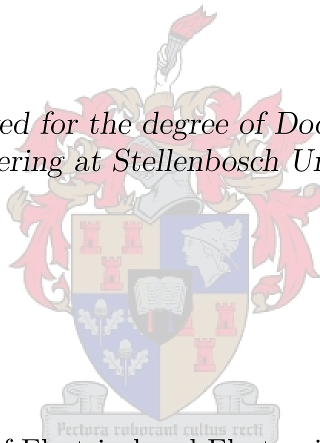
GPU Acceleration of Matrix-based Methods in Computational Electromagnetics

by

Evan Lezar



Dissertation approved for the degree of Doctor of Philosophy in Engineering at Stellenbosch University

Department of Electrical and Electronic Engineering,
Stellenbosch University,
Private Bag X1, 7602 Matieland, South Africa.

Promoter: Prof. D.B. Davidson
Department of Electrical and Electronic Engineering
Stellenbosch University

March 2011

Declaration

By submitting this dissertation electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature: ....

E. Lezar

Date: February 23, 2011

Copyright © 2011 Stellenbosch University
All rights reserved.

Abstract

This work considers the acceleration of matrix-based computational electromagnetic (CEM) techniques using graphics processing units (GPUs). These massively parallel processors have gained much support since late 2006, with software tools such as CUDA and OpenCL greatly simplifying the process of harnessing the computational power of these devices. As with any advances in computation, the use of these devices enables the modelling of more complex problems, which in turn should give rise to better solutions to a number of global challenges faced at present.

For the purpose of this dissertation, CUDA is used in an investigation of the acceleration of two methods in CEM that are used to tackle a variety of problems. The first of these is the Method of Moments (MOM) which is typically used to model radiation and scattering problems, with the latter begin considered here. For the CUDA acceleration of the MOM presented here, the assembly and subsequent solution of the matrix equation associated with the method are considered. This is done for both single and double precision floating point matrices.

For the solution of the matrix equation, general dense linear algebra techniques are used, which allow for the use of a vast expanse of existing knowledge on the subject. This also means that implementations developed here along with the results presented are immediately applicable to the same wide array of applications where these methods are employed.

Both the assembly and solution of the matrix equation implementations presented result in significant speedups over multi-core CPU implementations, with speedups of up to 300× and 10×, respectively, being measured. The implementations presented also overcome one of the major limitations in the use of GPUs as accelerators – that of limited memory capacity – with problems up to 16 times larger than would normally be possible being solved.

The second matrix-based technique considered is the Finite Element Method (FEM), which allows for the accurate modelling of complex geometric structures including non-uniform dielectric and magnetic properties of materials, and is particularly well suited to handling bounded structures such as waveguide. In this work the CUDA acceleration of the cutoff and dispersion analysis of three waveguide configurations is presented. The modelling of these problems using an open-source software package, FEniCS, is also discussed.

Once again, the problem can be approached from a linear algebra perspective, with the formulation in this case resulting in a generalised eigenvalue (GEV) problem. For the problems considered, a total solution speedup of up to 7× is measured for the solution of the generalised eigenvalue problem, with up to 22× being attained for the solution of the standard eigenvalue problem that forms part of the GEV problem.

Opsomming

In hierdie werkstuk word die versnelling van matriksmetodes in numeriese elektromagnetika (NEM) deur die gebruik van grafiese verwerkingseenhede (GVEe) oorweeg. Die gebruik van hierdie verwerkingseenhede is aansienlik vergemaklik in 2006 deur sagteware pakette soos CUDA en OpenCL. Hierdie toestelle, soos ander verbeterings in verwerkings vermoë, maak dit moontlik om meer komplekse probleme op te los. Hierdie stel wetenskaplikes weer in staat om globale uitdagins beter aan te pak.

In hierdie proefskrif word CUDA gebruik om ondersoek in te stel na die versnelling van twee metodes in NEM, naamlik die Moment Metode (MOM) en die Eindige Element Metode (EEM). Die MOM word tipies gebruik om stralings- en weerkaatsingsprobleme op te los. Hier word slegs na die weerkaatsingsprobleme gekyk. CUDA word gebruik om die opstel van die MOM matriks en ook die daaropvolgende oplossing van die matriksvergelyking wat met die metode gepaard gaan te bespoedig.

Algemene digte lineêre algebra tegnieke word benut om die matriksvergelykings op te los. Dit stel die magdom bestaande kennis in die vagebied beskikbaar vir die oplossing, en gee ook aanleiding daartoe dat enige implementasies wat ontwikkel word en resultate wat verkry word ook betrekking het tot 'n wye verskeidenheid probleme wat dié lineêre algebra metodes gebruik.

Daar is gevind dat beide die opstelling van die matriks en die oplossing van die matriksvergelyking aansienlik vinniger is as veelverwerker SVE implementasies. 'n Verselling van tot 300× en 10× onderkeidelik is gemeet vir die opstel en oplos fases. Die hoeveelheid geheue beskikbaar tot die GVE is een van die belangrike beperkinge vir die gebruik van GVEe vir groot probleme. Hierdie beperking word hierin oorkom en probleme wat selfs 16 keer grootter is as die GVE se beskikbare geheue word geakkommodeer en suksesvol opgelos.

Die Eindige Element Metode word op sy beurd gebruik om komplekse geometrieë as ook nie-uniforme materiaaleienskappe te modelleer. Die EEM is ook baie geskik om begrensde strukture soos golfgeleiers te hanteer. Hier word CUDA gebruik of om die afsny- en dispersieanalise van drie golfleierkonfigurasies te versnel. Die implementasie van hierdie probleme word gedoen deur 'n versameling oopbronkode wat bekend staan as FEniCS, wat ook hierin bespreek word.

Die probleme wat ontstaan in die EEM kan weereens vanaf 'n lineêre algebra uitganspunt benader word. In hierdie geval lei die formulering tot 'n algemene eiewaardeprobleem. Vir die golfleier probleme wat ondersoek word is gevind dat die algemene eiewaardeprobleem met tot 7× versnel word. Die standaard eiewaardeprobleem wat 'n stap is in die oplossing van die algemene eiewaardeprobleem is met tot 22× versnel.

Acknowledgements

I would like to thank the following people, who all played some role in making this research possible:

Ingrid for everything that she has done for me, especially in the last few months when pressure was at its highest.

Prof. D.B. Davidson for the guidance through the years.

Renier Marchand for his proofreading, useful comments, and ensuring that this manuscript was handed in on time.

Mary Lancaster for her proofreading and caramel brownies.

My family for there continued support.

My in-laws for providing me with board and lodging while in Stellenbosch.

Danie Ludick for useful discussions and code samples.

Braam and Delita Otto, Paul van der Merwe, and Josh Swart for their continued friendship and the odd cup of coffee or tea.

André Young for his countless lifts to and from the airport and insightful discussions.

Prof. Albert Groenwold for his willingness to listen and excitement about the results that really kept me motivated.

Ulrich Jakobus for his discussions on the MOM and LU decomposition implementations.

Willem Burger for his help in debugging many of the implementations.

Heine de Jager for his assistance in the administration of the new GPU-enabled node at the university's HPC facility.

Kevin Colville and Sebastian Wyngaard for interesting discussions on HPC in general.

The NRF, CHPC, and EMSS-SA (Pty) Ltd for providing financial support for this research in the form of bursaries, contract employment, or funding for hardware used for testing.

The CEMAGG group for the great group lunches.

The occupants of E212 for putting up with my tea and coffee habits.

The residents of Farside for providing a little life outside the office.

and to anyone that I have forgotten. Thank you too.

Contents

Declaration i

Abstract ii

Opsomming iii

Acknowledgements iv

Contents v

Nomenclature viii

1 Introduction 1
- 1.1 Research objectives . . . 2
- 1.2 Contributions . . . 3
- 1.3 Chapter summary . . . 4

2 General purpose GPU computing 6
- 2.1 An overview of parallel computing . . . 6
- 2.2 History of GPU computing . . . 8
  - 2.2.1 History of NVIDIA CUDA . . . 9
  - 2.2.2 History of the ATI Stream SDK . . . 10
- 2.3 NVIDIA CUDA . . . 11
  - 2.3.1 Programming model . . . 11
  - 2.3.2 Hardware implementation . . . 17
  - 2.3.3 Mapping software to hardware . . . 20
  - 2.3.4 Software ecosystem . . . 21
- 2.4 ATI Stream SDK . . . 23
  - 2.4.1 Hardware . . . 23
  - 2.4.2 Software . . . 25
- 2.5 OpenCL . . . 26
  - 2.5.1 Platform model . . . 26
  - 2.5.2 Execution model . . . 27
  - 2.5.3 Memory model . . . 28
- 2.6 Outlook for GPGPU computing . . . 28
- 2.7 Conclusion . . . 29

3 GPU-accelerated dense NLA 30
- 3.1 Dense numerical linear algebra . . . 31
  - 3.1.1 Optimised implementations . . . 34
- 3.2 A note on CUDA-based dense numerical linear algebra implementations . . . 36

3.3 Overcoming GPU memory limitations for dense LU decomposition 37
3.3.1 Left-looking LU decomposition 39
3.3.2 Adapting for GPU acceleration 40
3.3.3 The MAGMA-panel-based hybrid 42
3.3.4 Implementation analysis 44
3.4 Benchmarking 45
3.4.1 The test platforms 46
3.4.2 Results 46
3.5 Conclusion 56

4 MOM scattering analysis 58
4.1 Related work 59
4.2 Problem overview 60
4.2.1 Monostatic scattering 61
4.2.2 The Method of Moments 62
4.3 The implementation of an accelerated MOM solution process 66
4.3.1 The development process 67
4.3.2 The matrix assembly computational process 68
4.3.3 CPU and CUDA co-development 68
4.3.4 Parallel CPU implementation 73
4.3.5 Parallel CUDA implementation 74
4.3.6 Implementation of the other phases 80
4.4 Verification results 81
4.5 Performance results 85
4.5.1 Runtime contributions 85
4.5.2 Speedups 87
4.5.3 Discussion of results 93
4.6 Conclusion 95

5 FEM waveguide analysis 97
5.1 Related work 98
5.2 FEM formulation for cutoff and dispersion analysis 98
5.2.1 Cutoff analysis 100
5.2.2 Dispersion analysis 101
5.2.3 The generalised eigenvalue problem 102
5.3 Implementation 104
5.3.1 FEniCS 104
5.3.2 Eigensolver implementations 113
5.4 Verification results 117
5.5 Performance results 123
5.5.1 Runtime contribution 123
5.5.2 Speedup 125
5.5.3 Discussion of results 126
5.6 Conclusion 128

6 Conclusion and future work 130
6.1 Research observations 131
6.2 Future work 131

List of References 133

Additional MOM performance results **144**

Additional FEM performance results **154**

Nomenclature

Notation

$\lVert\cdot\rVert$	the Frobenius norm
$[X]$	a matrix
$(X)_{ij}$	the entry at row i and column j of the matrix $[X]$
$\{x\}$	a column vector
$(x)_i$	the ith element in the vector $\{x\}$
$\vec{x}$	a spatial vector
$\hat{x}$	a spatial vector with unit norm

Constants

$c_0 \approx$	$299792458\ \text{m} \cdot \text{s}^{-1}$ (the speed of light in vacuum)
$\pi \approx$	3.14159265

Symbols

$\vec{E}$	electric field
$\vec{E}_{inc}$	incident electric field
$\vec{E}_s$	scattered electric field
ϵ_r	relative permittivity of a medium
f, f_c, f_o	frequency, cutoff frequency, operating frequency
$\vec{f}_n$	the RWG basis function associated with edge the nth edge
G_0	the scalar free-space Green's function
$\gamma = \alpha + j\beta$	propagation constant
Γ_e	a boundary that is an electric wall
Γ_m	a boundary that is a magnetic wall
$\vec{J}$	surface current density
k, k_c, k_o	wavenumber, cutoff wavenumber, operating wavenumber
$\vec{k}$	propagation vector
L_i	the ith scalar Lagrange finite element basis function
μ_r	relative permeability of a medium
$\vec{N}_i$	the ith curl-conforming Nédélec finite element vector basis function of the first kind
∇	the gradient operator
$\nabla\times$	the curl operator
∇_t	the transverse gradient operator

$\nabla_t \times$	the transverse curl operator
ω, ω_c, ω_o	angular frequency, angular cutoff frequency, angular operating frequency
Ω	the domain defined by a waveguide cross-section
Ω_v	the domain defined by the interior of a waveguide
$\vec{r}$	the position vector
r, θ, ϕ	coordinates in the spherical coordinate system
$\hat{r}$, $\hat{\theta}$, $\hat{\phi}$	the unit coordinate vectors in the spherical coordinate system
σ	the real valued shift used to transform a generalised eigenvalue problem to a standard eigenvalue problem
σ_{RCS}	the radar cross-section
TE_{mn}	a transverse electric waveguide mode
TM_{mn}	a transverse magnetic waveguide mode
x, y, z	coordinates in the Cartesian coordinate system
$\hat{x}$, $\hat{y}$, $\hat{z}$	the unit coordinate vectors in the Cartesian coordinate system

Abbreviations and acronyms

ALU	arithmetic and logic unit
API	application programming interface
BVP	boundary-value problem
CEM	computational electromagnetics
CPU	central processing unit
DP	double precision
EFIE	electric field integral equation
FDTD	finite-difference time-domain
FEM	finite element method
FP	floating point
FLOP	floating point operation
FLOPs	floating point operations
FLOPS	floating point operations per second
MFLOPS	10^6 FLOPS
GFLOPS	10^9 FLOPS
TFLOPS	10^{12} FLOPS
PFLOPS	10^{15} FLOPS
GB	gigabyte (1024 MB)
GB/s	gigabyte per second
GEV	generalised eigenvalue problem
GPU	graphics processing unit
GPUs	graphics processing units
GPGPU	general purpose GPU
ISA	instruction set architecture
MB	megabyte $(1024 \cdot 1024$ bytes)
MB/s	megabyte per second

MIMD	multiple-instruction-multiple-data
MOM	method of moments
NLA	numerical linear algebra
OOC	out-of-core
PEC	perfect electrical conductor
RCS	radar cross-section
RWG	Rao-Wilton-Glission
SDK	software development kit
SEV	standard eigenvalue problem
SIMD	single-instruction-multiple-data
SMP	symmetric multiprocessor/multiprocessing
SP	single precision
SPMD	single-program-multiple-data
TE	transverse electric
TM	transverse magnetic

NVIDIA-specific abbreviations

CUDA	Compute Unified Device Architecture
GPC	graphics processing cluster
SFU	special function unit
SIMT	single-instruction-multiple-threads
SM	symmetric multiprocessor
SP	stream processor
TP	thread processor
TPC	thread/texture processing cluster

AMD-specific abbreviations

CD	compute device
CU	compute unit
PE	processing element
SC	stream core
T-PE	thick PE

Chapter 1

Introduction

In the field of computational electromagnetics (CEM), like most scientific or engineering disciplines, there is a constant drive for the solution and understanding of more complex problems. Unfortunately, the solution of these problems comes at an increased cost in terms of computational as well as storage requirements.

Although the computational power at our immediate disposal has increased dramatically with time, much of that improved performance has been due to frequency scaling which was halted just after the start of this century due to power requirements [1]. In its stead, computer system and architecture designers have made ever increasing use of parallelism to keep the growth in performance alive. The power of this parallelism is evident in the performance of the fastest – according to the Top500 list [2] – supercomputers in the world, where the performance has increased by four orders of magnitude over the past 17 years.

It is clear then at this point that the future, and in fact the present, are both highly parallel. One of the consequences of the shift to increasing the number of cores or processors in a computer system, instead of simply scaling the frequency at which they operate, is that software that is not designed to run in parallel no longer gets any faster for nothing – hence the phrase *"The Free Lunch is Over"*, that is the title of [1]. It is then imperative that existing implementations and techniques be revisited and their feasibility for parallel execution considered.

One of the fields where the move towards massive parallelism is most evident is in the field of general purpose GPU (GPGPU) computing. Although this discipline has been around in some form or another since the late 1970s, the release of more powerful hardware as well as a number of high-level languages over the past 10 years has really seen the field move from strength to strength. One of the most notable additions, and the one used in this dissertation, is the CUDA software and hardware architecture released by NVIDIA in November of 2006 [3]. A quick glance at the CUDA Zone [4] gives an indication of the vast array of applications that now make use of this acceleration technology.

Although a number of competing architectures exist, CUDA was one of the first to provide a software environment that was straightforward enough for anybody with a reasonable amount of general programming knowledge to grasp. Furthermore, a number of accelerated versions of common routines in linear algebra and signal processing were made available early on and resulted in out-of-the-box speedups for many applications. It is this software advantage, coupled with incremental increases in hardware capabilities, that has been the main driving force in the high adoption rate of CUDA and lead to the point where three of the top five fastest computers in the world – including the number one – now make use of these accelerators.

In this dissertation two well known matrix-based techniques in the field of computational electromagnetics are considered, namely the Method of Moments (MOM) for scattering problems and the Finite Element Method (FEM) for waveguide cutoff and dispersion analysis. More specifically, the parallelisation and resultant acceleration of these methods using CUDA-capable

GPUs is investigated. The fact that both methods are matrix-based allows for the adaptation and reuse of many of the advancements from GPU-based linear algebra to achieve speedups in some of the phases of the respective solution processes. This also means that improvements made here can be directly applied to a wide range of other applications – as long as the same class of linear algebra routines are used.

For the finite element problems considered, the part of the solution that can be expressed in terms of linear algebra contributes most significantly to the total computational cost. This is not always so for the MOM, and as such, the other phases of the solution process are also considered for acceleration. In this case, there are no libraries available the allow for the rapid development of an accelerated implementation, and instead an implementation must be developed from scratch. This does however allow for a better understanding of the parallel implementation.

It should be noted that although GPUs are an attractive option at present, largely due to the fact that their prices are kept low by the computer gaming industry, in their current implementation they still represent a relatively young technology – especially in the field of general purpose computing. Although there is evidence to the contrary [5, 6], it may be that these architectures will fade away to give rise to something else. Even if this is the case, there is a good chance that whatever replaces them will be even more parallel and any knowledge and experience gained in the porting of existing applications or the development of new code for GPU execution will prove invaluable.

1.1 Research objectives

Apart from the overarching aim of investigating the use of GPU-based acceleration in the field of computational electromagnetics, this research sets out to address a number of other points of interest. It should be noted that only a small sub-set of computational electromagnetic problems are considered, namely the Method of Moments (MOM) as applied to scattering analysis and the Finite Element Method (FEM) for the analysis of bounded waveguide structures.

For any GPU-based implementation, a positive result would be a speedup of more than 1× relative to a CPU implementation of a given method. Although this is the case here, it is not the only factor considered. A number of comparative benchmarks across multiple platforms allow for the investigation of the relative performance of GPU and CPU implementations on a single platform, as well as to the relative strengths of the different machines with and without GPU acceleration.

This benchmarking strategy allows for two issues to be addressed. The first of these is whether a GPU-accelerated implementation can enable high-end compute node performance on more standard systems. Secondly, the viability of GPUs as an upgrade path for existing systems is considered, allowing for an increase in the useful – and performance-competitive – lifespan of older systems at the fraction of the cost of a full replacements.

One of the factors limiting the usefulness of GPUs in general purpose computing is the amount of memory available on such devices. The implementations presented here attempt – wherever possible – to overcome this limitation. At this point it should be noted that this research is limited to investigating the performance of single systems each with a single GPU in use. Since the distributed use-case is not considered here, an implementation that aims to overcome the memory limitations of a GPU is considered to be successful if it is able to deal with problem sizes comparable to those that can be accommodated in the primary memory installed in the system.

A second factor impeding the widespread adoption of GPUs as accelerators is the (perceived) steepness of the learning curve associated with their programming. Although this may have been true in the past, where GPUs were bent to the will of a programmer using a number of graphics programming tricks, the development of compute-specific languages such as CUDA and OpenCL

has greatly improved the ease with which these devices can be programmed. The development of a number of high-level libraries that are widely used has also lowered the barrier to entry and increased the adoption rate.

The work presented here aims to address this second hurdle in a number of ways. Firstly, existing, freely available libraries are used. This route offers the most plug-and-play approach to first-time developers or people wishing to experiment with various implementations. An attempt is also made to keep much of the discussion and implementations as general as possible so as to ensure that the results are of interest to a wide range of applications.

An important aspect that is also considered in the work is that of CPU and GPU co-development. This stems from the realisation that no – or at least very few – GPU-based implementations exist alone. Generally they are either ported from an existing CPU implementation, or a CPU implementation is also required for compatibility reasons. The time and effort involved in developing and maintaining two versions of the same code can be prohibitive, especially in a commercial setting or any application where requirements and features change often.

1.2 Contributions

This work represents a number of contributions that result from the consideration of the research objectives discussed in Section 1.1. When considered in its entirety, an investigation into the use of GPU-acceleration in computational electromagnetic matrix-based methods is presented. Apart from addressing the overarching aim of this dissertation, this can be considered a useful contribution in it own right, as it summarises the work presented in a number of sources dealing with the GPU acceleration of a single CEM method. That is, either the Finite Element Method or the Method of Moments.

The GPU acceleration of the CEM methods considered uses a set of typical problems from each method as the starting point for the implementation as well as the results presented. This use of simple problems and a tutorial-like approach to many of the implementation discussions means that this research should be useful to anyone wishing to develop their own GPU-enabled versions of existing methods.

Since both the MOM and the FEM are matrix-based, they allow for the use of existing accelerated linear algebra routines such as CUBLAS [7] and MAGMA [8]. In the case of the LU decomposition, the MAGMA implementation is extended to allow for the solutions of problems that are larger than can be accommodated by the memory installed on the GPU. This is in keeping with one of the aims of this research with the accelerated GPU implementation able to outperform a multi-core platform-optimised CPU implementation even in cases where the amount of memory required is 16 times greater than the amount available to the GPU. This memory-resilient LU decomposition is also used as part of the Method of Moments solution process. The research into the LU decomposition led to a publication in *Electronics Letters* [9].

In the case of the Method of Moments, a seminal work in the field – the 1982 paper by Rao, Wilton, and Glisson [10] – is considered as a starting point for the discussion of the GPU acceleration of the MOM solution process. When compared to existing GPU-based MOM implementations, the work presented here is novel in a number of ways. These include the implementation of the entire solution process in both native double precision – as supported by the latest generation of GPUs – as well as the direct application to the formulation and problems presented in [10] resulting in the publication of [11] and [12]. The MOM process presented here is also implemented in such a way so as to overcome the restrictions on the problem size due to limited GPU memory.

For the Finite Element Method, the GPU acceleration of another set of canonical problems is considered. That of cutoff and dispersion analysis of waveguide structures [13]. The acceleration

of these problems is approached from a linear algebra perspective, and focuses on the solution of the eigenvalue problems that results as part of the FEM solution process. The GPU-accelerated (using both CUBLAS and MAGMA) ARPACK [14] implementation that results is to the author's knowledge the first to deal with generalised eigenvalue problems, and more specifically those that results from the FEM analysis of waveguide structures. The work presented is a continuation of that published in [15], and although only relatively small dense matrix problems are considered, the findings lay excellent groundwork for continued research.

Another contribution of both the MOM and the FEM implementations is in addressing the research objective of CPU and CUDA co-development. In these implementations CPU- and GPU-accelerated implementations are developed in parallel, with a methodology followed in each case that allows for the sharing of much of the code – especially in the case of the MOM implementation – between the two versions. This sharing of code greatly simplifies the addition of new features and the unavoidable debugging that will be required to implement them correctly.

A final general contribution is a set of comparative benchmarks over three (two in the case of the FEM results) different systems with different CPU and GPU capabilities. Of these systems, one is significantly older than the other two with a CPU performance that is lower by at least a factor two. The results presented show that not only does the addition of a GPU to the older system improve its achievable performance, but that the improved performance is such that the old machine is able to compete with the (unaccelerated) newer systems for most of the problems considered. The exception here is that the most expensive of the new systems still offers better performance for double precision computation, although it should be added that the cost of the new system is about 25 times higher than the GPU used to upgrade the slower system. One of the major contributions of these comparative performance results is that they allow for better informed upgrade or replacement decisions, which can result not only in a monetary saving, but also in the reduction of waste.

Although not mentioned in Section 1.1, one of the objectives for the FEM component of this research is the identification of a suitable open-source software environment to be used for finite element modelling in CEM. The software chosen, FEniCS, offers a set of powerful tools such as the ability to describe the finite element formulations in a very high-level language. The FEM implementations presented here use FEniCS and represent its first application to electromagnetic waveguide problems. The examples considered here are also to be included as part of the set of demos available at [16], thus contributing directly to the open-source community.

1.3 Chapter summary

The main body of work is divided into six chapters, with this chapter constituting the fist and providing a general overview of the dissertation. A number of points are addressed including a general motivation, as well as discussing the research aims and contributions of the work.

Chapter 2 provides an introduction to general purpose GPU computing and provides a history of the field, as well as a comparison of two current technologies, namely CUDA and OpenCL, of which the former is used for the implementations in this dissertation. The chapter serves to introduce much of the CUDA-specific terminology used in the rest of the dissertation, and provides a few simple examples to explain the workings of the architecture.

In Chapter 3 the application of CUDA to acceleration of dense numerical linear algebra (NLA) is considered. This includes a quick discussion on NLA in general including the BLAS and LAPACK libraries. As far as the CUDA aspects are concerned, libraries such as CULA Tools, MAGMA, and CUBLAS are discussed with the latter two subsequently used in the implementation that follows. The implementation sees the mapping of traditional out-of-core LU decomposition techniques to the CUDA device-host relationship and result in two versions of the LAPACK-compatible LU decomposition. These are not only faster than multi-core CPU imple-

mentations, but are ,in addition, not limited by the amount of memory available on the GPUs used. The chapter also introduces the benchmark systems used for the performance comparisons in this dissertation, with performance results for the LU decomposition implementations given.

The LU decomposition presented in Chapter 3 is used as part of the Method of Moments solution process which is discussed in Chapter 4. Here, the formulation from [10] is adapted for execution on CUDA GPUs, with a parallel CPU implementation also developed. This allows for the introduction of the CPU and GPU co-development methods, where much of the code is shared between the two implementations. This section also sees a large number of comparative benchmarks obtained for the sample problem considered. Results indicate that the implementation is resilient in terms of the limited amount of memory available on the CUDA devices used.

Chapter 5 continues the discussion of CUDA-accelerated matrix-based methods in CEM with an investigation of the acceleration of the finite element analysis of waveguide structure. Here, both cutoff and dispersion problems are considered. The CUDA acceleration is achieved by using the dense NLA routines supplied by CUBLAS and MAGMA to construct a generalised eigenvalue problem solver that is based on ARPACK. A set of comparative benchmarking results is also presented. In addition to the accelerated eigensolver, the chapter introduces the FEniCS software that is used for the implementation of the finite element formulations discussed. General conclusions and recommendations for future work are presented in Chapter 6, followed by two appendices, Appendix 6.2 and Appendix 6.2, which provide additional results not included as part of the MOM and FEM chapters (Chapter 4 and Chapter 5), respectively.

Chapter 2

General purpose GPU computing

General purpose GPU (GPGPU) computing is the use of graphics processing units to perform computations for which they were not initially designed. Although this practice has been around for some time, its adoption in a wide variety of fields including scientific computing, computational finance, and engineering has of late seen a remarkable increase. Much of this is a result of the very attractive performance promises made by such devices, as well as the improvement in the increasing ease with which they can be employed to perform arbitrary computational tasks.

This chapter aims to present an overview of GPGPU computing so as to provide a background for its application to computational electromagnetic problems in later chapters. Discussion starts with a general overview of parallel computation in Section 2.1, which is followed by the history of general purpose GPU computing in Section 2.2.

Sections 2.3 through 2.5 address CUDA by NVIDIA, the ATI Stream SDK from AMD, and OpenCL. Special attention is given to NVIDIA CUDA with much of the discussion going into great detail, as this is the language of choice for the implementations and results presented in Chapter 3, Chapter 4, and Chapter 5. Many of the concepts illustrated can, however, be applied to other technologies with one-to-one correspondences existing in many cases. The chapter is concluded with a brief discussion on possible future developments in the field of GPU computing.

2.1 An overview of parallel computing

In this section, a brief overview of parallel computing in general is presented. This serves to provide a basis on which the subsequent sections on GPGPU computing build. Furthermore, much of the terminology that is used in this chapter and the remainder of the dissertation is introduced.

When parallel computing is mentioned, one typically envisions rooms filled with a large number of connected computers with computational capabilities far outstripping a regular desktop or notebook computer. Twice a year a list is released where the performance of the top 500 of these supercomputers is showcased [2] and there is often much competition for the coveted top position. The performance – measured in floating-point operations per second (FLOPS) – of the machines on the list has increased exponentially since its inception in June of 1993, with the current (November 2010) number 1 system achieving a performance of 2.57 PFLOPS (thousand trillion floating-point operations per second). This is up from 59.7 GFLOPS (billion floating-point operations per second) in June of 1993 and represents an improvement of more than 43000 times over less than 20 years.

Most of the systems in the current Top500 list are built as clusters of thousands of compute nodes and thus follow the distributed computing model [17]. In this model each of the nodes has its own local memory and software running on the system must rely on a message passing protocol such as MPI [18] to ensure that each of the nodes has the data it requires. In such

systems each of the compute nodes is typically a fully-fledged multi-core system with a small number (10–20) of cores. Since each of these cores has access to the memory local to the node, message passing is not required for communication on a single node [17]. In this case, the shared memory parallel model or symmetric multiprocessor (SMP) model can be used, with tools such as OpenMP [19] available to realise such implementations.

It should be noted that although supercomputers such as those on the Top500 list have been making use of multiple processors to achieve performance for some time now, it represents a relatively recent development in the main-stream computing sector. The first consumer-level multi-core processors appeared in the early 2000s as a direct results of the increasing power requirements associated with ramping up the frequency as which the processors operate (which had been one of the main sources of the performance improvements up to that point) [1]. The multi-core processor is now ubiquitous in everyday computing and most desktop machines can be seen as an implementation of the shared memory parallel model. These multi-core nodes or hosts are examples of multiple-instruction-multiple-data (MIMD) machines, that allows for multiple separate instructions to operate on different data in parallel [20].

Just as there are different models for the description of parallel computers, whether they be the distributed or massively parallel processor systems of the Top500 list or a standard desktop computer, there are a number of models that define the way in which parallelisation can be achieved. These are bit-level, instruction-level, task, and data parallelism [17]. The first two of these are generally closely tied to hardware and are only discussed briefly before addressing task and data parallelism. Bit-level parallelism is leveraged to increase the performance of a processor by increasing the word size (the number of bits that can be operated on at a time) of the processor. This is evident in the increase from 8 bits in early x86 computers to 64 bits in the current x86-64 architecture. Instruction-level parallelism relies on the fact that any program executing on a computer consists of a stream of instructions and that many of the instructions can be reordered and executed in parallel without changing the results of the program.

