: *Speedup of the ILU setup and a single Krylov iteration for different mesh sizes of the case T106C on the K40 GPU compared to the Xeon E5 CPU.*

System Matrix N_{row}	speedup of 1 Krylov iteration $T_{\text{ITR}}^{\text{CPU}}/T_{\text{ITR}}^{\text{GPU}}$	speedup of ILU setup $T_{\text{ILU}}^{\text{CPU}}/T_{\text{ILU}}^{\text{GPU}}$
264640	0.584	0.230
1058560	1.856	0.548
2249440	2.944	0.681
4631200	3.980	0.778

Table 5.17: *Execution time on the Xeon E5 CPU for the Jacobi setup and a single Krylov iteration for different mesh sizes of the case T106C.*

System Matrix N_{row}	1 Krylov iteration T_{itr} [s]	preconditioner setup T_{Jacobi}[s]	Ratio $T_{\text{Jacobi}}/T_{\text{itr}}$
264640	0.014	0.022	1.543
1058560	0.066	0.097	1.416
2249440	0.144	0.206	1.426
4631200	0.328	0.418	1.255

Table 5.18: *Execution time on the Tesla K40 GPU using the preconditioned FGMRES of Paralution: the Jacobi setup and a single Krylov iteration for different mesh sizes of the case T106C.*

System Matrix N_{row}	1 Krylov iteration T_{itr} [s]	preconditioner setup T_{Jacobi}[s]	Ratio $T_{\text{Jacobi}}/T_{\text{itr}}$
264640	0.002	0.001	0.278
1058560	0.008	0.001	0.113
2249440	0.016	0.002	0.103
4631200	0.029	0.003	0.086

Table 5.19: *Speedup of the Jacobi setup and a single Krylov iteration for different mesh sizes of the case T106C on the K40 GPU compared to the Xeon E5 CPU.*

System Matrix N_{row}	speedup of 1 Krylov iteration $T_{\text{ITR}}^{\text{CPU}}/T_{\text{ITR}}^{\text{GPU}}$	speedup of preconditioner setup $T_{\text{Jacobi}}^{\text{CPU}}/T_{\text{Jacobi}}^{\text{GPU}}$
264640	6.52	34.60
1058560	8.63	107.15
2249440	8.84	118.19
4631200	11.34	156.16

for small meshes as reported in Table 5.18 and 5.19. In order to assess the Jacobi preconditioned FGMRES per flow iteration, the same case used for previous benchmarking will be used with different but low CFL numbers $\in (5, 10, 15, 20, 25, 50)$ both on the CPU and the GPU.

Figure 5.9 shows the simulation speedup for different CFL numbers along with the local assembly and linear solver speedup for different mesh sizes. With the increase in the CFL number, the number of linear iterations per flow iteration increases as the system matrix is more and more ill-conditioned. The highest speedup is performed with the lowest CFL as the portion of the assembly is the highest. The lower the CFL the closer is the global speedup to the assembly speedup. The global speedup is then bounded by the assembly speedup as the highest value and the linear solver speedup as the lowest value. The global speedup for all CFL numbers grows with the increase in the number of cells influenced especially by the increase in the assembly speedup.

While this preconditioner shows a cheaper setup time (see Table 5.17 and 5.18), it can not handle very ill-conditioned matrices. Therefore, when small time steps are used (see equation (5.2)) a Krylov solver converges with fast Jacobi preconditioner without the need for a factorization-based preconditioner. However, large time steps decrease the diagonal dominance and with it the condition number making thus the incomplete LU factorization essential to the convergence of the linear solver. With a lower limit on the CFL number than an ILU preconditioned flow solver, the flow solver with Jacobi preconditioner performs more flow iterations while performing also much more linear iterations per flow iteration compared to the ILU preconditioned linear solver. Finally, per linear iteration the Jacobi version is very attractive on the GPU (see Table 5.19) but for the whole simulation the Jacobi alternative is more time-consuming than the ILU preconditioned one.

To further accelerate the preconditioned linear solver while preserving the accuracy of the solution, the LU matrix should be provided at a lower cost. As reported by many authors [Chow and Patel, 2015; Saad, 2003], a low accuracy for the Lower Upper matrices affects the conditioning of the system leading to a larger number of linear system iterations. Since the GPU runs the iterations of the linear solver faster than it performs the incomplete LU factorization, the cost of the additional inner iterations can be traded against the ILU setup time. One possibility to get a faster and lower quality ILU is to apply multicoloring in order to identify independent rows that can be processed in parallel during the factorization. The multicoloring ILU (MC-ILU) shows more parallelism than the standard ILU, which is reflected on the faster setup (see Table 5.20 and Table 5.21). At the same time, the number of Krylov iterations increases for the same CFL number compared to the ILU preconditioned linear solver. The number of flow iterations, on the other hand, is the same as the CFL number has not been changed, which makes the MC-ILU more interesting than the Jacobi preconditioner.

The setup of the MC-ILU preconditioner is faster than the setup of the standard ILU for a small mesh (see Table 5.22). This advantage becomes smaller when the size of the matrix increases. Probably the coloring cost increases more than the advantage of the coloring for the ILU setup for large matrices. Also for the Krylov iterations, the multi-coloring acceleration fades with the increase in the matrix size.

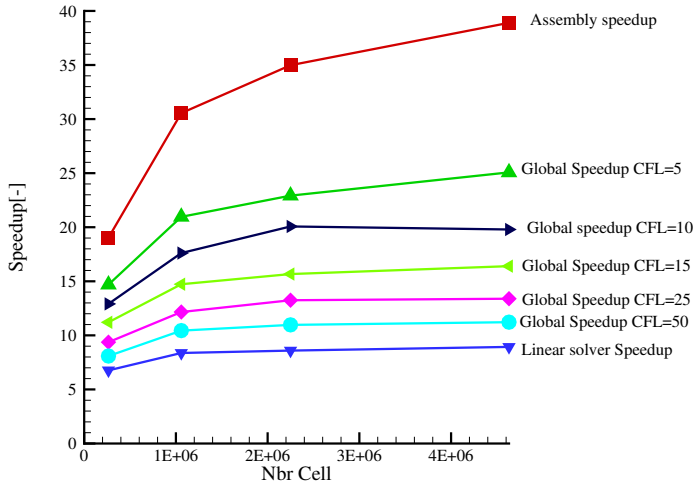


Figure 5.9: *Effect of the CFL increase on the global speedup for a fixed number of flow iterations of the RANS simulation on the case T106C with different mesh resolutions using the Jacobi preconditioned FGMRES linear solver.*

Table 5.20: *Execution time on the Tesla K40 GPU using preconditioned FGMRES of Paralution: the MC-ILU setup and a single Krylov iteration for different mesh sizes of the case T106C.*

System Matrix N_{row}	1 Krylov iteration T_{itr} [s]	preconditioner setup T_{MCILU}[s]	Ratio $T_{\text{MCILU}}/T_{\text{itr}}$
264640	0.004	0.279	65.137
1058560	0.014	1.030	71.990
2249440	0.030	2.114	71.645
4631200	0.059	4.292	72.280

Table 5.21: *Speedup of the MC-ILU setup and a single Krylov iteration for different mesh sizes of the case T106C on the K40 GPU compared to the Xeon E5 CPU.*

System Matrix N_{row}	speedup of 1 Krylov iteration $T_{\text{ITR}}^{\text{CPU}}/T_{\text{ITR}}^{\text{GPU}}$	speedup of preconditioner setup $T_{\text{ILU}}^{\text{CPU}}/T_{\text{MCILU}}^{\text{GPU}}$
264640	3.341	0.079
1058560	6.104	0.573
2249440	6.680	0.649
4631200	7.135	0.717

Table 5.22: *Speedup of the MC-ILU setup and a single Krylov iteration for different mesh sizes of the case T106C on the K40 GPU compared to the ILU on k40.*

System Matrix N_{row}	speedup of 1 Krylov iteration $T^{\text{GPU}}_{\text{ITR}}/T^{\text{GPU}}_{\text{MC-ITR}}$	speedup of preconditioner setup $T^{\text{GPU}}_{\text{ILU}}/T^{\text{GPU}}_{\text{MCILU}}$
264640	7.802	1.771
1058560	3.288	1.045
2249440	2.269	0.953
4631200	1.793	0.921

While the choice of the solver is not very relevant as most Krylov solvers have a comparable performance, the choice of the preconditioner is very decisive for the studied test case. The performance of some preconditioners from a prominent library namely Magma sparse have been benchmarked (see Table 5.23). Magma sparse relies on the cuSparse[†] implementation of the ILU factorization with level-scheduling. For the application of the preconditioner within the linear solver iteration, the library proposes also an Incomplete Sparse Approximate Inverse method (ISAI [Anzt et al., 2016a]). All methods use the flexible GMRES algorithm. For small meshes on the GPU, an ILU with ISAI triangular solver of Magma sparse could be competitive but for large meshes, the Paralution library provides the best GPU performance. Based on the results of the benchmark, the chosen linear solver is the preconditioned GMRES of the Paralution library [PARALUTION Labs, 2016] with a restart Krylov subspace of 10 to 30 vectors.

As a reaction to the high execution time of the ILU, an attempt is made to lower the required number of recomputations or updates of the preconditioner. It is not a new topic as many researchers tried to reduce the execution time spent for the preconditioning in a context of a set of consecutive linear systems such as a steady CFD simulation [Birken et al., 2008; Tebbens and Tuma, 2007]. It is possible to recalculate the ILU only periodically with a period that can be set from the beginning or dynamically changing based on some parameters such as the number of linear solver iterations. Some methods [Birken et al., 2008; Tebbens and Tuma, 2007] update the ILU preconditioner using former factorization in a way which is lighter than a recalculation and should result in fewer linear solver iterations than the periodic recalculation. Anzt et al. [Anzt et al., 2016b] use an iterative ILU on the GPU for which the preconditioner of one system is the first guess for the next system. They show that updating a the preconditioner of a previous time step could be fast and effective.

Since the preconditioner setup is very costly and the system matrix is only slightly changing from one flow iteration to the next, it is possible to recycle the preconditioner from previous iterations without altering the flow convergence. Table 5.24 and Table 5.25 show the effect of the preconditioner freezing both on the CPU and on the GPU for the LS89 test case. For both hardware, the cost of solving the linear system has decreased and with it the portion of the system solving from the total

[†]http://docs.nvidia.com/cuda/cusparse/

Table 5.23: *Execution time on the Tesla K40 GPU using preconditioned FGMRES of Paralution: the ILU setup and a single Krylov iteration for different mesh sizes of the case T106C.*

System Matrix N_{row}	Triangular Solver	1 Krylov iteration T_{itr} [s]	preconditioner setup T_{PC}[s]	Ratio $T_{\text{PC}}/T_{\text{itr}}$
264640	CUSPARSE	0.049	0.61	12.50
1058560	CUSPARSE	0.072	2.63	36.27
2249440	CUSPARSE	0.107	5.66	52.71
4631200	CUSPARSE	0.174	11.59	66.61
264640	ISAI	0.013	0.77	56.75
1058560	ISAI	0.056	3.64	64.76
2249440	ISAI	0.121	8.08	66.83
4631200	ISAI	0.250	16.89	67.56

execution time. The striking fact is that the portion of the linear solver is less than 15% for the CPU while more than 80% for the GPU. This is related mainly to the cost of the preconditioner setup on the GPU.

Simply skipping the factorization for a preset number of flow iterations improves, indeed, the performance as observed above. Having a very low update rate, however, can make the linear solver struggle to converge especially during the early flow iterations.

To decrease the time spent in the factorization while guaranteeing the convergence of the linear solver in a decent number of linear iterations, a mechanism is implemented to decide when an update of the ILU is necessary. The factorization is performed thus only *on-demand* when the LU quality is so decreased that the linear solver needs more iterations to converge than a user defined threshold:

Pseudo-code of the on-demand LU factorization

```
if (itr> MAX_ITR ) M <-LU_Factorization (A)
(x, itr) <- FGMRES (A,M,b)
```

Table 5.24: *Different preconditioner update rates for LS89 simulation on the Xeon E5 CPU.*

preconditioner update rate	Mean flow integration [s]	Total time [s]	Time integration portion
100.0%	91.15	618.59	14.74%
50.0%	69.75	592.23	11.78%
25.0%	62.81	560.22	11.21%
12.5%	56.89	527.95	10.78%

Table 5.25: *Different preconditioner update rates for LS89 simulation on the Tesla K40 GPU.*

preconditioner update rate	Mean flow integration [s]	Total time [s]	Time integration portion
100.0%	294.87	352.76	90.65%
50.0%	200.91	258.65	87.30%
25.0%	170.74	228.75	85.52%
12.5%	155.13	213.14	84.46%

where A, M and b are defined in equation (5.9). The maximum number of iterations `MAX_ITR` depends on the condition number and thus on the time step. As the time step depends on the CFL and the mesh cell size a relation between `MAX_ITR` and CFL number can be found for a given mesh:

$$\text{MAX\_ITR} = \text{a} + \text{b} * \text{CFL} \,, \tag{5.11}$$

with a and b two tuning parameters. Parameter a plays an important role for applications with a low CFL number and b increases with the mesh refinement. The *on-demand* factorization changes only the entries of L and U matrices, and not the ordering of the non-zero elements.

For the 2-stage Runge-Kutta solver, the flow required 637 flow iterations to converge with a 6 orders of magnitude decrease in the relative flow residual. The standard ILU performs one factorization per flow iteration, while the *on-demand* ILU (ILU-OD) performs a factorization only every 20 flow iterations and if the number of linear iterations required for the last linear solver call exceeds 8 [†]. A global speedup of 2.26x is reached between the OD-ILU and the ILU both on the GPU (see Table 5.26). Figure 5.10a shows the growing difference in the execution time for the mean flow integration responsible for the performance improvement of the OD-ILU. Figure 5.10b shows the number of linear solver iterations per call for both solvers (with ILU and with OD-ILU as preconditioner). At the beginning of the simulation, the OD-ILU performs more linear iterations because of the outdated ILU within a fast-changing flow and related system matrix. But after 200 flow iterations, the simulation is close to the stationary state and both preconditioners almost do the same number of linear iterations per call to the linear solver, which reflects the small difference between the up-to-date and the outdated ILU preconditioner. Indeed, after the initial phase the ILU is updated only every 20 flow iterations as the number of GMRES linear iterations never exceeds 8, which is the threshold value set for a preconditioner update. In total, the OD-ILU performs 50 calls to the ILU factorization, which corresponds to a decrease of 92%. The effect of the OD-ILU is expected to be more pronounced the higher is the required decrease in the flow residual as for low residuals the solution is only slightly changing and the preconditioner can be reused for a large number of flow iterations.

[†]Based on Equation (5.11) with $CFL = 50$, $b = 0.15$ and $a = 0.5$.

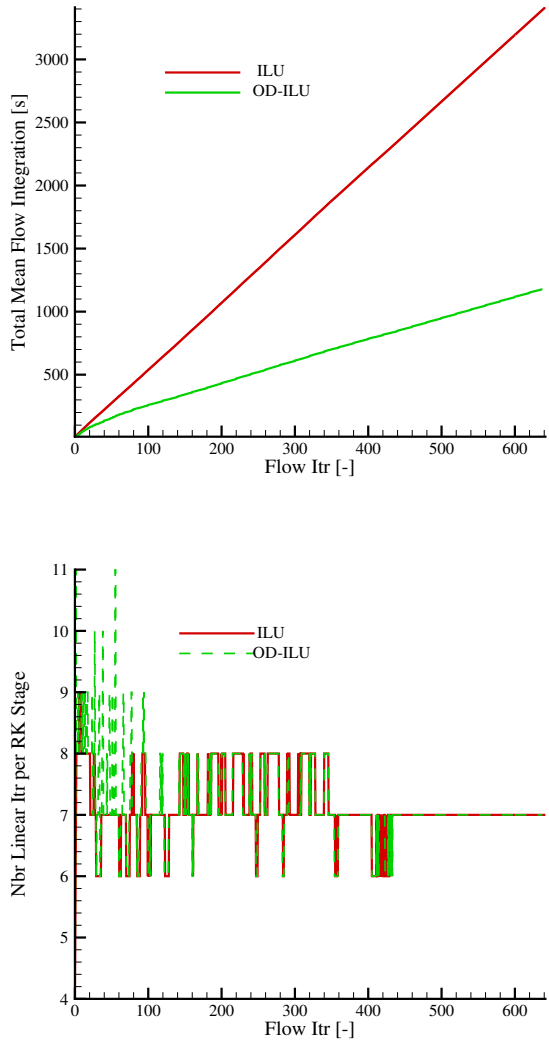


Figure 5.10: *Comparison of the standard ILU and the on-demand ILU for (top) the total simulation time and (down) the number of linear iterations per Runge-Kutta stage.*

Table 5.26: *Execution time of a converged simulation on the K40 using the ILU and the OD-ILU preconditioned FGMRES of Paralution for the case T106C with* $CFL = 100$*, no-sorting assembly and mesh 4.*

PC name	periodic update	Threshold value	Time Integration[s]	Total [s]
OD-ILU	20	8	1177.72	1768.66
ILU	1	-	3412.43	3991.39

In the next step, the same OD-ILU mechanism is implemented in the CPU version with an ILU update every 20 flow iterations and an update based on the number of linear iterations not exceeding 8 per call to the linear solver (see Table 5.27). The GPU OD-ILU simulation reaches a speedup of 11.47x over the OD-ILU on the CPU mainly boosted by the increase in the linear solver speedup.

5.5 Results

After analyzing the cost of the preconditioner setup and a single Krylov iteration for a set of preconditioners, it has been observed that the MC-ILU is interesting for small meshes while ILU is the best choice for large meshes. In order to generate the ultimate speedup, an implicit RANS simulation on the turbine stator T106C is run on different meshes with increasing size. The stopping criterion for the linear solver is a 10^{-1} reduction of the relative residual and the flow solver stops when the minimization of the L_2 norm of the residual reaches 10^{-6}. The 2-stage Runge-Kutta (RK) time-stepping method is chosen for the benchmark. Tables 5.28, 5.29, 5.30 and 5.31 report on the execution time of the CPU ILU, the GPU MC-ILU and the GPU ILU. The reached speedup of the GPU ILU solver against the CPU ILU is reported in Table 5.32 for different mesh sizes.

Figure 5.11 compares the three GPU alternatives for the total execution time, the execution time of the system solving and the assembly. For small meshes, the MC-ILU preconditioned FGMRES is the fastest solver preceding the Jacobi and the ILU preconditioned FGMRES (see Figure 5.11a). The Jacobi preconditioned

Table 5.27: *Execution time of a converged simulation on the Xeon E5 CPU OD-ILU preconditioner with FGMRES of PETSc for the case T106C with* $CFL = 100$ *and mesh 4.*

PC name	periodic update	Threshold value	Time Integration[s]	Total [s]
CPU OD-ILU	20	8	3586.7	20276.9 (5.6h)
CPU ILU	1	-	5602.14	22169.84 (6.48h)

Table 5.28: *Execution time on the K40 using MC-ILU preconditioned FGMRES of Paralution for different mesh sizes of the case T106C with* $CFL = 100$ *and no sorting assembly.*

Mesh Size	# flow iterations	Assembly [s]	Time Integration[s]	Total [s]
52928	770	57.16	412.57	469.73
211712	777	150.39	1575.11	1725.51
449888	755	270.51	3190.74	3461.26
926240	756	509.33	6346.10	6855.44

Table 5.29: *Execution time on the K40 using the ILU preconditioned FGMRES of Paralution for different mesh sizes of the case T106C with* $CFL = 100$ *and no sorting assembly.*

Mesh Size	# flow iterations	Assembly [s]	Time Integration[s]	Total [s]
52928	771	56.36	724.2075	780.566
211712	770	142.12	1345.5491	1487.67
449888	746	257.76	2235.921	2493.68
926240	746	497.95	4099.955	4597.91

Table 5.30: *Execution time on the K40 using Jacobi preconditioned FGMRES of Paralution for different mesh sizes on the case T106C with* $CFL = 100$ *and no sorting assembly.*

Mesh Size	# flow iterations	Assembly [s]	Time Integration[s]	Total [s]
52928	3611	266.10	495.09	761.19
211712	4111	773.07	1971.67	2744.74
449888	4031	1423.55	4127.36	5550.91
926240	4095	2810.00	8441.50	11251.50

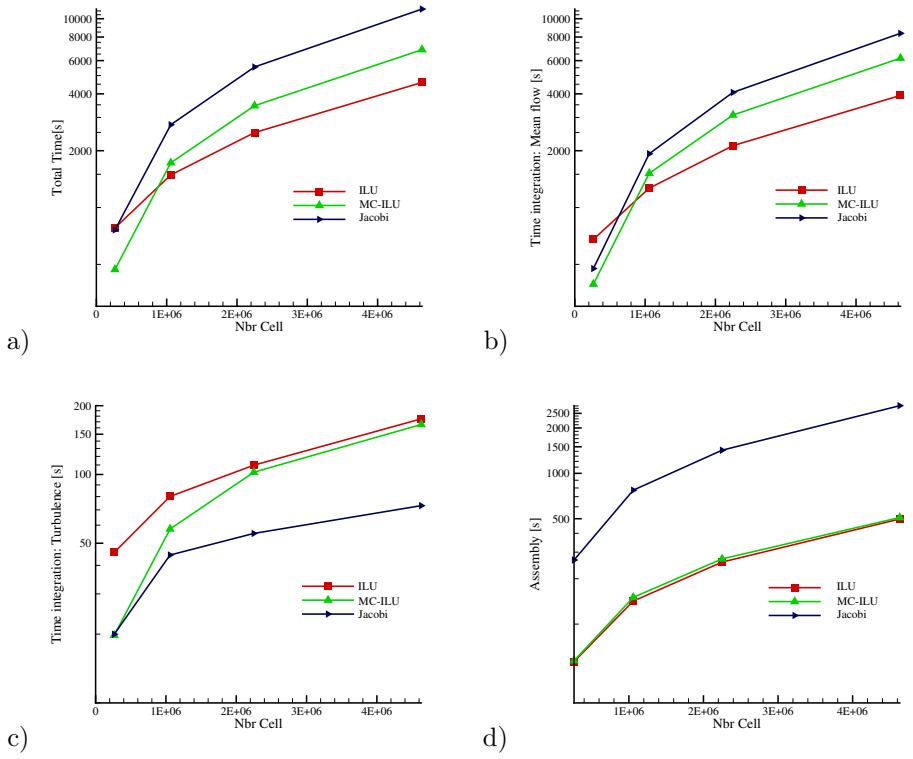


Figure 5.11: *Execution time of the T106C case for different mesh sizes: (a) total execution time, (b) execution time of the time-stepping of mean flow variables, (c)execution time of the time-stepping of the turbulence and (d) execution time of the assembly.*

Table 5.31: *Execution time on the Xeon E5 CPU using the ILU preconditioned FGMRES of PETSc for different mesh sizes of the case T106C with* $CFL = 100$.

Mesh Size	# flow iterations	Assembly [s]	Time Integration[s]	Total [s]
52928	764	1037.05	235.35	1276.79
211712	770	4247.95	1211.75	5469.51
449888	746	9093.07	2969.08	12085.20
926240	746	19870.87	6883.08	26827.65

Table 5.32: *Speedup of the ILU preconditioned FGMRES of Paralution for different mesh sizes of the case T106C with* $CFL = 100$ *over PETSc on the Xeon E5 CPU.*

Mesh Size	Assembly Speedup[-]	Time integration speedup[-]	Total speedup[-]
52928	18.40	0.32	1.64
211712	29.89	0.90	3.68
449888	35.28	1.33	4.85
926240	39.91	1.68	5.83

version performs more linear iterations per flow iteration, nevertheless, it is faster on the time integration (see Figure 5.11(b-c)) since it has a very fast setup and linear iteration on the GPU. It has, however, a poor performance on the assembly as it is the same procedure run by all solvers and the Jacobi version requires more flow iterations because of the lower CFL number (see Figure 5.11d).

For large meshes, the ILU preconditioned solver is the fastest in the assembly and the mean flow time integration. For the turbulence time-stepping, it performs as good as the MC-ILU alternative.

5.6 Discussion

The global speedup reached in this work is a combination of the high assembly speedup and the low linear solver speedup. Applications spending more time on the assembly phase rather than the linear solver benefit more from the GPU acceleration since looping over faces or cells in a large mesh could be easily adapted to the GPU hardware.

To sum up, three types of operations on the GPU are here present: (1) slow preconditioner setup, (2) fast linear solver iterations and (3) very fast system assembly. The final speedup is a linear combination of the speedup of these three operations as follows:

$$S = \alpha_{\text{PC}} S_{\text{PC}} + \alpha_{\text{itr}} S_{\text{itr}} + \alpha_{\text{assembly}} S_{\text{assembly}}, \tag{5.12}$$

with S the global speedup, S_{PC} the preconditioner speedup (actually slowdown), S_{itr} the linear iteration speedup, S_{assemly} the assembly speedup. α_{PC}, α_{itr} and α_{assembly} are the portions of the execution time of the preconditioner, the linear iteration and the assembly, respectively. The best speedup is reached when the portion of the assembly is at its maximum while the portion of the preconditioner setup is at its minimum. In light of these facts, a discussion is presented, for which the Paralution library is used, and the effect of the CFL number, the number of RK stages and the linear stop condition are analyzed.

5.6.1 Effect of the CFL number on the GPU speedup

A higher CFL number is generally related to a faster convergence and a shorter execution time. The general recommendation is then to chose the largest possible CFL that still guarantees a stable simulation. This subsection treats solely the effect of increasing the CFL number on the GPU speedup. The CFL number reduces the number of flow iterations required to reach a stationary solution at the cost of an increase in the number of linear solver iterations per flow iteration. In total, the steady simulation requires fewer factorization while the cost of the ILU preconditioning is independent of the CFL number. On the other hand, the portion of the linear solver from the execution time of the flow iteration increases at the expense of the system assembly which has a better GPU acceleration. Table 5.33 shows a small decrease of the global speedup for higher CFL numbers. For very high CFL numbers the simulation can be largely dominated by the linear solver resulting in a small GPU speedup or even a slowdown.

5.6.2 Effect of the RK stages number on the GPU speedup

In general the higher the number of Runge-Kutta stages is, the higher the CFL number could be. The ideal number of RK stages for a faster simulation on the GPU is case-dependent. In this subsection, the effect of the number of stages is analyzed for a fixed number of flow iterations not taking into account the expected

Table 5.33: *Speedup for different CFL numbers of the ILU preconditioned FGMRES of Paralution for the case T106C on the K40 GPU compared to the Xeon E5 for mesh 1.*

CFL Number	linear solver speedup	Assembly speedup	Global speedup
100	0.34	18.43	1.61
200	0.36	18.73	1.58
300	0.37	18.91	1.56
400	0.37	18.93	1.53
500	0.38	19.51	1.55
1000	0.37	18.58	1.42

higher stable CFL number with more RK stages.

As indicated in the solver algorithm, the system matrix is computed only on the first stage and for all further stages, only the RHS is changing. When the system matrix does not change so does the preconditioning matrix. Consequently, having more stages adds more residual computation and linear system iterations both advantaging the GPU. Table 5.34 confirms that the linear solver speedup increased because of the reuse of the same preconditioner for more stages. Moreover, the speedup of the assembly improved since the second stage and higher need only the well-accelerated space integration and no Jacobian calculation. Both improvements are combined to increase the global speedup the more RK stages a simulation has.

5.6.3 Effect of the linear solver stop condition on the GPU speedup

The linear stop condition is the condition that has to be fulfilled by the linear solver, here the FGMRES, in order to consider the linear system as solved (see Algorithm 4). Higher levels of convergence of the linear solver lead therefore to a large number of linear iterations without substantially altering the number of flow iterations.

Equation (5.12) breaks the global speedup into three types of operations: the preconditioner, the linear iterations and the assembly. The increase of the level of convergence for the linear solver increases the portion of execution time devoted to the linear iteration at the cost of the assembly and the preconditioning. As a results the global speedup decreased as shown in Table 5.35. In general the mildest possible stop condition should be chosen, which however should not deteriorate the convergence of the flow solver.

5.7 Validation

The test case is a transonic flow over the LS89 inlet guide vane cascade [Arts et al., 1990], which experiences a turning of 74 degree through the NGV geometry and a passage shock with a peak Mach number of 1.15. Figure 5.12 shows the validation of the method on the LS89 case.

Table 5.34: *Speedup for different numbers of Runge-Kutta stages of the ILU preconditioned FGMRES of Paralution for the case T106C on the K40 GPU compared to the Xeon E5 for a mesh size of almost 1M cells.*

Nbr RK Stages	linear solver speedup	Assembly speedup	Global speedup
3	1.70	37.41	5.50
4	1.92	36.96	5.66
5	3.18	41.43	7.24
6	3.45	41.28	7.47

Table 5.35: *Speedup for different linear solver stopping conditions of the ILU preconditioned FGMRES of Paralution for the case T106C on the K40 GPU compared to the Xeon E5 for a mesh size of almost 1M cells.*

relative Stop Condition $\|\|r_i\|\|/\|\|r_1\|\|$	linear solver speedup	Assembly speedup	Global speedup
1	1.51	39.22	5.50
2	2.04	39.19	5.11
3	2.41	38.70	4.81

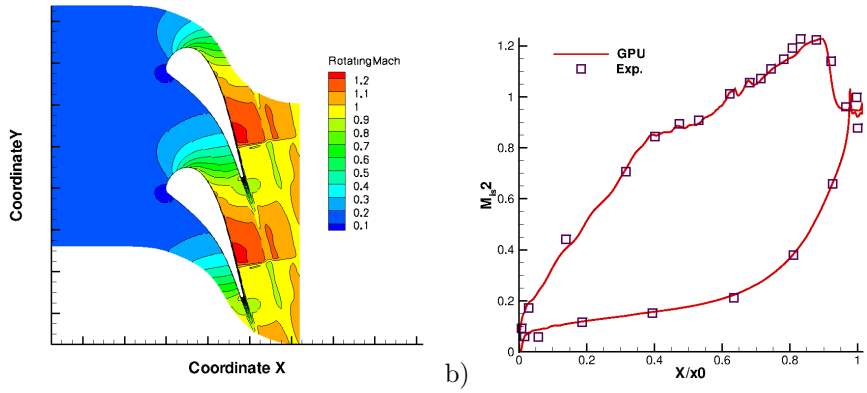


Figure 5.12: *(a) Mach contours of the transonic LS89 [Arts et al., 1990] turbine guide vane test case and (b) Computed and experimental distributions of isentropic Mach number on LS89 inlet guidance vane surface ($M_{2_{is}} = 1.02$ $P_{01} = 1.605bar$).*

5.8 Summary

The operations required by an implicit solver belong to three groups based on the reached speedup on the GPU: very fast (assembly), fast (linear iteration) and slow operations (factorization). The assembly reached a speedup of almost 40x which is lower than the two orders of magnitude reached for the explicit solver, even though both are embarrassingly parallel. Indeed, the assembly includes the same space integration as for the explicit solver but at the same time computes the Jacobian and form the system matrix. The Jacobian are slower to be computed than the fluxes as the Jacobian kernel is more resource consuming and thus fewer cells are computed concurrently. Moreover, the direct insertion or the sorting are also time-consuming which results in the speedup gap between the space integration and the assembly. The effect of the Jacobian decreases when more Runge-Kutta stages are used since they are computed only for the first stage.

The slow factorization has the major influence on the global implicit solver speedup, which hardly exceeds one order of magnitudes. At the time of writing the promising iterative factorization has not been stable in this work and the only optimization is the reduction of the number of performed standard factorizations for flow convergence. A lot of potential is in these techniques as sometimes 90% of the execution time of a RANS simulation is spent solving the linear system with most of this time spent in the incomplete factorization of the system matrix.

5.9 Conclusion

This chapter presented an implicit RANS CFD solver fully running on the GPU with the benefit of avoiding any CPU-GPU penalizing communication. The flexible GMRES implementation of the Paralution package is used to solve the linear system of equations along with the incomplete LU factorization for the preconditioning. A proposed control mechanism is capable of accelerating the linear solver without altering the flow convergence by reducing the number of times an incomplete LU factorization is performed. Speedups of 11.47x compared to a single-core CPU is measured for 3-D flow predictions in turbine applications.

In addition to the reached final acceleration results, the chapter analyzed thoroughly the performance of multiple implementations for the assembly and the system solving. The benchmark demonstrated that implicit time-stepping in CFD applications can profit from the GPU computational power, provided an appropriate GPU occupancy is reached and a good mesh in terms of surface-area-to-volume ratio is used. As the bottleneck of the GPU flow solver is the incomplete LU factorization used as preconditioner, the *on-demand* ILU factorization presented in this chapter improved the overall speedup by 60% to 80%.

6

Explicit versus Implicit CFD Simulations, the GPU dimension

A Computational Fluid Dynamics (CFD) code for steady simulations solves a set of non-linear partial differential equations using an iterative time-stepping process, which could follow an explicit or an implicit scheme. On the CPU, the difference between both time-stepping methods with respect to stability and performance has been well covered in the literature. However, it has not been extended to consider modern high-performance computing systems such as Graphics Processing Units (GPU). In this chapter, the GPU implementations of the two time-stepping methods, presented in chapter 4 and 5, are used to study the difference between the two methods on the GPU. A classification of basic CFD operations is introduced, which is based on the degree of parallelism they expose and is used to study the potential of GPU acceleration for every class. The classification provides local speedups of the basic operations, which are finally used to compare the performance of both methods on the GPU. The target is to enable an informed-decision on the most efficient combination of hardware and method when facing a new application.

6.1 Introduction

CFD is commonplace in engineering activities, as many products are nowadays designed by relying heavily on numerical preconditions with reduced wind tunnel testing in order to cut down the product development cost. Computations are, neverthe-

This chapter is based on the article:

M.H. Aissa, T. Verstraete, and C. Vuik. Toward a GPU-aware comparison of explicit and implicit CFD simulations on structured meshes. *Computers & Mathematics with Applications*, 74(1):201–2017, 2017b .

less, still expensive and a trade-off is continuously sought between fast turnaround time and high accuracy. New hardware with a high computational power, such as the GPU, can be effectively employed to target fast computations without compromising the accuracy. The GPU, originally developed from graphics pipelines, excels on processing a large amount of independent data with a very regular and simple memory access pattern. Not all CFD solvers offer the required workflow to profit from the high GPU performance.

The basic task of CFD solvers is indeed to advance iteratively an initial solution by performing a space and a time integrations of the governing equations. In order to update the solution, the solver requires a memory access to the neighboring cells. For the explicit time integration, the update depends only on few neighbor cells, while for the implicit time integration, the update depends on all cells and can be formulated as a solution of a linear system of equations.

The locality of explicit solvers reflects on the memory usage, which is very low compared to the implicit solver memory footprint (apart from special cases of matrix-free implicit solvers [Johan and Hughes, 1991]). The computations performed within this scheme have a stencil-based character, for which neighbor cells are used to update a central cell following a regular pattern. As the operations are repeated for all mesh cells, it generates a large number of independent operations. Due to this simple regular workflow, explicit solvers are suited to the GPU massively parallel architecture and can benefit largely from its computational power. Interesting speedups of one to two orders of magnitudes have been reported in the literature [Brandvik and Pullan, 2011; Brock et al., 2015; Karantasis et al., 2014; Elsen et al., 2008; Lefebvre et al., 2012]. In general, explicit solvers are stable only with a small time step, as the latter is controlled by relatively low Courant-Friedrichs-Lewy (CFL) conditions (e.g. CFL=0.92 [Van Leer et al., 1992]). This method requires, thus, a large number of iterations to converge. When combining the GPU acceleration with some flow convergence acceleration techniques, such as residual smoothing, multigrid and local time-stepping, the explicit method can be very efficient.

Implicit solvers have less stringent stability limits, allowing to speed up the transient process with an increased CFL number. The acceleration reaches, however, asymptotically a limit, due to the inherent nonlinearity of the equations, as depicted in Figure 6.1. Implicit solvers benefit less from the GPU acceleration especially when factorization based preconditioners are used that rely on the Gaussian elimination such as the incomplete LU factorization (ILU). New algorithms are, nevertheless, trying to expose more parallelism and improve the performance of linear solvers on the GPU by accelerating the incomplete factorization [Chow et al., 2015] and its use in the linear solver [Anzt et al., 2015, 2016a]. Reported speedups for implicit solvers are of one order of magnitude [Luo et al., 2015; Fu et al., 2014; Aissa et al., 2017].