In contrast to bit-level and instruction-level parallelism, task and data parallelism are more closely related to the algorithm or application being implemented [17] and not the underlying hardware architecture. It should, however, be noted that there do exists architectures (including the GPUs discussed later in this chapter) that are specifically designed for data-parallel execution. An application or algorithm exhibits task parallelism (also called function parallelism) when entirely different calculations are to be performed on the same or different data [17] and these tasks can be performed in parallel. A contrived example of this is an application that needs to compute the sum and the products of lists of numbers. Regardless of whether the numbers are from different or the same lists, these operations (the sum and the product) are independent and can be performed concurrently.

Data parallelism involves performing the same (or similar) operations or functions on individual elements of a large data structure [17]. This type of parallelism is often achieved by examining loops present in an implementation and executing independent portions in parallel. An example would be the addition of the corresponding elements in two arrays, where it is clear that each of the new data elements to be computed are independent and the operations can thus be performed in parallel. Note that the data parallel model as discussed in [17] is a generalisation of the single-instruction-multiple-data (SIMD) model that forms part of *Flynn's taxonomy* [20], and later evolved into the single-program-multiple-data (SPMD) approach. An alternative is the MIMD approach – which can be compared to task parallelism in some cases – already mentioned with reference to general shared-memory and multi-core machines.

Applications usually exhibit both task and data parallelism and often have a number of different levels at which these parallelisations can be implemented [17]. That is to say that a particular algorithm could be implemented as a set of parallel tasks performed per data element (task parallelism within data parallelism) or as a set of separate tasks which are each data parallel

(data parallelism within task parallelism). It should be noted that, although task parallelism is often relatively easy to implement using message passing or a multi-threaded implementation, the amount of parallelism available is usually modest and does not increase much with the size of the problem being considered [17]. Data parallelism on the other hand does grow with the size of the data set and allows for the utilisation of massively parallel processors such as GPUs.

Although this section aims to provide a relatively complete overview of the parallel computing landscape, the scope of the field makes this impossible. An additional source of information is [21], which provides a number of insights including the matching of algorithms to a specific architecture. A field that has not been mentioned at all in this section is that of grid computing. This is addressed with specific reference to computational electromagnetic problems in [22].

2.2 History of GPU computing

Any history of GPGPU computing should at least mention the Ikonas system of 1978 [23]. This system was developed as a programmable raster display system for cockpit instrumentation and was quite large by today's standards. Although other such systems have been developed [24, 25], the rest of the history addresses the use of consumer-level hardware such as the GPUs originally introduced by NVIDIA in 1999 [26].

Due to the inherently parallel nature of graphics rendering – the processing of geometric, texture and lighting data to generate a scene for display on a computer monitor by calculating the required properties of each pixel [27] – and the ever increasing capabilities of GPUs, the scientific community started investigating their use in general purpose computation shortly after their introduction. Initial attempts were based on utilising the rendering pipeline of the GPU directly using OpenGL, which was originally intended for writing 2D and 3D graphics applications. Although this approach did result in some impressive speedups for the time, the process was quite complex and this hindered large scale adoption.

As the hardware developed and became more programmable (predominantly for applying effects in computer visualisations and games), a number of higher level languages were also developed that greatly simplified the programming process. These include shader languages, such as Cg (2002), GLSL (2002), and HLSL (2002), whose primary goal was not general GPU computation, but the description of more complex scenes for use in computer generated imagery.

Later, a number of academic languages were developed that the were specifically targeted to GPGPU applications. These languages typically considered the GPU as a stream processor performing some operation (or kernel) on a stream of input data to produce the desired output data stream. In many cases the same operation is performed on each element of the input stream. This process of applying the same operation to a number of data elements (usually in parallel) is referred to as the single-instruction-multiple-data (SIMD) model, as opposed to the multiple-instruction-multiple-data (MIMD) model typically followed by conventional CPUs. Two such languages that deserve some mention are BrookGPU (2003) and Sh (2003) which have each since been developed into commercial offerings. The Sh language and API was released as PeakStream, soon after which it was purchased by Google with no public development now taking place. BrookGPU became RapidMind and is now owned by Intel. This seemingly forms the core of Ct (C for Throughput Computing), intended as a data-parallel programming language to leverage the power of multi-core processors.

The end of 2006 saw the birth of what can be considered modern general purpose GPU computing, with both NVIDIA and AMD releasing products that offered previously unheard of programmability and promises of computational power [3, 28]. The history of what would become the ATI Stream SDK from AMD and the Compute Unified Device Architecture (CUDA) from NVIDIA is presented briefly, after which CUDA is considered in some detail in Section 2.3. A comparative overview of the ATI Stream SDK is presented in Section 2.4.

A comparative performance timeline for the two technologies is given in Figure 2.1, with data taken from [29] and [30]. As can be seen from the figure, the performance promised by the devices is one of the primary reasons that there has been so much interest in utilising these technologies for general purpose computation. It should be pointed out that there are a number of caveats that should be considered when comparing theoretical peak performance figures such as these. A comparative study of GPU vs CPU performance is provided in [31], and lists a number of these.

One of the indications of the coming of age of general purpose GPU computing was the inclusion of systems using this technology in the twice yearly Top500 list discussed in Section 2.1 [2]. The Tsubame cluster from Tokyo Institute of Technology improved its position to 29$^{\text{th}}$ in the November 2008 list with the inclusion of 170 Tesla S1070 units from NVIDIA. The November 2009 list saw the inclusion in the fifth position of the Tianhe-1 system from National Super Computer Centre in China which used AMD/ATI GPUs to accelerate computation. Where the Tsubame system used Tesla units designed for the high-performance computing segment, the Tianhe-1 used two Radeon 4870 (high-end consumer) devices in each node [2].

The high point in terms of GPU computing and the Top500 list came with the announcement of the November 2010 results which has 10 of the systems using NVIDIA GPUs for acceleration with AMD/ATI devices being used by two systems [2]. The systems using NVIDIA GPUs include three of the top five systems. The Tianhe-1A system at the National Supercomputing Center in Tianjin, China occupies the first place, with the Nebulae system, also from China, in third (down from second in June 2010). The final GPU-enabled system in the top five is the Tsubame 2.0 from the Tokyo Institute of Technology in fifth position.

2.2.1 History of NVIDIA CUDA

Although a number of NVIDIA GPUs had been used for the purpose of GPGPU computation with some success up until that point [32, 33], the real birth of NVIDIA's GPGPU computing

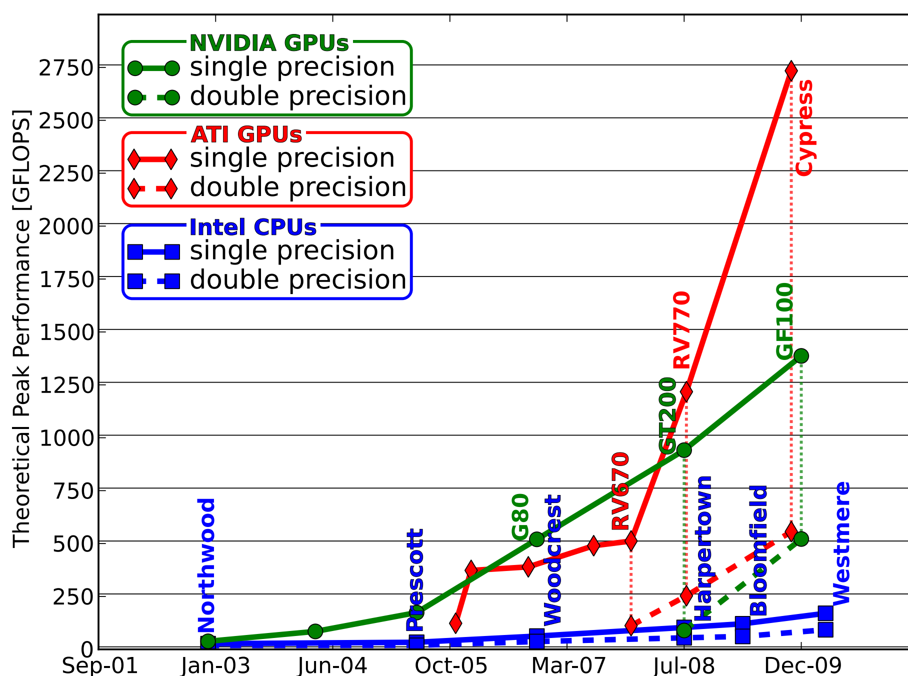


Figure 2.1: Figure showing the performance timelines of NVIDIA GPUs, ATI (AMD) GPUs, and Intel CPUs over the past seven years. Theoretical peak performance curves are shown for both single precision (solid curves) and double precision (dashed curves) and are adapted from [29] and [30].

effort came at the end of 2006 with the release of their G80 architecture [3]. From a graphics processing point of view, this new architecture was an answer to the requirements imposed by the DirectX 10 API – which is to say that all the operations in a graphics pipeline be performed by unified shaders, and not by separate vertex and pixel shaders as was the case in the past [27, 34]. Thus instead of the graphics pipeline containing (possibly hundreds [3] of) sequential pipeline stages, all the required operations are performed by a standardised shader core – hence the term "*unified shader*".

Although the G80 architecture was originally intended for graphics, it was also designed in such a way as to provide an entry into the GPGPU market that, in 2006, was really still in its infancy [3]. The hardware formed part of a larger architecture called the Compute Unified Device Architecture (CUDA), providing a well-defined programming model and software tools to harness the power of the GPUs for general purpose computation.

Since the first public release of the CUDA toolkit at the beginning of 2007, the API has developed steadily, providing more functionality and being consistantly updated to accommodate the capabilities of new iterations of the hardware architecture. These include the addition of double precision floating-point support in the GT200 architecture [35], as well as a unified memory model in the Fermi architecture [26].

Although the original G80 architecture was a graphics architecture with the ability to perform general purpose computation, the interest shown from the high performance computing (HPC) sector was phenomenal and led to the development of the Tesla range of products specifically targeted at HPC. These devices are in essences NVIDIA GPU-based devices with no graphics output capabilities. They also offer more memory than their consumer-level counterparts and thus allow for the processing of larger computational problems.

2.2.2 History of the ATI Stream SDK

At about the same time that NVIDIA released the G80 architecture and CUDA, AMD/ATI released CTM (Close To Metal), which gave programmers relatively low-level control over the graphics processing hardware for general purpose GPU computing [28]. This low-level language later evolved into the Compute Abstraction Layer (CAL) and became the supporting API layer for the high-level ATI Brook+. Their combination was called the ATI Stream SDK [36] – AMD's answer to CUDA.

In addition to advances on the software front, hardware from AMD also evolved rapidly. AMD introduced the first ever GPU device with native double precision support, the FireStream 9710. It was six months before NVIDIA brought the GT200 – their first double precision device – to market. The next FireSteam device, the FireStream 9250 was the first device to offer more than 1 TFLOP (one trillion floating-point operations per second) in peak theoretical single precision performance in a very compact form factor [37].

Soon after the ratification of OpenCL (Open Compute Language) [38] in December 2008, AMD decided to replace ATI Brook+ with OpenCL as the development language for programming AMD devices [39]. One reason for this change may be the hope that the use of an industry standard, multi-platform development language will be able to reduce the lead that NVIDIA has enjoyed in the GPGPU segment. Since multi-core CPUs can also be used as OpenCL compute devices [40], this also means that AMD can offer a unified heterogeneous computing solution [39].

Note that, although ATI was originally a separate company, AMD acquired ATI in mid 2006, only a few months before the release of CTM [28]. Although the ATI brand was retained for some time, it was replaced by AMD in August of 2010. Thus there exist a number of devices that – although developed by AMD – have ATI branding, whereas future devices will bear the AMD moniker. As such AMD and ATI are used interchangeably throughout the rest of this

chapter. The purchase of ATI seems to have come to fruition, with the release of AMD's Fusion platform at the end of 2010 [5]. This platform sees the integration of a GPU – acting as a parallel floating-point accelerator – and a number of x86 cores onto a single processor.

2.3 NVIDIA CUDA

As already mentioned, changes to the rendering requirements brought about by APIs such as DirectX 10 lead to new architectures being developed by hardware vendors such as NVIDIA and ATI. The G80 architecture by NVIDIA saw the implementation of a unified shader design [3]. This design saw the traditional rendering pipeling consisting of various stages being replaced by a number of identical shading cores. These cores are highly programmable, which allows them to perform any of the functions that would normally be executed by special-purpose pipeline stages and include both pixel and vertex shading, previously handled by separate hardware.

This change in architecture also had the advantage that the programmability of the hardware for performing general purpose computational tasks was greatly improved. The Compute Unified Device Architecture (CUDA) from NVIDIA is a general purpose parallel computing architecture and provides a computing model, as well as an instruction set architecture (ISA) to leverage the power of the underlying parallel hardware [29], a CUDA-capable device such as a GPU. Due to the multiple market segments in the GPU industry, it is important that such an architecture is scalable, as it allows for development resources to be shared across an entire line of GPUs. In addition, this lends itself well to scalable implementations in terms of performance - often quite difficult to achieve in a traditional parallel programming environment.

2.3.1 Programming model

Before considering the actual hardware associated with a CUDA device, the programming model is addressed. The reasoning for this is that although future generations of hardware may be more complex and provide advanced capabilities, their support for the programming model is guaranteed. Thus if one can develop code for a current generation, this should run on future hardware with little or no modification.

The CUDA programming model consists of three key abstractions, namely, a hierarchy of thread groups, shared memories, and barrier synchronisation, all of which are exposed to the programmer using a series of programming language extensions. The abstractions allow for the management of fine-grained data parallelism at a thread level, nested within coarse-grained data parallelism and task parallelism (which have both been discussed in Section 2.1). This forces the programmer to consider the problem domain and partition it into coarse sub-problems that can be solved independently (possibly in parallel) with each of these sub-problems also being decomposed to a thread level. This is similar to the approach followed by stream processing [41] and since the coarse blocks of work must be independent by definition, their parallel computation should scale well.

It should be noted that the concept of a thread used here is not the same as a thread in a classical CPU-based multi-threaded sense (as in the case of OpenMP [19]). A CUDA GPU thread is a lightweight entity whose context can be switched with zero overhead [27]. This is made possible by the GigaThread engine by NVIDIA which allows for thousands of threads to be in flight during execution [3, 26, 35]. Where the term thread is used in this chapter, it refers to a CUDA thread and not a CPU thread unless otherwise specified.

In order to further the discussion of the CUDA programming model, consider a simple example of adding the contents of two `float` arrays. The CPU-based C routines `add()` is shown in Listing 2.1 with the array `c` being calculated as the sum of `a` and `b` (for each of the `N` elements in the arrays). This is the same example that serves as an introduction to CUDA in [42].

Listing 2.1: An example for adding two `float` arrays of length `N` implemented in C.

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

CUDA kernel definition

As with stream processing [43], the basic unit where computation can take place in CUDA is known as a kernel. In CUDA terms a kernel is a routine, usually written in C, that is executed on a CUDA-capable device. Listing 2.2 shows the CUDA kernel, `add_kernel()`, for the example discussed and shown in Listing 2.1.

Listing 2.2: A simple example of a CUDA kernel for adding two `float` arrays of length `N`.

```
__global__ void add_kernel ( int N, const float *pdev_a, const float *pdev_b,
    float *pdev_c )
{
  // calculate the data index for this instance of the kernel
  int i = threadIdx.x + bockDim.x * blockIdx.x;

  if ( i < N )
  {
    // calculate the output value for the given data element
    pdev_c[i] = pdev_a[i] + pdev_b[i];
  }
}
```

Listing 2.2 illustrates another important point – the programmability of CUDA-capable devices is provided by a number of C language extensions. This will be discussed in more detail in Section 2.3.4. For now it is sufficient to note that a CUDA kernel is indicated by the `__global__` qualifier and must be of return type `void`.

Also evident in Listing 2.2 is that no explicit iterations are performed over the data elements in the arrays. This is due to the fact that the CUDA environment automatically handles the execution of the same kernel for each data element in the problem domain, with many of the executions occurring concurrently. The kernel invocation will be discussed in more detail later.

CUDA thread organisation

Before continuing the discussion of this kernel example, the data segmentation options provided by CUDA need to be discussed. In terms of segmentation, the finest granularity is that of a single CUDA thread. When a kernel, such as the one given in Listing 2.2, is executed, it is executed for every thread that has been defined.

Although these threads allow a fine-grained access to data, it is impractical to use them to individually address the elements in large datasets - especially if the size of the dataset exceeds the number of available threads. Threads are thus grouped into one-, two- or three-dimensional blocks, each with identical dimensions, providing a coarser level of control over the segmentation of the problem space. The blocks are further arranged in a one- or two-dimensional grid at the highest level. An example of such a partitioning is shown in Figure 2.2. This style of

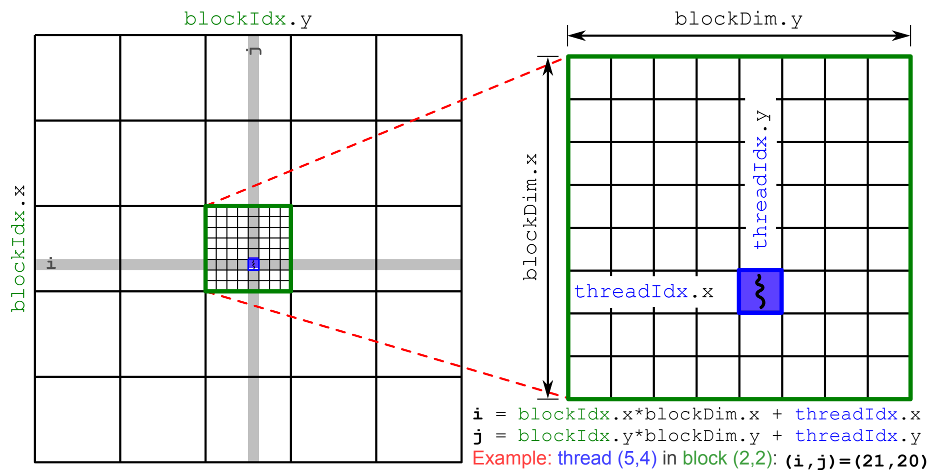


Figure 2.2: Diagram showing the grouping of threads into 8×8 blocks in a 5×5 grid for computation on a CUDA device. Also shown is the calculation of the global index (`(i,j)`) of a thread from local thread and block information. Note that although the blocks and grid shown are square, this is not a prerequisite [29].

organisation lends itself particularly well to partitioning one- or two-dimensional data such as vectors, matrices or images. Furthermore, since the local index (in each dimension) of a thread within a block, as well size and position of the block in the grid can be accessed at runtime, the global index of the thread can easily be calculated. This global index could then correspond to a position of a data element in the input or output stream, for example.

Referring to the simple kernel example presented in Listing 2.2, the use of built-in variables `threadIdx`, `blockDim`, and `blockIdx` is shown to calculate the global index of a thread to identify the indices of the array elements that are being operated on by the kernel. These variable are structures with fields `x`, `y`, and `z` depending on their dimension. The use of these variables to calculate the global indices for a two-dimensional problem is also illustrated in Figure 2.2. Here, the global index of a thread in the `x` direction is calculated as: `blockIdx.x*blockDim.x + threadIdx .x`, for example.

Hierarchical CUDA memory model

As with many other processor architectures, CUDA devices have a hierarchical memory model, with some of the memory types shown in Table 2.1. A number of other characteristics of the memory types are also shown, including whether or not the memory offers cached access and whether it is read-only. If we consider as an example the shared memory, it is located on chip (explained in more detail in Section 2.3.2), offers fast low latency read-write access, and thus does not need to be cached. In fact, in many algorithms this shared memory is used as a user managed cache for optimal performance [27]. Furthermore, the shared memory can be accessed by all the threads in a block allowing for data reuse and communication between these threads. Due to the fact that the execution order of threads is not known beforehand, it is often required to make use of synchronisation barriers to ensure that the contents of shared memory is consistent. This is achieved by making use of a `__syncthreads()` function which halts execution for a thread until all threads in a block have reached that point in the code.

Figure 2.3 shows a graphical representation of the CUDA memory hierarchy, showing the allowed access for each memory types (indicated by the colours of the connecting lines). Also shown in the figure is an abstracted representation of a host which consists of a multi-core CPU, connected to the host memory through a hierarchy of cache memory. In the CUDA sense, the

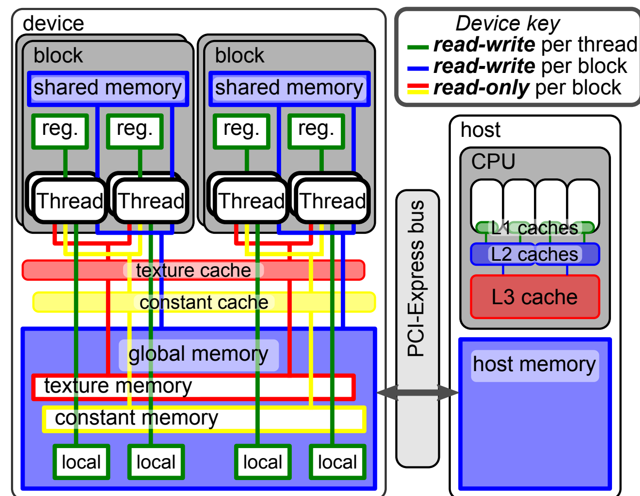


Figure 2.3: Graphical representation showing the memory hierarchy associated with a CUDA device. Red (–) and yellow (–) pathways represent per-thread read-only memories whereas green (–) and blue (–) pathways represent read-write memories that are accessible by a single thread or a block of threads, respectively. Also shown is the host and the communication with the device's global memory space over the PCI-Express bus.

host is the system where the *device* (CUDA GPU) is installed, with its own memory.

As seen in Table 2.1 and Figure 2.3, certain parts of the CUDA device's memory (specifically those that reside in global off-chip memory) can be accessed from the host. As part of the CUDA computational process – to be discussed in more detail shortly – data are trasferred from host memory over a bus (such as the PCI Express bus) to the required area of global memory, where they can be accessed by the threads executing on the device [29]. This need to transfer data before they can be operated on often limits the total performance (including data transfer overheads) of CUDA devices, especially for problems that only require a small amount of computation per data-element. The relatively small amount of global device memory installed, when compared to host memory, can also be a limiting factor for computation on the device.

The CUDA computational process

At this stage we have discussed the concept of a kernel and the segmentation of data into a grid of thread blocks. It has also been stated that CUDA automatically executes the kernel for each thread in the resultant grid. To further illustrate the CUDA computational process, consider again the kernel of Listing 2.2 used as an example thus far. The inclusion of this simple kernel

Table 2.1: A summary of the type and locality of various CUDA memory areas.

Memory	Location (on/off chip)	Cached	Access	Locality	Accessible from host
Register	On	No	R/W	Thread	No
Local	Off	No	R/W	Thread	No
Shared	On	No	R/W	Block	No
Global	Off	No	R/W	All	Yes
Constant	Off	Yes	R	All	Yes
Texture	Off	Yes	R	All	Yes

in a computation is now addressed and introduces a number of other concepts related to GPU computing.

In Listing 2.3 all the auxiliary functions required to run the kernel on the GPU are shown. The first thing to note is that the declaration of the `add()` function is identical to that of the CPU-based example given in Listing 2.1. This is intentional, as the two functions are designed to perform the same calculations and are thus interchangeable.

Listing 2.3: An implementation of the `add()` routine in Listing 2.1, illustrating the invocation of the simple kernel presented in Listing 2.2 for GPU computation.

```
void add ( int N, const float *a, const float *b, float *c )
{
  // Pointers for device memory
  float *pdev_a = 0; float *pdev_b = 0; float *pdev_c = 0;
  // Allocate the device memory
  cudaMalloc ( (void**) &pdev_a, N*sizeof(float) );
  cudaMalloc ( (void**) &pdev_b, N*sizeof(float) );
  cudaMalloc ( (void**) &pdev_c, N*sizeof(float) );
  // Transfer the input data to the device
  cudaMemcpy ( pdev_a, a, N*sizeof(float), cudaMemcpyHostToDevice );
  cudaMemcpy ( pdev_b, b, N*sizeof(float), cudaMemcpyHostToDevice );
  // Setup the data segmentation
  int threads_per_block = 64;
  int blocks_per_grid = div_to_next ( N, threads_per_block );
  // setup the block and grid using the type dim3
  dim3 block ( threads_per_block, 1, 1 );
  dim3 grid ( blocks_per_grid, 1, 1 );
  // invoke the kernel
  add_kernel <<< grid, block >>> ( N, pdev_a, pdev_b, pdev_c );
  // Transfer the output data to the host
  cudaMemcpy ( c, pdev_c, N*sizeof(float), cudaMemcpyDeviceToHost );
  // Free the device memory
  cudaFree ( pdev_a ); cudaFree ( pdev_b ); cudaFree ( pdev_c );
}
```

Since the CUDA device is an external accelerator to the host and can, in general, not access the memory of the host directly [29], it is necessary to allocate memory in the device's global memory (see Table 2.1). This can be done using the CUDA `cudaMalloc()` function. Listing 2.3 contains three such calls. Each call allocates the requested number of bytes in the device's global memory and modifies a pointer accordingly [29]. Thus, after the calls to `cudaMalloc()`, the pointers `pdev_a`, `pdev_b`, and `pdev_c`, each contain the addresses in global GPU memory where space has been allocated for `N` single precision floating-point values. Here, the convention of prefixing `pdev_` to pointers to device memory is used. As is the case with the standard C `malloc()` function, the memory that has been allocated must be freed once it is no longer needed. This is done using the `cudaFree()` function as shown at the end of the listing.

Once the memory has been allocated on the device, the input data (the contents of the arrays `a` and `b`, in this case) must be specified. In order to make it available on the device the data must reside in GPU memory. In order to copy the data to the device from the host, the function `cudaMemcpy()` is used. This is equivalent to the standard C `memcpy()` routine with the exception that it requires an additional parameter indicating the direction of the copy that must be performed. The possible values for this parameter are:

- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`

- `cudaMemcpyDeviceToDevice`
- `cudaMemcpyHostToHost`

with the names self-explanatory [44]. Note that in the code sample given in Listing 2.3, `cudaMemcpyHostToDevice` is used to copy the input data to the device, whereas `cudaMemcpyDeviceToHost` is later used to copy the computed result (stored in `pdev_c` on the device) back to the host overwriting the contents of the array `c`.

The next step in the computational process is to segment the problem domain by dividing it into a grid of thread blocks. Since the example presented here is linear in nature, this process is quite simple, and the code to calculate the division is also given in Listing 2.3. In this case, the number of threads per block (`threads_per_block`) is fixed at 64. Using this value and the `div_to_next()` routine of Listing 2.4, the number of blocks in the grid (`blocks_per_grid`) is determined. Note that all the blocks in the grid must be of equal size and that the total number of threads must be at least equal to the length of the arrays, `N`.

Listing 2.4: A routine to calculate and return the smallest integer value `k` that satisfies the condition `k*step` $\geq$ `N`. This routine is used to calculate the number of fixed-size blocks are required in a CUDA grid for a given `N` and with `step` set to the block dimension.

```
int div_to_next ( int N, int step )
{
  if ( N % step )
    // if N is divisible by step then simply return the quotient
    return N / step;
  else
    // if N is not divisible by step then return the integer part of the
        quotient plus 1
    return N / step + 1;
}
```

As seen in Listing 2.4, if `N` is divisible by `step` (with `step` equal to `threads_per_block` when called from Listing 2.3), then the return value (and thus `blocks_per_grid`) is simply `N/threads_per_block`. If this is not the case, `blocks_per_grid` of Listing 2.3 is assigned the return value of `N/threads_per_block + 1`. In the latter case, the last block in the grid will have `blocks_per_grid *threads_per_block - N` threads that will have no data elements associated with them. For this reason, it is important to ensure that when the kernel is executed, these threads do not attempt to access memory locations that are undefined. This is done in Listing 2.2 by checking if `i < N` before using `i` as an index in the array. An alternative is to always ensure that the amount of device memory allocated is always a multiple of the block size – effectively padding the input and output data on the device – and has been found to offer some performance improvements over the unpadded case [45].

With the number of threads and blocks determined and used to initialise the two `dim3` (a `struct` with three `int` fields `x`, `y`, and `z` [29]) variables, `block` and `grid`, the CUDA kernel `add_kernel()` of Listing 2.2 can be invoked. This is done in the same way as calling a standard C function, with the addition of specifying the block and grid configuration by including `<<< grid, block >>>` between the function name and the parameter list. This instructs the CUDA runtime library to launch the kernel for the number of threads defined by the grid. After the kernel execution, the output data can be transferred to the host as already discussed.

It should be noted that this kernel call is the first point where any code is executed on the device, with all code preceding and following it executing on the host. To clarify this, the process described here and corresponding to Listing 2.3 is depicted graphically in Figure 2.4.

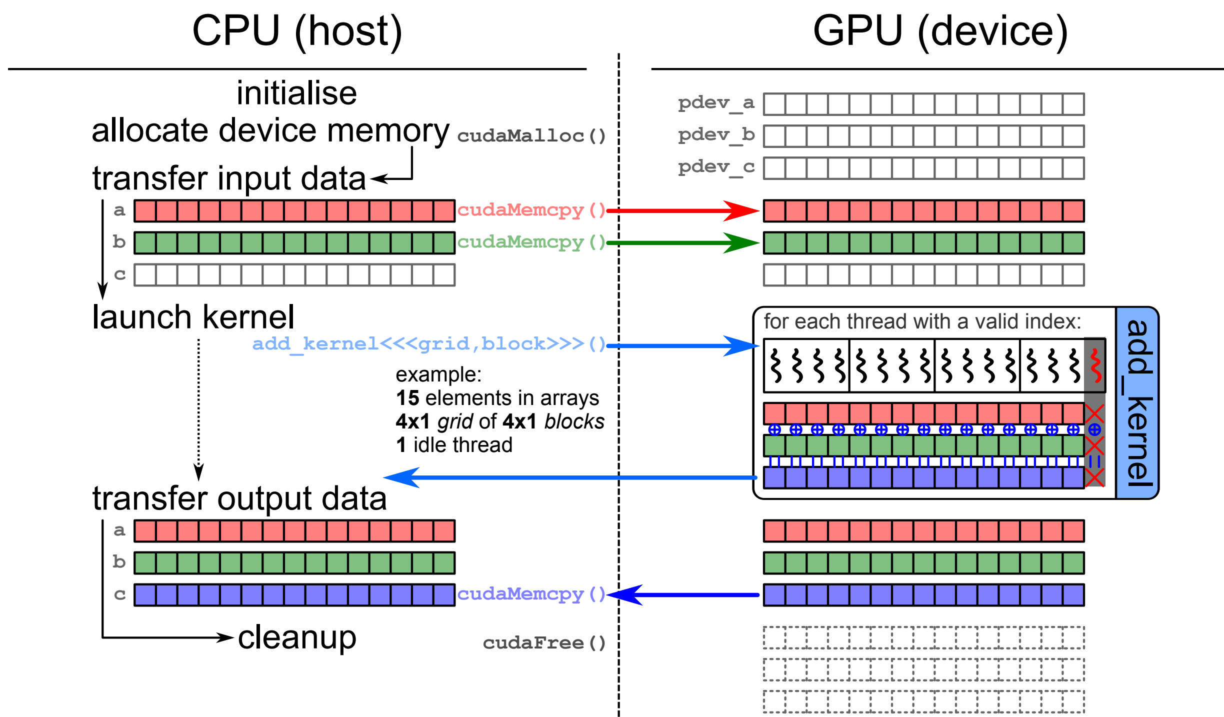


Figure 2.4: The computational process for calculating the sum of two arrays `a` and `b` and storing the result in a third array `c` using the CUDA kernel in Listing 2.2. The listing for the process is given in Listing 2.3.