The studied combinations cover the explicit and the implicit time-stepping both on the CPU and on the GPU, which results in four different approaches. Few authors compared the explicit to the implicit performance. Niemeyer and Sung [2014a] implemented a Finite Volume flow solver for chemical reactions on a GPU with explicit time-stepping. He compared the performance on a CPU and a GPU of the explicit time integration to a commercial implicit solver on a CPU. He showed, however, no results on the GPU version of the implicit solver. Brock et al. [2015] developed

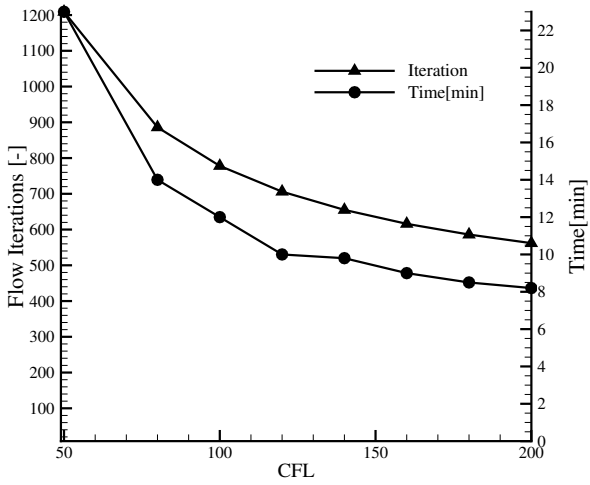


Figure 6.1: *Saturation of the iterations number with the increase of CFL number (case: turbine T106C [Michálek et al., 2012]).*

an explicit solver for an astrophysics application and compared the execution time of one iteration with estimations of the implicit performance on the CPU as the memory requirement for the solved case were prohibitive for the implicit integration. He showed a peak speedup of 2.5x for the GPU explicit over the CPU explicit and expected a speedup of 15x for the GPU explicit over the CPU implicit. He has not considered the possible faster convergence of the implicit method.

To bring the comparison forward a classification is introduced, which considers basic CFD operations following their suitability to the GPU architecture and studies their speedups. The classification of different CFD operations is inspired by the major differences of both hardware (see Table 6.1). The CPU is a general purpose processor able to handle different type of tasks. The large cache for a reduced number of CPU cores (e.g. Xeon E5-2640 has 15 MB of cache memory for 8 cores) is an efficient cure to the non regular data flow of some algorithms. In fact, it minimizes the cache misses defined as calls to variables relocated from the cache memory due to overuse [Hartstein et al., 2008]. The processor high clock rate of the CPU is an indicator for the speed, at which computations are executed and the memory is accessed. The GPU, on the other hand, is very specialized with a large computational power coming from the high number of GPU cores. The clock rate and cache capacities are in general lower than what CPUs offer and time-consuming cache misses can not be avoided for dispersed memory accesses. Under these circumstances, a regular data flow pattern is essential in order to achieve a good performance. As the computation power exceeds by far the GPU memory bandwidth, algorithms need to balance the slow memory access with a large number of computations. The large number of cores with a reduced power for each core are

efficiently used when a large number of independent data is available with a relatively simple and regular calculation for every piece of data. These specificities of the GPU motivated a classification of the CFD operations regarding three criteria: (1) the amount of data to process, (2) the data dependency level and (3) the regularity of the access pattern. The amount of data is used to maximize the possibilities for the GPU to hide the slow memory access with computations. A low data dependency provides a large number of independent instructions ideal for the large number of GPU cores. For a regular access pattern, the GPU can combine multiple expensive memory loads into one single load. The only condition is that consecutive threads of half a warp access consecutive memory positions (Cf. Section 2.5).

The entire explicit/implicit comparison is based on the proportion in execution time of each class in every scheme, which is used to estimate the GPU speedup. Similar classification has been introduced by Elsen et al. [2008] for vehicle aerodynamics. He classified kernels based only on memory access pattern and neglected the amount of data to be processed.

The performance estimation for the GPU is an active domain. Some authors analyze the GPU code ([Hong and Kim, 2009; Baghsorkhi et al., 2010]) to build an estimate of the GPU performance. Li et al. [2015] predicts the performance of Sparse Matrix-Vector operations (SpMV) using a trained probabilistic model. Baghsorkhi et al. [2010] interprets a GPU kernel as a workflow graph and estimates its execution time. The CPU and the GPU solvers -presented in the two last chapters- are used to accumulate enough benchmark data of basic operations to correlate it with the global performance of CFD simulations on the GPU. The covered simulations are steady CFD simulations on turbomachinery with structured meshes. The knowledge over the underlying CFD operations is used to match the performance bottlenecks on a GPU with their sources from the basic CFD operations. In that way, the use of automated methods to analyze applications code or executable is avoided.

The classification is used also to empower readers for a better appreciation of reported speedups in the literature. As discussed by Lee et al. [2010], reported GPU speedups are not to be taken as raw numbers. The reference CPU code optimization is important along with the used GPU. At the same time, some reported accelerations are only local and specific to a certain operation with sometimes little influence on the global speedup. The followed qualitative approach focuses on transmitting key know-how on CFD operations on the GPU. The aim is to help the developer to estimate the expected speedup of steady CFD simulations on GPUs using only the profiling results of the CFD application on a single CPU.

6.2 The classification

In computer science, problems which easily run in parallel, are called *embarrassingly parallel*. For those type of problems, it is clear how to divide the algorithm into small independent pieces of calculations, which are performed on different data. The algorithm shows a low level of data dependency peculiar to this type of operations. An *inherently sequential* problem, on the other hand, has highly interdependent operations, which have to be executed in a given order to get accurate results. A practical approach to classify CFD operations as *embarrassingly parallel* or *inher-*

ently sequential is to follow the GPU utilization of the operation given by the GPU profiler. The NVIDIA Visual Profiler (NVVP) can deliver information about the utilization of the memory and the computation units of the GPU for every kernel. A kernel with a high utilization of both units is suited to the GPU architecture and very efficiently written and executed. Figure 6.2 depicts this ideal case in addition to three realistic cases of performance limitations for GPU kernels: a latency-limited, a compute-limited and a memory-limited kernels. Kernels with a low memory and a low computation utilization are latency-limited. Such a low utilization can be caused by a non-efficient code (memory non-coalesced, important thread divergence) or by the use of a low number of threads. Indeed, when a kernel starts only a few threads, these are unable to hide the memory latency. When warps of threads are waiting for memory loads, not enough warps are available to fill the waiting time efficiently with computations. Kernels with a high compute utilization and a poor memory utilization are said to be compute-bound. For such a kernel, the performance optimization should first target the reduction of the number of computations per memory load. Kernels with a high memory utilization and a low compute utilization are said to be bandwidth-bound. For such a kernel, few computations are performed per memory load. In that case, improving the memory coalescence and using the shared-memory could improve the performance of the kernel.

In this section, elementary functions used by the explicit or the implicit CFD solver are first classified into three classes: compute-bound, memory-bound and latency-bound kernels. The classification reflects the differentiation between *embarrassingly parallel* and *inherently sequential* algorithms using the GPU utilization as a measure of the GPU suitability of studied CFD operations. Every class is analyzed regarding solely its performance on the GPU.

Compute-Bound Embarrassingly Parallel operations are suited to the GPU architecture and make an intensive use of arithmetic operations boosting the compute utilization of the GPU (see Figure 6.2). This class contains explicit time-stepping methods (e.g. Runge-Kutta scheme used in this work), convective and viscous flux calculation and turbulence calculation. These operations are stencil-based, they involve few neighbor cells for the computations related to a central cell. This class of functions is able to provide a large amount of independent data, which increases with the mesh size. The data dependency is relatively low as only a second-order scheme is used for the space integration (requiring access to 17 neighbors). Most of these functions involve an intensive use of arithmetic operations for a reduced stencil leading to a set of compute-bound functions. As the access pattern is regular on all data, it is very rare to have a thread divergence. Even though some algorithms can still impose divergence through a conditional statement. The Roe scheme, for instance, performs an entropy correction to better capture flow shocks as the original formulation does not recognize the sonic point [Harten et al., 1997]. This could lead to different execution paths within a warp. These kernels can have a high performance in terms of updated cells per second, which leads to a relatively high speedup of two orders of magnitude (See Figures 6.3 and 6.4).

The convective flux calculation is a compute-bound kernel, as the Roe scheme (see equation (4.9)) has to be evaluated at the cell face involving a lengthy computation with a large number of arithmetic operations. The flux calculation is based on a

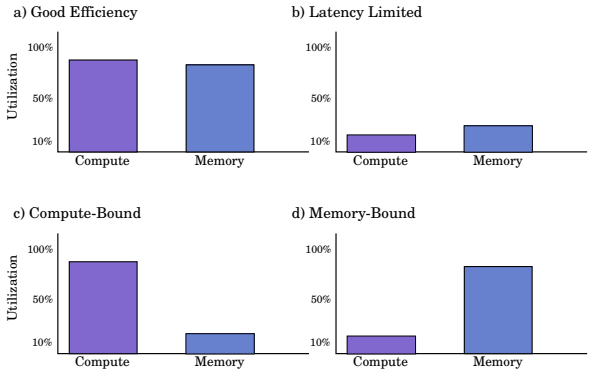


Figure 6.2: *Four cases of utilization of the memory and the compute unit of a GPU delivered by the NVida Visual Profilier.*

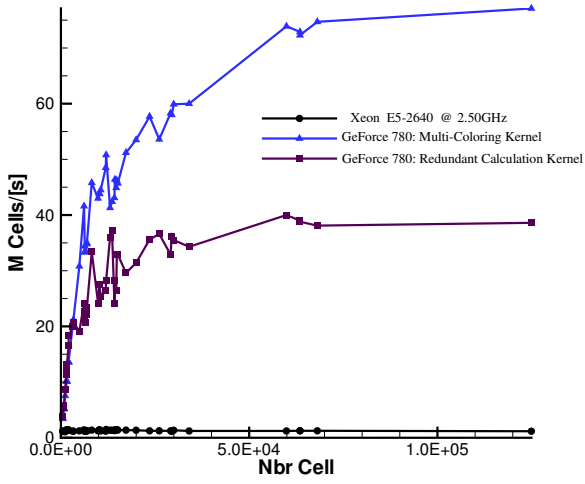


Figure 6.3: *Performance in terms of updated cells per second for one inviscid flux calculation on a single CPU compared to a multi-coloring based kernel and a redundant calculation on the GeForce 780 GPU for a RANS simulation of a 2D flow on a supersonic compressor cascade (see case 2 in the Appendix).*

summation of surface contributions and thus not thread-safe as two or more faces could add face contributions at the same time to the same cell. One possibility to avoid this race condition is to compute the flux cell-wise which is thread-safe but requires a redundant calculation of the flux. Recalling that the kernel is compute-bound, redundant computation has to be avoided. The second approach called the multi-coloring consists of creating groups (colors) of faces, which do not share cells in common. For every color, the computation is thread-safe at the expense of less coalesced access. The second approach is proven to be more efficient (see Figure 6.3 and the extended benchmarking of Section 4.2).

Memory-Bound Embarrassingly Parallel operations provide well defined independent work units for the GPU but make few computations per loaded byte of memory. This class includes the Jacobian calculation and Sparse Matrix-Vector operations (SpMV). Compared to the flux calculation which generates a vector of N_v values per cell [†], the Jacobian calculation updates a matrix of $N_v \times N_v$ values and performs thus many memory operations. Within the SpMV, a dot product is computed for every row of the sparse matrix. In CFD time integration, the rows correspond to mesh cells. Even though they can be reordered to favor a better performance, SpMV operations are known for being memory bound [Bell and Garland, 2009] and benefit averagely from the GPU. It is very important here to specify the data storage layout, as it can improve the data dependency and the memory access. Linear solvers, such as GMRES, are based on SpMV operations and belong thus to the same class.

Latency-Bound Inherently Sequential operations provide a very limited fine-grained parallelism, insufficient to run the GPU in a profitable regime. This third class deals with functions operating on a small amount of data with a high dependency and a non-regular memory access. Classical factorization algorithms, used for instance in implicit solvers as a preconditioner, belong to this class since they are based on the Gaussian elimination [Saad, 2003; Li and Saad, 2013; Chow et al., 2015] and handle sparsely populated matrices. The peculiar aspect of factorization is that the algorithm, in general, is recursive and the entries are computed serially.

Some linear solvers, such as the GPU version of PETSc [Minden et al., 2013], perform the ILU factorization on the CPU as a response to its difficult adaptation to massively parallel hardware. In this case, the entire system matrix needs to be transferred to the host and back. The communication through the PCI bus between host and GPU is not encouraged for optimized performance. For large systems, this alternative has no benefits as the fast CPU ILU factorization is not compensating the expensive communication costs. The other approach is to optimize the ILU factorization on the GPU. In order to expose more parallelism during the assembly of the ILU matrices, it is possible to identify independent rows that can be updated concurrently [Saad, 2003]. Few options are available to improve the performance of the ILU factorization on the GPU among them multi-coloring [Naumov et al., 2015; Lukarski, 2012; Li and Saad, 2013] and level-scheduling [Naumov, 2011].

While the methods cited above increase slightly the parallelism of the factorization, other methods such as the iterative ILU [Chow and Patel, 2015] propose a novel algorithm. The factorization is replaced by a minimization problem able to provide

[†] N_v is the number of flow variables

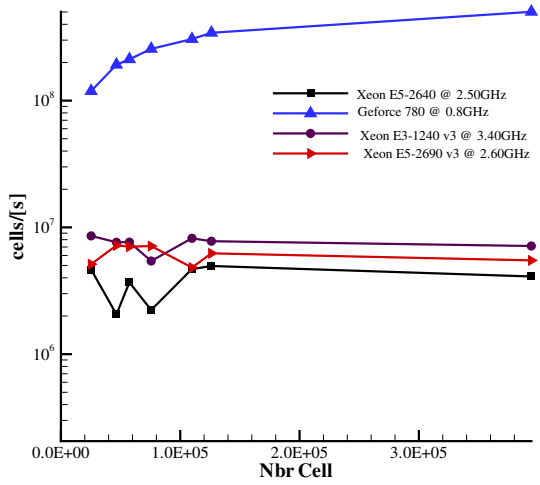


Figure 6.4: *Performance in terms of updated cells per second for one Runge-Kutta stage on a Geforce 780 compared to three other single CPUs for a RANS simulation of a 2D flow on a supersonic compressor cascade (see case 2 in Appendix).*

an approximate L and U with more fine-grained parallelism. The method relies on a fixed point iteration $x^{n+1} = G(x^n)$ which is guaranteed to converge [Chow and Patel, 2015; Frommer and Szyld, 2000] after a set of sweeps[†]. The GPU implementation of this method, presented in [Chow et al., 2015, p.5], evokes a trade-off between convergence and parallelism as the fixed point iteration tends to use less frequently updated values when more threads are used.

Within this work, the iterative ILU factorization was not stable and the multi-coloring version of ILU had no performance gain for large scale problems. The ILU factorization remains therefore in the latency-bound category of kernels.

Figure 6.5 summarizes the benchmark data collected on four different CFD operations belonging to the three categories mentioned above. The level of utilization of the memory and the compute units can be related to a degree of parallelism and consequently to a speedup. For 2D/3D RANS simulations on structured meshes, the sparse matrix factorization experiences a slowdown, while the GMRES iterations can be up to 10x times faster on the GPU [‡]. The Jacobian calculation and the flux calculation belong to different kernel types (the first is memory-bound while the second is compute-bound) but to the same speedup category [10x .. 100x]. Some of the operations, such as the flux calculation, are used in the space integration in the explicit and the implicit solvers. Other operations related to the linear system solving are proper to the implicit solver. The speedup categories identified for these different operations will be the start material for a performance comparison of the

[†]a sweep is one full update of the L and U matrices
[‡] K40 GPU compared to single core Xeon E5.

explicit solvers first qualitatively in the next subsection and then quantitatively in Section 6.4.

The trends in the classification can be generalized over the different hardware of similar computational power. The quantitative aspect of the classification is, however, hardware-dependent. A compute-bound kernel in a GPU card with a high memory bandwidth, for instance, can be memory-bound in another GPU card with a lower memory-bandwidth. Moreover, A well-balanced kernel in a weak GPU can be latency-bound in a powerful GPU card. The classification introduced in this work is based on benchmark data collected on Tesla K40 and Geforce GTX780 of NVIDIA. Both cards have a relatively high memory bandwidth of 280 Gb/s, while the K40 provides also 1.2 Tflops in double precision. In short, the classification holds for quite recent GPUs and can be probably challenged by Pascal generation of NVIDIA cards. The principles used to generate the classification remain, however, the same and it depends upon new benchmark data to update it.

6.3 Acceleration of the time integration method: a qualitative analysis

This section presents a qualitative model able to give some insights into the performance of the two integration methods on the GPU and on the CPU. It is based on the classification introduced in the previous subsection and makes use of the performance of all methods for one flow iteration which can be then extrapolated to cover realistic test cases. For that purpose, this section introduces a set of ratios, which will be used to link the performance of different methods.

For the explicit solver, the time integration and the space integration present similar stencil-based operations. Both are embarrassingly parallel and only few memory loads and few arithmetic operations are done per cell for the explicit time integration compared to the space integration. The execution time of this operation (t_T^{Exp}) can be thus neglected in the definition of the execution time ratio relating the implicit solver to the explicit solver duration for a single flow iteration on the CPU:

$$R_{CPU}^{ITR} \approx \frac{t_S + t_T^{Imp}}{t_S} = 1 + \frac{t_T^{Imp}}{t_S} \,, \tag{6.1}$$

with t_S the duration of the space integration for both solvers. The approximation (see Equation (6.1)) guarantees a faster explicit flow iteration on the CPU ($R_{CPU}^{ITR} > 1$). Theoretically, very high values of R_{CPU}^{ITR} are possible but in practice researchers [Cecka et al., 2011; Aissa et al., 2017] report a 30% to 80% of the global execution time spent on the time integration. Consequently, R_{CPU}^{ITR} could reach values of 5 and possibly one order of magnitude for a dominant implicit time integration of 90% of the total execution time. The analysis of the different execution times of both methods on the CPU revealed the well-known fact: Explicit solvers can not compete with implicit solvers unless the convergence rates are similar ($N_{itr}^{Exp} \approx N_{itr}^{Imp}$) which is rather unusual.