The example considered here consists of only a single CUDA kernel, and once completed the calculated results are transferred from the device to the host using the `cudaMemcpy` function. This function includes an implicit global barrier that ensures that all threads have completed executing before commencing with the data transfer. If a more complex problem is being considered, it may be necessary to call multiple CUDA kernels before transferring the final results from the device. If this is the case, it is important to use explicit global barriers such as the `cudaThreadSynchronize()` routine to ensure that data required for a subsequent kernel has been calculated [29].

2.3.2 Hardware implementation

To this point we have considered a CUDA-capable device as an abstraction that is compatible with the programming model discussed. Some aspects of the hardware itself are now covered. This serves to provide a better understanding not only as to where the power of these devices come from, but also as to what the inherent limitations of the architecture are.

As already stated, the G80 hardware architecture was introduced by NVIDIA along with CUDA [3]. This hardware saw a move away from the pipelined design in original GPU implementations. In its place, a unified shader architecture was developed where a shader was not restricted to performing one particular kind of operation, either vertex or pixel shading, for example. Now, each shader was identical to the others and could be programmed to perform any required operation.

Since the G80, the CUDA architecture has seen two major redesign iterations, with each new generation of hardware offering many new features and performance improvements over the previous one. Due to the rapidly evolving nature of the landscape, it will not be long before the hardware discussed here will once more be out of date. For this reason, the discussion that follows will be of a general nature so as to (hopefully) be applicable to future hardware versions.

Following this, a quick overview of the current state of the art will be presented at the end of the section.

Building a massively parallel processor

At the heart of every CUDA device are many identical CUDA cores – also called a thread processors (TPs) or streaming processors (SPs). These are the device elements that are responsible for actually performing calculation and consist of 32bit floating-point and integer ALUs and some control circuitry – although much of the latter is shared between a number of these cores [27]. This control logic and and instruction cache shared between the SPs leads to the next logical grouping to consider – namely the symmetric multiprocessor (SM[1]) [29].

Simplified representations of the SMs of the three architectures considered here are shown in Figure 2.5. Here, it can be seen that in addition to a fixed number of SPs, an SM also contains a number of registers and a shared memory space, already mentioned in Section 2.3.1, that can be accessed by the SPs. Note that, since the shared memory is accessible by all the SPs in an SM, it can be used to share data between them. Additional components of an SM include special function units (SPUs) which are used to perform more complex floating-point operations such as calculating trigonometric functions [29]. It should be noted that each SM of the GT200 also includes one double precision floating-point unit and although the Fermi architecture also supports double precision computation, this seems to be achieved by using the 32bit floating-point ALUs from two SPs concurrently.

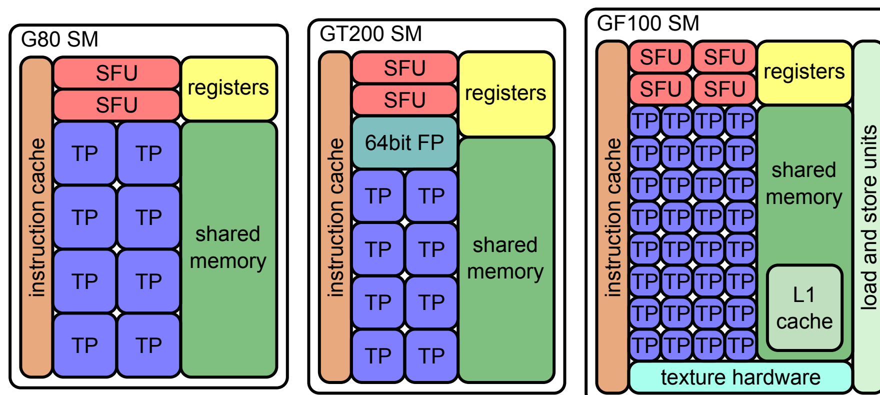


Figure 2.5: Simplified representations of the SMs for (from left to right) the G80, GT200, and GF100 architectures. Shown in each figure are the thread processors (TPs), special function units (SFUs), registers, shared memory and an instruction cache. The SM for the GT200 adds a single 64bit floating-point unit. The GF100 SM is more complex with load and store units, as well as a shared memory space that includes an L1 cache which does not require user management.

In the case of the G80 and GT200, the SMs are grouped together to form a texture (or thread) processing cluster (TPC) [35, 46] that adds texture sampling hardware, as well as a first level of texture cache. It should be noted that only texture references (as per Table 2.1) are cached and this can be used to improve performance of certain algorithms considerably [47]. The cluster also includes a shared instruction scheduler. For the GF100 architecture, the concept of a TPC is no longer required, as much of the hardware previously shared between the SMs of a

[1]Note that NVIDIA uses the abbreviation SM to refer to a physical (and logical) hardware groping on a CUDA device. In general parallel computing terms, the abbreviation SMP is used for symmetric multiprocessor or symmetric multiprocessing, in which case it refers to a more general computing model where a number of processors have access to a shared memory space.

texture processing cluster (TPC) are now included in the SMs themselves. This includes load and store units, as well as the texture cache. In addition, the shared memory area includes a cache for all memory operations. The SMs are still grouped together into graphics processing clusters (GPCs) which encapsulate all the required hardware for graphics processing [48].

For all the architectures, these clusters – be it texture processing or graphics processing clusters – of SMs are used as the building blocks for constructing CUDA-capable devices. In addition to the clusters, various other components are required - including memory controllers for accessing the device's global memory, as well as a second level of cache. In the case of the G80 and GT200 architectures, this cache is only used for the reads of textures stored in global memory. For the GF100 architecture, it is used for for all global memory accesses [48].

In Section 2.3.1, the concept of lightweight GPU-threads whose positions in a global grid are available at runtime was introduced. This fine-grained, data-parallel software approach is what is at the core of parallel execution on CUDA hardware [27]. CUDA-capable devices use the GigaThread Engine to realise the execution of the instructions that comprise a CUDA kernel over multiple data elements in parallel. The GigaThread Engine enables each SM of a CUDA device to store the contexts of hundreds of threads in hardware and allows for switching between these threads with zero overhead [27, 29]. A summary of the capabilities of the three CUDA architectures discussed is given in Table 2.2.

Table 2.2: Specifications for the three CUDA architectures discussed. Values indicate the maximum values for each architecture, and as such may differ for implementations targeting different market segments.

		G80	GT200	GF100
computational resources				
clusters per device		8	10	4
SMs per cluster		2	3	4
total		**16**	**30**	**16**
TPs per SM		8	8	32
total		**128**	**240**	**512**
SFUs per SM		2	2	4
total		**32**	**60**	**64**
64bit FP units per SM		–	1	16(1)
total		–	**30**	**256**(1)
per SM resources				
shared memory		16KB	16KB	64KB(2)
registers		32KB	64KB	128KB
thread contexts		768	1024	1536
theoretical peak FP performance @ 1GHz(3)				
GFLOPS	32bit	256	480	1024
	64bit	–	60	512

(1) Two TPs are used for a single double precision (DP) operation.

(2) Configurable as either 48KB shared memory and 16KB L1 cache, or 16KB shared memory and 48KB L1 cache.

(3) 1 multiply-add (MAD) or fused multiply-add (FMA) operation per clock per floating-point (FP) unit – 2 FP operations per clock per floating-point unit.

2.3.3 Mapping software to hardware

Although the SPMD programming model is a generalisation of a SIMD implementation (as discussed in Section 2.1 and in [17]), a CUDA device is not a pure SIMD device [27]. Furthermore, even though marketing material related to CUDA devices typically list the number of SPs as the number of processors, an SM is the smallest hardware grouping with an independent front-end (including a fetch unit and scheduling logic) [46]. Thus, a CUDA device can be seen as a MIMD array of SIMD cores. This means that although each SP in an SM is executing the same instruction in a SIMD fashion, different SMs on a device may be executing different instructions. NVIDIA goes further to state that the execution on an SM is in fact *single-instruction-multiple-threads* (SIMT) which differs from SIMD in that it relaxes some of the requirements imposed on inputs and outputs, as well as branch instructions [27]. This section discusses how the thread-block programming model discussed in Section 2.3.1 maps to execution on the CUDA hardware discussed in Section 2.3.2.

Block assignment

When a CUDA kernel is launched, and the grid of thread blocks initialised, the blocks of the grid are each assigned to an SM on the device [27]. As long as the resource (number of threads, registers, and shared memory) requirements of each block can be met, multiple blocks can be assigned to an SM. There is also an upper limit on the number of blocks that can be assigned to an SM. Blocks that cannot initially be assigned for execution, are kept in a queue until the kernel execution of another block completes, Furthermore, a block remains assigned to an SM and its allocated resources (shared memory and registers) remain reserved until execution completes for the block. This reservation of resources is what allows for the communication between the threads of a block though shared memory. Of course it follows that, for this same reason, a block cannot be transferred to another SM once it has been assigned for execution [27, 29].

Thread scheduling and execution

Since scheduling and execution depends heavily on the hardware implementation, it makes sense to discuss it with reference to a specific CUDA device family [27]. For the discussion that follows, the GT200 architecture will be used. This is also one of the architectures used to obtain the results presented in Chapter 3, Chapter 4, and Chapter 5. It should be noted that, although the exact values used may be device-specific, the concepts can be applied to any of the other devices discussed.

In order to schedule the threads of a block, they are grouped into warps [29]. In the case of the GT200 (and G80) the warps are made up of 32 threads. The same instruction is issued to each of the threads in a warp by the SM and is executed in a SIMT fashion. A single precision floating-point operation that takes one cycle to complete per thread (such as floating-point multiplication or addition [29]), will require four cycles to complete for an entire warp. This is because only eight threads can execute at a time on the available SPs, and is illustrated as a timing diagram in Figure 2.6 for a block of 64 threads.

Each instruction that is to be executed, as well as fetches from the various memories, have a latency associated with them [49, 50]. These can range from a few cycles for a simple floating-point multiplication to hundreds of cycles for more complex operations such as integer division or fetching an operand from global memory. For many of these cycles, there is no useful computation being done, and the processing hardware of the device is idle. While the threads of a warp are stalled the GigaThread Engine allows for the scheduling of another warp of threads at zero-cost. This new warp then continues execution. Having enough warps (or blocks) assigned to an SM, ensures that there are always threads executing while other warps wait for high la-

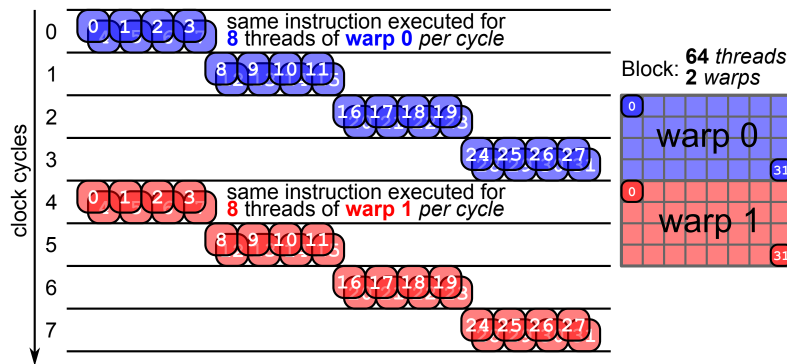


Figure 2.6: A simplified timing diagram illustrating the execution of threads in a warp. Since there are eight thread processors, the same instruction is issued over four cycles and executed for each of the 32 threads in the warp (assuming a single cycle single precision floating-point instruction). Also shown is a 64 thread block and its division into two warps, with the second warp executing after the first.

tency instructions to complete – effectively hiding the latency of these instructions [27]. This process is depicted graphically in Figure 2.7 and is similar to the use of out-of-order execution in a traditional CPU pipeline [17]. This latency-tolerant design is one of the reasons that there is so little of the surface area of the device's processor devoted to cache memory, but also means that in order to fully utilise the device the datasets being operated on and the number of threads must be sufficiently large [27].

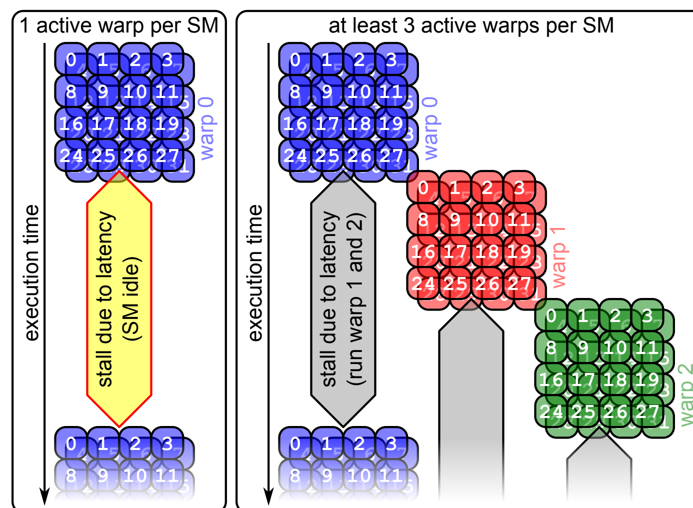


Figure 2.7: A diagram illustrating how assigning multiple warps for execution on an SM can hide the latency of long latency operations (including global memory reads). On the left – with only 1 active warp – the SM is idle while waiting for a long latency operation to complete. On the right, warps are scheduled ensuring that the SM is not idle during long latency operations.

2.3.4 Software ecosystem

Although the programming model and the hardware implementation of a parallel architecture such as CUDA are important aspects to consider, perhaps the most important component is the software that is built around the architecture. The ease of use of this software is often a significant factor in the adoption rate of the architecture as a whole. In this section we

consider the software – both from NVIDIA and third-party developers – that is designed for the CUDA architecture. This includes development tools, as well as libraries that provides specific functionality.

CUDA runtime and driver APIs

When developing CUDA applications, one makes use of one of two APIs, namely the CUDA driver API or the CUDA runtime API. These APIs provide functions to handle all the aspects associated with a CUDA device and include device management, memory management, and execution control [50]. Examples of calls to the runtime API have already been shown in Listing 2.3, where the `cudaMalloc()`, `cudaMemcpy()`, and `cudaFree()` routines were used.

In terms of functionality, all the functionality of the runtime API is provided by the driver API. The latter is a lower level API and the code required tends to be more verbose, but the API provides some additional functionality [50]. For the examples presented in this dissertation, the runtime API is used, as it is simpler and thus easier to follow. It should be noted that, although earlier versions of CUDA required that the driver and runtime APIs be mutually exclusive, with only one or the other used in an application, this is no longer the case [29].

A diagram representing the CUDA software stack showing the relationship between a CUDA application and the two APIs discussed, is shown in Figure 2.8. Also shown in Figure 2.8 are a number of CUDA libraries that are built using the CUDA runtime and driver APIs. These libraries provide additional functionality, hiding much of the implementation details from the user application.

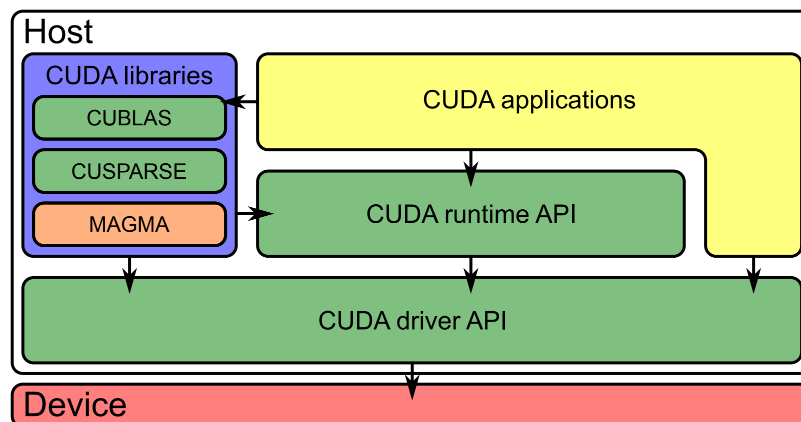


Figure 2.8: Schematic representation of the CUDA software stack showing both the driver and runtime APIs. Also shown are a number of higher level libraries used in this dissertation, as well as the interactions between the components. Note that only the driver API communicates directly with the device and that the APIs, libraries and applications reside on the host.

CUDA libraries

Perhaps the most influential factor in the adoption of CUDA over competing technologies was the early release of accelerated libraries for common computational tasks. Two such libraries are CUFFT and CUBLAS [7, 51] which are CUDA implementations of the Fast Fourier Transform [52] and the BLAS [53] set of linear algebra routines, respectively. These libraries allowed developers to implement GPU-enabled algorithms without having to spend much time getting to know the new architecture.

Subsequently, NVIDIA has released the CUSPARSE and CURAND [54, 55] libraries, which provide support for sparse matrices and random number generation, respectively. In addition, a number of third-party libraries have come to the fore, providing similar functionality to mainstream CPU-based libraries. One of the most notable ones relevant to the work discussed in this dissertation are MAGMA [8] and CULATools [56] which implement a subset of the LAPACK [57] routines. The CUBLAS and MAGMA libraries are discussed further in Chapter 3, where various aspects of numerical linear algebra on CUDA-based GPUs are considered.

2.4 ATI Stream SDK

Although the CPU-based aspects of the work presented in this dissertation are implemented in CUDA, it is useful to consider some of the alternatives available. For this reason the technology offered by AMD is also considered. Note that the discussion presented here does not go into as much detail as that of Section 2.3, but instead serves to highlight some of the key similarities and differences between the two technologies.

Many of the differences between the ATI Stream SDK (including OpenCL discussed in Section 2.5) and NVIDIA CUDA are simply a matter of name – at least as far as this discussion is concerned. As such, Table 2.3 provides a mapping between terminology from the different architectures [39, 40, 58]. In the case of the hardware elements, the table does not imply anything regarding the similarities of the different implementations.

Table 2.3: A list of equivalent terms or functionally equivalent hardware for NVIDIA CUDA and ATI Stream SDK.

NVIDIA CUDA	ATI Stream SDK
thread organisation	
thread	work-item
block	work-group
grid	NDRange
warp	wavefront
hardware	
thread processor	stream core *(processing element)*[1]
symmetric multiprocessor	compute unit
special function unit	T-processing element

[1] Although it is more fair to compare a thread processor (TP) to a stream core, there are some cases where the comparison to a processing element (PE) is more apt. In terms of single precision floating-point arithmetic, for example – a TP and a PE can be seen as nearly equivalent.

2.4.1 Hardware

The hardware from AMD follows similar design principles to those of the NVIDIA devices discussed in Section 2.3.2, in that there are a number of simplified processing cores that share a certain amount of control and instruction processing hardware. A diagram representing the ATI equivalent of an symmetric multiprocessor, called a Compute Unit (CU), is shown in Figure 2.9

[39]. Each CU contains 16 Stream Cores (SCs), as well as a local data store (LDS) that acts as a shared memory space for communication between the threads running on the CU.

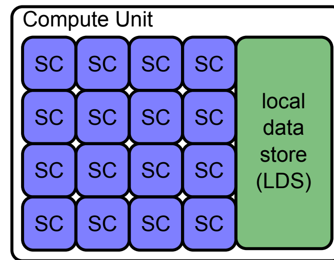


Figure 2.9: An ATI Cypress compute unit consisting of 16 stream cores (SCs) and a local data store (LDS). Since each stream core (shown in Figure 2.10) is more complex than a CUDA TP, the compute unit is simpler than a CUDA GT200 SM (see Figure 2.5). For example, the compute unit does not contain separate SFUs, since this functionality is implemented at an SC level.

The SCs are the equivalent of the CUDA stream processor, but an ATI SC is significantly more complex. Each one comprises five processing elements (PEs) (capable of performing single precision floating-point or integer operations), as well as a branch prediction unit. One of the PEs is more advanced, with this T-Processing Element (T-PE) performing some of the functions of the CUDA special function unit (SFU) [39]. A representation of an ATI SC is shown in Figure 2.10, where a small register file that is shared between the processing elements can also be seen. For double precision operations, two or four of the regular PEs are used together, depending on the operation required [39].

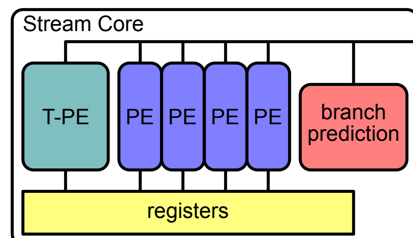


Figure 2.10: An ATI stream core (SC) showing four processing elements (PEs) and a T-processing element (T-PE) which performs the same operations as a CUDA SFU. Each PE has access to its own portion of a shared register file and the SC also contains branch prediction hardware.

With an ideal instruction mix, all the PEs of an SC can be used in parallel, and as such the AMD marketing material lists the number of PEs as the number of cores [39]. However, comparing the number of ATI stream cores (and not processing elements) to the number of CUDA thread processors (TPs) would be more fair.

As is the case with NVIDIA GPUs, an AMD device also contains various texture processing hardware (including two levels of texture cache), as well as a global device memory, resulting in a similar hierarchical memory model to the one presented in Table 2.1 and discussed in Section 2.3.1 for CUDA devices. Table 2.4 shows some specifications of the ATI Cypress GPU [39], with the equivalent specifications of the NVIDIA GT200 from Table 2.2, shown for comparison.

2.4.2 Software

For the current implementation of the ATI Stream SDK, OpenCL is used as the high-level language of choice. In addition, the Compute Abstraction Layer (CAL) [59] can be used to program the devices at a very low level. OpenCL is similar to the CUDA driver API in terms of verbosity, and since OpenCL is designed to run on a variety of different architectures and devices, the routines associated with device management tend to be more complex. A more complete analysis of OpenCL and its comparison with CUDA is covered in Section 2.5.

As discussed in Section 2.3.4, one of the factors that accelerated the adoption of CUDA was the presence of libraries such as CUBLAS and CUFFT. Although AMD does provide ACML-GPU [60], which implements BLAS routines that are accelerated by AMD GPUs, at present this is limited to matrix-matrix multiplication routines. Since the ratification of OpenCL and its subsequent adoption as the device programming language of choice by AMD, libraries such as ViennaCL have come to the fore [61]. ViennaCL provides functionality covered by many of the BLAS routines, as well as some more advanced linear algebra routines. As OpenCL gains more acceptance in the development community, it is expected that more third-party libraries will become available.

Table 2.4: Specifications for an ATI Cypress (RV870) device with the equivalent values (and architecture components) for an NVIDIA CUDA GT200 shown for comparison.

	Cypress	GT200	
computational resources			
CUs per device	20	30	(SMs)
SCs per CU	16	8	(TPs)
total	**320**	**240**	
PEs per CU	80	–	
total	**1600**	–	
T-PEs per CU	16(1)	2	(SFUs)
total	**320**	**60**	
64bit FP units per CU	16(2)	1	
total	**320**(2)	**30**	
per CU resources			
local data store	32KB	16KB	(shared memory)
registers	256KB	64KB	
theoretical peak FP performance @ 1GHz(3)			
GFLOPS 32bit	3200	480	
64bit	800	60	

(1) The T-PEs are included in the count of PEs per CU as they can perform the same operations.

(2) Two or four regular processing elements (PEs – 64 per CU) are used to perform double precision floating-point (FP) operations.

(3) 1 multiply-add (MAD) or fused multiply-add (FMA) operation per clock per floating-point (FP) unit – 2 FP operations per clock per floating-point unit. Note that Cypress devices tend to be clocked below 1GHz[39].

2.5 OpenCL

OpenCL is an open standard that aims to provide a means to implement general purpose parallel programming across a wide variety of platforms and devices including CPUs, GPUs and other processors [40]. Thus it could be said that OpenCL aims to be to heterogeneous parallel computing what the OpenGL standard is for graphics. Furthermore, OpenCL supports both data-based and task-based parallelism and is implemented as a subset of ISO C99 including certain parallel programming extensions. It should be noted that the standard has been ratified by a large number of players in the computing industry, including Apple, Intel, AMD, and NVIDIA [38].

Since OpenCL is designed to produce portable code that can run across a wide range of platforms, it can be defined through a series of models to which an OpenCL-compatible device must adhere [40]. Some of these models will now be discussed, with many of the concepts having one-to-one mappings to those present in CUDA and discussed in Section 2.3. Some of the mappings have already been mentioned in Table 2.3, since the latest iteration of the AMD hardware has been specifically designed for OpenCL compatibility.

Although the OpenCL programming model is not discussed in depth, it should be noted that it supports the same data parallel approach as discussed for CUDA in Section 2.3.1 and more generally in Section 2.1. In addition, OpenCL supports a task parallel programming model which sees parallelism achieved by executing a number of unrelated tasks concurrently instead of performing the same operation over a number of data elements at the same time [17, 40].

2.5.1 Platform model

The highest level abstraction in the OpenCL specification is the concept of a platform, which is depicted graphically in Figure 2.11. An OpenCL platform consists of one or more Compute Devices (such as NVIDIA or AMD GPUs discussed in Sections 2.3 or 2.4, respectively) connected to a host [40]. There is no requirement that these Compute Devices be identical – hence the suitability of OpenCL to heterogeneous computing. A Compute Device further consists of at least one Compute Unit, which in turn is made up of one or more Processing Elements. Note that although the host appears separately on the diagram in Figure 2.11, a single core CPU system can still act as a OpenCL platform with the host and the Compute Device being the same physical device. The host-device model is identical to the one used in CUDA, where a

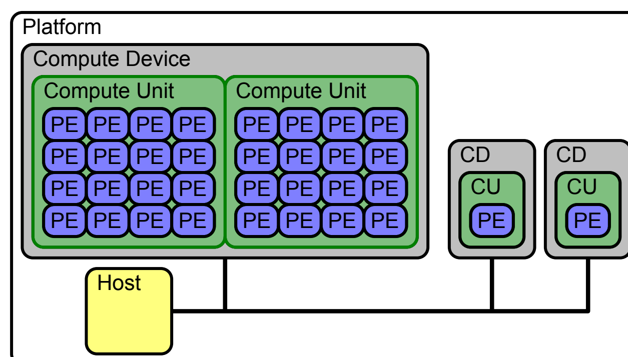


Figure 2.11: The OpenCL platform model showing a host attached to multiple more compute devices (CDs). Each compute device consists of one or more compute units (CUs) which in turn consists of at least one processing element (PE). The figure shows three compute devices, one with two compute units and a total of 32 PEs, and two more identical ones with one compute unit and processing element each. This is then an example of a heterogeneous system.

CUDA GPU acts as an external accelerator, although in this case, no explicit provision is made for heterogeneity.

2.5.2 Execution model

Since the platform model previously discussed is similar to that of CUDA devices, it follows that the execution model will have similarities as well. As is the case in CUDA (see Section 2.3) and in stream computing [41, 43], the pieces of code that execute on OpenCL devices are called kernels, with a host program that manages the execution of these kernels [40].

Many of the concepts discussed regarding the CUDA thread organisation model can be translated to OpenCL, as has already been indicated in Table 2.3. A brief overview of the OpenCL execution model is now given, with the OpenCL version of Figure 2.2 presented in Figure 2.12 [40]. Here, the global index space is an `N`-dimensional index space called an `NDRange` with the supported values of `N` being one, two (shown in Figure 2.12), or three. The `NDRange` thus contains `N` integers indicating the size of the index space in each of the `N` dimensions [40].

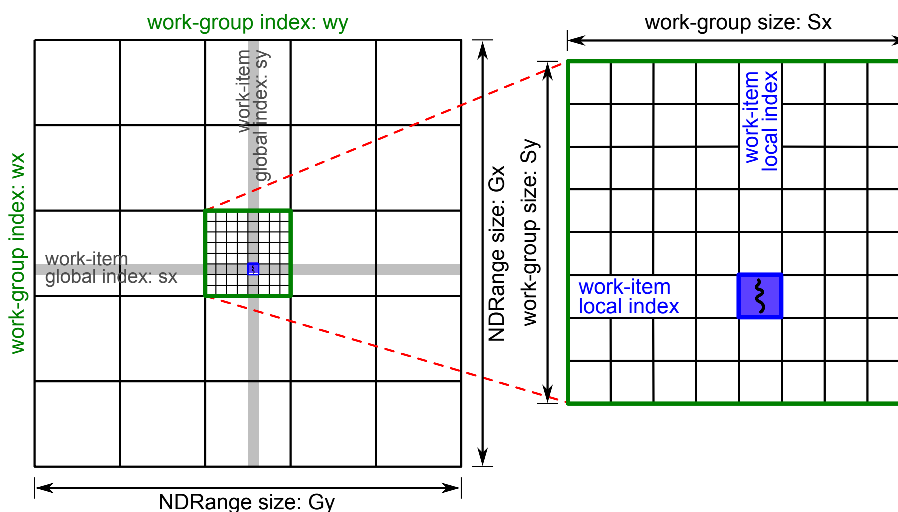


Figure 2.12: Organisation of OpenCL work-items into an work-groups and an NDRange. Note the differences in naming when compared to the CUDA case shown in Figure 2.2.

Although this `NDRange` is roughly equivalent to the CUDA grid, there are some differences. As mentioned, the `NDRange` can be one-, two-, or three-dimensional, whereas the CUDA grid is at present restricted to only two dimensions [29]. A more important difference is that while the CUDA grid can be seen as an arrangement of CUDA thread blocks, the `NDRange` specifies the global indices of the OpenCL work-items themselves [40].

The OpenCL work-items offer the finest level of granularity and are in effect an executing instance of an OpenCL kernel for each element of the index space (`NDRange`). Similarly to the case of a CUDA thread, the global ID (position in the index space) of a work item is known at runtime and although the same code is executed by each work-item, the data being operated on, as well as the specific execution pathways (in the case of conditional statements or loops, for example) may differ [40]. Work-items are organised into work-groups of equal dimension which provide a more coarse-grained segmentation of the index space, with each work-group being assigned an ID related to its position in the space. In CUDA, the global position (or ID) of a thread has to be explicitly calculated using the thread's local position in a block and the block's position in the grid. In OpenCL, a number of routines are provided that return the global and local ID of a work-item at runtime [40].

In terms of execution on an OpenCL device such as an NVIDIA or AMD GPU (Section 2.3.2 and Section 2.4.1, respectively), a work-group is assigned to a compute unit. Here, the instructions for the work-items are executed by the processing elements. Note that since there is no requirement that there be more than one processing element, it may be that the work-items of a group, and even the work-groups themselves, are executed sequentially.

2.5.3 Memory model

As is the case with CUDA, OpenCL defines a hierarchical memory model for an OpenCL device, with work-items having access to four memory regions. These are summarised in Table 2.5, which can be compared to Table 2.1.