In the following, the execution times ratio relating the implicit to the explicit

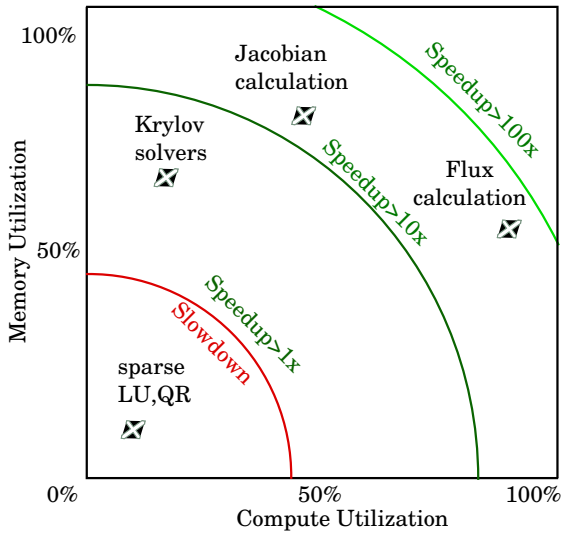


Figure 6.5: *Classification of four different CFD operations based on the compute and the memory utilizations leading to different speedup categories.*

solver for a flow iteration on a GPU is considered. Equation (6.2) relates R_{GPU}^{ITR} to the same ratio on the CPU (R_{CPU}^{ITR}) as follows:

$$R_{GPU}^{ITR} = R_{CPU}^{ITR} * \frac{a_{Exp}^{GPU}}{a_{Imp}^{GPU}} , \tag{6.2}$$

with a_{Exp}^{GPU} and a_{Imp}^{GPU} the speedups of the explicit and the implicit solvers on the GPU. Explicit solvers profit more from the GPU as they have only *compute-bound embarrassingly parallel* operations. Implicit solvers on the GPU profit averagely from the GPU as they contain *memory-bound embarrassingly parallel* operations for the linear solver and *latency-bound inherently sequential* operations when standard[†] factorization-based preconditioners are used. The more an implicit solver is dominated by the time integration the less it benefits from the GPU favoring the explicit solver on the GPU. A single explicit flow iteration ($R_C = 1$) is already faster than an implicit flow iteration on the CPU ($R_{CPU}^{ITR} > 1$) and the GPU increases the ratio by one to two orders of magnitude ($a_{Exp}^{GPU}/a_{Imp}^{GPU} >> 1$). The decisive aspect in the performance comparison is the portion of the linear solver from the total execution time of the implicit solver.

The speedup of one single flow iteration is, however, not determinant for the global performance as both methods have different convergence rates. The final comparison depends more on the ratio of convergence of both methods defined as :

$$R_C = \frac{N_{ITR}^{Exp}}{N_{ITR}^{Imp}}, \tag{6.3}$$

[†]Iterative incomplete factorization are not included

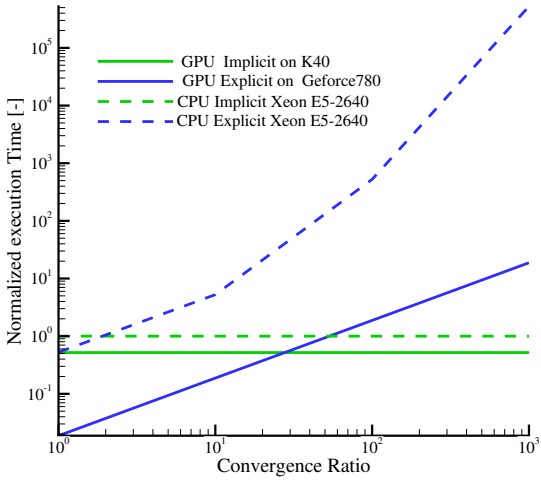


Figure 6.6: *Execution time until flow convergence of the GPU implicit solver and both explicit solvers normalized by the execution time of the CPU implicit solver for different convergence ratios ($N_{itr}^{Exp.}/N_{itr}^{Imp.}$) for a turbine nozzle guide Vane (Blade geometry from Arts et al. [1990]).*

with N_{ITR}^{Exp} and N_{ITR}^{Imp} the number of flow iterations required to reach a converged solution for an explicit and an implicit solver, respectively. This ratio is in practice large as explicit solvers are severely restricted in terms of CFL condition. If results are available for the two solvers both on the CPU and on the GPU for one flow iteration, different conclusions can be drawn depending on the ratio of convergence.

The application of the above-introduced model on a case of a turbine nozzle guide vane is depicted in Figure 6.6. It shows that one flow iteration of the GPU explicit solver is the fastest followed by the CPU explicit solver, the GPU implicit solver and finally the CPU implicit solver. When the convergence ratio R_C increases reflecting a faster converging implicit solver, the normalized execution time [†] of the implicit solver remains of course equal to one but the normalized execution time of the explicit solver is linearly increasing.

The GPU explicit solver is the fastest alternative within a certain range of values for R_C ($R_C < 20$). For $R_C >= 20$, the implicit GPU version is the fastest alternative. The maximum value of R_C, for which the GPU explicit solver is still the fastest choice is equal to its acceleration for one flow iteration compared to the direct competitor, here the GPU implicit solver.

Even though the current work does not provide results for the CPU parallelization nor for the effect of the convergence acceleration (e.g. residual smoothing and multigrid), the model shown in Figure 6.6 can handle these cases. The parallelization for the explicit and the implicit solvers on both the GPU and the many/multi-core

[†] normalized by the execution time of the CPU implicit solver.

CPU corresponds, indeed, to a translation of the performance curve† to regions of shorter execution times. Depending on the degree of acceleration the fastest combination of method and hardware can change. On the other hand, a convergence acceleration of the explicit or the implicit solver will change the convergence ratio R_C, defined in Equation (6.3). A faster converging explicit solver, for instance, will have a lower R_C and a different value is read from the same performance curve in Figure 6.6.

6.4 Numerical experiments: the subsonic turbine cascade T106C

For the benchmark, two Intel Xeon CPUs are used with different clock rates and two different GPUs are used with different local memory capacities (see Table 6.1). The Geforce GTX 780 with only 3GB of local memory is running the GPU explicit solver and the Kepler K40 card with 12 GB of local memory is running the GPU implicit solver.

T106C is a very high-lift, mid-loaded low-pressure turbine blade [Michálek et al., 2012]. The blade turns an incoming flow and reduces its pressure as depicted in Figure 6.7. Different mesh sizes are used for the benchmark. In a first step, an average execution time for one flow iteration has been averaged over 20 iterations. The explicit solver uses a 4-stage Runge-Kutta scheme with a CFL number of 2.5 and the implicit solver uses a 2-stage of Jacobian-Trained Krylov Implicit-Runge-Kutta scheme (JT-KIRK) with a CFL number of 50. The execution time of one flow iteration for different mesh sizes is depicted in Figure 6.8.

A difference of one to two orders of magnitudes is observed between the execution time of the explicit solver on the GPU and the implicit solver on the CPU. This is due to the combination of two facts: first, explicit solvers are inherently faster per iteration and second, they benefit largely from the GPU acceleration as they include mostly operation very suited to the GPU architecture. The performance of the implicit solver on the CPU can be improved with a higher clock frequency (e.g. Xeon E3 slightly outperforms the Xeon E5).

In order to compare all alternatives from different hardware, the slowest combi-

†Curve relating the execution time of a simulation to a its flow convergence

Table 6.1: *Hardware used in the benchmark.*

Reference	Clock rate [GHz]	Cache Size [MB]	Memory Bandwidth[GB/sec]	Global Memory [GB]
E3-1240	3.4	8	21	-
E5-2640	2.5	15	42.6	-
GTX 780	0.863	0.064	288.4	3
Tesla K40	0.745	0.064	288	12

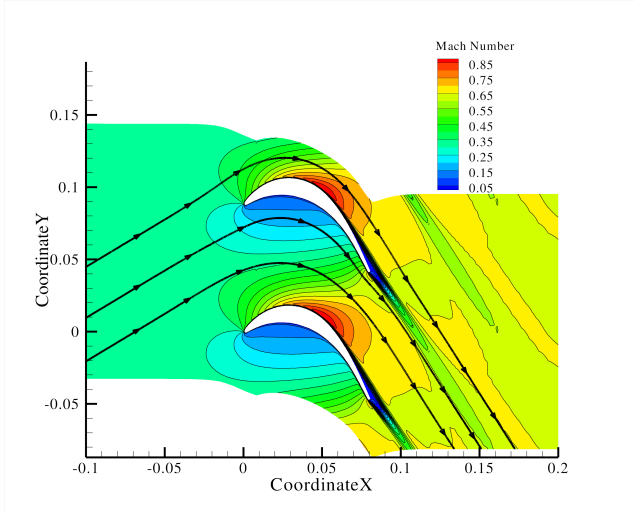


Figure 6.7: *Mach number contours around the T106C nozzle guide vane.*

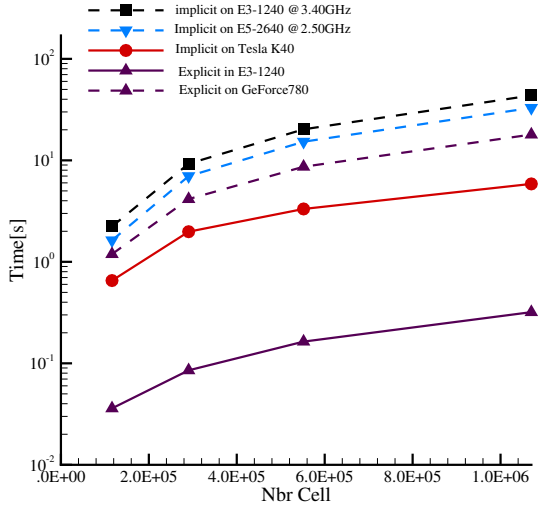


Figure 6.8: *Execution time for one flow iteration of the implicit solver and the explicit solver both on different CPUs (dashes line) and the GPU (full line).*

nation is chosen as a common reference, which is in this case the implicit solver on the CPU with the lower clock rate (Xeon E5). This approach leads to high speedups which do not reflect a GPU acceleration only, as usual speedups do. It reflects also the fact that an explicit flow iteration is lighter by nature while the implicit flow iteration solves a linear system of equations. Regardless of how efficiently the linear system is solved, an explicit Runge-Kutta stage is much lighter with its very few operations. Table 6.2 summarizes the performance of the combinations highlighting the growing gap between the explicit solver on the GPU and the other alternatives with the increase of the mesh size. The GPU, as a throughput-oriented device, is used more efficiently when the workload of embarrassingly parallel operations increases. The CPU, on the other hand, has rather a constant performance independent of the mesh size.

Explicit solvers are faster since they perform fewer operations per flow iteration than the implicit solvers and the GPU version of both methods is faster, even though the explicit solver takes more advantage from the GPU. The execution times of one flow iteration are used to extrapolate a performance comparison for different convergence ratios (see Figure 6.9). The convergence ratio is used to cover a wider range of applications as in some areas implicit solvers converge much faster than the explicit and in other areas the difference is not very pronounced. The initial ranking based on one flow iteration is valid for the unrealistic convergence ratios of one for which the explicit solver and the implicit solver converge after the same amount of flow iterations. Realistic ratios are in general between 20 to 100 for turbomachinery simulations. A common pattern is repeated for all mesh sizes predicting the explicit GPU solver to be the fastest alternative for low convergence ratios and the implicit GPU solver for large convergence ratios.

In the next step, the flow around T106C is solved for a relative residual drop of six orders of magnitude. Depending on the mesh size, the implicit solver required between 1328 and 1366 flow iterations for a CFL of 50. The explicit solver required between 22900 and 23400 flow iterations, which leads to a convergence ratio† $R_C = 17$. For this R_C value, the extrapolation based on the execution time of one flow iteration (see Figure 6.9) predicts the explicit solver on the GPU to be the fastest solver followed by the GPU implicit solver for all mesh sizes. This approxima-

†defined in Equation (6.3)

Table 6.2: *Speedup of the explicit and the implicit solvers on the GPU and on the CPU over the implicit solver on the Xeon E5-2640 CPU for one flow iteration.*

N_{Cells}[-]	Implicit E5-2640	Implicit E3-1240	Explicit E5-2640	Implicit Tesla K40	Explicit Geforce 780
116k	1	1.37	1.87	3.41	62.0
290k	1	1.32	2.23	4.67	108.5
552k	1	1.32	2.33	6.09	123.6
1070k	1	1.32	2.42	7.44	136.4

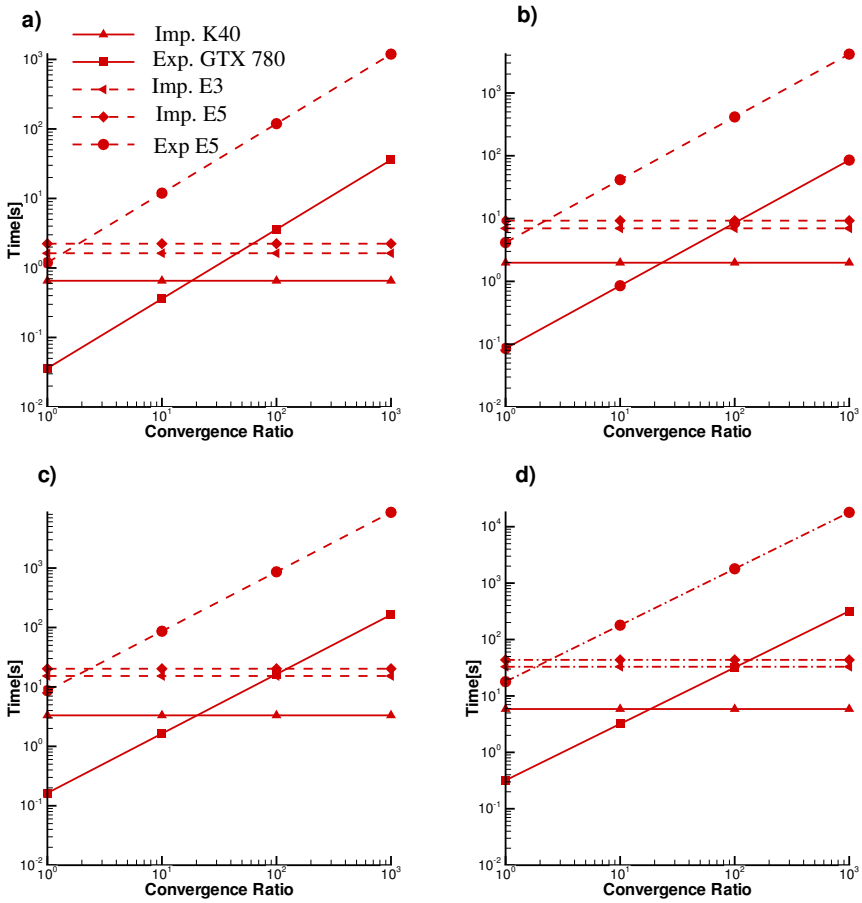
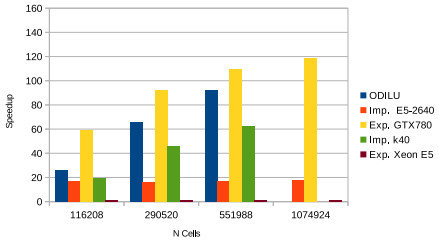


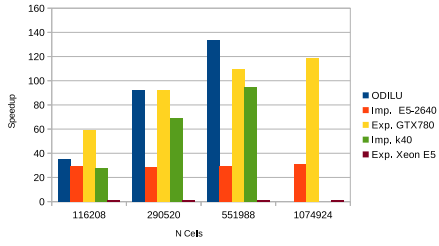
Figure 6.9: *Case T106C: Execution time of the implicit solver for one flow iteration and the explicit solver for equivalent flow iterations both on the CPU(dashes line) and the GPU (full line) as a function of the convergence ratio for increasing mesh resolution (a: smallest mesh, d: largest mesh).*

tion does not cover, however, the *on-demand* factorization (cf. section 5.4.2). This technique acts on reducing the global execution time of implicit solvers on the GPU by skipping most of the ILU factorization used for the preconditioner. It does not change the number of flow iterations and thus the convergence ratio is untouched. Figure 6.10a depicts the speedup (with reference to the explicit solver on E5-2640) of an entire flow simulation around the T106C blade including the performance of GPU solvers with the *on-demand* factorization (ODILU). The CPU delivers similar performance independently from the mesh size while the GPU has a better performance for larger meshes.

For a higher CFL number, every single implicit flow iteration is slower on both the GPU and the CPU but the convergence ratio (See Equation (6.3)) is larger favoring the implicit solver. For a $CFL = 100$, the implicit solver requires between 756 and 781 flow iterations which corresponds to a drop of 42% in the number of flow iterations compared with the number of flow iterations of the implicit solver with $CFL = 50$. Figure 6.10b summarizes the execution time until convergence of the used solvers scaled by the execution time of the explicit solver on the CPU for a $CFL = 100$. Depending on the mesh size the fastest combination is changing. The convergence ratio contributed largely to the improved GPU Implicit performance compared to the explicit CPU solver, which is used as a reference, as reflected by the increase of the speedup for the GPU implicit solver in Figures 6.10b compared to Figure 6.10a. Higher CFL values indeed reduce the total number of flow iterations required for the flow convergence, which entails fewer ILU builds. At the same time, the linear solver requires more linear iterations to converge, since the linear system is more ill-conditioned but this does not outweigh the reduction in time achieved by avoiding more ILU builds. As a result, the increase of the CFL number improved also the acceleration of the CPU implicit solver compared to the reference CPU explicit solver.



(a) $CFL_{Imp.} = 50$



(b) $CFL_{Imp.} = 100$

Figure 6.10: *Speedup of all solvers with reference to the explicit solver on E5-2640 for two different CFL numbers for the implicit solver and the same CFL number for the explicit solvers ($CFL_{Exp.} = 2.5$).*

6.5 Discussion

Explicit solvers on a single core CPU are the slowest alternative in all studied test cases since stencil-based operations have a great potential of parallelization that the serial implementation is not using. The GPU parallelization enables the explicit solver to still compete with implicit solver, even though the explicit convergence rates are not good. Explicit solvers are so efficient in memory usage that very heavy meshes of millions of cells still fit in the GPU global memory. An interesting research direction is then, to work on improving the convergence rates of explicit solvers without dramatically penalizing the execution time of one flow iteration. Figure 4.26 in Subsection 4.2.6 showed a case of a 4x times slowdown in the explicit solver performance for a doubling of the convergence rate. The multigrid technique could be more promising as its additional cost is normally a portion of the execution time of the fine mesh (Cf. Subsection 4.2.6).

When the mesh fits in the GPU memory, the GPU implicit solver is the fastest in the two cases since it combines the accelerated stencil operation for the residual and the Jacobian calculation while not suffering very much from the preconditioning due to the *on-demand* factorization. Optimizing the preconditioner update within a large sequence of flow iterations within the steady CFD simulation is a promising research direction to further enhance the performance of the GPU implicit solver. Hartmann et al. [2009] assessed the impact of using a periodically updated preconditioner on the number of linear solver iterations on the CPU. Tebbens and Tuma [2007] introduced a similar performance assessment for a proposed method of approximate preconditioner update on the CPU.

The memory usage of implicit solvers limits, however, the mesh size to about one million cells for Tesla K40 and 0.25 million for GeForce 780 with the actual solver memory footprint. This number can fluctuate based on the memory usage optimization but unless no matrix storage is done it should remain in the same order of magnitude. Therefore, for large meshes the GPU device memory could be not enough and then the explicit GPU and the implicit CPU have comparable performance. For some applications (e.g. the adjoint method), the flow should be resolved to machine accuracy (a residual drop of 10^{-16}). In that case, the ratio of convergence between the explicit and the implicit schemes is much larger. At the same time, meshes are for other applications (e.g. chemical kinetics in reactive-flow simulations) very large for actual CPUs capacities to use an implicit solver, consequently for that case the explicit solver is the only alternative available.

The order of the space integration has an impact on the amount of computation for every flow iteration. This might change the ratio between the space and the time integration for the implicit solver. Nevertheless, as the same space integration is to be found in both the implicit and the explicit solvers, the order of the discretization scheme has no impact on the choice of the best combination between time integration method and hardware. The finding of this work apply mostly to Finite Volume discretized flows in turbomachinery. The convergence ratio, which is very important in the comparison depends also on acceleration techniques of both methods.

6.6 Conclusion

Different combinations of time integration methods and computational hardware have been presented. A classification has been introduced based on the suitability of CFD operations to the GPU hardware. The comparison highlighted the high possible acceleration of explicit solvers as based on embarrassingly parallel functions. Difficulties in accelerating implicit solvers have been discussed including inherently sequential incomplete factorization used as a preconditioner for ill-conditioned linear systems in the implicit time integration. It has been observed that the GPU is able to extend the range of usability of explicit solvers to averagely good converging solvers. The findings prove, that the choice between explicit and implicit time integration relies mainly on the convergence of explicit solvers and the efficiency of preconditioners on the GPU for the implicit solver.

7

Applications: GPU CFD solvers in Design Optimization

The previous chapter compared the two CFD solvers - ported in this work - both for one flow iteration and for a converged simulation. In this chapter, these solvers are used in an aerodynamic shape optimization of a compressor and a turbine cascades. For both optimization test cases, the convergence of the explicit and the implicit RANS solvers is compared and used with the model shown in the previous chapter to decide which CFD simulation is faster and will be chosen for the aerodynamic shape optimization. In a second phase, the chapter analyses the GPU potential of the Kriging interpolation method used in this work as a surrogate model for the optimization of a turbine cascade.

7.1 One-level optimization of a supersonic compressor cascade

The GPU-accelerated CFD solvers with explicit and implicit time-stepping have been integrated in the in-house optimizer CADO [Verstraete, 2010], which uses the differential evolution algorithm. The test case is a supersonic compressor cascade designed initially for an inlet Mach number of 1.3 and a pressure ratio of 1.67. Total pressure and temperature are imposed at the inlet in addition to the flow angle while static pressure is imposed at the outlet.

This chapter is based on the article:

M. H. Aissa, T. Verstraete, and C. Vuik. Aerodynamic optimization of supersonic compressor cascade using differential evolution on GPU. In *AIP Conference Proceedings. Eds. Theodore Simos, and Charalambos Tsitouras.*, volume 1738, page 480077. AIP Publishing, 2016 .

Before setting the optimization case, the performance of both CFD solvers has been compared. The convergence ratio, defined in this work as the ratio of the number of iterations until convergence for both solvers, is equal to 14.38. This relatively low ratio resulted from the low maximum allowed CFL number ($CFL = 15$) for the implicit solver due to convergence problems compared to a $CFL = 2.0$ for the explicit solver. Figure 7.1 shows the performance model corresponding to the compressor cascade case based on the execution time of one flow iteration as explained in Chapter 6. For a convergence ratio smaller than 100 the explicit GPU solver is the fastest choice followed by the explicit solver on the CPU, the implicit solver on the GPU and the implicit solver on the CPU.

Therefore, the explicit CFD solver on the GPU is chosen for the aerodynamic shape optimization of the compressor cascade.

The flow around the pre-compression baseline design (Figure 7.3a) experiences a bow shock ahead of the profile leading edge, which propagates toward the adjacent blade to form a passage shock. The passage shock interacts with the adjacent blade boundary layer and induces a boundary layer separation. The achieved flow turning is 4 degrees and the static pressure ratio is 1.67, which is mainly achieved through the normal passage shock.

The optimization objective is to minimize the losses in terms of mass averaged entropy generation ($\Delta S = \frac{S_{\text{OUT}} - S_{\text{IN}}}{S_{\text{REF}}}$) at the outlet for the same flow conditions. The optimization problem has one constraint, which is to keep the flow turning equal or greater than the baseline case. The blade geometry is defined by superimposing a parametric thickness profile to a parametric camberline as sketched in Figure 7.2. The optimization variables are the y-coordinate of 5 control points of the blade angle distribution B-spline. The thickness distribution remains on the other hand fixed during the optimization. A population of 40 individuals has been evolved for 9 generations through the differential evolution method to reach at the end a total improvement of 20% in terms of entropy generation.

A comparison of the flow around the baseline and around the optimized blade shows two main differences: the normal shock turned into a partially oblique one and the passage shock moved downstream approaching the trailing edge. The change in the flow is due to a reduction in the incidence angle on the leading edge (LE) area which reduces the losses by bringing the bow shock next to the LE and decreasing the expansion at the suction side responsible for the flow acceleration before it reaches the passage chock. Starting from the stagnation point on the LE area the flow is first accelerated on the original blade suction side due to an inappropriate incidence reaching a peak Mach number as high as 1.7. Contrary, on the optimized blade the incidence has been adjusted to the flow direction reducing the peak Mach number to 1.5 only. Further downstream both profiles decelerate the flow on the suction side through a curvature opposed to classical subsonic profiles, achieving a pre-shock Mach number of 1.6 to 1.3 for the original and optimized blade respectively by Prandl-Meyer compression waves (Figure 7.3).

While this optimization, which required a total of 360 flow evaluations, would run 32 days on a single CPU, it took only 2 days on 2xK40 GPUs.

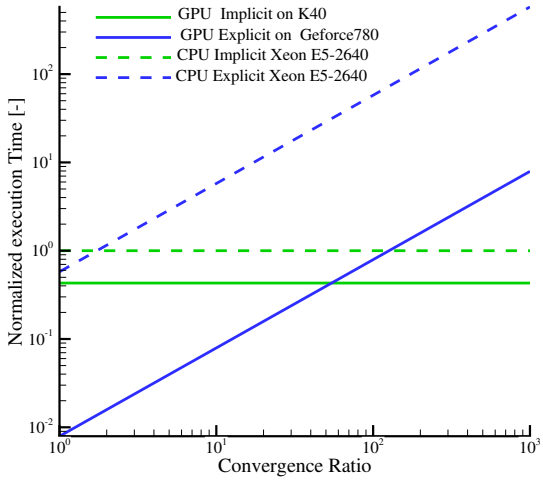


Figure 7.1: *Execution time of converged simulations normalized by the execution time of the implicit simulation on the CPU for the compressor cascade test case.*

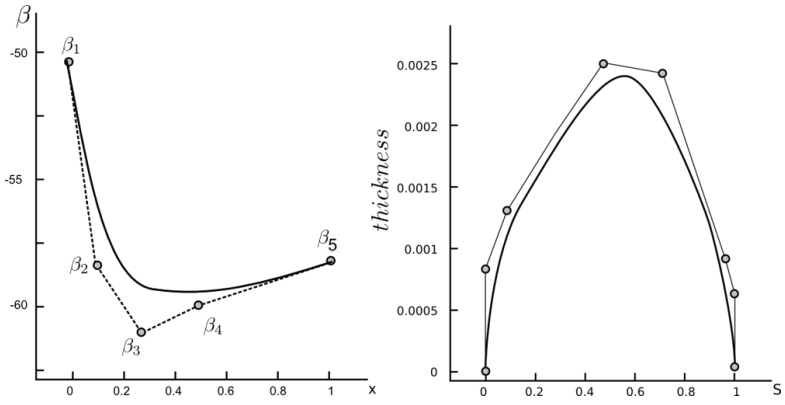


Figure 7.2: *Parameterization using Bezier splines for the camberline and the thickness of the compressor blade.*

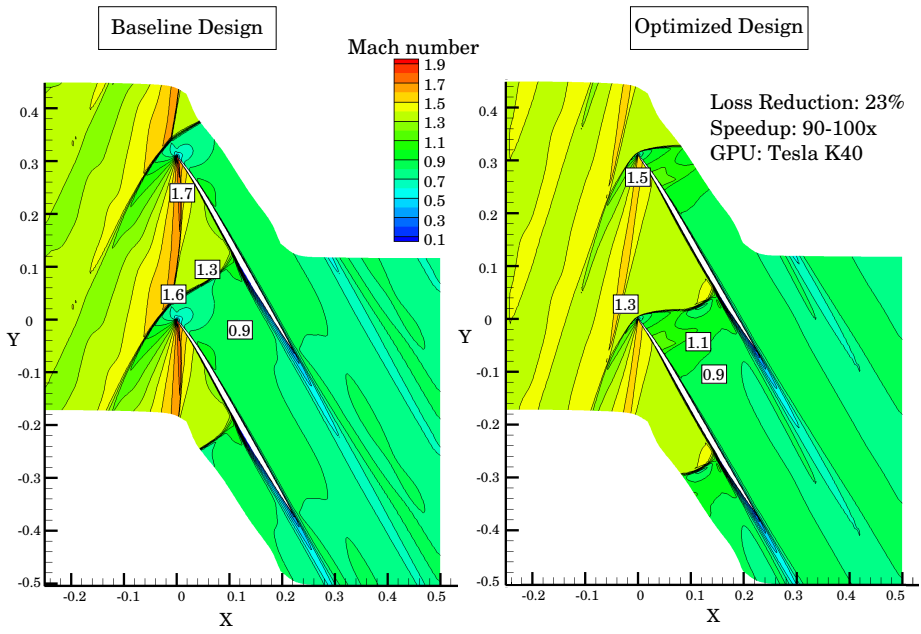


Figure 7.3: *Mach contours of the baseline and optimized design of the supersonic compressor cascade ($P_{01} = 1bar$ $T_{01} = 300K$ and $\alpha_1 = -64.5$).*

7.2 Metamodel-assisted optimization using Kriging

This section introduces shortly the Kriging interpolation method and the way it is used within CADO. The exposed parallelism of this method is discussed before the 2-level optimization case of the inlet guidance vane LS82 [FOTTNER, 1990] is presented.

7.2.1 Kriging

Interpolation methods are widely used to build surrogate models capable of emulating the response of an expensive simulation. Kriging is a powerful interpolation method praised for being capable of exactly predicting provided samples. The method offers also a possibility to estimate the error and the expected improvements [Jones, 2001] of a prediction. Figure 7.4 shows the algorithm of the ordinary Kriging method as used in this work. The method builds first a correlation matrix to map available data, which in this work cover the different design variables (thickness distribution, camber line, sweep ...).

$$\mathbf{R}_{ij} = exp(-\sum_{l=1}^{d} 10^{\theta^{(l)}} ||\vec{x}_i^l - \vec{x}_j^l||^{p^l}) \tag{7.1}$$

with d the number of dimensions and θ and p two hyper-parameters controlling the range of the correlation and its smoothness. In general the case $p = 2$ is used to guarantee an indefinitely differentiable response [Sacks et al., 1989]. The diagonal elements of the correlation matrix are equal to one and the closer two points are the larger is the entry.

This correlation, more precisely its inverse (R^{-1}), is the base to compute a variance ($\hat{\sigma}^2$) and a mean ($\hat{\mu}$) both used to formulate a prediction $y(x^*)$:

$$\hat{\mu} = \frac{\mathbf{1}^T\mathbf{R}^{-1}\mathbf{y}}{\mathbf{1}^T\mathbf{R}^{-1}\mathbf{1}} \quad (7.2) \qquad \hat{\sigma}^2 = \frac{1}{n}(y - \mathbf{1}\hat{\mu})^T\mathbf{R}^{-1}(y - \mathbf{1}\hat{\mu}) \quad (7.3)$$

$$y(x^*) = \hat{\mu} + \mathbf{r}^T\mathbf{R}^{-1}(y - \mathbf{1}\hat{\mu}) \quad (7.4)$$

The quality of the prediction can be assessed by cross-validation (CR) or Maximum Likelihood Estimation (MLE) and this work implements the MLE using the following likelihood formulation:

$$\phi = -\frac{n}{2}ln(\hat{\sigma}^2) - \frac{1}{2}ln(|\mathbf{R}|) \quad (7.5)$$

Independently of the method used to train the hyper-parameters some variables are always needed such as the mean, the variance, the correlation matrix and its inverse. Toal [2016] compares the performance of 5 formulations of Kriging on the CPU and on the GPU. The author found that GPUs outperform CPUs for high dimensional problems when the problem size is relatively large, while small problems suffer from the slow matrix inversion on the GPU. The parallelization is, however, done with Matlab with no access to low-level code optimization relying solely on the GPU and CPU optimization of Matlab. The authors suggested an automated hardware choice based on the performance of both the GPU and the CPU on a small set of data. The algorithm chooses the fastest hardware to perform the preliminary test.

As shown in Figure 7.4 ordinary Kriging requires a set of samples (input x) and related output y in addition to a start value for θ, which is altered within an optimization process. Within the optimization loop, first the correlation matrix is built and its inverse is computed using, for instance, the LU factorization. The inverted correlation matrix and the output vector are essential to compute the mean $\hat{\mu}$, which is used to compute the variance $\hat{\sigma}^2$ (see Equations (7.2) and (7.3)). Once the variance $\hat{\sigma}$ is available the likelihood function can be calculated as it requires only the variance and the determinant of the correlation matrix $\mathbf{R}$. An optimization algorithm can be used to maximize iteratively the likelihood. In this work the finite difference method is used. When an optimal value of θ is reached the training phase of the Kriging model is finished and the condensed data inside the model is ready to be used to search for the best design.

The most time consuming operation in Kriging is the inversion of the correlation matrix, for which the CPU version of CADO uses the LU factorization. The LU decomposition ($A = LU$) is also used to compute the determinant of the correlation matrix, which is needed for the prediction through Kriging (see Equation (7.4)). The benchmark showed in Table 7.1 covers the LU inversion on a single core CPU (in-house implementation) and the GPU LU factorization of CUSOLVER for different

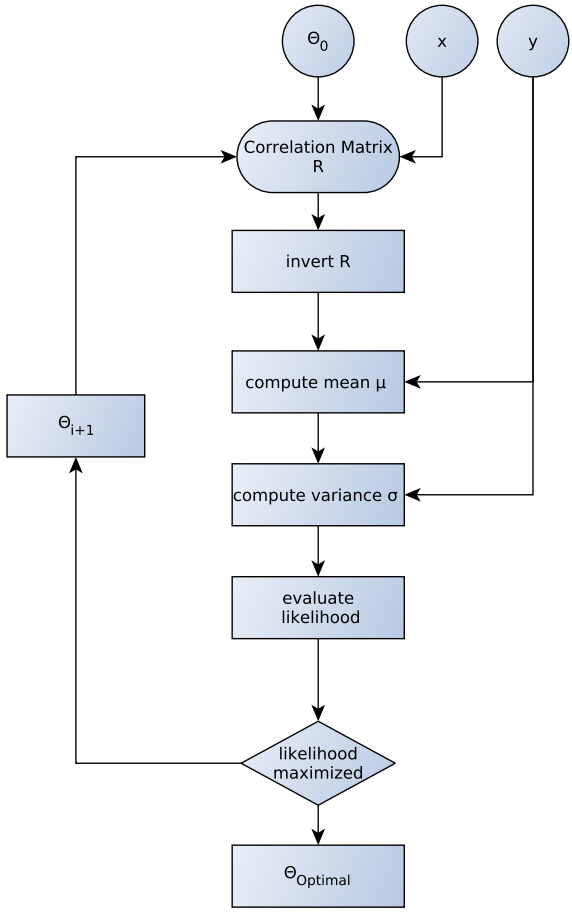


Figure 7.4: *Chart of the ordinary Kriging algorithm.*

sizes of the correlation matrix. The correlation matrices have been filled with realistic data samples extracted from 5D Dejong case [Jong, 1975].

The factorization of a matrix is, however, an inherently serial procedure and important speedups are measured only for matrices of $N_{Row} >= 400$. Toal [2016] found similar results for a Cholesky decomposition, as the Tesla K20C GPU is faster than the i7-2860 CPU only for data sets of more than 700 elements. The profiling of the GPU version showed that the GPU is having low compute and memory utilization for $n < 400$.

7.2.2 The LS82 cascade

The LS82 turbine inlet vane is a very high turning blade design for Mach numbers ranging from 0.7 to 1.4. The blade turns an incoming flow with 1.9 degree to an outflow angle of -80. Different outlet static pressures result into different test cases from subsonic to supersonic. In this work, the subsonic case with $M_{is2} = 0.85$ is considered. The subsonic flow is accelerated in the blades passage with no supersonic pockets and therefore no shocks. The trailing edge (TE) wake is then responsible for the total pressure loss.

The parameterization is similar to the previous optimization case with a combination of a thickness distribution and a camber angle distribution. The optimization has 9 design variables controlling the blade thickness and an objective of a loss minimization. The loss is quantified with the mass-averaged entropy increase. The only constraint is to keep the outflow angle smaller than -80 ($\alpha_2 <= -80$).