Table 2.5: A summary of the OpenCL memory types showing allowed access, as well as locality. Also indicated is whether the memory is accessible from the host and what the CUDA equivalent memory is. The CUDA version of this table is given in Table 2.1.

Memory	Access	Locality	Accessible from host	CUDA equivalent
Private	R/W	work-item	No	Register
Local	R/W	work-group	No	Shared[1]
Global	R/W	All	Yes	Global
Constant	R	All	Yes	Constant

[1] Note that CUDA local memory is reserved per-thread and resides in global device memory. Thus the equivalent of OpenCL local memory is CUDA shared memory – although OpenCL does not require that this be implemented on-chip.

Note that even though the names are similar, CUDA local memory and OpenCL local memory are not related. OpenCL local memory is the equivalent of CUDA shared memory. There is, however, no requirement in the OpenCL specification that the former be implemented as fast, low-latency, on-chip memory, and may simply be mapped to sections of global memory [40]. Some OpenCL devices such as the AMD and CUDA GPUs do implement this local memory as small areas of fast memory for performance reasons [39, 58].

2.6 Outlook for GPGPU computing

No discussion on GPU computing would be complete without at least some mention of the future of the technology. One aspect that can be assumed is that the devices will become more and more capable – as can already be attested due to their rapid evolution to this point. It is expected that the performance of these devices will improve for some time. Since this performance improvement will likely be due to increased levels of parallelism and not frequency scaling, the power requirement issues that hampered the improvement of conventional CPUs should be less of a problem, although memory bandwidth limitations will play an ever increasing role.

Perhaps the most important development will be in the programmability of these devices. This is already evident from the great advances made when comparing CUDA and OpenCL to earlier options such as using OpenGL or shading languages like Cg to force a GPU to perform general purpose computation. Particular attention should be paid to OpenCL with its cross-platform, device-independent approach, which is very attractive to developers as it means it is not necessary to bet on one particular hardware vendor.

OpenCL is also an interesting option when it comes to heterogeneous computing – where a system used for computation consists of a number of different compute devices – as it allows not only for operating system- or device-independent development (in theory), but also for multiple devices with different capabilities to be used at the same time. The importance of heterogeneous computing is further illustrated by developments such as the Fusion platform by AMD [5], and the fact that NVIDIA is also including support for x86 CPUs in their CUDA compilers [6].

2.7 Conclusion

In this chapter, a very brief overview of general purpose GPU (GPU) computing has been presented, with the promise of improved performance at least prompting an investigation into the use of such techniques in the field of computational electromagnetics. Although the chapter provided more detail regarding NVIDIA CUDA, it is hoped that the introduction into competing technologies such as the OpenCL standard – from a software point of view – and the hardware from AMD gives a better understanding of the landscape. In the end it is the software maturity of CUDA (with a two year head-start over OpenCL) and the readily-available third-party libraries that win out for our intended applications.

Although this chapter was brief, there are a number of resources that can be used as a starting point for getting into GPGPU computing. The websites from both NVIDIA [4] and AMD [36] offer a number of sample applications, links to tutorials, as well as examples of what is possible with the various technologies. For NVIDIA CUDA development, both [27] and [42] can be recommended as good sources for getting started, with the former approaching the problems from the more general perspective of massively parallel processors.

Chapter 3

GPU-accelerated dense numerical linear algebra

Many problems in scientific analysis and engineering result in matrix equations that need to be solved – be it as part of a linear system or an eigenvalue problem. The field of computational electromagnetics is no different, and although matrix-free methods do exist, techniques such as the Method of Moments (MOM) and the Finite Element Method (FEM) both involve assembling a matrix of some sort and using it to find the desired solution to the problem being modelled.

In this chapter, an overview of dense numerical linear algebra (NLA) is presented, with specific attention given to the libraries that are considered the de-facto standard in this field – BLAS [53] and LAPACK [57] – and their acceleration using CUDA GPUs. Although much of the desired functionality is provided by existing libraries (most notably CUBLAS [7] and MAGMA [8]), these do still have some limitations. One of the limitations addressed here is that imposed by the amount of memory available on CUDA devices. This limitation results in a restriction on the sizes of the matrices for which the LU decomposition can be performed. The LU decomposition specifically, is an important operation in the field of computational electromagnetics as it can be used as part of the MOM and FEM solution processes. These methods will be discussed in Chapter 4 and Chapter 5, respectively.

The implementations of a CUDA-based LU decomposition discussed here are the result of mapping a traditional out-of-core (OOC) implementation presented in [62] to the GPU computing model. Although results for these implementations have already been presented in [9], this chapter contains a more in-depth discussion of the implementation details, as well as additional result sets, which are obtained on three different systems with varying levels of computational power.

One of the systems considered is relatively old, and as such the measured CPU-only performance lags considerably behind current systems. The use of this system in the benchmarking study allows investigation into the feasibility of using consumer-level GPUs to boost the performance of ageing systems. The results presented indicate that even with only a GT200-based GPU installed, older systems are able to provide competitive performance when compared to CPU-only results for more current systems. Furthermore, the relatively low price of a drop-in GPU upgrade, when compared to a replacement system, makes it an enticing option – especially when one's budget is constrained.

The introduction to dense numerical linear algebra is presented in Section 3.1, including a discussion on the libraries used, and some points relevant to the implementations presented later in the chapter. This is followed by a short discussion in Section 3.2 on existing GPU-based dense NLA with an emphasis on the LU decomposition. Following this, the details of the adaptation of out-of-core methods and their subsequent implementation are presented in Section 3.3.1. Section 3.4 introduces the systems used for benchmarking the implementations discussed, as

well as the methods presented in subsequent chapters. It also shows the results obtained for four variants – in terms of supported data type and precision – of the two LU decomposition implementations discussed. The significance of the performance results is addressed and general comments and conclusions for the chapter are given in Section 3.5.

3.1 Dense numerical linear algebra

Since many problems in science and engineering can be reduced to (or include) linear algebra operations, a lot of time and effort has gone into implementing these routines on computers. Two packages that have become synonymous with dense numerical linear algebra are the Basic Linear Algebra Subprograms (BLAS) [53] – providing a number of basic linear algebra routines, including vector and matrix multiplication – and the Linear Algebra Package (LAPACK) [57], in which the BLAS routines are used as the basis for a number of more complex linear algebra operations such as matrix factorisations and singular value decompositions.

The BLAS routines are divided into three levels and are classified according to the order of computation required [63]. The level 1 routines implement linear operations on vectors and, for example, include the inner product of two vectors. The second and third level routines consist of quadratic and cubic operations, respectively. Level 2 operations include matrix-vector operations, while level 3 operations of interest are matrix-matrix operations. Table 3.1 gives a summary of this information, as well as the order of the storage and computational cost associated with each level.

Table 3.1: A summary of the type and storage and computational cost of the three levels of BLAS routines.

Level	Type	Storage	Computation
1	vector-vector	$\mathcal{O}(N)$	$\mathcal{O}(N)$
2	matrix-vector	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
3	matrix-matrix	$\mathcal{O}(N^2)$	$\mathcal{O}(N^3)$

As previously mentioned, LAPACK is a collection of common routines for solving linear algebra problems. These include a number of matrix factorisations (Cholesky, QR and LU decompositions), routines for solving linear systems, as well as eigenvalue calculations and singular value decompositions for dense or banded matrices [57, 64]. Many of these LAPACK routines are written in terms of level 3 BLAS routines and as such can take advantage of optimized BLAS implementations [64].

Most of the BLAS and LAPACK routines support a number of data types for the elements of the matrices or vectors that they operate on. The data type for which a routine is intended, is typically indicated by the first letter of the routine name, and is chosen according to the precision required (single or double precision), as well as whether the routine operates on real or complex values [57]. A summary of the letters used for each of the four possible cases is given in Table 3.2.

When discussions are of a general nature and not pertaining to a particular implementation of a BLAS or LAPACK routine, the type specifier is sometimes replaced by an underscore (_) as is the case in [62]. In this dissertation, the type specifier is replaced by a lower-case `x` with the rest of the routine name in all-caps. Thus `xFUNC` will be used in cases where `SFUNC`, `CFUNC`, `DFUNC`, or `ZFUNC` can be used interchangeably for this fictional routine.

The functionality of some of the BLAS and LAPACK routines used in the various implementations of this dissertation are discussed briefly here. Note that although they often have a

Table 3.2: A summary of the prefixes used for BLAS and LAPACK routines depending on the precision used, as well as whether values are real or complex. Also shown are the number of bytes required for the storage of data elements of each type.

	real	**complex**
single precision	S (4 bytes)	C (8 bytes)
double precision	D (8 bytes)	Z (16 bytes)

variety of modes in which they can operate, only the one most relevant to the implementations will be discussed in detail.

General matrix-matrix multiplication

Of the level 3 routines, the most important is undoubtedly the set of general matrix-matrix multiplication routines, `SGEMM`, `DGEMM`, `CGEMM`, and `ZGEMM`, which perform a general matrix-matrix multiplication on single precision real, double precision real, single precision complex, and double precision complex matrices, respectively [65]. These routines are the building blocks for many other computational routines, including other level 3 BLAS routines [65]. Due to their importance, one finds that hardware vendors attempt to optimise the execution of these routines on their platforms.

The `xGEMM` routines take three matrices (say $[A]$, $[B]$, and $[C]$) and two scalars (say α and β) as inputs and perform the following operation

$$[C] = \alpha [A] [B] + \beta [C] . \tag{3.1}$$

Note that other variants (involving the transpose of $[A]$, for example) of (3.1) are also possible but are not considered here [53].

Triangular system solve

Another important set of BLAS routines – especially in the case of the LU decomposition [8, 62] – are the triangular matrix solve routines `xTRSM`. These can be used to perform operations including the solution of the linear system

$$[L] [X] = \alpha [B] , \tag{3.2}$$

with α a scalar value, and the unknown $[X]$ and $[B]$ both $N \times M$ matrices. The matrix $[L]$ is a lower triangular $N \times N$ matrix with an assumed unit diagonal. As is the case with the `xGEMM` routines, `xTRSM` supports a number of other formats, including the case where $[L]$ is upper triangular and does not have a unit diagonal [53].

It should be noted that solving (3.2) is equivalent to solving

$$[L] \{x\} = \alpha \{b\} , \tag{3.3}$$

with $\{x\}$ and $\{b\}$ both N-vectors, for M right-hand sides using Gaussian elimination [63]. Since the computational cost for each right-hand side is $\mathcal{O}(N^2)$ [63], the total computational cost for performing (3.2) is $\mathcal{O}(N^2M)$.

LU decomposition

The LU decomposition (with partial pivoting) of an $N \times N$ matrix $\left[A\right]$ computes two matrices $\left[L\right]$ and $\left[U\right]$ such that

$$\left[P\right]\left[A\right] = \left[L\right]\left[U\right], \tag{3.4}$$

with $\left[P\right]$ a matrix containing only zeros and ones that applies a series of row interchanges to $\left[A\right]$ [63]. The matrices $\left[L\right]$ and $\left[U\right]$ are lower triangular with a unit diagonal and upper triangular, respectively. The computational cost for the LU decomposition is $\mathcal{O}(N^3)$ and, more specifically, can be performed in $\frac{2}{3}N^3$ FLOPs for a real matrix [63].

The LU decomposition is implemented in the `xGETRF` series of routines in LAPACK. Here, the row pivots, although represented by matrix pre-multiplication in (3.4), are implemented using a one-dimensional integer array `IPIV`. Each `IPIV[i]` contains the (1-based) index of the row which which row `i` must be exchanged. These row interchanges are implemented by the `xLASWP` routines, and take as parameters a pointer to the array `IPIV`, the matrix to which the row pivots must be applied, and the indices indicating the range of rows that must be pivoted.

A factorisation such as the LU decomposition allows one to avoid explicitly computing the inverse of a matrix, as would be necessary, for example, when solving a linear system [63]. Using the LU decomposition, finding an unknown vector $\left\{x\right\}$ by solving the linear system

$$\left[A\right]\left\{x\right\} = \left\{b\right\}, \tag{3.5}$$

for a given $\left[A\right]$ and right hand side $\left\{b\right\}$, can be performed in two steps. The first is to solve

$$\left[L\right]\left\{y\right\} = \left[P\right]\left\{b\right\}, \tag{3.6}$$

for $\left\{y\right\} = \left[U\right]\left\{x\right\}$, and then solve

$$\left[U\right]\left\{x\right\} = \left\{y\right\}, \tag{3.7}$$

for the desired $\left\{x\right\}$. Here, $\left[L\right]$, $\left[U\right]$, and $\left[P\right]$ are obtained by the LU decomposition of $\left[A\right]$ as shown in (3.4). Note that since the computational cost of solving both triangular systems using Gaussian elimination is $\mathcal{O}(N^2)$, the total computational complexity of obtaining $\left\{x\right\}$ remains $\mathcal{O}(N^3)$.

In addition, the factorisation can be reused and the linear system of (3.5) can be solved for multiple (M) right-hand sides, as in

$$\left[A\right]\left[X\right] = \left[B\right], \tag{3.8}$$

with $\left[X\right]$ and $\left[B\right]$ both $N \times M$ matrices. In this case (3.6) and (3.7) are written as

$$\left[L\right]\left[Y\right] = \left[P\right]\left[B\right], \tag{3.9}$$

with $\left[Y\right]$ also $N \times M$ and

$$\left[U\right]\left[X\right] = \left[Y\right], \tag{3.10}$$

respectively. This process is implemented in the LAPACK `xGETRS` routines, with the triangular solve steps including calls to the `xTRSM` routine. Since the factorisation is performed only once, the solution of the linear system can be performed with a computational complexity of $\mathcal{O}(N^3 + N^2M)$.

3.1.1 Optimised implementations

Due to the importance of the BLAS routines, and especially matrix-matrix products in a wide range of linear algebra problems [65], a number of optimised implementations of the BLAS library exist. These optimised implementations are typically targeted at specific platforms or architectures and allow applications developers to achieve high performance on a wide range of targets with little additional development.

Although libraries targeted at large clusters exist, this work is more concerned with the use of GPUs in accelerating computational electromagnetic calculations on ordinary workstations, and, as such, the implementations considered are targeted at these platforms. A number of libraries that provide the desired functionality are shown in Table 3.3. The table also indicates which of the libraries are discussed in more detail in the sections that follow.

Table 3.3: A feature matrix for a number of numerical linear algebra libraries. The table indicates support for CUDA GPUs, whether the library provides BLAS or LAPACK functionality, and if the library is freely available. Here, a • indicates a yes, and a × indicates a no. The names of the libraries that are discussed further are shown in **bold**.

Library	Platform	BLAS	LAPACK	Free
ATLAS	CPU	•	•	•
ACML	CPU(1)	•	•	•
MKL	CPU	•	•	×(2)
CUBLAS	CUDA GPU	•	×	•
CULA Tools	CUDA GPU	×	•	×(3)
MAGMA	CUDA GPU	×(4)	•	•

(1) Implementations of the SGEMM and DGEMM routines are also available for AMD GPUs in ACML-GPU.
(2) A free trial of the MKL is available for personal use – which does not include academic research.
(3) A free basic version is available but only includes a limited number of single precision LAPACK routines.
(4) A number of BLAS routines have been implemented, but have been subsequently included in implementations of CUBLAS.

A notable exclusion from the discussion is the Intel Math Kernel Library (MKL) which provides BLAS and LAPACK implementations (amongst others) that are optimised for Intel's processors [66]. Although the performance of MKL tends to be quite good, it is not considered here due to licensing restrictions.

Also not discussed in detail is the CULA Tools library from EM Photonics [56]. This is a CUDA-based implementation of a large number of LAPACK routines and as such provides functionality similar to that of MAGMA. In terms of performance, CULA Tools attains an LU decomposition (xGETRF) performance of around 330 GFLOPS and 140 GFLOPS for single and double precision, respectively, on a GF100 device, while almost 200 GFLOPS and 70 GFLOPS (again for single and double precision real matrices, respectively) have been measured on the older GT200 architecture. These figures compare favourably to a CPU-based MKL implementation, with performance peaking at around 75 GFLOPS and 20 GFLOPS for the single and double precision case on a modern 4-core system [67]. As is the case with MKL, CULA Tools is not free, and although a free, basic version is available, it only provides a limited number of single precision routines and will not be considered further [56].

ATLAS

One of the most widely used BLAS implementations (that also provides LAPACK functionality) is the Automatically Tuned Linear Algebra Software (ATLAS) [65]. One of the reasons for its popularity is that it uses "Automated Empirical Optimisation of Software" to automatically adjust the BLAS (and LAPACK) implementations to provide similar performance to vendor-tuned libraries across a wide range of platforms. This involves applying a number of optimisation techniques, including software pipelining, cache and register blocking, and loop unrolling to a number of kernel prototypes at compile time and using empirical timings to select the best implementation for a given platform from a wide range of options [65].

This automatic tuning does away with the need for hand-tuned optimisations, which can be increasingly difficult to keep up to date in a rapidly evolving hardware landscape. ATLAS is used in a wide range of software including MATLAB [68] and its GNU alternative, Octave [69], and is available (precompiled and optimised for a wide range of common platforms) on a number of popular Linux distributions such as Ubuntu.

ACML

The AMD Core Math Library (ACML) is a set of numerical routines supplied by AMD. Although these can be used on all modern x86 processors (including those from Intel), they are tuned specifically for use on AMD64 platform processors [64]. ACML includes a full set of BLAS and LAPACK routines, as well as a set of Fast Fourier Transform (FFT) and random number generation routines.

Many of the BLAS and LAPACK routines in ACML include multi-threaded implementations that utilise OpenMP to make use of more than one core in a multi-core (or multiprocessor) machine [64]. Also available is a GPU-accelerated version called ACML-GPU, which uses AMD GPUs (see Section 2.4) to perform matrix-matrix multiplications [60]. Although ACML is not open-source, it is available free of charge for personal, academic, and commercial use.

CUBLAS

As discussed in 2.3, an implementation of BLAS that is optimised to run on NVIDIA GPUs (see Section 2.3.2) is included with CUDA [7]. Although this CUDA BLAS (CUBLAS) library initially only provided a subset of the BLAS routines, it has subsequently been extended to include all of them. The level 3 routines, especially, are able to take advantage of the immense computational power of CUDA GPUs, resulting in a measured performance of around 300 GFLOPS for the `DGEMM` routine on a C2050 (a GF100 device) [70]. On the older GT200 architecture, about 75 GFLOPS and 375 GFLOPS have been measured for `DGEMM` and `SGEMM`, respectively [71].

Note that the performance for the `DGEMM` routines does not always carry over to the other lower level BLAS routines. This is largely due to the data transfer overhead involved when making use of GPUs. In the case of the level 3 BLAS routines, the data transfer cost is $\mathcal{O}(N^2)$, whereas the cost of computation is $\mathcal{O}(N^3)$ (see Table 3.1). As such, for large enough problems, the computational component will dominate. In the case of the level 1 and level 2 routines this is not so, although in some algorithms it may be able to amortise the cost of the data transfer over a number of calls to the routines and still obtain a speedup in the application [15, 47]. This is demonstrated in the eigenvalue solvers considered as part of Chapter 5.

In Section 3.3, the CUBLAS implementations of the `xTRSM` and `xGEMM` routines are used to add GPU acceleration to a left-looking LU decomposition. In addition, this LU decomposition is implemented so as to enable the factorisation of much larger matrices than can be accommodated in GPU memory.

MAGMA

Although still in an early version, the MAGMA (Matrix Algebra on GPU and Multicore Architectures) project provides promising implementations of many LAPACK routines for execution on heterogeneous architectures consisting of multicore CPUs and CUDA GPUs [8]. MAGMA includes the one-sided matrix factorisations (including the LU decomposition discussed in Section 3.1) and their associated solvers. In contrast to CULA Tools [56], which provides similar functionality, MAGMA is freely available under the same licensing conditions as LAPACK.

In addition to a number of LAPACK routines, MAGMA includes MAGMA BLAS (intended to complement CUBLAS), which provides optimised versions of the `xGEMM` and `xTRSM` routines that were not available as part of CUBLAS at one time. Although the `xTRSM` routines, for example, have since been implemented as part of CUBLAS, some of the MAGMA BLAS implementations still offer performance advantages over CUBLAS [8]. If the current implementation pattern continues, however, these changes can be expected to themselves be included in subsequent versions of CUBLAS by NVIDIA.

In terms of performance, the MAGMA library offers exceptional performance compared to CPU-based implementations and is even able to outperform the commercial offering CULA Tools [56]. For the LU decomposition on a GF100 device, the double precision real (`DGETRF`) performance has been measured at 224 GFLOPS [70] – more than 60% higher than the CULA Tools implementation on the same device and almost double the performance of MKL on a 48 core system with a similar theoretical peak performance as the NVIDIA device used. For GT200 devices, 320 GFLOPS and 70 GFLOPS are attainable for single and double precision real matrices, respectively – compared to 100 GFLOPS and 55 GFLOPS for MKL on an 8-core CPU [72].

Although the performance of MAGMA is impressive, the current implementation is still limited by the amount of memory available on the device. As such, a LAPACK-compatible implementation of the LU decomposition for CUDA devices is presented in Section 3.3.1 and a discussion on the effects on the problem sizes that can be solved is given in Section 3.3.4. In some of the other sections of this dissertation, the MAGMA routines are used as is to investigate the acceleration of various computational electromagnetic techniques.

3.2 A note on CUDA-based dense numerical linear algebra implementations

The use of linear algebra routines over a wide range of disciplines has generated much research in their GPU acceleration. This is especially true for matrix-matrix multiplications and three matrix factorisations – the LU decomposition, Cholesky decomposition, and the QR factorisation [63] – with some of these implementations pre-dating CUDA by at least a year [73, 74, 75].

Even when we restrict our investigation to more recent CUDA implementations the options are numerous, these include the CUBLAS library supplied by NVIDIA, the MAGMA library, and CULA Tools – already discussed in the previous section – as well as others such as the FLAME Project [76]. The majority remain fairly similar in their implementation and calling conventions to standard BLAS and LAPACK [53, 57]. The FLAME Project, however, attempts to rethink the development and programming of linear algebra libraries [77]. As such, it presents a means to express the algorithms used in linear algebra in terms of blocks and provides a set of APIs to allow written code to closely resemble these algorithms. Although this approach is promising and has been applied to a wide range of problems on different architectures (including CUDA GPUs) [77, 78, 79, 80, 81], the commitment of the scientific community to standard BLAS and LAPACK (and their respective optimised implementations discussed in Section 3.1) seems to be impeding the adoption of the FLAME Project.

One of the primary concerns addressed in this chapter is that of the amount of available GPU memory limiting the size of the LU decomposition that can be accelerated using a CUDA device. Positive results for the methods presented here (for the double precision complex case) are reported in [9]. This work also aims to provide a drop-in replacement for the LAPACK or MAGMA routines responsible for performing the LU decomposition, thereby allowing these modified routines to be used with minimal changes to existing application code.

The StarPU runtime system also addresses the concern of limited device memory and thus overlaps with the methods presented here [82]. It is designed specifically for heterogeneous computing environments and allows for the definition of computational tasks (for example, matrix-matrix multiplications and solving triangular linear systems) which are offloaded by a scheduler to various computational devices. Recently, StarPU has been used in conjunction with MAGMA to target CUDA-based GPUs [82, 83], with results for both the LU decomposition and Cholesky factorisation showing significant speedups over CPU only multi-core implementations. However, only real-valued single precision results were presented in the case of the LU decomposition in [82].

Note that implementations such as the one discussed in [84] also allow for the solution of very large problems, but are targeted at distributed environments. In [84], results are shown comparing LU decomposition performance for a BlueGene/P cluster (2048 to 16384 cores), an Opteron cluster (256 to 4096 cores), and a GPU-based cluster consisting of between 8 and 4096 CUDA devices (C1060s and C2050s) with communication implemented using MPI. This contrasts sharply with the implementation and results presented here, where the performance of a single node and a single CUDA device is considered.

A topic of interest that is not discussed here, is the matter of mixed-precision linear system solvers [63]. With such solvers an attempt is made to perform most of a desired implementation in a lower precision than is required in the final solution, and to only perform certain critical sections at full precision. The advantages of this are twofold. Firstly, single precision computation is typically at least a factor two faster than double precision computation, with this performance gap even greater for many CUDA GPUs. Secondly, the amount of memory required in the algorithm can be reduced significantly, which will aid in further overcoming the restrictions imposed by the amount of device memory available. Such a solver is implemented as part of MAGMA for the solution of double precision real systems of equations and this solver exhibits performance comparable to the single precision case [8].

3.3 Overcoming GPU memory limitations for dense LU decomposition

As discussed in Chapter 2, although GPGPU computing shows much promise in many cases, there are a number of limitations that need to be addressed. The foremost of these is the limited memory typically available on these devices. Even in the case of a CUDA GPU installed in a relatively low-end host, the amount of host memory typically exceeds the GPU memory by at least a factor of four. Furthermore, even though the high-end Tesla devices have device memories that can be three or four times as large as a consumer device of the same architecture, the amount of memory installed in the hosts housing these devices tend to be greater than regular desktops by about the same margin, resulting in similar memory constraints.

The aim of this section is to present a CUDA-based implementation of the LU decomposition as provided by the LAPACK `xGETRF` routines discussed in Section 3.1 that overcome these memory limitations. It should be noted that, although CUDA-based implementations of the `xGETRF` routines exist in the MAGMA library [8], these are limited by the amount of memory available on the device.

A logical starting point for dealing with the limited memory available on a GPU is traditional out-of-core (OOC) methods. One such method is presented for the LU decomposition in [62], where a panel-based left-looking LU decomposition is described. Such a process is used in parallel and out-of-core implementations such as ScaLAPACK [85], and is discussed further in Section 3.3.1.

In a typical desktop (or server) computer system, there exists a memory hierarchy with each tier classified according to speed, cost, and volatility [86]. When we consider memories that do not form part of the CPU (cache and registers), two types of memory can be identified. The first is primary or main system memory and the second is secondary storage. Main memory consists of a high-speed volatile memory that is used for the temporary storage of data and results when performing a computation, as well as the code that is being run. The secondary storage consists of non-volatile memory such as magnetic disks or flash-based memory and is used for long term storage of data.

In a modern computer system such as the ones used here for benchmarking (introduced in Section 3.4), typical sizes for the main system memory range in the tens of gigabytes (GBs), with magnetic disks capable of storing hundreds or thousands of gigabytes commonplace. Although the storage capacity of primary memory is significantly less than that of secondary storage devices, its performance is much greater – bandwidth of 20 GB/s and upward versus 600 MB/s in the case of modern magnetic disks. Note that in the case of the latter, measured performance differs greatly from the theoretical peak, with sustained rates measured at tens of megabytes per second (MB/s). The relative size and performance of these memories are summarised in Table 3.4.

Table 3.4: A summary of the storage capacities and bandwidth ranges of typical modern memory technologies for both the host and device memories.

	Capacity	Bandwidth
host main memory	16–96 GB	20–50 GB/s
host secondary memory	300–2000 GB	150–600 MB/s
host–device interconnect	N/A	4–12 GB/s
device global memory	1–6 GB	100–150 GB/s

Also shown in Table 3.4 are typical capacity and bandwidth ranges for CUDA devices, as well as the interconnect (the PCI Express bus) between the device and the host. Note that although CUDA devices have lower memory capacities than the hosts in which they are installed, they do offer a slight memory bandwidth advantage. This advantage is, however, not as great as the one exhibited by host main memory over host secondary memory. For the values given in Table 3.4, for example, the memory bandwidth of a CUDA device's global memory is 2–7.5× higher than for the host's main memory, whereas the host's main memory offers 34–340× as much bandwidth as its secondary memory.

In the case of traditional OOC methods, the primary system memory is too small to meet the memory requirements of the problem being considered. With the LU decomposition considered here, for example, the whole matrix (which has a storage requirement of $\mathcal{O}(N^2)$) being factored cannot reside in main system memory. In the panel-based implementation presented in [62], the matrix is stored in secondary memory and only parts (panels) of the matrix need to be stored in primary memory and operated on in order to complete the operation. In this way, the main memory storage requirement for the LU decomposition is reduced to $\mathcal{O}(N)$. It should be noted that in such implementations the input-output cost also needs to be taken into account [81].

In the CUDA implementation considered here, the algorithms are similar, with the primary difference being that the role of the main system memory is now fulfilled by the device memory. The main system memory is then used in the same way that the secondary memory is used in the traditional out-of-core case. This should not be seen as a GPU-based extension of the OOC implementation, but instead as a mapping of the OOC approach to GPU computing in order to overcome memory limitations. What this does mean is that the implementation presented is still limited by the amount of system memory available, in the same way that traditional OOC implementations were limited by the amount of secondary storage available.

3.3.1 Left-looking LU decomposition

Although the panel-based left-looking variant of the LU decomposition is discussed in some depth in [62], the discussion is summarised here as background to the GPU implementation that follows in Section 3.3. As as starting point, consider an $N \times N$ matrix of double precision complex values $\left[Z\right]$ and a desired panel width $NB > 0$. The matrix is then divided into M panels. The first $M-1$ panels have a width of NB and the last panel has a width of $N-(M-1)NB$. Thus if N is a multiple of NB, the matrix is divided into M panels of width NB. If this is not the case, the last panel will be made up of the remaining columns of $\left[Z\right]$.

The technicalities of the last panel aside (due to N not being a multiple of NB), the process required to perform the LU decomposition on the matrix $\left[Z\right]$ using the panel-based approach is quite simple. Figure 3.1 shows the configuration of the matrix as it has progressed to an arbitrary point, with the panels i and k being designated the temporary and active panels, respectively. In a simple out-of-core implementation, these two panels are the only parts of the matrix $\left[Z\right]$ that need to reside in main system memory [62], and require storage for $2 \times N \times NB$ matrix elements.

Every active panel k needs to be updated using all the panels that have already been processed - these are all panels of the matrix $\left[Z\right]$ to the left of k. For an arbitrary temporary panel i to the left of k, the following operations are required to update the active panel [62]

$$\left[C_{0k}\right]' \longleftarrow \left[T_{0i}\right]^{-1} \left[C_{0k}\right], \tag{3.11}$$

$$\left[C_{1k}\right]' \longleftarrow \left[C_{1k}\right] - \left[T_{1i}\right] \left[C_{0k}\right]', \tag{3.12}$$

$$\left[E_{k}\right]' \longleftarrow \left[E_{k}\right] - \left[D_{i}\right] \left[C_{0k}\right]'. \tag{3.13}$$

Here, $\left[C_{0k}\right]'$, $\left[C_{1k}\right]'$, and $\left[E_k\right]'$ are the matrices $\left[C_{0k}\right]$, $\left[C_{1k}\right]$, and $\left[E_k\right]$ updated in-place as in (3.11), (3.12), and (3.13), respectively. The matrix $\left[T_{0i}\right]$ represents the $NB \times NB$ lower triangular submatrix on the diagonal of $\left[Z\right]$. Furthermore, (3.12) and (3.13) can be combined into a single operation as

$$\begin{bmatrix} C_{1k} \\ E_k \end{bmatrix}' \longleftarrow \begin{bmatrix} C_{1k} \\ E_k \end{bmatrix} - \begin{bmatrix} T_{1i} \\ D_i \end{bmatrix} \left[C_{0k}\right]', \tag{3.14}$$

with the prime superscript once again indicating an in-place update.

Once the update of the active panel k has been completed, by performing the operations described in (3.11) and (3.14) for each of the panels to the left of the active panel ($i < k$), the LU decomposition of $\left[E_k\right]$ is calculated. The resultant row pivots then need to be applied to the submatrix $\left[P_k\right]$, after which the panel to the right of k becomes the active panel and the process is repeated.

In terms of implementation, both (3.11) and (3.14) have equivalent BLAS routines that can be used [53], namely `xTRSM` and `xGEMM`, and were introduced in Section 3.1. For the LU decomposition of the matrix $\left[E_k\right]$ and the application of the row pivots, the LAPACK routines `xGETRF` and `xLASWP` ,respectively, are used (see Section 3.1).

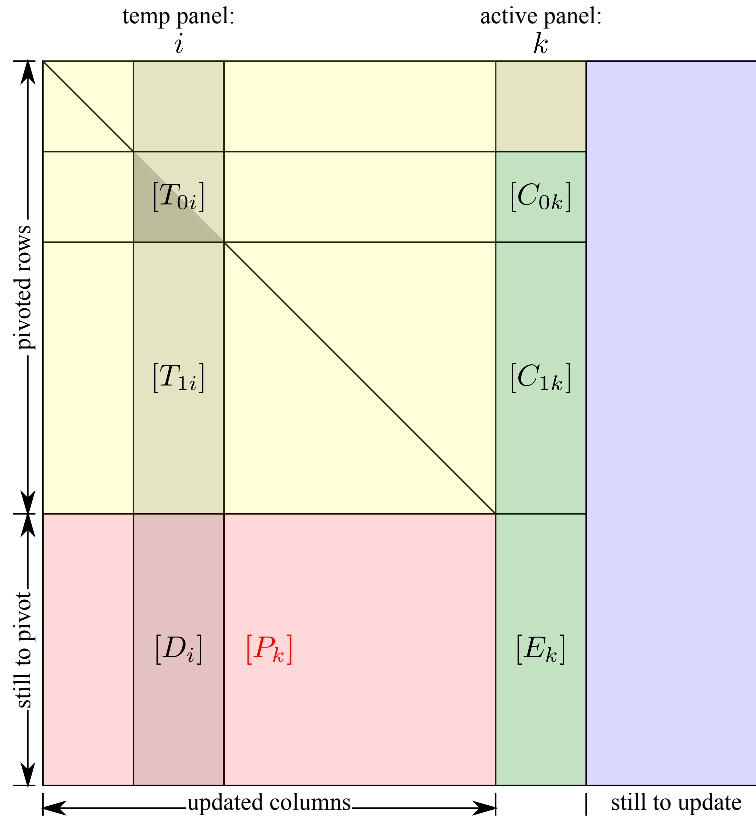


Figure 3.1: A diagram showing the state of a matrix that has been partially factorised (adapted from [62]) as discussed in Listing 3.1. Shown are the temporary and active panels, as well as the submatrices used in (3.11) to (3.13). The areas of the matrix that are up to date are shown in yellow, with the columns of the matrix that have yet to be updated shown in blue. The green areas indicate the parts of the active panel that are involved in the current update step, with the pink submatrix ($[P_k]$) showing the matrix that will be pivoted after the LU decomposition of $[E_k]$ has been calculated.

An overview of the required algorithm is presented in Listing 3.1. It should be noted that for the algorithm as it is presented here, the matrix is stored in pivoted form, whereas in [62] it is stored in unpivoted form.

From timing results presented in [62], it is clear that the four BLAS and LAPACK routines used to perform the updates of (3.11) and (3.14), as well as the LU decompostion of the submatrix $\left[E_k\right]$, contribute significantly to the total runtime of obtaining the LU decomposition of the large matrix $\left[Z\right]$. Thus one would expect that their GPU acceleration will go a long way in improving the overall performance of the decomposition. Other aspects such as the input and output costs associated with the out-of-core implementation summarised here are not considered further. More information on these aspects can be found in [62] and [81].

3.3.2 Adapting for GPU acceleration

As mentioned, the primary objective is to accelerate the LU decomposition using CUDA GPUs (more specifically CUBLAS), while not being limited by the amount of memory installed on the device, but instead by the amount of main memory in the host. Although libraries such as MAGMA [8] do provide CUDA-accelerated `xGETRF` routines for performing the LU decomposition, the amount of device memory remains a limitation.

In the case of the out-of-core left-looking LU decomposition as discussed in Section 3.3.1, the secondary memory of the host is used in conjunction with the main system memory to

Listing 3.1: Algorithm for an out-of-core panel-based left-looking LU decomposition (adapted from [62]).

```
for each panel k:
  1. load panel k into the active panel
  2. apply row pivots to the active panel
  3. for each panel i to the left of k:
    a. load panel i into the temporary panel
    b. perform the update in (3.11)
    c. perform the update in (3.14)
  4. perform the LU decomposition of [E_k]
  5. store the active panel overwriting panel k
  6. apply the row pivots to [P_k]
  ( for each panel i to the left of k:
      a. load panel i into the temporary panel
      b. apply the pivots to the rows of the temporary panel that correspond with
         [P_k]
      c. store the temporary panel overwriting i
  )
```

calculate the LU decomposition of a matrix that is too large to fit into main memory alone. In this section, the implementation is adapted to a CUDA-based LU decomposition, where it is now the device memory that is limited and the entire matrix is able to reside in main system memory. The algorithm for the GPU panel-based left-looking LU decomposition is shown in Listing 3.2, with some notable differences to the out-of-core algorithm of Listing 3.1. The first

Listing 3.2: Algorithm for the panel-based left-looking LU decomposition including GPU acceleration.

```
A. allocate device memory for two panels
B. for each panel k:
  1. apply row pivots to the panel k on the host
  2. transfer panel k to the GPU active panel
  3. for each panel i to the left of k:
    a. transfer panel i to the GPU temporary panel
    b. perform the update in (3.11) on the device
    c. perform the update in (3.14) on the device
  4. transfer the GPU active panel to panel k on the host
  5. perform the LU decomposition of [E_k] on the host
  6. apply the row pivots to [P_k] on the host
C. cleanup device memory
```

of these is that the loading of the active panel – which now equates to transferring the panel to the device – and the application of the pivots to the active panel (panel k) have been switched around. The main purpose of this is to avoid the development of custom CUDA kernels that perform the required row interchanges which cannot be performed efficiently on current GPUs [72], although examples of such routines for transposed matrices are available as part of the MAGMA source code [8] and are discussed in [72, 87]. The row pivots are now applied on the host before transferring the panel to the device. The next difference is that the updated active panel is transferred from the device before the LU decomposition of $[E_k]$ is performed on the host. Lastly, since the entire matrix now resides in host memory, the pivots can be applied to the matrix $[P_k]$ in one step, and not one panel at a time like in the out-of-core implementation.

Since the submatrices to which the updates of (3.11) and (3.14) must be applied now reside in device memory, these are performed using the relevant CUBLAS routines. The LU decomposition of the submatrix $[E_k]$ and the application of the row-pivots, however, are still applied

on the host. This functionality is thus supplied by the LAPACK routines implemented as part of ACML.

3.3.3 The MAGMA-panel-based hybrid

Although the CUBLAS-based implementation discussed in Section 3.3.2 is able to outperform a CPU-based implemented by some margin [9], it is slower than the MAGMA implementation for matrices that can fit into GPU memory. In fact, in [9] it was found that the MAGMA implementation is about 50% faster on the hardware used for testing.

A possible way to achieve performance comparable to MAGMA is to simply check if the matrix for which the LU decomposition must be computed is small enough to factor using MAGMA. If this is the case, then MAGMA is used. If not, then the panel-based implementation of Section 3.3.2 is used. This will, however, result in a drastic performance drop once MAGMA can no longer be used and as such, an improved hybrid scheme is devised and discussed here. The aim of this MAGMA-panel-based hybrid approach is to try to continue to leverage MAGMA's performance once the matrix can no longer occupy device memory.

Consider the representation of a matrix in Figure 3.2, which is similar to Figure 3.1 except that areas pertaining to the hybrid implementation are also highlighted. The lower-right square submatrix, labeled as $\left[E_{MAGMA}\right]$, represents the largest matrix that can be solved using MAGMA alone. If the total matrix size is less than the MAGMA limit – the number of rows or columns of the largest matrix that can be factorised directly using MAGMA – then only MAGMA is used and a hybrid method is not required. The case where the matrix exceeds this limit is now discussed further.

Figure 3.2 shows the MAGMA panel labelled as m. This is the first panel for which the number of remaining columns to the right (including this columns of the panel itself) is less than or equal to the MAGMA limit. In contrast to the panel-based implementation already discussed, where the LU decomposition of each $\left[E_k\right]$ is performed by the CPU, the MAGMA-panel-based hybrid implementation uses MAGMA to perform the LU decomposition of the entire submatrix $\left[E_{MAGMA}\right]$ in one step. Thus the CPU-based LU decomposition is only used for factorising $\left[E_k\right]$ if the active panel is to the left of the MAGMA panel ($k < m$). Since MAGMA has been found to be faster than the CPU version, especially for large matrices, its use give the hybrid method a performance boost over the standard panel-based approach.

For active panels to the left of panel m, the algorithm is identical to the standard panel-based implementation given in Listing 3.2, and requires no further discussion. If, however, this is not the case ($k \geq m$), the update procedure is adjusted slightly. Such panels need to be updated as in (3.11) and (3.14), but only for the temporary panels to the left of the MAGMA panel ($i < m$). Following this, the LU decomposition of $\left[E_{MAGMA}\right]$ is calculated, and the matrix to the left of m ($\left[P_{MAGMA}\right]$) is pivoted in host memory accordingly. The algorithmic representation of this hybrid approach is given in Listing 3.3 and will now be discussed in more detail.

It should be noted that while the panel-based implementations discussed in these sections make use of the left-looking variant of the LU decomposition, the MAGMA implementation of `xGETRF` uses the right-looking BLAS level 3 version of the algorithm [8]. This means that internally, MAGMA also makes use of CUDA-based BLAS `xGEMM` and `xGETRF` routines – although these may not be the CUBLAS implementations.

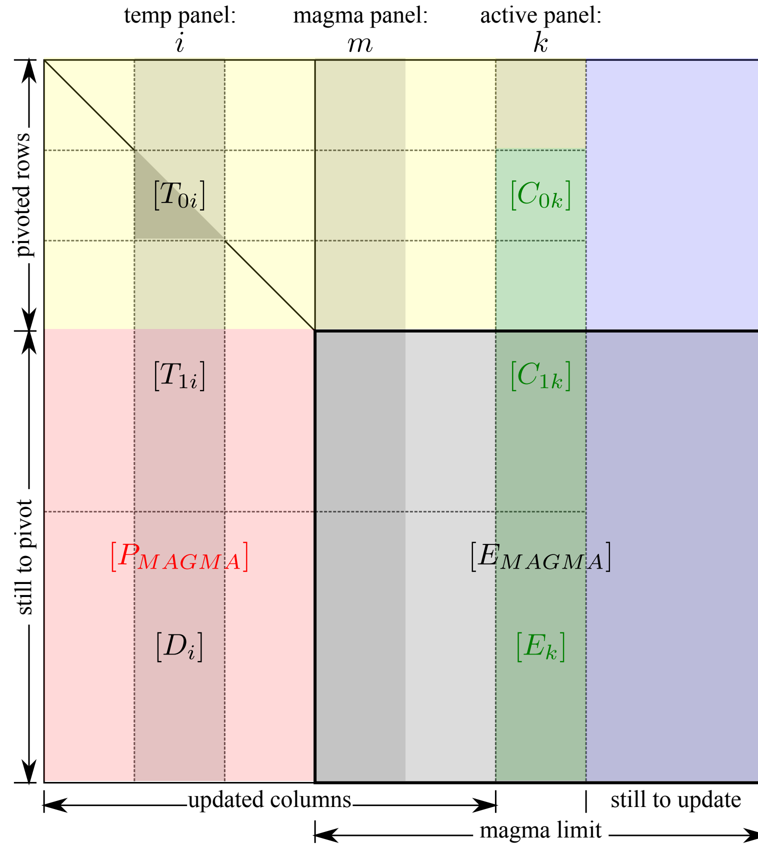


Figure 3.2: A diagram showing the state of a matrix that has been partially factorised using the MAGMA-panel-based hybrid algorithm as given in Listing 3.3. The active and temporary panels (shown) are the same as for the matrix given in Figure 3.1. The MAGMA panel, as well as the matrix $[E_{MAGMA}]$ used in this version of the algorithm are also indicated. As in Figure 3.1, the areas of the matrix that are up to date are shown in yellow, with the columns of the matrix that have yet to be updated shown in blue. The green areas indicate the parts of the active panel that are involved in the current update step, with the pink submatrix ($[P_{MAGMA}]$) showing the matrix that will be pivoted after the LU decomposition of $[E_{MAGMA}]$ (in this case) has been calculated using MAGMA.

Listing 3.3: Algorithm for the MAGMA-panel-based hybrid LU decomposition that uses CUBLAS and MAGMA routines as a means of GPU acceleration.

```
A. allocate GPU memory
B. for each panel k:
   1. apply row pivots to the panel k on the host
   2. transfer panel k to the GPU active panel
   3. for each panel i to the left of k and m:
      a. transfer panel i to the GPU temporary panel
      b. perform the update in (3.11) on the device
      c. perform the update in (3.14) on the device
   4. transfer the GPU active panel to panel k on the host
   5. if k is to the left of m:
      a. perform the LU decomposition of [E_k] on the host
      b. apply the row pivots to [P_k] on the host
C. perform the LU decomposition of [E_MAGMA] using MAGMA
D. apply the row pivots to [P_MAGMA] on the host
E. cleanup GPU memory
```

3.3.4 Implementation analysis

As already mentioned, the aim of the panel-based CUDA implementations discussed in Section 3.3.2 and Section 3.3.3 is to overcome the restrictions imposed by the limited memory available on CUDA devices. Although it will never be possible to completely overcome these limitations – as the amount of memory installed will never be infinite – it is hoped that a GPU-based implementation would at least be able to handle matrices that are able to reside in main system memory.

For the purpose of this discussion, consider a device with 1 GB of memory installed in a host with 16 GB of main system memory. The largest full matrices that can be stored for these two memory sizes and all the BLAS data types (see Table 3.2) are given in Table 3.5. Note that since the memory required is $\mathcal{O}(N^2)$, with N the number of rows or columns in the square matrix, multiplying the amount of memory available by 16 only results in an increase by a factor four in N. Conversely, to store a matrix with twice as many rows and columns, one would need four times as much memory.

Table 3.5: The size of the largest $N \times N$ matrix that can be stored in 1GB and 16GB of memory for different element types. The element types are indicated using the BLAS convention shown in Table 3.2. Also shown are the sizes (in bytes) of each element.

	S (4 bytes)	C (8 bytes)	D (8 bytes)	Z (16 bytes)
1 GB	16384×16384	11585×11585	11585×11585	8192×8192
16 GB	65536×65536	46340×46340	46340×46340	32768×32768

To achieve the stated goal of being able to factor matrices that can fit into system memory (16 GB) using the panel-based implementations on the GPU (with 1 GB of memory), the size of the largest matrices that can be solved using the panel-based approaches must at least equal the size in the 16 GB row of Table 3.5. The maximum matrix sizes that can (in theory) be handled by the panel-based approaches on devices with 1 GB of memory and a panel width of 256 are given in Table 3.6 for the four different data types considered.

From the table, it is clear that the sizes of the matrices that can be solved using the panel-based implementations by far exceed the sizes of the matrices that can occupy main system memory in each case for the panel size chosen. The panel width ($NB = 256$) was chosen after some experimentation on the testing hardware used and may require some adjustments for good performance on other platforms. Using narrower panels will further increase the size of the matrices that can be solved with the panel-based methods, but will also increase the number

Table 3.6: Summary of the maximum matrix sizes that can be solved using a panel-based approach on a CUDA GPU with 1 GB of memory and a panel width of $NB = 256$. Also shown is the amount of memory required to store the full matrix, the number of panels the matrix would be split into and the square matrix limits for 16 GB of memory from Table 3.5 (for comparison).

	matrix size	memory required[(1)]	number of panels	16 GB size limit
S	524288×524288	1024 GB	2048	65536×65536
C	262144×262144	512 GB	1024	46340×46340
D	262144×262144	512 GB	1024	46340×46340
Z	131072×131072	256 GB	512	32768×32768

(1) for the storage of the full matrix.

of panels used, which may affect performance due to increased input-output costs [81]. Wider panels, on the other hand, will reduce the size of the problems that can be solved (although according to Table 3.6 there is still a lot of leeway) and should reduce the input-output costs [81]. This may have the consequence of reducing the effectiveness of the MAGMA-panel-based hybrid implementation as the switch to MAGMA is currently made on a panel boundary (see Section 3.3.3).

When considering a MAGMA-only implementation (or the calculation of the MAGMA limit), the matrices that can be solved using MAGMA alone have sizes similar to those presented in the 1 GB row in Table 3.5 (assuming 1 GB of device memory) since the entire matrix needs to occupy device memory. In practice, MAGMA requires some additional work space allocated on the device [8], and as such the matrices that can be handled are smaller by a few hundred elements in each dimension. Further, since the MAGMA-panel-based hybrid approach has the same limitations as the regular panel-based approach, it also serves to extend MAGMA's functionality to matrices that it would not normally be able to factor.

3.4 Benchmarking

In the brief discussion on CUBLAS and MAGMA in Section 3.1.1, the performance characteristics of each have already been mentioned and as such are not discussed further. In this section performance results for both the CUBLAS panel-based implementation of Section 3.3.2 and the MAGMA-panel-based hybrid implementation of Section 3.3.3 are presented. These results aim to illustrate two things, namely that the implementations are successful in overcoming the GPU memory limitations, and that this can be done while still maintaining a performance advantages over a CPU-based implementation. A further aim is to show that the MAGMA-panel-based hybrid implementation adds to the performance of the standard CUBLAS panel-based approach.

The performance of implementations of all the `xGETRF` routines is investigated. This is done for random matrices ranging from 1024×1024 to 32768×32768 in size. These sizes are chosen so as to exceed the matrix sizes that are able to be accommodated by 1 GB of memory while still occupying less than 16 GB for double precision complex matrices (see Table 3.5).

For a subset of the test runs, an LU backward error measure is calculated as

$$err_{LU} = \frac{\left\| \left[P\right] \left[A\right] - \left[L\right] \left[U\right] \right\|}{N * \left\| \left[A\right] \right\|}, \tag{3.15}$$

with the matrices as discussed in Section 3.1 (see specifically (3.4)) and N the size of $\left[A\right]$. This error measure is the same metric used for verification by MAGMA [8]. The $\left\|\cdot\right\|$ operator indicates the matrix Frobenius norm and is implemented by a call to the `xLANGE` LAPACK routines.

It should be clear from (3.15) that the computation of this error requires that the original matrix ($\left[A\right]$) be stored. Furthermore, additional storage space is required to compute the matrix product $\left[L\right]\left[U\right]$ and as such the error cannot be computed in-core for matrices whose storage requirements exceed about a third of the installed system memory. For this reason, the errors for some of the larger matrices are calculated on the test system with more memory installed.

The performance measure used is the number of floating-point operations performed per second (FLOPS) in the computation of the LU decomposition of the input matrix. For real valued matrices the number of floating-point operations required for the LU decomposition of an $N\times N$ real matrix is given by [63]

$$n_{\text{fp,real}} = \frac{2N}{3}. \tag{3.16}$$

Since complex multiplications and additions require more floating-point operations, the number of operations for a complex matrix is given by [8]

$$n_{\mathrm{fp,complex}} = \frac{8N}{3}. \tag{3.17}$$

The number of floating-point operations performed per second in the LU decomposition is then given by

$$\mathrm{GFLOPS} = \frac{n_{\mathrm{fp},\cdot}}{t_{\mathrm{LU}}} \times 10^{-9}, \tag{3.18}$$

with t_{LU} the measured wall-clock time in seconds for the LU decomposition (the call to the `xGETRF` routine) and $n_{\mathrm{fp},\cdot}$ given by either (3.16) or (3.17), depending on the data type of the matrix. Note that the post-multiplication by 10^{-9} results in the calculated operation rate being in billions of operations per second (GFLOPS).

It should be pointed out, that in order to allow for a fair comparison between CPU and GPU-based implementations, timing is performed at the last common point between the various implementations. This means that the CUDA timing results include all data transfer and initialisation costs.

3.4.1 The test platforms

A summary of the three systems used for benchmarking and testing purposes is given in Table 3.7. These systems offer some variety in terms of computational hardware. The first system is a 5-year old workstation that was originally used as a computational server for small- and medium-sized problems. An NVIDIA consumer graphics card, a GeForce GTX 280, was later added for the purpose of GPU computing research. This card represented a high-end device at the time and retailed for around \$400. The cost of the original system was in the region of \$6000.

The second system represents a relatively current mid to high-end consumer-level desktop system, and although the specifications of the components are similar to those of the first system, the price was significantly lower. The system was purchased for around \$1600 and included a high-end AMD consumer graphics card (\$400 alone). This card was later replaced by an NVIDIA card, the GeForce GTX 465, which represents a mid-range consumer card with a retail price of about \$300.

The final system is a high-end node in a compute cluster and offers significantly more processing power and memory than the first two systems. The approximate purchase price for the system was \$11000 and includes two Tesla M1060 compute devices. As mentioned in Chapter 2, these Tesla devices are effectively GPUs with more memory and no graphics capabilities. In the tests conducted, only a single M1060 was used.

All the test systems run a 64 bit installation of Linux, the first two systems run Ubuntu and the third SUSE. The BLAS and LAPACK functionality for the tests is provided by ACML (version 4.4.0), which was introduced in Section 3.1. Freely available GNU compilers are used and include `gcc`, `g++`, and `gfortran`.

3.4.2 Results

This section presents the benchmarking results for the `SGETRF`, `CGETRF`, `DGETRF`, and `ZGETRF` routines on the three systems described in Section 3.4.1. Unless otherwise stated, the benchmarks are conducted on random matrices ranging from 1024×1024 to 32768×32768 elements in size. In each successive benchmarking step, the number of rows and columns of each matrix are increased by 512.

A figure is included for each test system and data type combination, with four performance curves shown per figure. These correspond with results obtained using a single core of the CPU

Table 3.7: Summary of the hardware used for testing

	System 1	
	AMD Opteron 275	**GeForce GTX 280**
architecture	AMD64	GT200
number of cores	4 (2 dual-cores)	240 SPs (30 SMs)
clock speed	2.2 GHz	1.4 GHz
memory	16 GB	1 GB
release year	2005	2008
	System 2	
	AMD Phenom II X4 945	**GeForce GTX 465**
architecture	AMD64	GF100
number of cores	4	352 SPs (11 SMs)
clock speed	3.0 GHz	1.22 GHz
memory	16 GB	1 GB
release year	2009	2010
	System 3	
	Intel Xeon X5550	**Tesla M1060**
architecture		GT200
number of cores	8 (2 quad-cores)	240 SPs (30 SMs)
clock speed	2.66 GHz	1.3 GHz
memory	48 GB	4 GB
release year	2009	2008

installed in the host (labelled **ACML 1-core** and provided as a baseline and not run for all steps of 512), using all the cores available on the host (**ACML 4-core** in the case of System 1 and System 3 and **ACML 8-core** for System 3), using the CUBLAS panel-based approach of Section 3.3.2 (**CUBLAS panel-based**), and lastly the MAGMA-panel-based hybrid implementation (**MAGMA-panel hybrid**) discussed in Section 3.3.3. Each performance curve shows the measured performance (in GFLOPS) as a function of the number of rows (and columns) in the matrix. The theoretical limits for matrices of the specified data type that can occupy 1 GB of memory (as in Table 3.5) and 4 GB of memory (where applicable) are indicated by vertical black lines labelled as **1 GB** and **4 GB**, respectively.

SGETRF

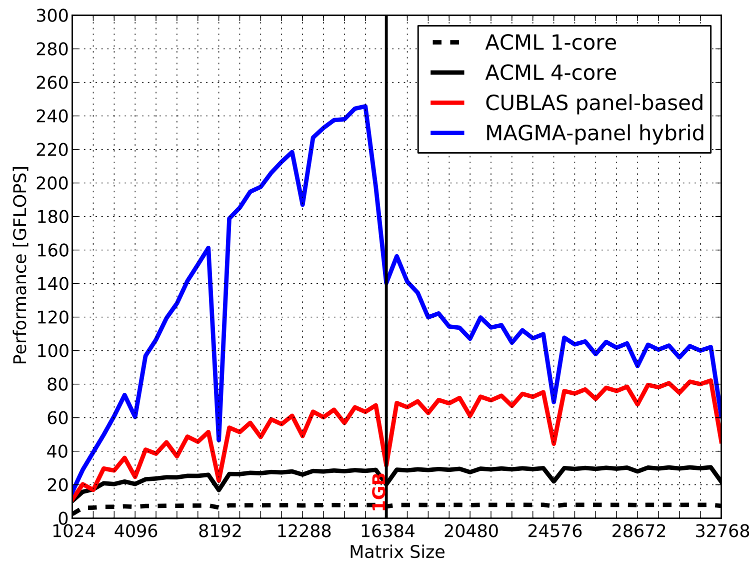


Figure 3.3: Measured SGETRF performance of **System 1** as a function of matrix size for random input matrices. Also shown is a vertical line indicating the 1 GB size limit for single precision real matrices. The 4 GB size line is not shown here as it corresponds with a matrix size of 32768×32768 which is the limit of the horizontal axis.

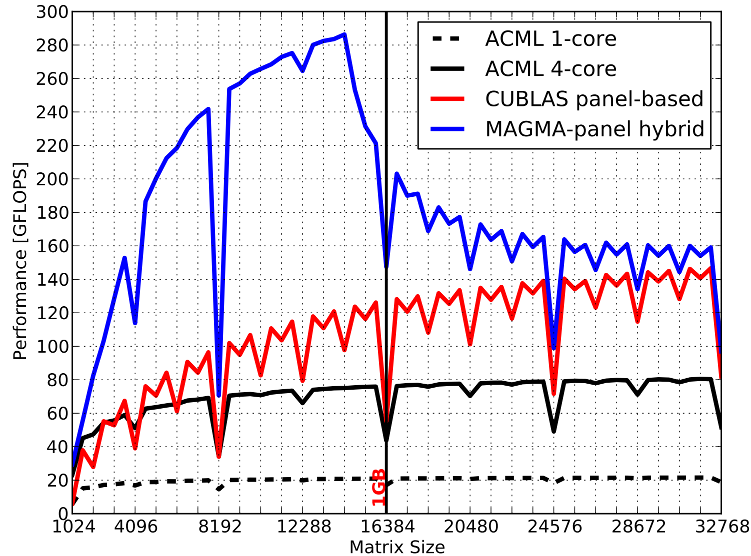


Figure 3.4: Measured SGETRF performance of **System 2** as a function of matrix size for random input matrices. Also shown is a vertical line indicating the 1 GB size limit for single precision real matrices. The 4 GB size line is not shown here as it corresponds with a matrix size of 32768×32768 which is the limit of the horizontal axis.

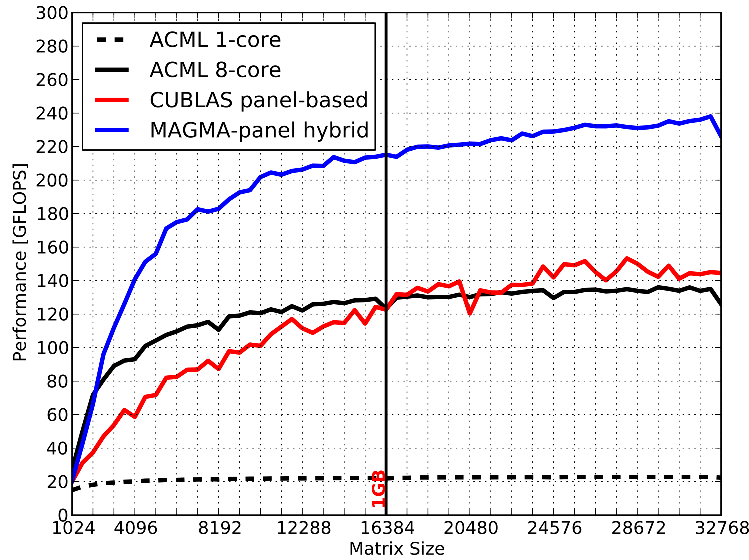


Figure 3.5: Measured `SGETRF` performance of **System 3** as a function of matrix size for random input matrices. Also shown is a vertical line indicating the 1 GB size limit for single precision real matrices.

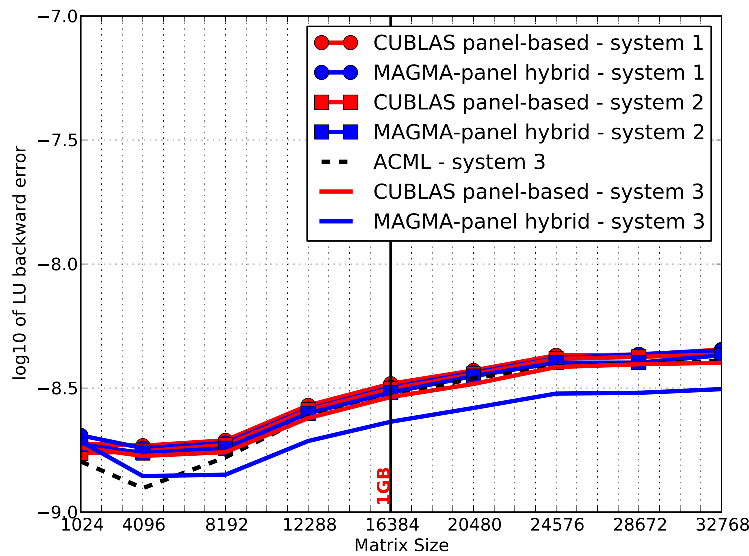


Figure 3.6: $\log_{10}$ of the `SGETRF` error measure (see (3.15)) as a functions of matrix size for random input matrices on the three test systems considered. ACML-based CPU results are only shown for **System 3**. Also shown is a vertical line indicating the 1 GB size limit for single precision real matrices. The 4 GB size line is not shown here as it corresponds with a matrix size of 32768×32768 which is the limit of the horizontal axis.