The first step is about choosing the appropriate time integration for the CFD evaluation. Figure 7.5 shows the performance model of the LS82 case [†]. For a convergence ratio ($N_{itr}^{Exp}/N_{itr}^{Imp}$) larger than 36, the implicit solver on the GPU is the fastest alternative. For the LS82 case the convergence ratio has been calculated and is equal to 457.5 favoring obviously the implicit solver[‡]. Therefore, the GPU implicit solver is used for the CFD evaluation.

The 1-level optimization algorithm of CADO has been run in addition to the metamodel-assisted algorithm using Kriging. Both optimizers, running independently, reduced the losses by decreasing the flow turning drifting, however, into regions of non-allowed outflow angles (see Figure 7.6).

The 1-level optimization with 20 CFD evaluations per optimization iteration required 29 iterations (582 CFD evaluations in total) to reach similar regions on the

[†]Defined in last chapter as the execution time of converged simulations (Explicit/Implicit on the CPU/GPU) normalized by the execution time of the implicit solver on the CPU.

[‡]The explicit solver require 457.5 times more iterations to converge than the implicit one.

Table 7.1: *Speedup* $S_{LU} = T_{\text{CADO}}/T_{\text{CUSOLVER}}$ *of the LU factorization on the K40 over Xeon E5 for different matrix sizes.*

N_{Row}	100	200	400	800	1000
S_{LU}	0.63	2.25	7.17	25.05	52.43

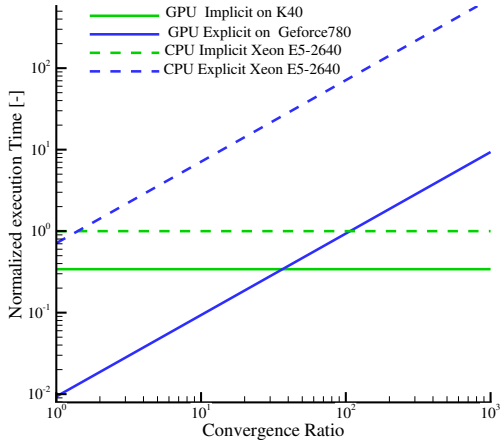


Figure 7.5: *Execution time of converged simulations normalized by the execution time of the implicit simulation on the CPU for the LS82 test case.*

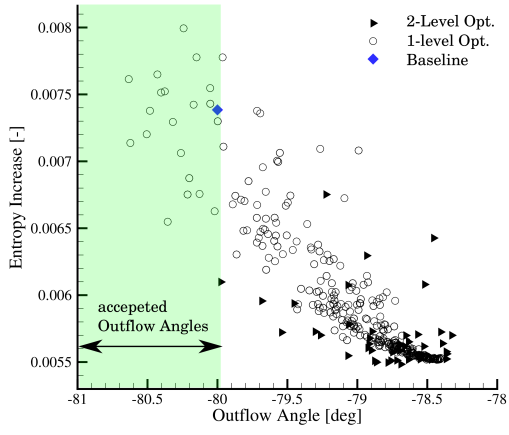


Figure 7.6: *Comparison of the results of both the 1-level and 2-levels optimization algorithm for the LS82 case highlighting the region of accepted designs.*

objective space the Kriging reached within few iterations and using only one CFD evaluation per iteration (see Figure 7.7). After few optimization iterations, the 1-level optimizer lost all accepted designs from the population and the mutation, as a diversity enabler in the differential evolution algorithm [Storn and Price, 1997], was not enough to recover those designs. Moreover, the Kriging is more explorative and even if it drifted into the region of non-accepted outflow angles, it delivered in 42 iterations an accepted design with 17.4% of improvement in the loss minimization compared to the baseline design. Figure 7.8 shows also that Kriging has been well approximating the outflow angle and to a smaller extend the entropy increase.

The optimized blade delivered by the metamodel-assisted optimizer at iteration 42 shows larger Mach numbers on the suction side (see Figure 7.9). The same effect is observed on the isentropic Mach number at the blade surface (see Figure 7.10) with a delayed but more pronounced acceleration on the suction side and a following smoother deceleration. This is caused by the decrease of the volume of the leading edge which reduces the converging nozzle effect. This had an effect on the TE wake which is less turbulent for the optimized case generating less entropy as shown in Figure 7.11. Figure 7.12 shows the total pressure for the optimized case and the total pressure at some locations extracted following the axial direction from the passage until the outlet. The total pressure experiences a decrease when crossing the wakes generated by the cascade blades with the first wake being the most important while the rest are less pronounced.

7.3 Final remarks

The GPU simulations experience a speedup ranging from one to two orders of magnitude. The comparison of all 4 alternatives in the previous chapter showed that for a fast converging explicit solver, the GPU explicit solver is the fastest alternative. In general the difference in convergence rates is outbalancing the GPU speedup making the GPU implicit solver the most convenient for use in design optimization. With this kind of solver a limit of around one million cells for the mesh is present.

As the speedup for implicit solvers is less than one order of magnitude, it can be outbalanced when a set of few CPUs is used. Especially in population based design optimization, an entire generation of designs needs to be evaluated not forgetting the initial database. In that case, running on every CPU core the slower implicit CPU evaluation is more advantageous since as many designs will be analyzed as the CPU has cores. The scaling is not perfect when using shared memory as all simulation share the same cache and RAM. These scalability issues disappear when using clusters of nodes.

Therefore, it is not recommended to let the GPU evaluate one by one a large set of designs, within a 1-level generation or a database, while an actual CPU can evaluate multiple designs at the same time. On the other hand, the two level optimization making use of meta-models uses only very few evaluations for every iteration after it builds the database. In that configuration it is possible to make a good use of the GPU as these few individuals can run faster on the GPU than on the multi or many-cores CPU systems.

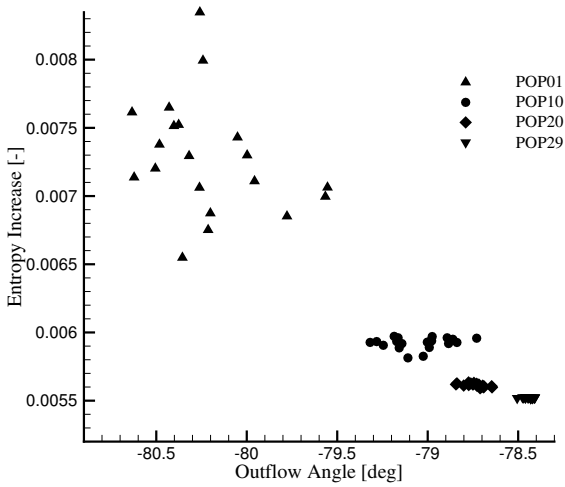


Figure 7.7: *Results of the 1-level optimization algorithm for the LS82 case.*

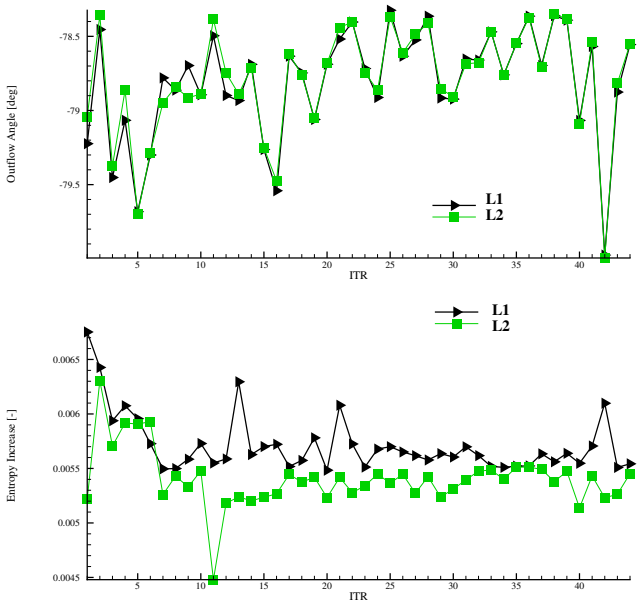


Figure 7.8: *Comparison of the high-fidelity and Kriging values for the outlet angle (top) and entropy increase (down).*

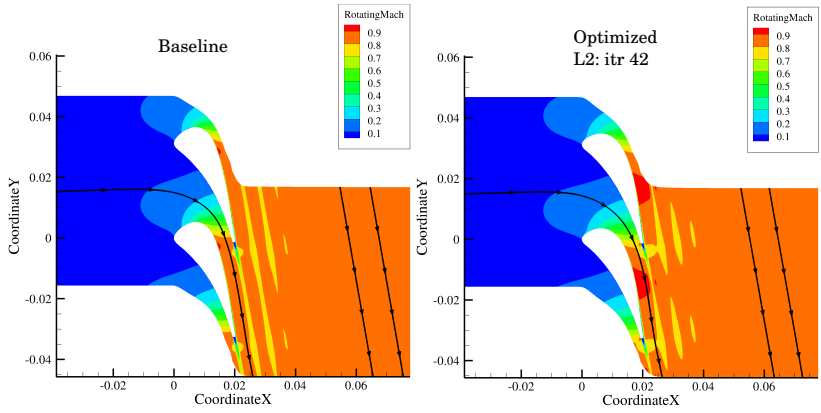


Figure 7.9: *Plot of the Mach number contours of the baseline and optimized design (Kriging assisted Iteration 42).*

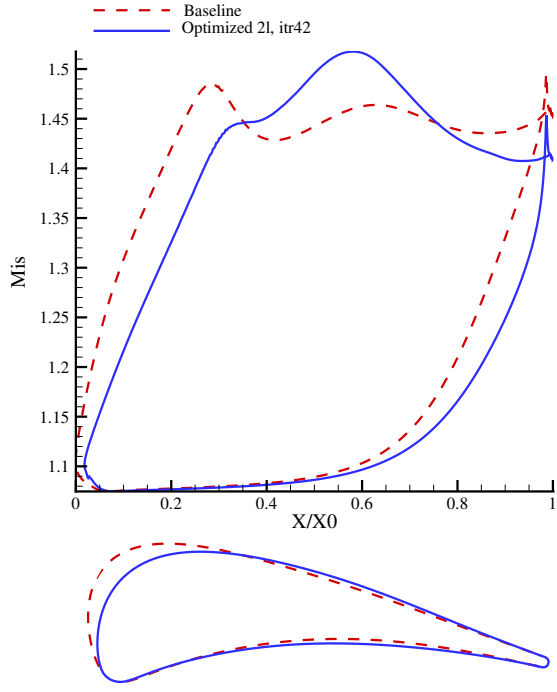


Figure 7.10: *Plot of the isentropic Mach number on the blade surface for the baseline and the optimized design (Kriging assisted Iteration 42) with the two profiles shown on the bottom.*

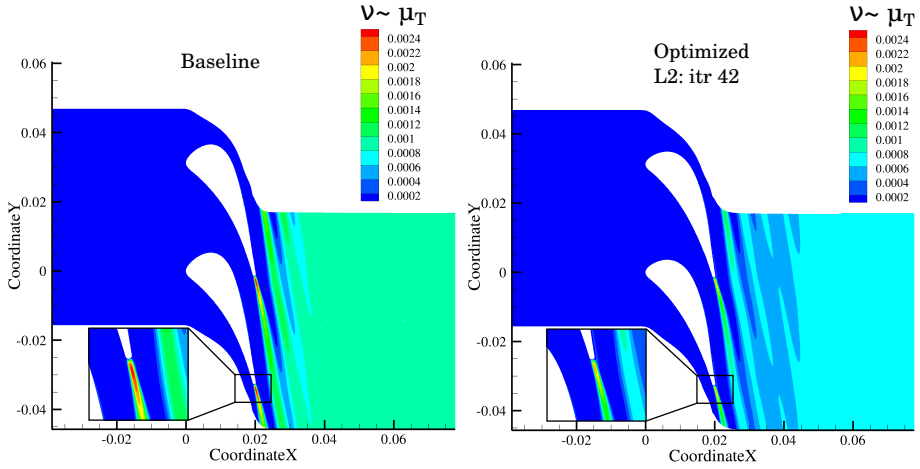


Figure 7.11: *Plot of the Spalart-Allmaras transport variable $\tilde{\nu}$ contours (proportional to the turbulent eddy viscosity) of the baseline and optimized design (Kriging assisted Iteration 42).*

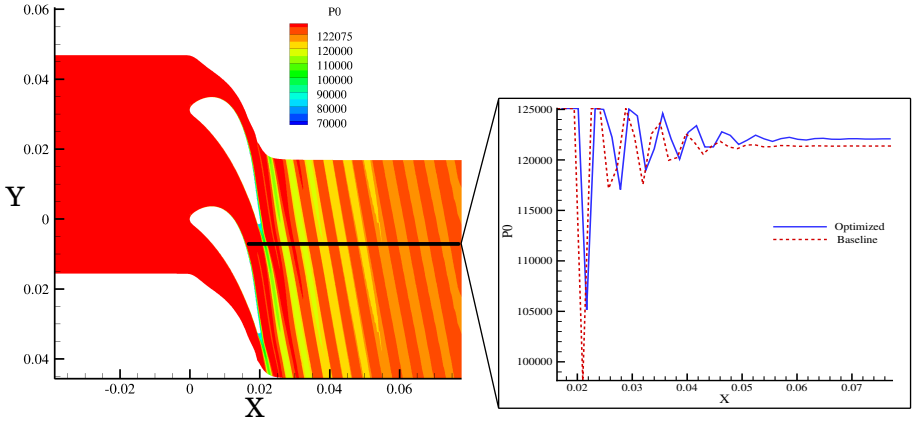


Figure 7.12: *Plot of the total pressure contours of the optimized design with Kriging assisted Iteration 42 (left) and a comparison of the total pressure along axial direction with the baseline design (right).*

8 Conclusions

The throughput-oriented latency-tolerant GPU is different from the cache-based CPU architecture. Some algorithms (e.g. flux computation) are inherently parallel and suited to the execution on the GPU, while others need to be optimized following the profiler guidance to get an average acceleration. Some other algorithms (e.g. matrix factorization) are inherently serial and can not get accelerated unless the algorithm is substantially changed. Large scale problems run the GPU more efficiently and stencil-based option emanating from large-scale problems are the best suit for the GPU architecture.

The general pattern seen in the literature is that the CPU governs the MDO method and starts the GPU as a workhorse to perform the bulk of the arithmetic operations. Consequently, GPUs have no tangible impact on the MDO methods themselves beside the fact that they accelerate single-field simulations in a similar way traditional clusters do. The limited use of GPUs as co-processors for the acceleration of special parts of an established algorithm is thus self-explanatory.

The CFD solver with explicit time-stepping provided the GPU with the stencil-based operations needed to boost the compute and memory utilization and reach large speedups. Moreover, the lookup table for the interface update made it possible to have a GPU resident solver, which is mandatory to avoid the costly CPU-GPU data transfer. An additional code optimization concerned the solving of the race condition, for which multi-coloring showed better results than redundant computation as most of the explicit kernels are compute-bound. Explicit solvers are consequently the candidate for a substantial GPU speedup, while the flow convergence is the bottleneck. Multigrid and IRS showed an improve of the flow convergence at the expense of a reduced fine-grained parallelism. A GPU-friendly convergence acceptation method has the potential of creating a super fast method, a serious competitor for the implicit time-stepping.

The CFD solver with implicit time-stepping experienced smaller speedups because of the factorization based preconditioning. Even the assembly part which offers stencil-based operations expressed less speedup than the explicit solver since the memory consumption per stencil is much larger (e.g. Jacobian computation). Unlike the explicit space and time integration, the assembly was faster using redun-

dant computation than multi-coloring in solving the race-condition as the occupancy was the main bottleneck. The sorting of the data into CSR format was a bottleneck for the assembly and has to be replaced by a direct insertion algorithm. For the system solving, an on-demand option improved the performance but a GPU-friendly and efficient preconditioner should be available to reach interesting speedups with the GPU. Boosted by the assembly speedup the implicit solver can be beneficial for an application having a large portion of the execution time on the assembly.

Comparing both solvers resulted into a classification of CFD operations based on the exposed parallelism. The more a solver relies on a well-accelerated operation the larger is the expected speedup while it can not exceed the one of the operation itself. In this work, a limit of 2 orders of magnitude has been observed. Very large speedups reported in the literature would reflect rather an unusually slow CPU reference code (e.g. performance of Matlab large scale code in general). For one flow iteration every solver makes use of some of the classified CFD operations and having the explicit solver on the GPU to run the fastest flow iteration is a direct result of the classification. For a whole simulation the performance ranking depends on the convergence of the explicit solver. If they converge well they should be used otherwise the implicit time-stepping remains the fastest choice. However very large meshes could exceed the GPU memory and then the explicit GPU solver would have to be compared with the implicit solver on the CPU. The concept of the classification can be used for other disciplines and applications in order to examine the potential acceleration an emergent hardware can bring. It gives an operation-specific acceleration that brings more insights than a simple overall speedup, which hides performance bottlenecks. Both the classification and the performance model help pick the fastest combination of hardware and algorithm.

The application of these tools to two optimization cases showed that the explicit solver was best suited for the compressor cascade while the implicit solver was best suited for the turbine cascade. It required a full run to get the convergence behavior of both solvers and a run for one flow iteration to build the performance model. In general, some heuristics would help estimate the convergence behavior. In this work, the explicit solver was better performing for transonic compressors and struggling to converge for turbine cascades in general. The contrary is true for the implicit solver.

Finally for the GPU accelerated optimization, the GPU is a serious alternative for the acceleration of high-fidelity evaluations within a metamodel-assisted optimization. In that case, a cooperation is essential between both systems. Therefore a directive-based method for the porting of a code which could run on multiple systems is a promising research direction. OpenACC and similar software are maturing and probably at one point, they will reach the maturity of similar systems for the CPU such as openMP. The low-level programming similar to CPU would be probably only the topic of a reduced community and their output will flow into a better high-level compiler. Like CUDA was a relieve from the use of graphical languages other concepts will emerge to translate CPU code to the GPU. Nevertheless, similar to the need to master the assembly language to get the last drop of performance of a CPU code the same will remain true for the GPU with the CUDA language.