CGETRF

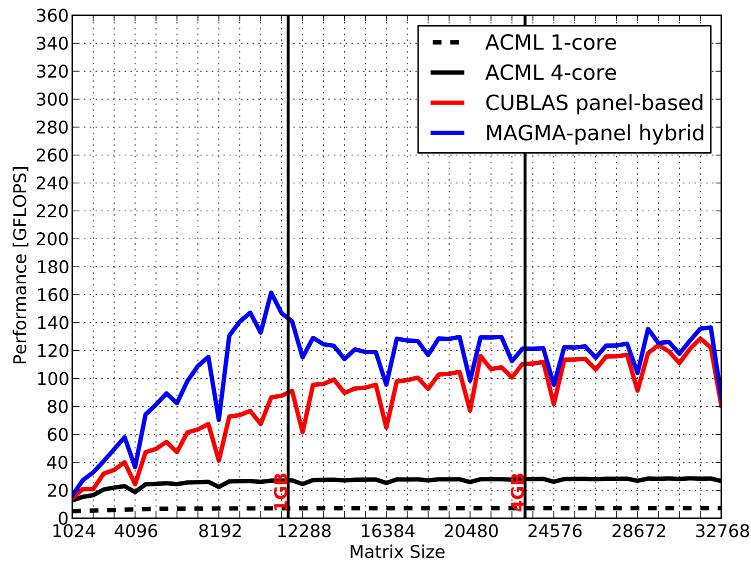


Figure 3.7: Measured CGETRF performance of **System 1** as a function of matrix size for random input matrices. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for single precision complex matrices.

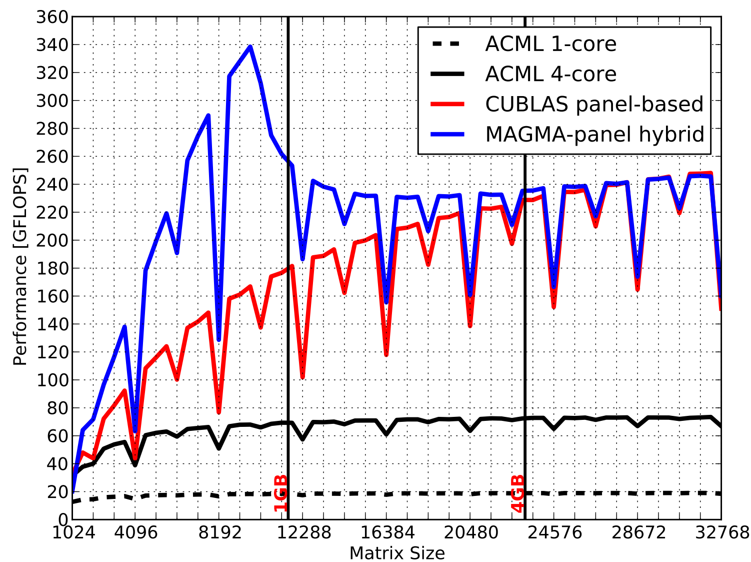


Figure 3.8: Measured CGETRF performance of **System 2** as a function of matrix size for random input matrices. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for single precision complex matrices.

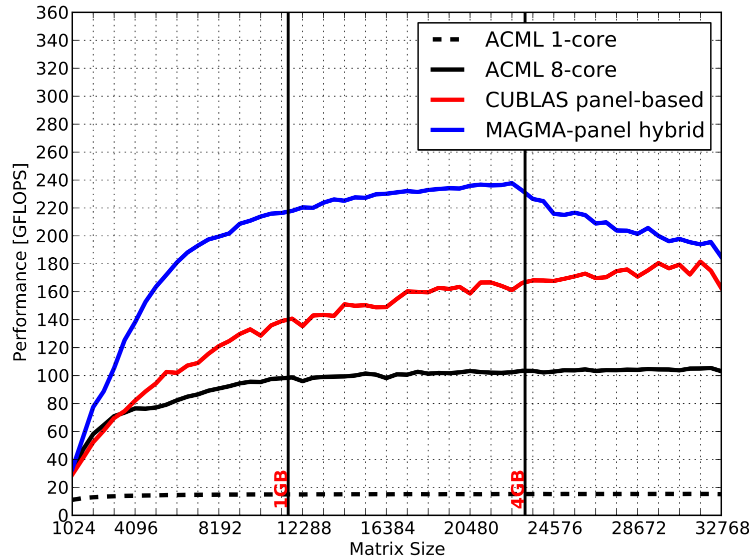


Figure 3.9: Measured `CGETRF` performance of **System 3** as a function of matrix size for random input matrices. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for single precision complex matrices.

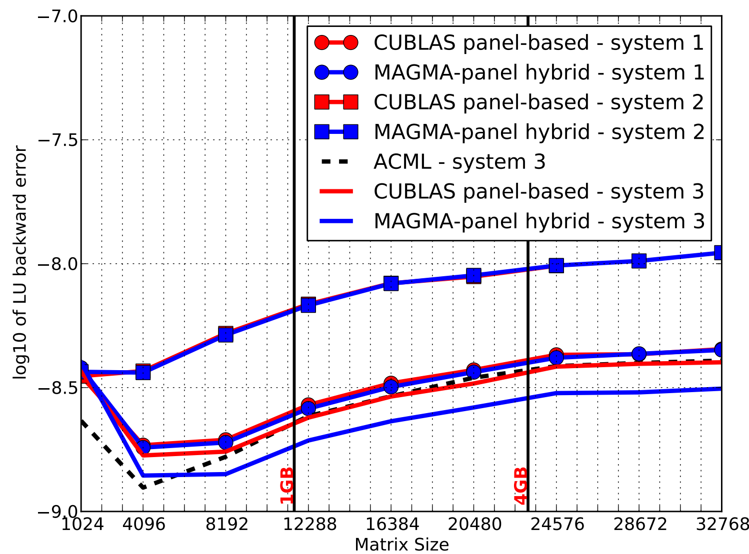


Figure 3.10: $\log_{10}$ of the `CGETRF` error measure (see (3.15)) as a functions of matrix size for random input matrices on the three test systems considered. ACML-based CPU results are only shown for **System 3**. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for single precision complex matrices.

DGETRF

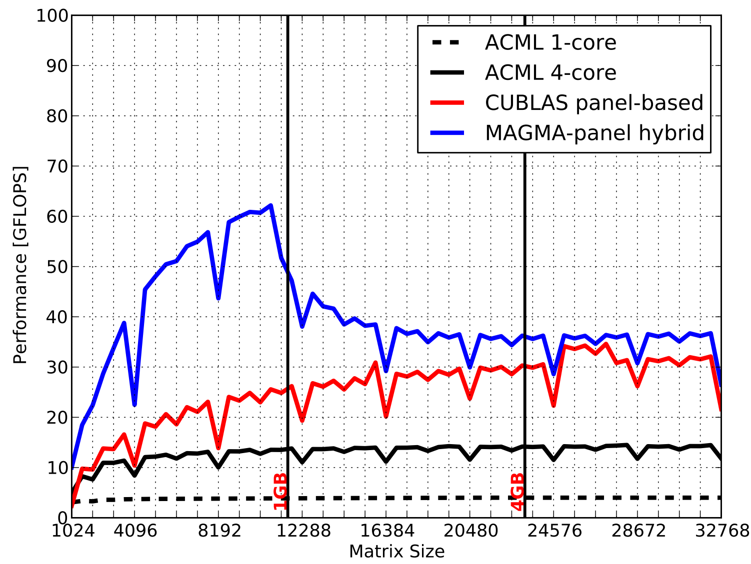


Figure 3.11: Measured `DGETRF` performance of **System 1** as a function of matrix size for random input matrices. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for double precision real matrices.

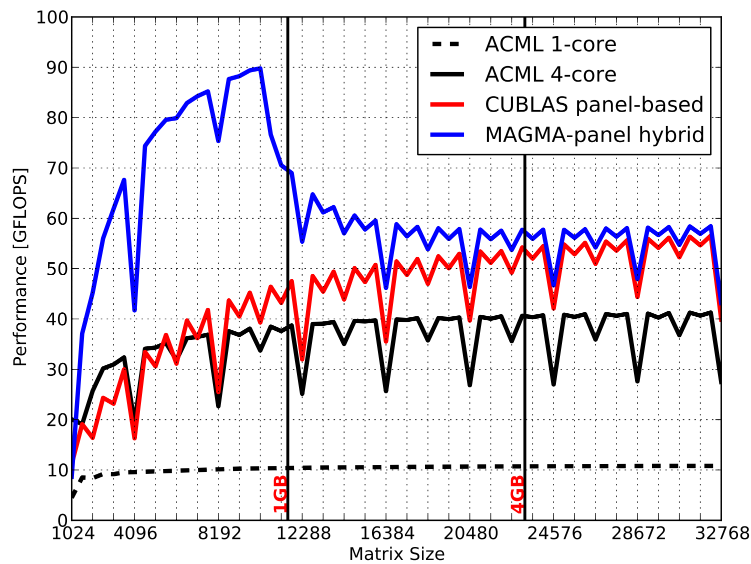


Figure 3.12: Measured `DGETRF` performance of **System 2** as a function of matrix size for random input matrices. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for double precision real matrices.

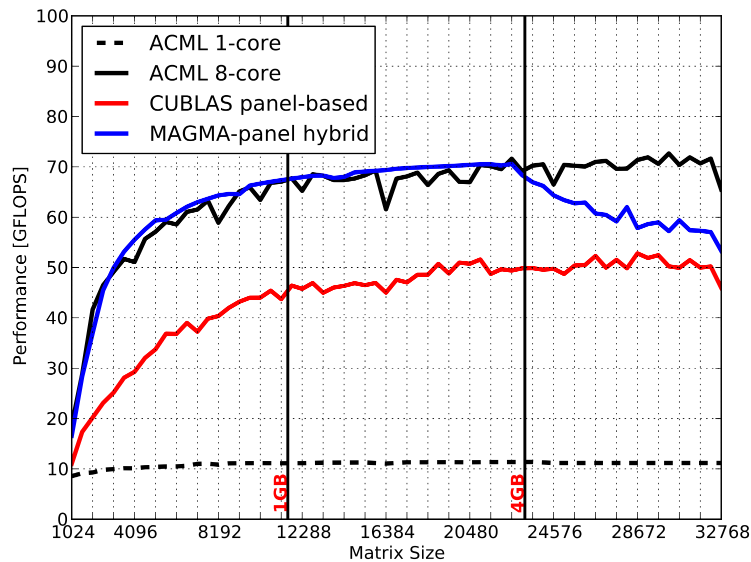


Figure 3.13: Measured DGETRF performance of **System 3** as a function of matrix size for random input matrices. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for double precision real matrices.

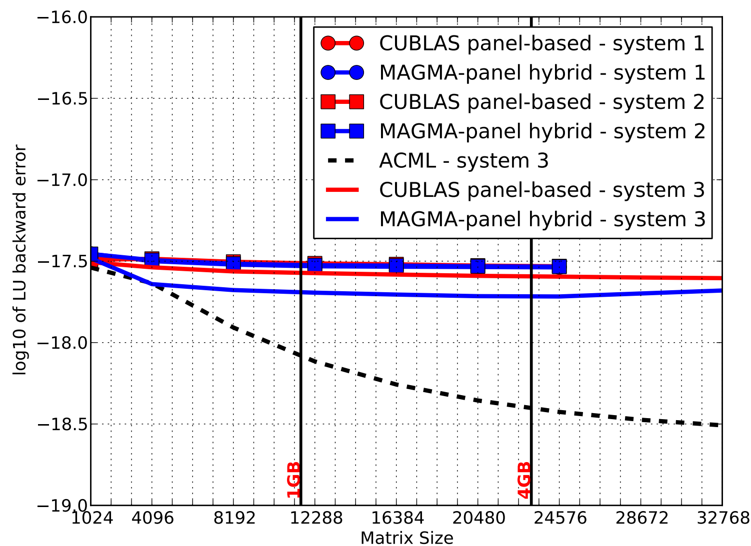


Figure 3.14: $\log_{10}$ of the DGETRF error measure (see (3.15)) as a functions of matrix size for random input matrices on the three test systems considered. ACML-based CPU results are only shown for **System 3** and results for **System 1** and **System 2** are only computed to 24576×24576 elements due to memory limitations of the hosts. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for double precision real matrices.

ZGETRF

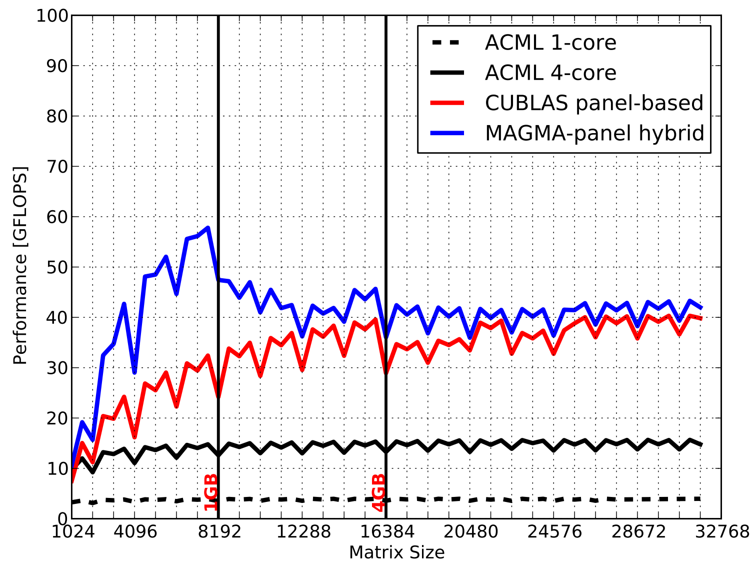


Figure 3.15: Measured `ZGETRF` performance of **System 1** as a function of matrix size for random input matrices. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for double precision complex matrices. Note that results are only shown for matrices up to 31744×31744 elements in size since the system ran out of memory beyond this point.

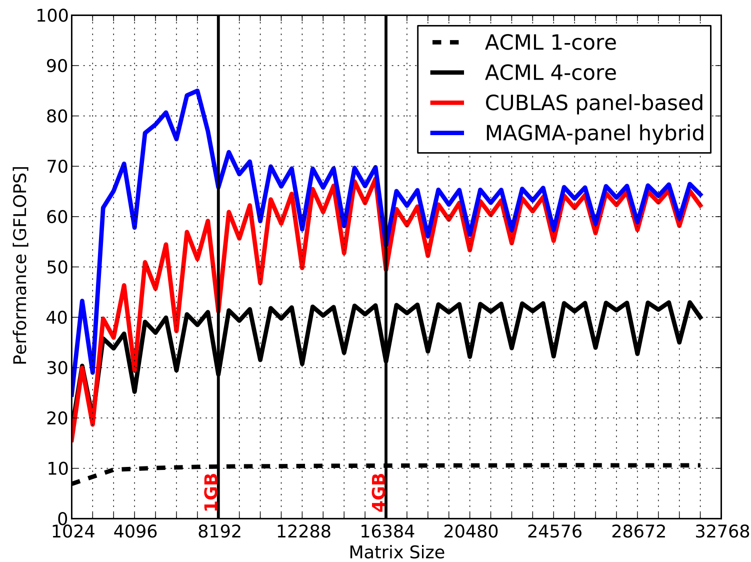


Figure 3.16: Measured `ZGETRF` performance of **System 2** as a function of matrix size for random input matrices. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for double precision complex matrices. Note that results are only shown for matrices up to 31744×31744 elements in size since the system ran out of memory beyond this point.

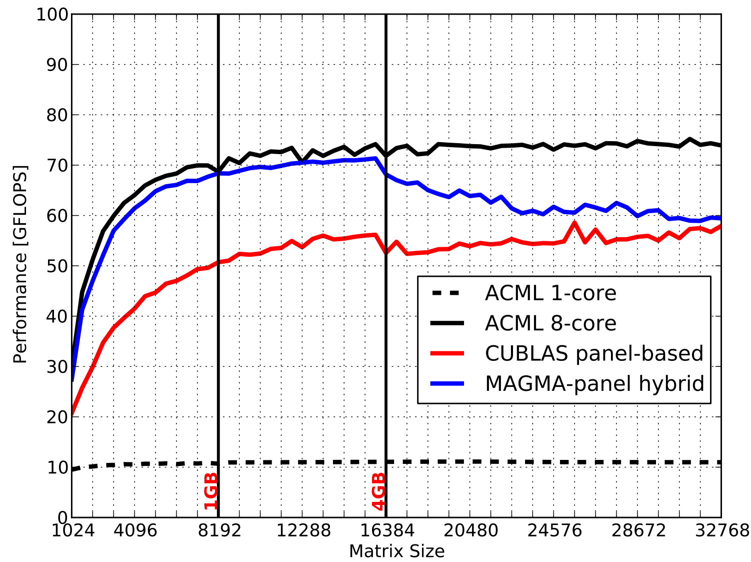


Figure 3.17: Measured ZGETRF performance of **System 3** as a function of matrix size for random input matrices. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for double precision complex matrices.

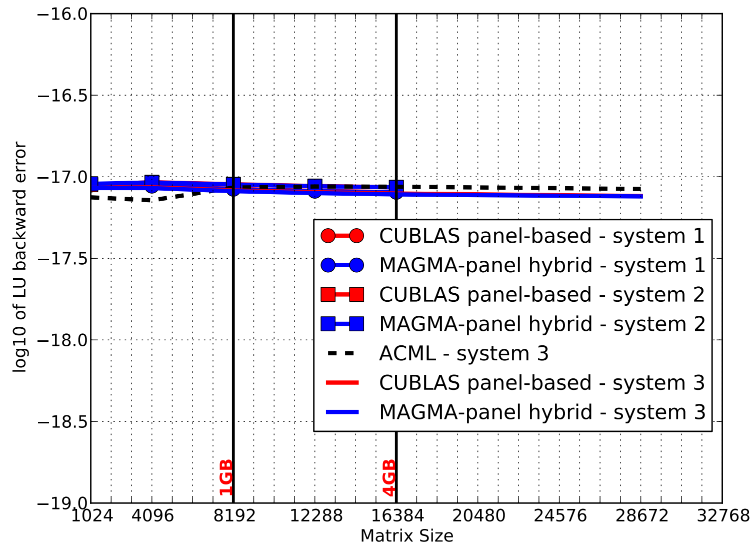


Figure 3.18: $\log_{10}$ of the ZGETRF error measure (see (3.15)) as a functions of matrix size for random input matrices on the three test systems considered. ACML-based CPU results are only shown for **System 3** and results for **System 1** and **System 2** are only computed to 16384×16384 (28672×28672 for System 3) elements due to memory limitations of the hosts. Also shown are vertical lines indicating the 1 GB and 4 GB size limits for double precision complex matrices.

A discussion of the results

A number of interesting points are illustrated by the performance results presented here. When considering the CPU-based ACML results alone, it is clear that there have been a number of advances made in terms of processor architectures over the past five years. This is illustrated by the more than doubling in performance when comparing the results (single and quad-core) from System 1 and System 2. This improvement cannot be attributed to clock speed alone, since this would only account for less than 40% improvement. The multi-core ACML implementation also offers a clear advantage over the single core one – even in the case of the Intel architecture of System 3.

In terms of GPU-based performance, the results follow a similar trend to previously published ones (see for example, the MAGMA documentation [8]) with GPU-based implementations outperforming ACML for most of the cases considered. The exceptions here are for the double precision results obtained on System 3 – where the peak GPU performance is at best able to equal the performance of the eight-core ACML implementation. This can be attributed to the relatively poor double precision performance (when compared to single precision performance) of the GT200-based M1060 used, since there is only one double precision floating-point unit per SM (see Section 2.3.2). This is no longer the case with the latest GF100-based devices such as the Tesla C2050, as is clearly demonstrated in [70].

One use case highlighted by the results that lends itself well to the server environment (for which System 3 is designed), is its use as a shared resource in larger system with mixed computational requirements, such as a computational cluster at a department of a university. In this case it is entirely possible to run two single-core jobs, each utilising one of the installed GPUs, and possibly attaining performance comparable to that of all eight cores while a number of other jobs are also utilising the remaining CPU resources. This hypothetical situation is obviously also affected by the availability of other resources such as memory and bus bandwidth.

The panel-based implementation of Section 3.3.2 is able to outperform multi-core ACML on System 1 and System 2. On all systems, the panel-based implementation is further outperformed by MAGMA, as is expected since the amount of communication between the host and the device is drastically reduced. The MAGMA-panel-based hybrid implementation of Section 3.3.3 achieves its goal of improving the performance of the panel-based implementation once the amount of memory available on the device has been exceeded. As the matrices get larger, and the percentage of the matrix that can be accommodated in device memory decreases, the advantage of the hybrid implementation over the standard panel-based implementation is reduced.

In the case of System 1, the results presented show that with the addition of a relatively cheap consumer graphics card, performance comparable to more up-to-date systems can be achieved. For the GTX 280 used in System 1, this is especially true for single precision floating-point performance, where the MAGMA-only results for this system are significantly faster than the ACML (CPU-based) results for the more modern System 2 and System 3. In the case of double precision computation, the GPU-enabled routines are able to at least equal the ACML results for System 2. As already discussed with regards to the GPU-based performance of System 3, the use of a GF100 device (such as the GTX 465 used in System 2) should result in much improved double precision performance, as is illustrated by the `DGETRF` and `ZGETRF` results for System 2.

3.5 Conclusion

In this chapter, an introduction to dense numerical linear algebra has been presented. The BLAS and LAPACK libraries, as well as their use in performing matrix operations such as matrix-matrix products and LU decompositions were discussed, with CUDA-based libraries – CUBLAS and MAGMA – implementing the desired functionality introduced. The use of these

libraries in the realisation of two panel-based LU decomposition implementations (adapted from a traditional out-of-core implementation) was presented, allowing the methods to be limited by the amount of system memory installed instead of the amount of memory available on the device.

Using these implementations, a comparison of CUDA GPU-based performance and CPU-based performance using the ACML library was conducted on three different systems used for testing. In all cases, the GPU-based performance results are significantly higher than single-core ACML results. In the case of single precision (`SGETRF` and `CGETRF`) results, the GPU implementations were between 2× and 12× faster than the multi-core results obtained on the respective systems. Even when double precision (`DGETRF` and `ZGETRF`) results are considered, the CUDA implementations were faster than the multi-core implementations on two of the three systems, with only the high-end eight-core server system being able to best the performance of the CUDA GT200 device used.

As expected for the two panel-based implementations considered, the MAGMA-panel-based hybrid implementation performs significantly better than the standard CUBLAS-based implementation especially for problem sizes where the matrix can occupy device memory (and only MAGMA is used) as the amount of data transferred between the host and the device is greatly reduced. It should be noted that the hybrid method continues to provide a performance advantage over the CUBLAS implementation once the matrix can no longer occupy device memory in its entirety.

As stated at the start of the chapter, one of the aims in the benchmarking of GPU-based linear algebra routines on the three systems considered is to ascertain whether a GPU can be used as a drop-in upgrade to extend the usefulness of ageing computing hardware. For the LU decomposition considered here, this is the case, and the older system is able to at least match the performance of the others in most of the variants considered. Considering that the GT200 architecture employed on the card used for these tests has itself been superseded by the GF100 architecture (as discussed in Section 2.3.2), a second GPU upgrade should further improve performance.

One of the shortcomings in the current implementation is that it makes use of a fixed panel size for all matrix sizes. As illustrated in [82], the optimal panel size for block-based algorithms is dependent on problem size. Although this can be extended to the panel-based implementations considered here, this issue has not been addressed at present. Another factor to consider is the overlapping of CPU and GPU computation in the panel-based approach to allow for a better utilisation of all the computational resources installed in the system.

The work presented here, provides simple CUDA-based implementations of the LU decomposition that are directly compatible with LAPACK. These provide a quick fix to developers wishing to add GPU capabilities to existing code, with the added benefit of performance improvements. That said, the implementations are not necessarily optimal – as illustrated by the difference in performance between the two implementations discussed – and the use of an operation scheduler and runtime such as StarPU [82] – recently used in combination with MAGMA [83] – should be considered. This further has the advantage of supporting multiple CUDA devices in a single host (as was the case for one of the testing systems). This would provide additional upgrade options for existing systems. The older system used here, for example, supports up to two CUDA devices even though only one is currently installed.

Chapter 4

MOM scattering analysis

The Method of Moments (MOM) or boundary element method (BEM) is a widely used technique in computational electromagnetics. Its ability to handle infinite domains exactly is one of its strengths and makes it ideal for modelling radiation and scattering problems[88, 89]. The method also does not require that free-space be discretised, greatly improving the modelling efficiency of such problems when compared to other full wave methods such as the Finite Difference Time Domain method (FDTD) or the Finite Element Method (FEM) [88].

Like the FEM, the MOM is a matrix-based method and the solution of a typical radiation or scattering problem results in a dense linear system which must be solved [88]. Due to its dense nature, methods such as those discussed in Chapter 3 can be used to implement the desired routines, and more importantly, provide GPU acceleration using CUDA. In this chapter a discussion on the GPU acceleration of the parts of the MOM solution process not related to solving the linear system is presented, although this is included for the purpose of performance analysis. This accelerated implementation is then applied to the computation of the radar cross section of a simple scatterer.

Since the focus of this work is not on the actual formulation of the Method of Moments, it starts with an existing formulation and documents the process of developing both CPU- and GPU-based implementations of the formulation, with the latter utilising CUDA. For this purpose, the seminal paper presented by Rao, Wilton, and Glisson [10] is used as a starting point. That work saw the discussion of the use of triangular patches and basis functions – named the Rao-Wilton-Glisson (RWG) basis functions – defined on pairs of patches in the discretisation and subsequent solution of vector surface current densities on arbitrarily shaped scatterers. The selection of [10] as a starting point is further justified by the use of the methods and the RWG basis functions in subsequent publications such as [90] and [91] as well as their implementation in commercial software packages such as FEKO [92].

The relatively simple framework of [10] also allows for a number of other issues to be addressed without spending too much time lost in the details of the formulation. Most notable of these is the introduction of a co-development method used here, where much of the detailed implementation sections are shared between the CPU and CUDA versions. This allows for the development tools – such as debuggers and profilers – of both targets to be leveraged in order to obtain a stable, accurate implementation. Furthermore, once the framework is in place, the addition of new functionality that is targeted at both platforms is almost trivial and as such the framework plays an important role in the development and maintenance process. This is especially true if it is applied to larger more complex development problems.

Although prior works on the acceleration of the Method of Moments – specifically the assembly of the matrices used in the solution process – do exist and are discussed in Section 4.1, this work is one of the first to utilise the double precision hardware support of the GT200 and later CUDA hardware for performing the matrix assembly in double precision. Furthermore,

the implementation presented here is of such a nature that the amount of GPU memory available on the CUDA devices used does not limit the size of the problems that can be addressed. When this is combined with the MAGMA-panel-based hybrid LU decomposition introduced in Chapter 3, the result is a CUDA-accelerated Method of Moments solution process that offers speedups over the entire range of problem sizes considered. These problem sizes exceed the limitations imposed by the device memory for 1 GB and 4 GB devices for both single and double precision computations (see Section 3.3.4).

As already shown in the previous chapter, one of the aims of this work is to investigate the feasibility of low-cost GPU upgrades of older systems as an alternative to system replacement. For this reason, the relative performance of the three systems introduced in Section 3.4.1 is once again considered. The comparative benchmarks presented indicate that – as is the case for the LU decompositions of Chapter 3 – the oldest system using GPU acceleration is able to best the multi-core CPU-only performance of at least one, if not both, of the newer systems for all the problems considered.

This chapter commences by discussing a number of previous works concerning the GPU acceleration of the Method of Moments in Section 4.1. The sample problem considerd, as well as an overview of the Method of Moments solution process are presented in Section 4.2. To this end, the formulations of [10] are touched upon, with specific attention paid to the matrix assembly. The co-development process and details pertaining to both the CUDA and multi-core CPU implementations of the MOM are given in Section 4.3, with verification and performance results presented in Section 4.4 and Section 4.5, respectively. Section 4.5 also includes a discussion on the implications of the performance results, with the general conclusions of the chapter presented in Section 4.6.

4.1 Related work

Although a number of prior works that make use of GPUs in the acceleration of the Method of Moments exist, the work presented here has many novel aspects. Some of the earliest sources can be found in [90] and [91], where the solution of electromagnetic scattering problems – specifically the radar cross section (both monostatic and bi-static in the case of [91] and only bi-static in [90]) – using the electric field integral equation (EFIE) formulation of the MOM is considered. Although both works use the same Rao-Wilton-Glisson basis functions applied to triangular patches with a conjugate gradient method used to solve the resultant linear system, they do not employ CUDA for the implementation of the GPU accelerated portions of the code and instead rely on Brook and the OpenGL Shading Language, respectively.

One of the first examples of the application of CUDA to the Method of Moments is in [89], where the method is applied to the solution of boundary value problems associated with the Helmholtz equation in acoustics and electromagnetics. Scattering problems are, however, not considered. The hardware used in [89] is based on the G80 architecture discussed in Section 2.3.2 and as such does not support native double precision computation. For this reason the double-single approach of [93] is followed, where two single precision values are used in an attempt to improve the precision of the implementation. The matrix assembly routine also forms part of an iterative solution process, where the matrix elements are calculated on the fly for use in a matrix-vector product.

In [94], mention is also made of a CUDA-based EFIE solver in the analysis of a two-dimensional transverse-magnetic (TM) cylinder, however, not much detail is given regarding the implementation or the exact formulation used. Here, a conjugate gradient method is also used as part of the solution process.

In [95], [96], and [97], direct LU decomposition-based solvers are applied to the linear system associated with the Method of Moments. Although the authors of [95] are the only ones

that specifically address double precision computation, they do so by considering only the acceleration of the complex double precision LU decomposition (which is discussed in detail in Chapter 3). This is then included as part of the solution process of the MOM analysis of a simple one-dimensional wire antenna. Both [96] and [97] consider the matrix assembly and LU decomposition phases in their publications, with [97] using MAGMA to provide the required accelerated LU decomposition routines.

As just mentioned, the work in [97] uses MAGMA to perform the LU decomposition required as part of the solution process and also considers the construction of the impedance matrix, but the study is restricted to single precision floating-point computations by the hardware used. In addition to this, roof-top functions are applied to the problems involving microstrip patch antennas and thus the calculations reduce to the computation of quasi-one-dimensional integrals [98]. It should be noted that the authors of both [95] and [97] have also shown results for the GPU acceleration of the finite difference time domain (FDTD) technique in [99] and [100], respectively, although only the latter uses CUDA.

Although the work of [96] also presents a detailed discussion of the CUDA acceleration of the Method of Moments as applied to electromagnetics scattering problems, these are restricted to the analysis of two-dimensional TM problems in single precision using pulse basis functions. In this case the use of single precision is not due to the hardware used as the GT200-based device used does support double precision. The implementation of the LU decomposition is dealt with at a very low level, with the authors opting to implement their own algorithm for subsequent GPU acceleration. No comparative results with existing – even CPU-based – methods are provided.

The work presented here is the evolution of other research on the topic, most notably [11] and [9] where the matrix assembly and LU decomposition phases of the Method of Moments are considered separately. The discussion of the methods and results of the latter have already been dealt with in Chapter 3, with the implementation and extension of the functionality supplied by CUBLAS and MAGMA. In both cases, the double precision performance of the respective phases are considered, with an overview of their combination, as well as a more detailed discussion on the implementation details of the matrix assembly phase given in [12]. This chapter then sees an extension of the methods presented in these publications in order to realise its aims.

As a final point, it should be noted that although algorithmic acceleration methods for the Method of Moments, such as the multi-level fast multi-pole method (MLFMM) exist, these are not applicable to all practical problems and a standard MOM implementation is often required [88]. These methods are not considered further here, although their CUDA acceleration has shown much promise [101].

4.2 Problem overview

This section aims to lay the groundwork for the discussion of the multi-core CPU and CUDA implementations discussed in Section 4.3. To this end, the sample problem considered is presented, and an overview of the Method of Moments discussed. For the latter, the seminal 1982 paper by Rao, Wilton, and Glisson [10] forms the basis of the discussion.

Although the sample problem and the theoretical framework used are each relatively simple, they are well suited to this particular chapter as they allow for the focus to be directed at the implementation details, instead of complicating matters with the intricacies of the mathematical formulations. The sample problem further illustrates the application of the CUDA-based LU decomposition of Chapter 3 – where only random input data was considered – to the matrices associated with the Method of Moments solution process.

4.2.1 Monostatic scattering

In order to develop the discussion regarding the acceleration of the Method of Moments using NVIDIA CUDA GPUs, a simple problem is considered – that is the monostatic scattering of a square PEC plate. Such a plate, measuring one wavelength on each side and located in the xy-plane, is shown in Figure 4.1. Also shown is an electric field with an arbitrary linear polarisation with propagation vector $\vec{k}$. The same configuration (with $\vec{k} = -k\hat{z}$) is used in [10] and as such allows for the verification of the computed results.

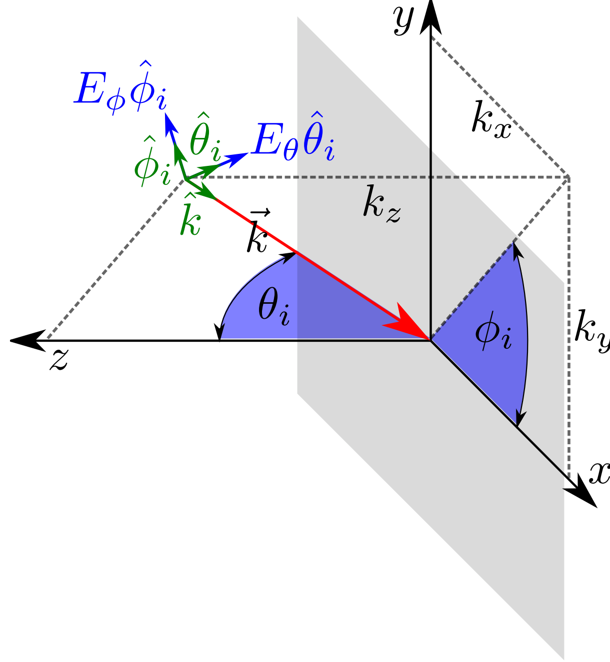


Figure 4.1: A diagram showing an incident plane wave ($\vec{E}_{inc}(\vec{r}) = (E_\theta \hat{\theta}_i + E_\phi \hat{\phi}_i)e^{j\vec{k}\cdot\vec{r}}$) with arbitrary linear polarisation and propagation vector ($\vec{k} = k_x\hat{x} + k_y\hat{y} + k_z\hat{z} = k\hat{k}$). The definitions of the angles (blue arcs) used in the calculation of the radar cross section of an object located at the origin, and the square PEC plate (grey) considered in this chapter are also shown. Note the unit vectors $\hat{k}$, $\hat{\theta}_i$, and $\hat{\phi}_i$, that define the incident field terms of the spherical coordinate system.

One quantity of interest in scattering problems such as these, is the radar cross-section (RCS) of the object. Its calculation involves determining the ratio of scattered power density to the incident power density at a given frequency, and gives an indication of how visible an object is to radar from a specific angle [88, 102]. It can also be defined in terms of the incident ($\vec{E}_{inc}$) and scattered ($\vec{E}_s$) electric fields as [102]

$$\sigma_{RCS}(\theta_s, \phi_s) = \lim_{R\to\infty} 4\pi R^2 \frac{|\vec{E}_s(\theta_s, \phi_s)|^2}{|\vec{E}_{inc}(\theta_i, \phi_i)|^2}, \tag{4.1}$$

with $R = |\vec{r}|$ the distance to the observation point and (θ_i, ϕ_i) and (θ_s, ϕ_s) defining the angle of incidence and angle of observation, respectively. The incident field – assuming harmonic time dependence at a single frequency (f) – has a position dependent phase and is given by [10]

$$\vec{E}_{inc}(\vec{r}) = (E_\theta \hat{\theta}_i + E_\phi \hat{\phi}_i)e^{j\vec{k}\cdot\vec{r}}, \tag{4.2}$$

with $\vec{k}$ the propagation vector of the field as shown in Figure 4.1. This propagation vector is simply the propagation constant (k) at the frequency of the incident field multiplied by a unit

vector in the direction of propagation and is defined as

$$\begin{aligned}\vec{k} &= k(\sin\theta_i\cos\phi_i\hat{x} + \sin\theta_i\sin\phi_i\hat{y} + \cos\theta_i\hat{z}) \\ &= \frac{2\pi f}{c_0}(\sin\theta_i\cos\phi_i\hat{x} + \sin\theta_i\sin\phi_i\hat{y} + \cos\theta_i\hat{z}).\end{aligned} \tag{4.3}$$

The vector along which the scattered field is evaluated can similarly be defined as

$$\vec{r} = R(\sin\theta_s\cos\phi_s\hat{x} + \sin\theta_s\sin\phi_s\hat{y} + \cos\theta_s\hat{z}). \tag{4.4}$$

In the case of the monostatic RCS (as is considered here), the direction of propagation of the incident field and the direction in which the scattered field is measured (or calculated) is the same ($\theta_i = \theta_s$ and $\phi_i = \phi_s$) [102], with such a calculation or measurement usually performed over a range of incident/measurement angles.

4.2.2 The Method of Moments

Scattering problems such as the one described in Section 4.2.1 can be solved numerically using the electric field integral equation (EFIE) formulation of the Method of Moments [103]. A simplified block diagram of the MOM solution process is given in Figure 4.2. As seen in the figure, the Method of Moments consists of five main steps

- Initialisation
- Calculation of the impedance matrix
- Calculation of the excitation vector
- Solving for the unknown current
- Post processing

with each of the steps offering its own challenges and potential for improving performance when considering GPU acceleration. It should be noted that not all the steps in the solution process contribute equally to the total time required to obtain a solution – as will be demonstrated in Section 4.5 – and as such their relative importance in terms of GPU acceleration is also affected. In many cases, the assembly of the impedance matrix as well as solving for the unknown current contribute most significantly to the computational requirement and are thus the primary focus of this investigation.

The block diagram also illustrates that some parts of the Method of Moments process are independent of the angle of incidence for a given frequency of the electric field. This gives an indication as to where data can be reused to improve the computational efficiency of the solution. Consider, for example, the calculation of the impedance matrix. This matrix is independent of incident angle, and thus if the RCS is being calculated over a range of angles, it is only necessary to assemble it once. Furthermore, if a technique such as the LU decomposition (see Section 3.1) is used to solve the resultant linear system, the factorisation can be reused for each of the angles of incidence. This is discussed further in subsequent sections.

The phases of the MOM process, and specifically their GPU acceleration using CUDA, are now discussed. Some phases, such as the matrix assembly phases, are discussed in more detail than others. A number of texts are available providing more information on all the phases. These include a good overview of the method in [88] and more details pertaining to implementation in [88, 103].

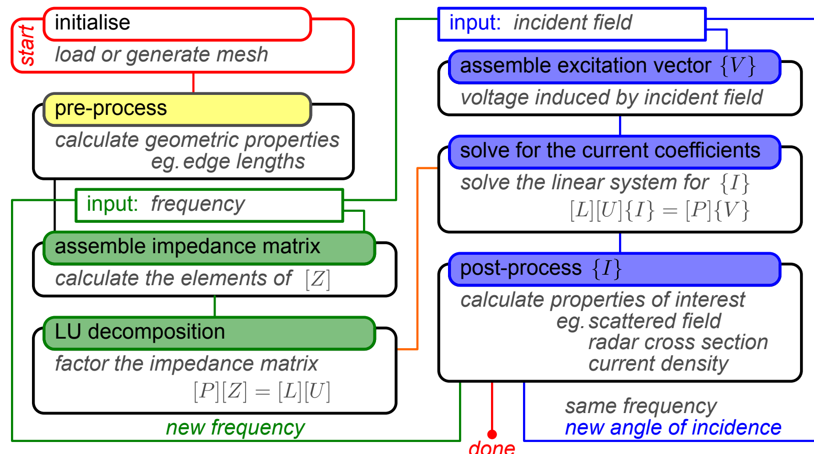


Figure 4.2: A block diagram showing the steps involved in obtaining the Method of Moments solution for a typical scattering problem. The steps in the left column are not dependent on the incident field and only need to be performed again if the frequency of the incident field changes. The steps in the right-hand column are dependent both on frequency and the angle of incidence of the applied electric field. The initialisation step shown bordered in red at the top left of the figure does not form part of the timed process for obtaining the results of Section 4.5.

Matrix assembly

As an introduction to the matrix assembly process, the basis functions introduced in [10] and discussed in a number of sources including [88, 103] are reviewed. To this end, consider the diagram in Figure 4.3 that depicts two adjacent triangles in a triangulation (mesh) of a dielectric surface and the nth edge that is shared by them. In the MOM solution process such a triangulation is typically obtained as part of the initialisation step (shown in Figure 4.2) and can either be created from scratch using some definition of the surface to be meshed, or in more complex cases loaded from a mesh generated by external CAD software.

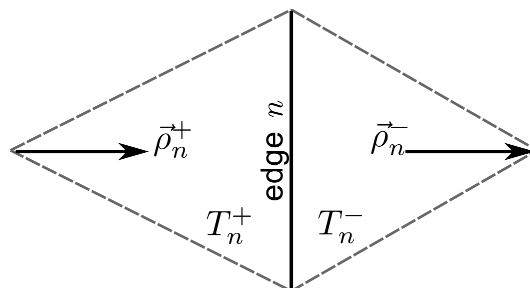


Figure 4.3: A diagram of a free edge in a surface mesh as well as the designation of the positive and negative triangles. Also shown are the vectors used in the definition of the basis function associated with the edge.

Choose one of the triangles associated with the nth edge as the positive triangle, and denote it as T_n^+. The other triangle is the negative triangle T_n^-. The basis function associated with the nth edge, $\vec{f}_n(\vec{r})$, at any point $\vec{r}$ in space is then given by [10]

$$\vec{f}_n(\vec{r}) = \begin{cases} \frac{l_n}{2A_n^+}\vec{\rho}_n^+(\vec{r}), & \vec{r} \text{ in } T_n^+ \\ \frac{l_n}{2A_n^-}\vec{\rho}_n^-(\vec{r}), & \vec{r} \text{ in } T_n^- \\ 0, & \text{otherwise.} \end{cases} \tag{4.5}$$

Here, l_n is the length of the nth edge, and A_n^+ and A_n^- are the areas of the positive and negative triangles associated with the edge, respectively. The vector $\vec{r}$ is the position where the basis function is to be evaluated and the $\vec{\rho}_n^{\pm}(\vec{r})$ vectors are calculated as the difference between $\vec{r}$ and their associated nodes opposite edge n, with orientations as indicated in Figure 4.3. This basis function then represents a unit current density flowing across the edge, and is only non-zero if $\vec{r}$ falls in the face of one of the triangles that share the edge.

With the basis functions defined, the vector current density on the surface of the mesh can be discretised and approximated by the weighted sum of the basis functions of all the triangles on the surface as follows [10]

$$\vec{J}(\vec{r}) = \sum_{n=1}^{N} I_n \vec{f}_n(\vec{r}), \tag{4.6}$$

where N is the number of degrees of freedom (DOFs). The number of DOFs is equal to the number of non-boundary edges in the surface triangulation of the geometry in the case considered here. Solving for the unknown coefficients I_n is the goal of the MOM solutions phases highlighted in grey in Figure 4.2.

For the scattering problem considered, these unknown coefficients are dependent on the incident electric field, $\vec{E}_{inc}$ of (4.2), and the geometry of the problem. Thus the problem of finding the unknown current coefficients can be written as [10, 103]

$$\left[Z\right]\left\{I\right\} = \left\{V\right\}, \tag{4.7}$$

where $\left\{I\right\}$ is an N-vector containing the unknown coefficients I_n of (4.6), $\left\{V\right\}$ is a voltage excitation vector dependent on the incident electric field, and $\left[Z\right]$ is an impedance matrix – the assembly of which will now be discussed. The solution of this linear system and the assembly of the excitation vector will be considered briefly in subsequent sections.

Although more detailed derivations for the expressions of the matrix elements is presented in [10], only the key points are summarised here. In summary, the matrix element equations are obtained by applying a Galerkin testing procedure to an equation of the time harmonic electric field in terms of a scalar electric ($\Phi(\vec{r})$) and magnetic vector potential ($\vec{A}(\vec{r})$). The required boundary conditions – such as tangential electric field continuity on a PEC surface [104] – are also applied. Since the technicalities of this derivation are not the focus of this research, many of the expressions from [10] are used verbatim.

Using the basis function definitions of (4.5), the entries (Z_{mn}) of the impedance matrix $\left[Z\right]$ in (4.7) can be calculated. In the classic RWG formulation, this entry is approximated as the combination of the magnetic vector potentials as well as the electric scalar potentials at the centres of the triangles associated with the mth edge as a result of the current density associated with the nth edge [10], expressed mathematically as

$$Z_{mn} = l_m \left[\frac{j\omega}{2} \left(\vec{A}_{mn}^{+} \cdot \vec{\rho}_m^{+} + \vec{A}_{mn}^{-} \cdot \vec{\rho}_m^{-} \right) - \left(\Phi_{mn}^{+} - \Phi_{mn}^{-} \right) \right]. \tag{4.8}$$

As such, it is required to integrate over both triangles associated with edge n. The integrands have the form

$$\vec{A}_{mn}^{\pm} = \frac{\mu}{4\pi} \int_{T_n^{\pm}} \vec{f}_n(\vec{r}) \frac{e^{-jk|\vec{r}_m^{c\pm} - \vec{r}|}}{|\vec{r}_m^{c\pm} - \vec{r}|} dT_n^{\pm}, \tag{4.9}$$

and

$$\Phi_{mn}^{\pm} = \mp \frac{1}{4\pi j\omega\epsilon} \int_{T_n^{\pm}} \frac{l_n}{A_n^{\pm}} \frac{e^{-jk|\vec{r}_m^{c\pm} - \vec{r}|}}{|\vec{r}_m^{c\pm} - \vec{r}|} dT_n^{\pm}, \tag{4.10}$$

for the magnetic vector potentials and electric scalar potentials, respectively. The vectors $\vec{r}_m^{c\pm}$ are the position of the centre of the positive and negative triangles of edge m and the integrals are a result of a two-point approximation applied to the integration over these triangles [10].

Note that part of the integrands in (4.9) and (4.10) are

$$G_0(\vec{r}_m^{c\pm}, \vec{r}) = \frac{e^{-jk|\vec{r}_m^{c\pm} - \vec{r}|}}{|\vec{r}_m^{c\pm} - \vec{r}|}, \tag{4.11}$$

and have the form of the free-space scalar Green's function [104]. Furthermore, since the influence of all the edges on each other must be considered, the computational and memory demands of calculating the $N \times N$ impedance matrix $\left[Z\right]$ are $\mathcal{O}(N^2)$.

Assembling the excitation vector

As discussed in the preceding section, where the expression for the elements of the impedance matrix was introduced, each matrix elements can be defined in terms of the RWG basis functions defined for an edge in the surface mesh, certain geometric properties of this edge, and the free-space scalar Green's function of (4.11). A similar expression for the elements of the excitations vector $\left\{V\right\}$ can be obtained, where in [10] the entry associated with the mth edge is given as

$$V_m = \frac{l_m}{2} \left(\vec{E}_m^+ \cdot \vec{\rho}_m^{\,+} + \vec{E}_m^- \cdot \vec{\rho}_m^{\,-} \right). \tag{4.12}$$

Here, l_m is once again the length of the edge, and the vectors $\vec{\rho}_m^{\,+}$ and $\vec{\rho}_m^{\,-}$ are from the basis function definition in (4.5). The electric field vectors $\vec{E}_m^{\pm}$ are defined in terms of the centres of the triangles of the mth edge ($\vec{r}_m^{c\pm}$) – as is the case in (4.9) and (4.10) – and the incident electric field of (4.2) as

$$\begin{aligned} \vec{E}_m^{\pm} &= \vec{E}_{inc}(\vec{r}_m^{c\pm}) \\ &= (E_\theta \hat{\theta}_i + E_\phi \hat{\phi}_i) e^{j\vec{k}\cdot\vec{r}_m^{c\pm}}. \end{aligned} \tag{4.13}$$

It should be noted that the expression for (4.12) is also a result of a two-point approximation to the integral over the triangles associated with edge m, after applying the Galerkin testing procedure. Since the computation must be performed once for each DOF, the storage and computational requirements of the excitation vector assembly are both $\mathcal{O}(N)$.

Solution of the linear system

As already mentioned, the Method of Moments reduces to solving the linear system of (4.7) for the vector of unknown current coefficients $\left\{I\right\}$ [103]. Furthermore it follows from (4.8) that $\left[Z\right]$ is, and by extension $\left\{I\right\}$ is also, complex. Such a linear system can be solved using the complex valued LU decomposition, as described in Section 3.1 with (3.4) – for $\left[Z\right]$ instead of $\left[A\right]$ – given here for clarity

$$\left[P\right]\left[Z\right] = \left[L\right]\left[U\right]. \tag{4.14}$$

Furthermore, since the impedance matrix is independent of the angle of incidence, the factorisation in (4.14) can be reused for multiple incident fields of the same frequency.

The LU decomposition and a number of CUDA-based GPU accelerated implementations were discussed in some detail in Chapter 3 and as such are not discussed further, although the implementations are used to obtain the results presented in Section 4.4 and Section 4.5. The unknown current coefficients can be found from the factorisation of the impedance matrix – using the `CGETRF` and `ZGETRF` routines – and the calculated voltage excitation vector.

Calculating the RCS

Since the radar cross section is dependent on both the incident and scattered fields as in (4.1), the first step in its calculation is to determine $\vec{E}_s(\vec{r})$ for a given incident field $\vec{E}_{inc}$. This can be done by substituting the current density discretisation of (4.6) into the electric field radiation integral [104]. This integral can then be evaluated using the calculated basis function coefficients $\{I\}$ [102].

Another approach discussed in both [102] and [103] ,and used here, is the equivalent dipole model. Here, each RWG basis function – associated with the nth edge and triangle pair – is replaced by an infinitesimal dipole with a dipole moment $\vec{m}_n$ given by

$$\vec{m}_n = l_n I_n (\vec{r}_n^{c-} - \vec{r}_n^{c+}), \tag{4.15}$$

where l_n is the length of edge n, and I_n is the calculated current coefficient (element of $\{I\}$) associated with the edge. The vectors $\vec{r}_n^{c\pm}$ are the coordinates of the centres of triangle $T_n^{\pm}$.

After defining

$$C_n = \frac{1}{r_d^2}\left[1 + \frac{1}{jkr_d}\right], \tag{4.16}$$

and

$$\vec{M}_n = \frac{(\vec{r}_d \cdot \vec{m}_n)\vec{r}_d}{r_d^2}, \tag{4.17}$$

the radiated (scattered) electric field for the given dipole is calculated as

$$\vec{E}_s^n(\vec{r}_d) = \frac{\eta}{4\pi}\left((\vec{M}_n - \vec{m}_n)\left[\frac{jk}{r_d} + C_n\right] + 2\vec{M}_n C_n\right) e^{-jkr_d}, \tag{4.18}$$

with $\vec{r}_d$ the vector from the dipole centre to the point where the electric field is to be calculated and $r_d = |\vec{r}_d|$ is the magnitude of this vector. In order to take the translation of the dipole into account, the substitution

$$\vec{r}_d = \vec{r} - \frac{1}{2}(\vec{r}_n^{c-} + \vec{r}_n^{c+}), \tag{4.19}$$

is made, giving an expression for the contribution to the scattered electric field by the nth edge at the point $\vec{r}$ in space. The other variables and constants in above equations are the wavenumber of the incident field k, and the wave-impedance of free-space η.

The total scattered electric field at a point $\vec{r}$ is then obtained from the superposition of the contributions from the various DOFs and is calculated as

$$\vec{E}_s(\vec{r}) = \sum_{n=1}^{N} \vec{E}_s^n(\vec{r}), \tag{4.20}$$

where the substitution of (4.19) has been made into (4.18). The calculated scattered field of (4.20) can now be substituted into (4.1), where, in practice, R is simply chosen far enough away from the PEC surface.

It should be noted that although the dipole approximation does not make any far-field assumptions, it does not perform well when the measurement distance r_d is of a similar order as the length of the mesh, or when the edge length is large compared to a wavelength [102, 103].

4.3 The implementation of an accelerated MOM solution process

For the panel-based and MAGMA-panel-based hybrid implementations of Chapter 3, a largely off-the-shelf approach was used. This was supported by the popularity and subsequent availability of CUDA-accelerated linear algebra routines such as the matrix-matrix multiplication and

the triangular system solve (all supplied by NVIDIA in CUBLAS [7]). The prolific nature of these routines result in a lot of attention from the GPU programming community – including NVIDIA themselves.

In the case of the matrix assembly and other phases of the methods of moments, this approach is not possible and the development of more of the low-level GPU code is necessary. Although there are other similar implementations, as discussed in Section 4.1, each of these deviate from the desired functionality in some way and thus cannot be used as is. This does, however, allow us to address another important issue in the development of GPU accelerated code – that of GPU and CPU-based co-development.

Thus the development of a CUDA-accelerated MOM routine that implements the RWG formulation of [10] is the focus of this section. Furthermore, the interdependence and co-development of CPU- and GPU-based code is considered throughout the discussion of this implementation. Since the matrix assembly phase is the most costly it will be considered in the most detail and used to guide the discussion of the more general aspects of the implemetation. The other phases of the MOM solution process, with the exception of the LU decomposition already discussed in Chapter 3, will also be touched upon.

4.3.1 The development process

In many cases today, GPU acceleration is added to an existing implementation of some algorithm or routine. Even if this is not the case, it is often easier to implement a CPU version of a desired code first and then move on to a GPU-based implementation as this serves to provide reference values which greatly aid in debugging, as well as providing a CPU implementation for performance comparisons, even if they are somewhat crude.

The approach followed here is to start with a Python implementation of the method as outlined in [10] and discussed in Section 4.2.2, and build both a more optimised CPU implementation as well as a CUDA-based GPU implementation from there. Python was chosen because it has has a number of strengths making it an ideal language for rapidly developing a prototype code. Even though experience shows that the absolute performance of the Python code is significantly lower than that of a compiled implementation such as FORTRAN or C/C++, it still allows for an analysis of the relative time required by the various phases in the solution process. One of the great advantages of Python is its ability to be combined with native C/C++ or FORTRAN libraries using modules such as Ctypes [105]. This makes it particularly suited to benchmarking and allows for an implementation to be ported to CPU-based C or CUDA in sections, greatly simplifying the development process.

Although one methodology to follow would be to start with a Python implementation and develop separate CPU- and GPU-based implementations of the MOM matrix assembly, this is not the approach followed here. The aim instead is to maximise the code shared between the CPU- and GPU-based versions. The reasoning behind this is that both implementations will be developed and debugged simultaneously, greatly reducing the amount of time required to obtain a working, stable implementation in each case. This is an important point to consider in large software projects, where not only is initial development complex, but maintenance and feature extensions are commonplace.

Since CUDA is provided as a set of C language extensions (see Section 2.3), C seems a logical choice for the co-development of the CPU- and GPU-based implementations. C does however lack operator overloading which, while not strictly necessary, does improve the ease with which the implementations can be developed. As such, C++ is chosen as the language of choice, although features such as templates and object orientation are not used extensively, and thus the source code presented in the subsequent sections will more closely resemble C than C++.

4.3.2 The matrix assembly computational process

The block diagram in Figure 4.4 serves to provide a better understanding of the computational process associated with the matrix assembly phase. It also ties the equations from [10] presented in Section 4.2.2 to specific conceptual blocks and better illustrates how they are related. The process is also shown as a simplified algorithm in Listing 4.1. In both cases it is shown to contain a number of integrations ((4.9) and (4.10)), which are implemented as simplex quadrature rules [106]. More complex integration schemes such as those presented in [107] or [108] can also be used. With these, measures are included to cancel the singularity associated with the free-space scalar Green's functions in (4.11) when its denominator approaches zero. These singularity cancellation schemes are not considered here. Evident in the figure and the listing is that all the computations required to calculate a single matrix element (`Z[m,n]`= Z_{mn} in (4.8)) can be grouped into a single computational unit or kernel ,which includes the evaluation of the integrals in (4.9) and (4.10). It should be noted that each call to the kernel is responsible for calculating a single matrix element and that the same calculations (algorithm) are used for each matrix element.

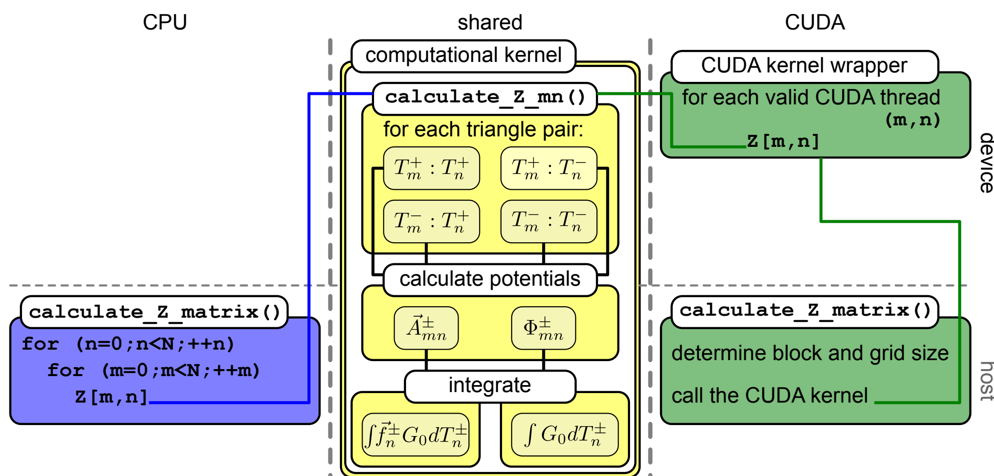


Figure 4.4: A block diagram showing the computational process of the Method of Moment matrix assembly phase for both the CPU and CUDA implementations. The separate `calculate_Z_matrix()` as well as the shared `calculate_Z_mn()` implementations are discussed in subsequent sections.

In [10], matrix assembly by faces, as opposed to edges, is suggested, as this allows for the reuse of computed integrals, which improves performance. The face-based approach is not considered here due to the communication that would be required between the GPU threads, making the implementation more complex but not impossible. For the problem considered here, the face-based implementation was found to be about four times faster than the edge-based variant when considering a single-core CPU implementation. In the edge-based implementation there is no relationship between the matrix elements and thus no synchronisation needs to be performed between the kernels. This makes it a data-parallel task for which GPUs are extremely well suited [27].

4.3.3 CPU and CUDA co-development

The aim is to implement as much of the computational part of the MOM matrix assembly (discussed in Section 4.3.2) as code that can be shared between both the CPU and the CUDA versions. Thus, referring to Figure 4.4, the routine `calculate_Z_mn()` – responsible for calculating the matrix element as in (4.8) – is used as a common entry point into the computational kernel.

Listing 4.1: An algorithm showing the basic outline of the computational process for calculating the Method of Moments impedance matrix, with element Z_{mn} calculated as in (4.8).

```
for each edge m:
  for each edge n:
    ( COMPUTATIONAL KERNEL ( m, n ) (4.8) )
    A. the positive triangle of edge m
      1. the positive triangle of edge n
        add the compute potential contributions:
          ( integrate (4.9) )
          ( integrate (4.10) )
      2. the negative triangle of edge n
        add the compute potential contributions:
          ( integrate (4.9) )
          ( integrate (4.10) )
    B. the negative triangle of edge m
      1. the positive triangle of edge n
        add the compute potential contributions:
          ( integrate (4.9) )
          ( integrate (4.10) )
      2. the negative triangle of edge n
        add the compute potential contributions:
          ( integrate (4.9) )
          ( integrate (4.10) )
```

In addition, any routines, such as quadrature integration routines, that are called by this function must also be shared between the CPU and the CUDA implementations.

It should be noted that although the implementation of `calculate_Z_mn()` is common to the two target devices, the code for calculating the full matrix (`calculate_Z_matrix()` in Figure 4.4) will not be the same. This is mainly due to the different implementations of the two outer loops (over `m` and `n`) in Listing 4.1 and will be discussed in more detail shortly.

In addition to the implementations of the outer loops that differ, there may be other device-specific code that cannot be shared between the two architectures. This includes type definitions – such as complex and vector data types – and operators defined on those types. As such, each target (CPU and CUDA) has its own header file containing compatible definitions. This file arrangement is depicted in Figure 4.5, with the two device specific header files (`cuda_defines.h` and `cpu_defines.h`) shown. Also shown in the figure is a header file containing the shared computational code (`computational_kernel.h`), a file containing the common definitions (`common_defines.h`), as well as the two source files containing the implementations of the individual `calculate_Z_matrix()` routines.

To better illustrate this, consider a code skeleton of the file `computational_kernel.h` shown in Listing 4.2. This skeleton shows the definition of a single function `calculate_Z_mn()` that has a return type of `double_complex`, a double precision complex value. Note that the only file included is `common_defines.h` and as such all code shown is device agnostic. Since this is only intended as a code skeleton, with more of the implementation details to follow, most of the function parameters as well as the function contents have been replaced by ellipsis dots. Note that this and subsequent code for this chapter shows only the double precision implementations. The single precision implementations are, however, very similar and in most cases only require that all occurrences of `double` be replaced with `float`.