8.1 Future work

The work on the deployment of the GPU computational power in aerodynamic shape optimization accumulated enough evidence to identify future promising research directions:

8.1.1 Convergence acceleration of explicit solver

The explicit solver showed the best acceleration among all the methods ported to the GPU. It has been discussed in chapter 6 that it belongs to a class of stencil-based operations best fitted to the GPU. Moreover, their low memory consumption compared to the implicit counterpart, make it possible to run realistic test cases with millions of cells in one GPU card. The implicit solver on the GPU can be faster propelled by its high CFL number. Therefore, the interesting research path is about improving the explicit solver convergence, while at the same time not limiting its fine-grained parallelism. This thesis presented a basic study on that direction benchmarking both IRS and MG and coming to the conclusion that IRS is damaging the GPU performance more than it increases the CFL. There is a need to find a GPU-friendly implementation for the IRS and at the same time look for other convergence acceleration methods, which have an embarrassingly parallel algorithm. A method by Swanson et al. Swanson et al. [2007] is able to reach a CFL number of 1000 with a multistage Runge-Kutta scheme through a preconditioning with fully implicit operator used as a smoother within a multigrid scheme. It could be interesting to see its performance on the GPU as it is used in many industrial CFD packages (e.g Numeca Fine)

8.1.2 Metamodeling on the GPU

The benchmarking of Kriging interpolation methods for the use as meta-models showed that the GPU can deliver interesting speedup exceeding two orders of magnitudes given a large scale correlation matrix. As the matrix has the size of the number of samples for Kriging, Kriging can not be substantially accelerated by the GPU unless multiple matrix inversions are done at the same time. Multi-fidelity Kriging also known as Co-Kriging could show a better GPU acceleration as the correlation matrix contains sample originating from a fast low fidelity simulation and therefore a large number of them could be generated in short time to build large scale correlation matrix. The most promising direction is to use gradient-enhanced Kriging which operates a correlation matrix of the size of the number of samples multiplied by the number of dimensions (equal to design variables). At the same time, the inversion should be implemented manually in order to avoid any time-consuming synchronization present now in used GPU linear algebra packages.

A

Appendix: Used Test cases

The test cases used throughout this work includes two cascades of turbine stators (t106c [Michálek et al., 2012] and LS89 [Arts et al., 1990]) and two cascades of compressor stators (CC2D [Aissa et al., 2016] and Turbolab[†]).

A.1 Turbine inlet guide vane: LS89

The experimental investigation for the transonic turbine guide vane cascade LS89 have been carried out in the von Karman Institute Isentropic Light Piston Compression Tube facility (CT-2) [Arts et al., 1990]. The blade shape has been optimized for an isentropic Mach number of 0.9 using an inverse method [Van den Braembussche et al., 1990]. Table A.1 gives few geometrical characteristics of the blade while the full documentation along with the manufacturing coordinates are found in the work of Arts et al. [1990]. Figure A.1 shows the blade cascade and Figure A.2 shows the meshing of the flow domain with more details on the leading and trailing edges.

[†] `http://aboutflow.sems.qmul.ac.uk/events/munich2016/benchmark/testcase3/`

Table A.1: *Geometrical characteristic of the LS89 blade*

Chord c [mm]	67.647
Pitch g [mm]	57.49995
Solidity c/g [-]	1.17647
Stagger angle γ	55.0 deg (from axial direction)

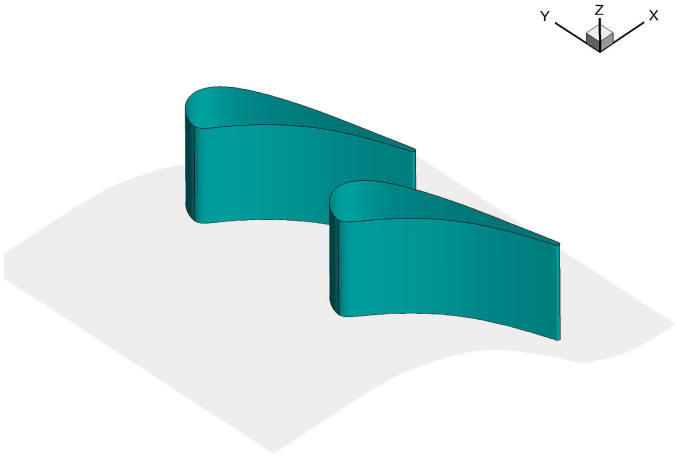


Figure A.1: *The LS89 blade cascade*

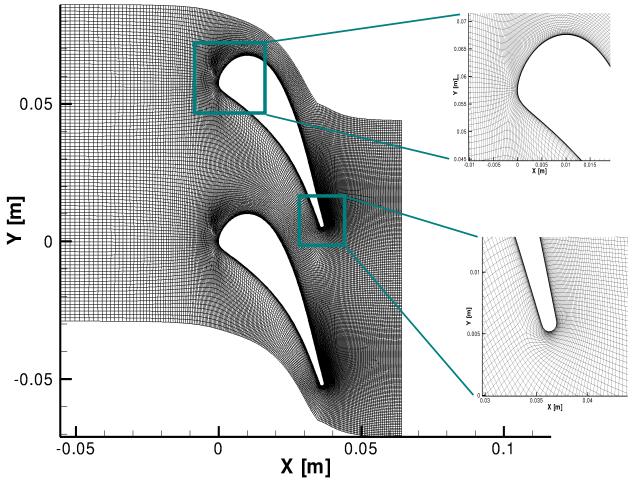


Figure A.2: *Meshing of the LS89 cascade*

A.2 Compressor stator cascade: CC2D

The test case is a supersonic compressor cascade designed initially for an inlet Mach number of 1.3 and a pressure ratio of 1.67. Total pressure and temperature are imposed at the inlet in addition to the flow angle while static pressure is imposed at the outlet. Table A.2 shows the set of used meshes with increasing mesh sizes. Figure A.3 the mesh number 2 with emphasis on the leading and trailing edge part.

A.3 Turbine stator cascade: T106C

The T106C is a very high lift, mid-loaded low pressure turbine blade. It has been studied at the VKI S1/C high-speed wind tunnel [Michálek et al., 2012]. The meshing kept the number of cells in the xy-plane constant while increasing the number of cells on the z-direction. The T106C turbine stator experiences a 2D flow, therefore the same flow phenomena are solved for all generated meshes. This ensures, that only the amount of computational work is increasing which is the focus of the computational performance study.

A.4 3D Compressor stator blade: Turbolab

The "TurboLab Stator" is a stator in a measurement rig at the TU Berlin in the TurboLab at the Chair for Aero Engines. The stator geometry has been designed based on a representative stator geometry as used in modern jet engine compressors[†]. As shown in Figure A.4, the flow entering the stator has an inlet angle of 42 degrees from the axial direction. The blade is required to turn the flow to the axial direction with a minimum loss.

[†] `http://aboutflow.sems.qmul.ac.uk/events/munich2016/benchmark/testcase3/`

Table A.2: *Mesh sizes for the CC2D case*

Grid number	N_{Cells}
1	25 k
2	46 k
3	57 k
4	75 k
5	110 k
6	126 k
7	394 k

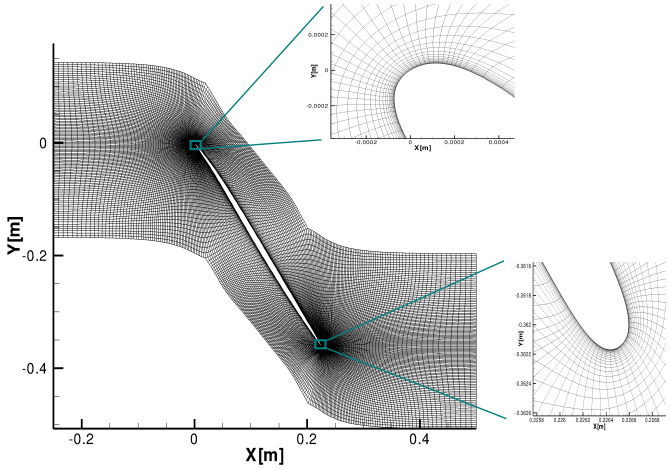


Figure A.3: *Meshing of the compressor cascade CC2D (mesh number 2)*

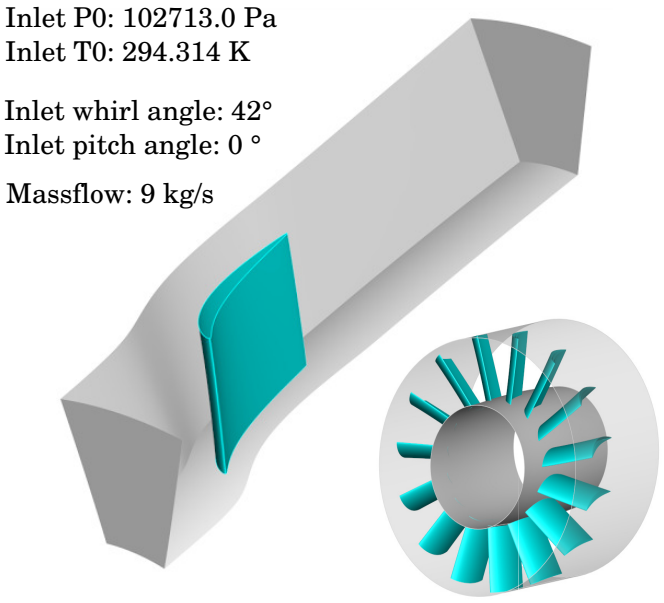


Figure A.4: *Turbolab turbine blade*

Curriculum vitae

- 2013/4- present: Early Stage Researcher at the Turbomachinery & Propulsion Department of the von Karman Institute of Fluid Dynamics, Sint-Genesius-Rode, Belgium.
  - Responsible for the porting of CFD modules to GPU using CUDA C.
  - Automatic design optimization using ported CFD simulations.
  - GPU implementation of interpolation methods (Kriging) for evolutionary Optimization methods.
- 2013/04-2017/10: Ph.D., Delft Institute of applied mathematics, Delft University of Technology, the Netherlands.
- 2007/10-2013/02: Diplom in Aerospace Engineering, Technical University of Stuttgart, Germany
  Specializations:
  - Fluid dynamics
  - Flight mechanics and control
- 2006/10 -2007/6: German language Certificate (DSH3), University of Heidelberg, Germany.
- 2002/9-2006/6: High-school diploma (Baccalaurat), Neapolis Elite school, Nabeul (Tunisia) predicate very good (Top 5%)
- Born on 08/03/1987 in Nabeul, Tunisia

List of publications and presentations

Journal papers:

- M.H. Aissa, T. Verstraete, and C. Vuik. Toward a GPU-aware comparison of explicit and implicit CFD simulations on structured meshes. Computers & Mathematics with Applications, 74(1):2012017, 2017b .
- M.H. Aissa, T. Verstraete, and C. Vuik. A thorough performance analysis of GPU-accelerated steady CFD simulations on structured meshes. Computers & Fluids, submitted.

Book chapter:

- Aissa M. H., Mller L., Verstraete T., Vuik C. (2017) Acceleration of Turbomachinery Steady Simulations on GPU. In: Desprez F. et al. (eds) Euro-Par 2016: Parallel Processing Workshops. Lecture Notes in Computer Science, vol 10104. Springer, Cham

International conference proceedings:

- Aissa, M. H., T. Verstraete, and C. Vuik. "Aerodynamic optimization of supersonic compressor cascade using differential evolution on GPU." AIP Conference Proceedings. Vol. 1738. No. 1. AIP Publishing, 2016.
- Aissa, M.H.; Verstraete, T.; Vuik, C. (2014) "Use of modern GPUs in Design Optimization", 10th ASMO-UK/ISSMO conference on Engineering Design Optimization, June 30th - July 1st 2014, Delft University of Technology, Delft, The Netherlands

List of conference presentations and talks:

- Aissa, M.H. ; Verstraete, T.; C. Chahine and C. Vuik (2016) Surrogate-Model Assisted Evolutionary Optimization of an Axial Compressor Stator 11th ASMO UK ISSMO, International Conference on Numerical Optimisation Methods for Engineering Design, Munich Germany 18-20/7/2016.

- Aissa, M.H. ; Verstraete, T. and C. Vuik (2016) How GPU Computational Power Can Alter the Established Comparison of Explicit to Implicit CFD Simulations?, 5th European Seminar on Computing June 5 - 10, 2016 Pilsen, Czech Republic.
- Aissa, M.H. ; Verstraete, T. and C. Vuik (2016) A GPU-Accelerated Steady CFD Solver for Turbomachinery Optimization, 28th International Conference on Parallel Computational Fluid Dynamics (Parallel CFD 2016), Kobe, Japan
- Aissa, M.H.; Verstraete, T.; Vuik, C. (2016) "Acceleration of turbomachinery steady simulations on GPU". 7th VKI PhD symposium, March 1-3rd 2016, von Karman Institute for Fluid Dynamics, Rhode-Saint-Gense, Belgium. Video of talk: `http://youtu.be/QRKrL44QLVw`
- Aissa, M.H.; Verstraete, T.; Vuik, C. (2015) "A GPU-Accelerated Navier-Stokes Solver for Multidisciplinary Design Optimization". 27th International Conference on Parallel Computational Fluid Dynamics (ParCFD 2015), May 17-21st 2015, Montreal, Canada

Bibliography

M.H. Aissa, T. Verstraete, and C. Vuik. Aerodynamic optimization of supersonic compressor cascade using differential evolution on GPU. In *AIP Conference Proceedings. Eds. T. Simos and C. Tsitouras.*, volume 1738, page 480077. AIP Publishing, 2016.

M.H. Aissa, L. Müller, T. Verstraete, and C. Vuik. Acceleration of turbomachinery steady simulations on GPU. In Desprez F. et al., editor, *Euro-Par 2016: Parallel Processing Workshops. Lecture Notes in Computer Science, vol 10104.*, pages 814–825. Springer, Cham, 2017.

G Allaire. Topology optimization with the homogenization and the level-set methods. In *Nonlinear homogenization and its applications to composites, polycrystals and smart materials*, pages 1–13. Springer, 2004.

S. R. Allmaras and F. T. Johnson. Modifications and clarifications for the implementation of the Spalart-Allmaras turbulence model. In *Seventh International Conference on Computational Fluid Dynamics (ICCFD7)*, pages 1–11, 2012.

J. D. Anderson. *Computational fluid dynamics: the basics with applications.* McGrawhill Inc, New York, 1995.

J. Andrews and N. Baker. Xbox 360 system architecture. *IEEE Micro*, 26(2):25–37, 2006.

H. Anzt, W. Sawyer, S. Tomov, P. Luszczek, I. Yamazaki, and J. Dongarra. Optimizing Krylov subspace solvers on Graphics Processing Units. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 941–949. IEEE, 2014.

H. Anzt, E. Chow, and J. Dongarra. Iterative sparse triangular solves for preconditioning. In *European Conference on Parallel Processing*, pages 650–661. Springer, 2015.

H. Anzt, E. Chow, T. Huckle, and J. Dongarra. Batched generation of incomplete sparse approximate inverses on GPUs. In *Proceedings of the 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 49–56. IEEE Press, 2016a.

H. Anzt, E. Chow, J. Saak, and J. Dongarra. Updating incomplete factorization preconditioners for model order reduction. *Numerical Algorithms*, 73(3):611–630, 2016b.

W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9(1):17–29, 1951.

T. Arts, M. Lambert de Rouvroit, and A.W. Rutherford. Aero-thermal investigation of a highly loaded transonic linear turbine guide vane cascade. TN 174, von Karman Institute for Fluid Dynamics, 1990.

V. Asouti, E. Kontoleontos, X. Trompoukis, and K. Giannakoglou. Shape optimization using the one–shot adjoint technique on Graphics Processing Units. In *7th GRACM International Congress on Computational Mechanics Conference, Athens*, 2011.

C.E. Augarde, A. Ramage, and J. Staudacher. An element-based displacement preconditioner for linear elasticity problems. *Computers & Structures*, 84(31): 2306–2315, 2006.

S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. H. Wen-mei. An adaptive performance modeling tool for GPU architectures. In *ACM Sigplan Notices*, volume 45, pages 105–114. ACM, 2010.

S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, et al. PETSc users manual revision 3.5. Technical report, Technical report, Argonne National Laboratory (ANL), 2014.

S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, K. Rupp, B. Smith, S. Zampini, and H. Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2015.

S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Solving dense linear systems on graphics processors. In *Euro-Par 2008–Parallel Processing*, pages 739–748. Springer, 2008.

R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, Philadelphia, PA, 1994.

N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA NVR-2008-004, Nvidia Corporation, 2008.

N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 18. ACM, 2009.

N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations version 0.5. 1, 2015.

N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing gems Jade Edition*, 2:359–371, 2011.

M. P. Bendsøe and N. Kikuchi. Generating optimal topologies in structural design using a homogenization method. *Computer Methods in Applied Mechanics and Engineering*, 71(2):197–224, 1988.

M. P. Bendsøe and O. Sigmund. *Topology optimization: theory, methods, and applications*. Springer Science & Business Media, 2013.

M. Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of Computational Physics*, 182(2):418–477, 2002.

K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), 2008.

P. Birken, J. D. Tebbens, A. Meister, and M. Tuma. Preconditioner updates applied to CFD model problems. *Applied Numerical Mathematics*, 58(11):1628–1641, 2008.

J. Blazek. *Computational Fluid Dynamics:Principles and Applications: Second edition*. Elsevier, 2005.

J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 917–924. ACM, 2003.

T. Brandvik and G. Pullan. Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 221(12):1745–1748, 2007.

T. Brandvik and G. Pullan. An accelerated 3D Navier–Stokes solver for flows in turbomachines. *Journal of Turbomachinery*, 133(2):021025, 2011.

B. Brock, A. Belt, J. J. Billings, and M. Guidry. Explicit integration with GPU acceleration for large kinetic networks. *Journal of Computational Physics*, 302: 591–602, 2015.

U. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.

C. Cecka, A. J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85(5): 640–669, 2011.

A. Cevahir, A. Nukada, and S. Matsuoka. High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. *Computer Science-Research and Development*, 25(1-2):83–91, 2010.

V. J. Challis, A. P. Roberts, and J. F. Grotowski. High resolution topology optimization using Graphics Processing Units (GPUs). *Structural and Multidisciplinary Optimization*, 49(2):315–325, 2014.

L. Chang and W. H. Wen-mei. A guide for implementing tridiagonal solvers on GPUs. In *Numerical Computations with GPUs*, pages 29–44. Springer, 2014.

L. Chang, J. A. Stratton, H. Kim, and W. H. Wen-mei. A scalable, numerically stable, high-performance tridiagonal solver using GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 27. IEEE Computer Society Press, 2012.

S. Chen and G. D. Doolen. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, 1998.

J. Cheng, M. Grossman, and T. McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

E. Chow and A. Patel. Fine-grained parallel incomplete LU factorization. *SIAM Journal on Scientific Computing*, 37(2):C169–C193, 2015.

E. Chow, H. Anzt, and J. Dongarra. Asynchronous iterative algorithm for computing incomplete factorizations on GPUs. In *International Conference on High Performance Computing*, pages 1–16. Springer, 2015.

S. Christgau, J. Spazier, B. Schnor, M. Hammitzsch, A. Babeyko, and J. Waechter. A comparison of cuda and openacc: accelerating the tsunami simulation easywave. In *Architecture of Computing Systems (ARCS), 2014 Workshop Proceedings*, pages 1–5. VDE, 2014.

M. Czapiński. An effective parallel multistart tabu search for quadratic assignment problem on CUDA platform. *Journal of Parallel and Distributed Computing*, 73 (11):1461–1468, 2013.

J.P. D'Amato and M. Vénere. A CPU-GPU framework for optimizing the quality of large meshes. *Journal of Parallel and Distributed Computing*, 73(8):1127–1134, 2013.

T. A. Davis. *Direct methods for sparse linear systems*, volume 2. Siam, Philadelphia, PA, 2006.

N. Delbosc. Personal communication, 2014.

Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, et al. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2011.

J. Dongarra. With extreme scale computing the rules have changed. In *International Congress on Mathematical Software*, pages 3–6. Springer, 2016.

L. S. Duarte, W. Celes, A. Pereira, I. Menezes, and G. H. Paulino. Polytop++: an efficient alternative for serial and parallel topology optimization on CPUs & GPUs. *Structural and Multidisciplinary Optimization*, 52(5):845–859, 2015.

L. C. Dutto. The effect of ordering on preconditioned GMRES algorithm, for solving the compressible Navier-Stokes equations. *International Journal for Numerical Methods in Engineering*, 36(3):457–497, 1993.

E. Elsen, P. LeGresley, and E. Darve. Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics*, 227(24):10148–10161, 2008.

H. A. Eschenauer, V. V. Kobelev, and A. Schumacher. Bubble method for topology and shape optimization of structures. *Structural Optimization*, 8(1):42–51, 1994.

J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*, pages 216–225. IEEE, 2011.

R. Farber. *CUDA application design and development*. Elsevier, 2011.

C. M. Fonseca and P. J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 3(1):1–16, 1995.

L. FOTTNER. *Test cases for computation of internal flows in aero engine components*. Number AGARD AR 275. Defense Technical Information Center, 1990.

A. Frommer and D. B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123(1):201–216, 2000.

L. Fu, Z. Gao, K. Xu, and F. Xu. A multi-block viscous flow solver based on GPU parallel methodology. *Computers & Fluids*, 95:19–39, 2014.

A. H. Gebremedhin, F. Manne, and A. Pothen. What color is your jacobian? graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.

S. Georgescu, P. Chow, and H. Okuda. GPU acceleration for FEM-based structural analysis. *Archives of Computational Methods in Engineering*, 20(2):111–121, 2013.

M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, and S. Turek. Efficient finite element geometric multigrid solvers for unstructured grids on GPUs. In *Second Int. Conf. on Parallel, Distributed, Grid and Cloud Computing for Engineering*, page 22, 2011.

M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, and S. Turek. Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses. *Computers & Fluids*, 80:327–332, 2013.

F. Glover. Tabu search-part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

D. Göddeke. *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*. PhD thesis, Technische Universität Dortmund, Fakultät für Mathematik, May 2010.

S. K. Godunov. A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics. *Matematicheskii Sbornik*, 89(3):271–306, 1959.

M. J. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, 1997.

A. Harten, P. D. Lax, and B. Van Leer. On upstream differencing and Godunov-type schemes for hyperbolic conservation laws. In *Upwind and High-Resolution Schemes*, pages 53–79. Springer, 1997.

S. Hartmann, J. Duintjer Tebbens, K. J. Quint, and A. Meister. Iterative solvers within sequences of large linear systems in non-linear structural mechanics. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 89(9):711–728, 2009.

A. Hartstein, V. Srinivasan, T. Puzak, and P. Emma. On the nature of cache miss behavior: Is it 2. *The Journal of Instruction-Level Parallelism*, 10:1–22, 2008.

X. He and L. Luo. Lattice Boltzmann model for the incompressible Navier–Stokes equation. *Journal of Statistical Physics*, 88(3):927–944, 1997.

D. Herrero, J. Martínez, and P. Martí. An implementation of level set based topology optimization using GPU. In *10th World Congress on Structural and Multidisciplinary Optimization, Orlando, Florida, USA*, pages 1–10, 2013.

C. Hirsch. *Numerical computation of internal and external flows: The fundamentals of computational fluid dynamics*. Butterworth-Heinemann, 2007.

S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

A. Jameson. Solution of the Euler equations for two dimensional transonic flow by a multigrid method. *Applied Mathematics and Computation*, 13(3-4):327–355, 1983.

A. Jameson, W. Schmidt, and E. Turkel. Numerical solutions of the Euler equations by finite volume methods using Runge-Kutta time-stepping schemes. *AIAA paper*, 1259:1981, 1981.

P. Jarzebski, K. Wisniewski, and R.L. Taylor. On parallelization of the loop over elements in FEAP. *Computational Mechanics*, 56(1):77–86, 2015.

J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

Z. Johan and T. Hughes. A globally convergent matrix-free algorithm for implicit time-marching schemes arising in finite element analysis in fluids. *Computer Methods in Applied Mechanics and Engineering*, 87(2):281–304, 1991.

D. R. Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of Global Optimization*, 21(4):345–383, 2001.

K. A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan Ann Arbor, 1975.

I.C. Kampolis, X.S. Trompoukis, V.G. Asouti, and K.C. Giannakoglou. CFD-based analysis and two-level aerodynamic optimization on Graphics Processing Units. *Computer Methods in Applied Mechanics and Engineering*, 199(9):712–722, 2010.

K. I. Karantasis, E. D. Polychronopoulos, and J. A. Ekaterinaris. High order accurate simulation of compressible flows on GPU clusters over software distributed shared memory. *Computers & Fluids*, 93:18–29, 2014.

K. Karimi, N. G. Dickson, and F. Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, 2010.

G. Karypis and V. Kumar. hMETIS 1.5: A hypergraph partitioning package. Technical report, Technical report, Department of Computer Science, University of Minnesota, 1998. Available on the WWW at URL http://www. cs. umn. edu/metis, 1998.

C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. RaleighN. C.: NorthCarolinaStateUniversity, 1995.

D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach, 2nd Edition*. Newnes, 2013.

P. Krömer, J. Platoš, and V. Snášel. Nature-inspired meta-heuristics on modern GPUs: state of the art and brief survey of selected algorithms. *International Journal of Parallel Programming*, 42(5):681–709, 2014.

W. B. Langdon. Graphics Processing Units and genetic programming: an overview. *Soft Computing*, 15(8):1657–1669, 2011.

J. J. Lee, S. P. Lukachko, I. A. Waitz, and A. Schafer. Historical and future trends in aircraft performance, cost, and emissions. *Annual Review of Energy and the Environment*, 26(1):167–200, 2001.

V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, 38(3):451–460, 2010.

M. Lefebvre, P. Guillen, J-M Le Gouez, and C. Basdevant. Optimizing 2D and 3D structured Euler CFD solvers on graphical processing units. *Computers & Fluids*, 70:136–147, 2012.

K. Li, W. Yang, and K. Li. Performance analysis and optimization for SpMV on GPU using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):196–205, 2015.

R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

F. Liu and A. Jameson. Multigrid Navier-Stokes calculations for three-dimensional cascades. *AIAA Journal*, 31(10):1785–1791, 1993.

D. Lukarski. *Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms*. PhD thesis, Georgia Institute of Technology, 2012.

M. Lukash, K. Rupp, and S. Selberherr. Sparse approximate inverse preconditioners for iterative solvers on GPUs. In *Proceedings of the 2012 Symposium on High Performance Computing*, page 13. Society for Computer Simulation International, 2012.

L. Luo, J. R. Edwards, H. Luo, and F. Mueller. A fine-grained block ILU scheme on regular structures for GPGPUs. *Computers & Fluids*, 119:149–161, 2015.

A. Mahdavi, R. Balaji, M. Frecker, and E.M. Mockensturm. Topology optimization of 2D continua for minimum compliance using parallel computing. *Structural and Multidisciplinary Optimization*, 32(2):121–132, 2006.

J. R.R.A. Martins and A. B. Lambe. Multidisciplinary design optimization: a survey of architectures. *AIAA Journal*, 51(9):2049–2075, 2013.

J. Michálek, M. Monaldi, and T. Arts. Aerodynamic performance of a very high lift low pressure turbine airfoil (T106C) at low Reynolds and high Mach number with effect of free stream turbulence intensity. *Journal of Turbomachinery*, 134 (6):061009, 2012.

P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd workshop on general purpose processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

V. Minden, B. Smith, and M. G. Knepley. Preliminary implementation of PETSc using GPUs. In *GPU Solutions to Multi-scale Problems in Science and Engineering*, pages 131–140. Springer, 2013.

G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

G. R. Mudalige, M. B. Giles, C. Bertolli, and P. H.J. Kelly. Predictive modeling and analysis of OP2 on distributed memory GPU clusters. *ACM SIGMETRICS Performance Evaluation Review*, 40(2):61–67, 2012.

B. Müller and A. Rizzi. *Runge-Kutta finite-volume simulation of laminar transonic flow over a NACA 0012 airfoil using the Navier-Stokes equations*. FFA TN / FFA, Flygtekniska Försöksanstalten, the Aeronautical Research Institute of Sweden. Aeronautical Research Institute of Sweden, 1986.

L. Mussi, F. Daolio, and S. Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. *Information Sciences*, 181 (20):4642–4657, 2011.

M. Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. Technical Report NVR-2011, NVIDIA Corp., Westford, MA, USA,, Westford, MA, USA,, 2011.

M. Naumov, P. Castonguay, and J. Cohen. Parallel graph coloring with applications to the incomplete-LU factorization on the GPU. Technical report, Nvidia Technical Report NVR-2015-001, Nvidia Corp., Santa Clara, CA, 2015.

K. E. Niemeyer and C. Sung. Accelerating moderately stiff chemical kinetics in reactive-flow simulations using GPUs. *Journal of Computational Physics*, 256: 854–871, 2014a.

K. E. Niemeyer and C. Sung. Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *The Journal of Supercomputing*, 67 (2):528–564, 2014b.

Fine NUMECA. Turbo.

CUDA NVIDIA. GPU occupancy calculator, 2017. `developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

ARB OpenMP. OpenMP application program interface version 4.0, 2013.

S. Osher and R. P. Fedkiw. Level set methods: an overview and some recent results. *Journal of Computational physics*, 169(2):463–502, 2001.

S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

PARALUTION Labs. PARALUTION v.1.1.0, 2016. `http://www.paralution.com/`.

D. A. Patterson and J. L. Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

E. Polizzi and A. H. Sameh. A parallel hybrid banded system solver: the SPIKE algorithm. *Parallel Computing*, 32(2):177–194, 2006.

S. B. Pope. *Turbulent flows*. IOP Publishing, 2001.

I. Reguly and M. Giles. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12. IEEE, 2012.

I. Reguly and M. Giles. Finite element algorithms and data structures on graphical processing units. *International Journal of Parallel Programming*, 43(2):203–239, 2015.

P. R. Rinaldi, E.A. Dari, M. J. Vénere, and A. Clausse. A Lattice-Boltzmann solver for 3D fluid simulation on GPU. *Simulation Modelling Practice and Theory*, 25: 163–171, 2012.

A. Rizzi and M. Inouye. Time-split finite-volume method for three-dimensional blunt-body flow. *AIAA journal*, 11(11):1478–1485, 1973.

P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43(2):357–372, 1981.

P.L. Roe and J. Pike. Efficient construction and utilization of approximate Riemann solutions. In *Proc. of the sixth international symposium on Computing methods in applied sciences and engineering, VI*, pages 499–518, Versailles, France, 1985. North-Holland Publishing Co.

A. J. Rueda, J. M. Noguera, and A. Luque. A comparison of native GPU computing versus OpenACC for implementing flow-routing algorithms in hydrological applications. *Computers & Geosciences*, 87:91–100, 2016.

K. Rupp. PETSc on GPUs and MIC: Current status and future directions. URL `http://www.mcs.anl.gov/petsc/meetings/2015/conference/Rupp_K.pdf`. Talk: Workshop: Celebrating 20 Years of Computational Science with PETSc Tutorial and Conference, Argonne National Laboratory, IL, USA,2015-06-15.

K. Rupp. Viennacl. http://viennacl.sourceforge.net, 2017.

Y. Saad. *Iterative methods for sparse linear systems*. Siam, Philadelphia, PA, 2003.

Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

Y. Saad and H. A. Van Der Vorst. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1):1–33, 2000.

J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn. Design and analysis of computer experiments. *Statistical Science*, 4(4):409–423, 1989.

J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

S. Schmidt and V. Schulz. A 2589 line topology optimization code written for the graphics card. *Computing and Visualization in Science*, 14(6):249–256, 2011.

C. P. Stone, E. Duque, Y. Zhang, D. Car, J. D. Owens, and R. L. Davis. GPGPU parallel algorithms for structured-grid CFD codes. In *Proceedings of the 20th AIAA Computational Fluid Dynamics Conference*, volume 3221, 2011.

R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11 (4):341–359, 1997.

E. Strohmaier, J. Dongarra, H. Simon, M. Meuer, and H. Meuer. Top500 list, november 2016, 2016. URL `https://www.top500.org/lists/2016/11/`.

K. Svanberg. The method of moving asymptotes - a new method for structural optimization. *International Journal for Numerical Methods in Engineering*, 24 (2):359–373, 1987.

R.C. Swanson, E. Turkel, and C. Rossow. Convergence acceleration of Runge–Kutta schemes for solving the Navier–Stokes equations. *Journal of Computational Physics*, 224(1):365–388, 2007.

E. Taillard, N. Melab, and E. Talbi. Parallelization strategies for hybrid metaheuristics using a single GPU and multi-core resources. In *International Conference on Parallel Problem Solving from Nature*, pages 368–377. Springer, 2012.

E. Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

E. Talbi. *Metaheuristics on Graphics Processing Units*. Wiley Publishing, 2014.

J. Tebbens and M. Tuma. Efficient preconditioning of sequences of nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 29(5):1918–1941, 2007.

D. Thevenin and G. Janiga, editors. *Optimization and Computational Fluid Dynamics*. Springer, 2008.

L.H. Thomas. Elliptic problems in linear differential equations over a network. Technical report, Watson Science Computer Laboratory Report; Columbia University, New York, NY, USA, 1949.

D. J. Toal. A study into the potential of GPUs for the efficient construction and evaluation of Kriging models. *Engineering with Computers*, 32(3):377–404, July 2016.

N. Trost, J. Jiménez, D. Lukarski, and V. Sanchez. Accelerating COBAYA3 on multicore CPU and GPU systems using PARALUTION. *Annals of Nuclear Energy*, 82:252–259, 2015.

RA Van den Braembussche, Olivier Léonard, and Lotvi Nekmouche. Subsonic and transonic blade design by means of analysis codes. *Computational Methods for Aerodynamic Design (Inverse) and Optimization, AGARD CP*, 463, 1990.

B. Van Leer. Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. *Journal of Computational Physics*, 32(1):101–136, 1979.

B. Van Leer, W. Lee, P. L. Roe, K. G. Powell, and C. Tai. Design of optimally smoothing multistage schemes for the euler equations. *Communications in Applied Numerical Methods*, 8(10):761–769, 1992.

T. Van Luong, N. Melab, and E. Talbi. GPU computing for parallel local search metaheuristic algorithms. *IEEE Transactions on Computers*, 62(1):173–185, 2013.

J. Vassberg and A. Jameson. *Introduction to Optimization and Multidisciplinary Design, VKI LS*. Von Karman Inst. for Fluid Dynamics Brussels, 2006.

V. Venkatakrishnan. Preconditioned conjugate gradient methods for the compressible Navier-Stokes equations. *AIAA Journal*, 29:pp. 1092–1110, 1991.

T. Verstraete. CADO: a computer aided design and optimization tool for turbomachinery applications. In *2nd Int. Conf. on Engineering Optimization, Lisbon, Portugal, September*, pages 6–9, 2010.

V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. On the limits of GPU acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, volume 13. USENIX Association, 2010.

E. Wadbro and M. Berggren. Megapixel topology optimization on a Graphics Processing Unit. *SIAM Review*, 51(4):707–721, 2009.

J. Wong, E. Kuhl, and E. Darve. A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems. *International Journal for Numerical Methods in Engineering*, 102(12):1784–1814, 2015.

Z. Xia, Y. Wang, Q. Wang, and C. Mei. GPU parallel strategy for parameterized LSM-based topology optimization using isogeometric analysis. *Structural and Multidisciplinary Optimization*, 2017.

Y. M. Xie and G. P. Steven. A simple evolutionary procedure for structural optimization. *Computers & Structures*, 49(5):885–896, 1993.

S. Xu, D. Radford, M. Meyer, and J-D Müller. Stabilisation of discrete steady adjoint solvers. *Journal of Computational Physics*, 299:175–195, 2015.

H. C. Yee. Upwind and symmetric shock-capturing schemes. Technical Report NASA-TM-89464, NASA Ames Research Center, Moffett Field, CA, United States, 1987.

L. Yin and W. Yang. Optimality criteria method for topology optimization under multiple constraints. *Computers & Structures*, 79(20):1839–1850, 2001.

T. Zegard and G. H. Paulino. Toward GPU accelerated topology optimization on unstructured meshes. *Structural and Multidisciplinary Optimization*, 48(3):473–485, 2013.

Y. Zhang, J. Cohen, and J. D. Owens. Fast tridiagonal solvers on the GPU. *ACM Sigplan Notices*, 45(5):127–136, 2010.

Ye Zhao. Lattice Boltzmann based PDE solver on the GPU. *The Visual Computer*, 24(5):323–333, May 2008.