Also not shown in Listing 4.2, are a number of other routines called from within `calculate_Z_mn()`. Some of these will be discussed in due course. Note that the `calculate_Z_mn ()` function definition also includes, in addition to the return type, `DEVICE_PREFIX` which is a macro defined in both `cuda_defines.h` and `cpu_defines.h`. In the case of the latter it is simply an empty macro whereas in the CUDA case it is used to add the `__device__` qualifier to all definitions of functions

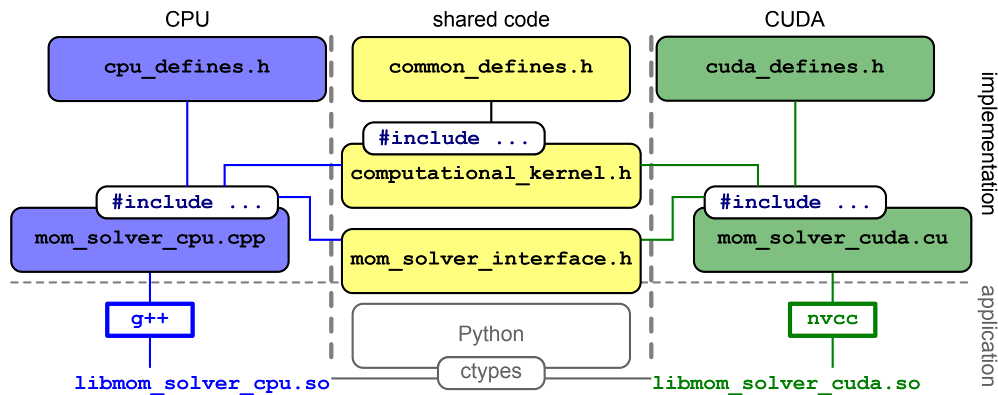


Figure 4.5: A diagram representing the organisation of the files used in the co-development process. The file names of the files for the implementation as discussed here are shown in the upper part of the diagram. The files that are specific to the CPU implementation are in blue rounded rectangles on the left of the image, with the CUDA files on the right in green. The files containing shared code – including the common interface (`mom_solver_interface.h`) are shown in the centre in yellow. Compiler information (`g++` or `nvcc`) as well as the names of the shared libraries is given in the lower part. Note that the file `mom_solver_interface.h` is shown extending into the application domain, as this is the interface that is used when the generated libraries are included in external programs such as a Python benchmarking framework using Ctypes.

Listing 4.2: Code skeleton for `computational_kernel.h`

```
#ifndef COMPUTATIONAL_KERNEL_H
#define COMPUTATIONAL_KERNEL_H
// include the common definitions
#include "common_defines.h"
...
// The computational kernel to calculate Z[m,n]
DEVICE_PREFIX double_complex calculate_Z_mn ( int m, int n, ... )
{
  double_complex Z_mn;
  ...
  return Z_mn;
}
...
#endif // #ifndef COMPUTATIONAL_KERNEL_H
```

that must execute on the CUDA device [29]. This `__device__` qualifier is similar to the `__global__` qualifier used with the CUDA kernels shown in Section 2.3, and is required for all routines called from a CUDA kernel.

The `complex_double` type used in Listing 4.2 and Listing 4.9 to signify a complex double precision floating-point value, is implementation dependent and, as such, is defined in both the `cpu_defines.h` and `cuda_defines.h` header files. The relevant parts of `cpu_defines.h` and `cuda_defines.h` are shown in Listing 4.3 and Listing 4.4, respectively. Most notable of these are the definitions of the `DEVICE_PREFIX` macro as blank in `cpu_defines.h` and as `__device__` in `cuda_defines.h`, and the definition of the `double_complex` type.

For the CUDA implementation of Listing 4.4, the complex value functionality is provided using the CUDA header file `cuComplex.h`, whereas the CPU implementation uses `complex.h` and `cmath.h`. It should be noted that the CPU header file of Listing 4.3 also contains a type definition for a `double3` type, used to represent a three-dimensional real vector in double precision, as a

Listing 4.3: Code skeleton showing some of the CPU definitions in `cpu_defines.h`

```
#ifndef CPU_DEFINES_H
#define CPU_DEFINES_H

#define DEVICE_PREFIX

#include <cmath.h>
#include <complex.h>

typedef std::complex<double> double_complex;

inline double_complex make_double_complex ( double r, double i )
{
  return double_complex ( r, i );
}

struct double3
{
  double x, y, z;
}
typedef struct double3 double3;

inline double3 make_double3 ( double x, double y, double z )
{
  double3 v = {x, y, z}
  return v;
}
...

#endif // #ifndef CPU_DEFINES_H
```

Listing 4.4: Code skeleton showing some of the CUDA definitions in `cuda_defines.h`

```
#ifndef CUDA_DEFINES_H
#define CUDA_DEFINES_H

#define DEVICE_PREFIX __device__

#include <cuComplex.h>

typedef cuDoubleComplex double_complex;

__inline__ __device__ double_complex make_double_complex ( double r, double i )
{
  return make_cuDoubleComplex ( r, i );
}
...

#endif // #ifndef CUDA_DEFINES_H
```

`struct`. This type is not defined in `cuda_defines.h` as this is one of the built-in vector types in CUDA [29]. A function included in CUDA that needs to be implemented for the CPU version (as shown in Listing 4.3) is `make_double3()`, which returns a `double3` variable initialised using the three `double` parameters passed.

In order to improve the readability of the code, operators are defined for the types used. Since these operators are not dependent on the specific implementation, they are included in the shared header file `common_defines.h` (as seen in Figure 4.5). A code skeleton for this file is given

in Listing 4.5, showing an addition operator for two double precision 3-vectors of type `double3`. Since this operator is used in both the CPU and CUDA implementation, and called from the computational kernel, the `DEVICE_PREFIX` macro is required. This operator calls the `make_double3()` routine that has either been defined as part of the built-in CUDA vector types, or implemented for the CPU version in `cpu_defines.h`. Listing 4.5 also shows the definition of a double precision complex 3-vector type `double_complex3` which is used in the representation of the electric field and magnetic vector potential.

Listing 4.5: Code skeleton showing some of the common definitions in `common_defines.h`

```
#ifndef COMMON_DEFINES_H
#define COMMON_DEFINES_H

struct double_complex3
{
  double_complex x, y, z;
}
typedef struct double_complex3 double_complex3;

DEVICE_PREFIX __inline__ double3 operator+( double3 v, double3 u )
{
    return make_double3 ( v.x + u.x, v.y + u.y, v.z + u.z );
}
...
#endif // #ifndef COMMON_DEFINES_H
```

In addition to the header files discussed so far, Figure 4.5 shows two files, `mom_solver_cpu.cpp` and `mom_solver_cuda.cu` for the CPU and CUDA implementations, respectively, and a header file, `mom_solver_interface.h`, where the common interface for these implementations is defined. An example of a routine that is declared in `mom_solver_interface.h`, shown in Listing 4.6, is the `calculate_Z_matrix()` routine.

Listing 4.6: A snippet from the `mom_solver_interface.h`file showing the declaration of the `calculate_Z_matrix()`function as part of the shared interface for the CPU and CUDA implementations in `mom_solver_interface.h`

```
extern "C"
int calculate_Z_matrix ( int, double*, int, ... );
```

The `calculate_Z_matrix()` function in `mom_solver_interface.h` is declared as an external C routine using the `extern "C"` keyword. This prevents the mangling of the function names by the C++ compiler and is required to link the library built from the code with a benchmarking framework implemented in Python using Ctypes [105]. Another important point is that all of the parameters of `calculate_Z_matrix()`, and the other routines declared in `mom_solver_interface.h`, are of standard C types, with the details of the implementations such as complex valued types hidden from a program that links to the generated libraries.

The simplified source code of Listing 4.7 and Listing 4.8 shows the inclusion of the interface declaration, as well as the subsequent definition of the `calculate_Z_matrix()` routines for the CPU and CUDA implementations, respectively. The parts of the two files shown are identical except for the inclusion of the CPU- and CUDA-specific definitions (`#include "cpu_defines.h"` and `#include "cuda_defines.h"`), which are included before the computational kernel header file in each case.

Listing 4.7: A code skeleton showing the definition of the CPU implementation of the `calculate_Z_matrix()` routine in `mom_solver_cpu.cpp` Note that `cpu_defines.h` is included before `computational_kernel.h`

```
#include "mom_solver_interface.h"
#include "cpu_defines.h"
#include "computational_kernel.h"

int calculate_Z_matrix ( int N, double* Z, int LDZ, ... )
{
  ...
}
```

Listing 4.8: A code skeleton showing the definition of the CUDA implementation of the `calculate_Z_matrix()` routine in `mom_solver_cuda.cu` Note that `cuda_defines.h` is included before `computational_kernel.h`

```
#include "mom_solver_interface.h"
#include "cuda_defines.h"
#include "computational_kernel.h"

int calculate_Z_matrix ( int N, double* Z, int LDZ, ... )
{
  ...
}
```

As indicated in Figure 4.5, `g++` and `nvcc` are used to compile the CPU and CUDA implementations, respectively. Using these compilers, two shared libraries, `libmom_solver_cpu.so` and `libmom_solver_cuda.so`, are created and these can be linked to by other applications. During the Python development and benchmarking process used here, these libraries are accessed from Python using Ctypes [105].

4.3.4 Parallel CPU implementation

Before continuing the discussion on the computational kernel and the CUDA specifics of the implementation, the multi-core CPU version is discussed briefly. It should be noted that this is specific to the CPU-based implementation, and in no way affects the shared computational kernel. This also serves to complete the CPU side of the organisation of files represented in Figure 4.5.

As already discussed, for the edge-based formulation considered here, the calls to the computational kernel routine `calculate_Z_mn()` are independent of each other and thus require no communication. This makes this process an ideal target for acceleration using multi-threaded methods such as OpenMP [19]. In terms of implementation, OpenMP has an advantage over a message passing implementation such as OpenMPI [18], in that much of the parallelisation can be automatically performed by the compiler. This does, however, restrict the parallel implementation to single shared memory or multi-core machines, as opposed to clusters of networked computers. Since this study is restricted to implementations on single machines, OpenMP is a logical choice for a simple paralellisation scheme.

The simplest way to obtain a parallel implementation using OpenMP is to specify which loops can be run in parallel. This is done using `#pragma omp` compiler directives, with the parallelisation of the loops shown in Listing 4.9. Here, two `#pragma omp` directives are visible, the first of which (`#pragma omp parallel default(shared)private(n,m)`) is used to indicate a block of code targeted

for parallel execution, as well as specifying which variables are private to the CPU threads – `m` and `n` – and that all other variables declared up until this point are to be shared between the threads by default (`default(shared)`). This makes sense in this case, since most of the other variables are constants or pointers to data that can be safely accessed by all threads as there are no dependencies between them.

Listing 4.9: Code to calculate the matrix `Z` using the shared computational kernel `calculate_Z_mn()` as part of a parallel CPU implementation. The computational kernel is called from parallelised nested loops to calculate the matrix `Z`. The matrix is stored in column-major order with a leading dimension `LDZ`.

```
...
// include omp.h for parallel execution
#include <omp.h>
...
int calculate_Z_matrix ( int N, double* Z, int LDZ, ... )
{
  int m, n;
  double_complex* dcZ = (double_complex*) Z;
#pragma omp parallel default(shared) private(n,m)
  {
#pragma omp for schedule(runtime)
    for ( n = 0; n < N; ++n )
    {
      for ( m = 0; m < N; ++m )
      {
          dcZ[n*LDZ + m] = calculate_Z_mn ( m, n, ... );
      }
    }
  }// #pragma omp parallel
  return 0;
}
```

The second OpenMP directive, `#pragma omp for schedule(runtime)`, informs the compiler that the outer loop over `n` is to be parallelised. The work allocation between threads is decided at runtime (`schedule(runtime)`), depending on the value of the `OMP_SCHEDULE` environment variable [19]. Another environment variable, `OMP_NUM_THREADS`, controls how many threads are launched for the particular section. Similarly to the CUDA driver, OpenMP handles the launching of these threads and this process is transparent to the user.

The routine thus contains the two parallelised nested loops over `n` and `m`, the index of the impedance matrix column and row, respectively. Each loop iteration sees a call to the `calculate_Z_mn()` routine of Listing 4.2 for each of the `N*N` combinations. Note that although the routine shown receives a a `double` pointer to the matrix to assemble, this is cast to a complex double precision pointer (`double_complex*`) for address calculation purposes. Some parameters, such as those containing geometric information, have been omitted.

4.3.5 Parallel CUDA implementation

In the preceding two sections, the shared computational kernel and some details of the parallel CPU implementation using OpenMP were introduced. In this section, specifics pertaining to the CUDA implementation are addressed.

Problem domain segmentation

As discussed in Section 2.3, one of the first steps in the development of a GPU program, or parallel program in general, often is to give some thought to the segmentation of the problem domain. As mentioned in the same section, one of the strengths of CUDA is its ability to handle a large number of lightweight threads and that each of these threads can be identified uniquely. Thus they could be used to perform a calculation on a separate data element. Furthermore, these threads can be grouped into one-, two-, or three-dimensional blocks, with the blocks then arranged in a one- or two-dimensional grid (see Section 2.3) [29]. A similar approach is followed for the parallel CPU implementation in Section 4.3.4, although there the threading is not explicitly defined.

Since the assembly of the impedance matrix, which is a two-dimensional structure with each data element a matrix entry, is being considered, it is intuitive to make use of two-dimensional blocks and a two-dimensional grid. This is illustrated graphically in Figure 4.6, with a 5×5 grid of 8×8 blocks shown. The darker grey areas on the right and the bottom of the grid indicate portions of blocks that do not correspond with matrix entries. This is as a result of the matrix not being a multiple of the block size, which must be the same for all blocks, although not necessarily the same in each dimension [29].

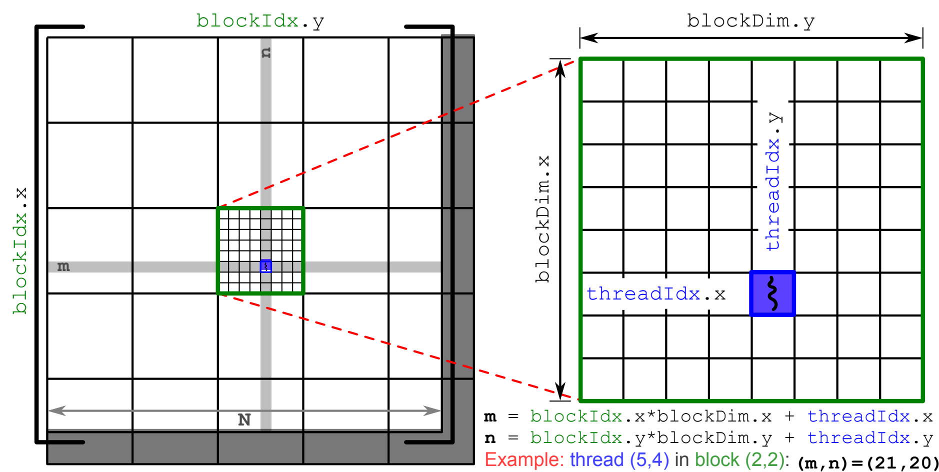


Figure 4.6: A diagram depicting the arrangement of threads into a two-dimensional grid of two-dimensional blocks and the use of this arrangement in the subdivision of an $N \times N$ matrix. Also shown are the built-in CUDA variables specifying the dimensions and indices of the blocks and threads and their use in determining the row (`m`) and column (`n`) index of an element in the matrix. The thread corresponding to matrix element $(21, 20)$ is highlighted in blue. The parts of the CUDA grid that do not correspond with matrix elements are shaded in grey.

Using this grid of thread blocks – with the two-dimensional global index of each thread corresponding to a row and column index of the impedance matrix $[Z]$ – CUDA can launch a large number of kernels. Each of these kernels then executes for a thread in the grid, and is used to calculate the matrix element at the position in the matrix defined by the index of that thread. Thus the global indices of the threads, excluding the areas that fall outside the matrix dimensions, can be seen as equivalent to the loop variables `m` and `n` introduced in Listing 4.1 and shown for the CPU implementation in Listing 4.9.

CUDA implementation details

The CUDA variant of Listing 4.9 is given in Listing 4.10, and serves both to introduce the CUDA specifics of the implementation, as well as to illustrate the differences between the CUDA- and the CPU-based implementations. Note that these differences only occur in these two listings and in the `cpu_defines.h` and `cuda_defines.h` files shown in Section 4.3.3. The contents of `computational_kernel.h` remain unchanged, so as to realise the aims of this co-developed implementation.

Listing 4.10: A code snippet showing an initial CUDA implementation of the `calculate_Z_matrix()` routine. The computational kernel is wrapped in a CUDA kernel which is shown in Listing 4.12.

```
...
// code shown in Listing 4.12 is inserted here
...
int calculate_Z_matrix ( int N, double* Z, int LDZ, ... )
{
  // allocate matrix in device memory
  double_complex* pdev_Z = 0;
  cudaMalloc ( (void**)&pdev_Z, N*LDZ*sizeof(double_complex) );
  // determine the block and grid size
  int threads_per_block = 8;
  int blocks_per_grid = div_to_next ( N, threads_per_block ); // see Listing 2.4
  // setup the block and grid using the CUDA type dim3
  dim3 block ( threads_per_block, threads_per_block, 1 );
  dim3 grid ( blocks_per_grid, blocks_per_grid, 1 );
  // call the kernel
  cuda_matrix_fill_kernel_wrapper <<< grid, block >>> ( N, pdev_Z, LDZ, 0, 0,
      ... )
  // transfer the result from the device to the host
  cudaMemcpy ( Z, pdev_Z, N*LDZ*sizeof(double_complex), cudaMemcpyDeviceToHost )
      ;
  // free the device memory
  cudaFree ( pdev_Z );
  return 0;
}
```

Listing 4.10 contains all the required steps to perform the calculation of the impedance matrix on a CUDA device. This routine follows the same computational process as the simple examples presented as part of the introduction to CUDA in Section 2.3 and include memory allocation, grid and block set-up, kernel invocation, transfer of the results, and freeing of the allocated memory. The kernel called in Listing 4.10 is `cuda_matrix_fill_kernel_wrapper()`, and will be discussed shortly.

Note that the routine name as well as the parameter list for all the implementations of `calculate_Z_matrix()`, including both the parallel CPU (Listing 4.9) and CUDA (Listing 4.10) versions, are identical. This is so that they can be used interchangeably as part of a higher-level implementation of the Method of Moments.

Memory limited CUDA implementation

One of the aims for this implementation is to overcome the restrictions on problem size due to the limited memory available on CUDA devices. To do this, an approach similar to that of [91] is followed. This involves splitting the impedance matrix into blocks that are small enough to fit into GPU memory, and handling these separately. Since the concept of a block is also included in CUDA, these matrix divisions are referred to as super-blocks. Furthermore, these square super-blocks are chosen so that their size (number of rows or columns) is a multiple of

the CUDA block size. In Figure 4.7, the use of a 16×16 super-block is shown for the matrix depicted graphically in Figure 4.6.

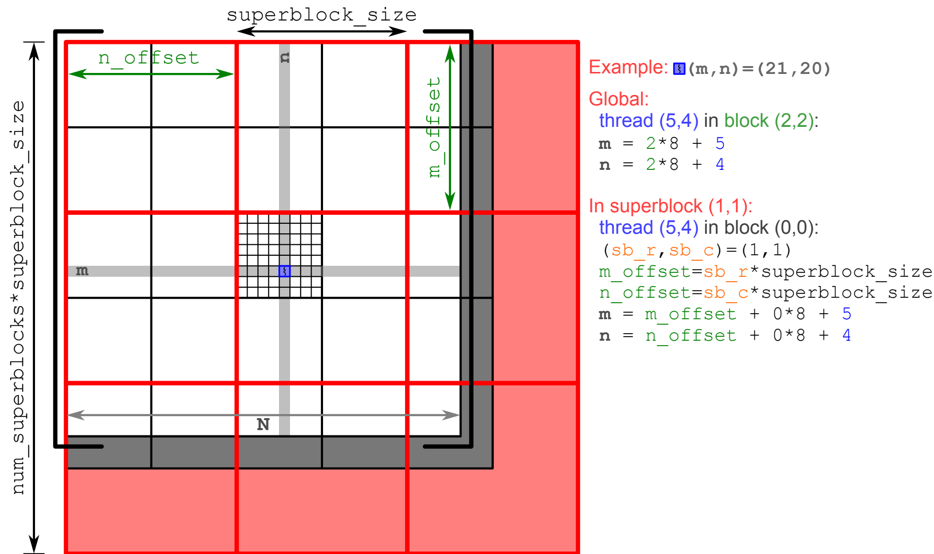


Figure 4.7: Figure showing the $N \times N$ matrix of Figure 4.6 split into a 3×3 grid of 16×16 super-blocks. Each super-block consists of 4 (2×2) CUDA blocks. The thread with global index (21,20) highlighted in Figure 4.6 is also shown along with an example of the calculation of its global index when super-blocks are used. In the figure, `sb_r`and `sb_c`are the row and column index of a super-block in the grid. As in Figure 4.6, the parts of the CUDA grid that fall outside the matrix boundaries are shaded grey, while the parts of the super-blocks that fall outside the original CUDA grid are filled in red.

The program and data flow between the device and the host for the assembly of the MOM impedance matrix is shown in Figure 4.8. At this point it should be mentioned that since the individual matrix elements are independent, it follows that the super-blocks themselves are also independent. It is then possible, in theory, to assemble multiple super-blocks simultaneously on multiple CUDA devices, although this is not considered further.

The incorporation of the super-block matrix assembly into the `calculate_Z_matrix()` routine in Listing 4.10 is straightforward. A code snippet of the modifications required is shown in Listing 4.11. The function `get_superblock_size()` checks the amount of available memory on the CUDA device and returns the number of rows of a square matrix that can be allocated successfully, while ensuring that this is also a multiple of the CUDA blocks size. If the available memory can accommodate the entire matrix, then the standard matrix assembly of Listing 4.10 is used. If this is not the case, then the matrix is assembled super-block by super-block by the routine `calculate_Z_matrix_superblock()` whose source code is not shown.

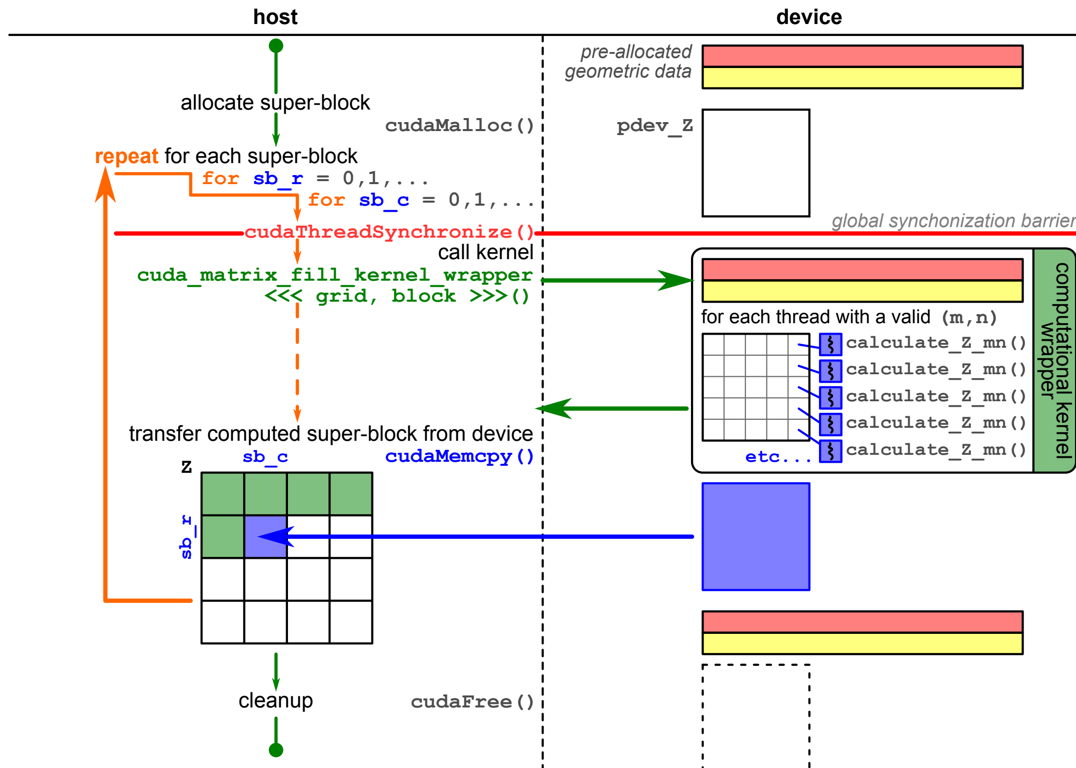


Figure 4.8: A diagram representing the program and data flow between the host and a CUDA device in the assembly of the system matrix using the super-block implementation. Note that the geometric data required for the calculation has been allocated and transferred as part of the pre-processing step shown in Figure 4.2 with only the super-block `pdev_Z`allocated and freed here. Note the loop over the super-blocks to assemble the matrix `Z` on the host by rows of super-blocks, as seen in the data transfer step. The figure also shows a global synchronisation barrier in the form of a `cudaThreadSynchronize()` call before the CUDA kernel is called.

Listing 4.11: Modifications to `calculate_Z_matrix()`of Listing 4.10 to allow for matrix assembly using super-blocks.

```
int calculate_Z_matrix ( int N, double* Z, int LDZ, ... )
{
  int superblock_size = get_superblock_size ();
  // check to see if the matrix will fit in GPU memory
  if ( N > superblock_size )
  {
    // allocate device memory
    double_complex* pdev_Z = 0;
    cudaMalloc ( (void**)&pdev_Z, superblock_size*superblock_size*sizeof(
        double_complex) );
    // fill the matrix using super-blocks
    calculate_Z_matrix_superblock ( N, Z, LDZ, pdev_Z, superblock_size, ... );
    // free the device memory
    cudaFree ( pdev_Z );
  }
  else
    // perform the computation as in Listing 4.10
  return 0;
}
```

The CUDA kernel

Both the standard CUDA and super-block implementations of `calculate_Z_matrix()` contain calls to a CUDA kernel `cuda_matrix_fill_kernel_wrapper()`. This routine acts as a wrapper for the device function ,`calculate_Z_mn()` of Listing 4.2, and also serves to provide the same functionality of the double-loop implementation in the CPU case (Listing 4.9). The code for the CUDA kernel is given in Listing 4.12, where, as is the case for the other code samples used in this discussion, the parameters pertaining to constants and geometric data have been omitted.

The parameters that are given are the size of the full matrix (`N`), a pointer to where the matrix is to be constructed in global device memory (`pdev_Z`), the leading dimension of this matrix (`LDZ`), and lastly row and column offsets (`m_offset` and `n_offset`) that are only applicable for the super-block implementation, and can be treated as zero if the standard assembly is being considered. In the case of the super-block variant, the device pointer `pdev_Z` does not represent device memory allocated for the entire matrix, and the leading dimension of the device matrix is not equal to that of the matrix on the host. The leading dimension, offsets, and the current thread index are all used to calculate the linear position, in column-major order, of the matrix element in the array `pdev_Z`.

Listing 4.12: Code to wrap the computational kernel `calculate_Z_mn()`as a CUDA kernel implemented in `mom_solver_cuda.cu`Note that the edge lengths are stored in global device memory and are passed as a parameter (not shown) `const float* pdev_edge_lengths`

```
__global__ void cuda_matrix_fill_kernel_wrapper ( int N, double_complex* pdev_Z,
    int LDZ, int m_offset, int n_offset, ... )
{
  int m = blockIdx.x*blockDim.x + threadIdx.x + m_offset;
  int n = blockIdx.y*blockDim.y + threadIdx.y + n_offset;
  // use shared memory to store the lengths of the edges for this block
  __shared__ double l_m[8];
  __shared__ double l_n[8];
  // only some of the threads read the lengths from global memory
  if ( threadIdx.y == 0 && m < N )
      l_m[threadIdx.x] = pdev_edge_lengths[m];
  if ( threadIdx.x == 0 && n < N )
      l_n[threadIdx.y] = pdev_edge_lengths[n];
  // synchronise the threads of the block
  __syncthreads();

  if ( ( m < N ) && ( n < N ) )
  {
      pdev_Z[(n-n_offset)*LDZ + (m-m_offset)] = calculate_Z_mn ( m, n, l_m[
         threadIdx.x], l_n[threadIdx.y], ...);
  }
}
```

Calling this CUDA kernel from the `calculate_Z_matrix()` routines, instructs the CUDA runtime to launch a kernel instance for each of the threads in the grid of thread blocks defined by `grid` and `block`. As discussed in Section 2.3, all the threads of a block are able to access the fast on-chip shared memory and this allows for communication between the threads. Although there is no dependence between the matrix elements and thus no communication is required between the threads of a block, Listing 4.12 does show the use of shared memory, as well as synchronisation between the threads.

The purpose of the use of shared memory in Listing 4.12 is not inter-thread communication, but instead to reduce the number of accesses to high-latency global memory by storing common

geometric data (in this case the length edges `m` and `n`) in shared memory. Since each of the threads requires the lengths of two edges (see (4.5), (4.8), (4.9), and (4.10)) this would normally involve `2*blockDim.x*blockDim.y` reads from the global memory array `pdev_edge_lengths`. With the use of the shared memory variables `l_m` and `l_n` – both declared using the `__shared__` CUDA qualifier [29] – the number of reads is reduced to `blockDim.x + blockDim.y` and is possible since `m` and `n` are constant over the rows (`x`-dimension) and columns (`y`-dimension) of a block, respectively.

Note that only the threads of the first row and column of the block actually read the data from global memory and the thread with local index `(0,0)` reads two values. The `__syncthreads()` barrier is included to ensure that shared memory values stored in the arrays `l_m` and `l_n` have been read successfully from global memory before commencing computation for the block. Although it is also possible to extend this to include other geometric data used in the computation of the matrix elements (such as the centres of the elements and the element areas), this is not considered at present.

4.3.6 Implementation of the other phases

In closing the discussion on the implementation of the accelerated MOM solution process, consider once again the block diagram of Figure 4.2 that illustrates the various phases. As already mentioned, the initialisation phase – which is responsible for constructing or loading the mesh used in the computation – is not considered here. The parts of the block diagram that are considered here are shown as part of an algorithm to calculate the RCS of a square plate in Listing 4.13. The `calculate_Z_matrix()` routine that has been discussed in some detail is shown in step 2 and the other steps in the algorithm are now addressed. Again, geometric data and constants have been omitted in most cases and replaced with ellipsis dots.

Listing 4.13: Pseudo-code representation of the computational process (derived from Figure 4.4) for calculating the mono-static RCS of a square plate.

```
1. pre_process ( ... );
2. Z ← calculate_Z_matrix ( N, ... );
3. LU, IPIV ← LU_decomposition ( N, Z, ... );
4. for each incident angle theta in list_of_angles:
  a. V ← calculate_V_vector ( N, theta, E_inc, ... );
  b. I ← solve_I_vector ( N, LU, IPIV, V, ...);
  c. E_s ← calculate_E_s ( N, I, theta, ... );
5. RCS ← post_process ( E_s, E_inc, list_of_angles )
6. cleanup ( ... );
```

The implementation of the remaining phases also follow the co-development framework as pointed out in Section 4.3.3. As such, they consist of computational implementations in `computational_kernel.h`, as well as entry functions and wrappers in either `mom_solver_cpu.cpp` or `mom_solver_cuda.cu` for each of the implementations.

Certain geometric properties of the mesh are required as part of the computational process. These include the length of the edges in the mesh, and centres of the triangles in the mesh. Although it is possible to calculate these on the fly this does lead to a lot of redundant calculations. Recall, for example, that all the matrix elements in the same column or row of the matrix are associated with the same edge and thus have an equal corresponding length. If the edges lengths are calculated each time they are required, the number of computations will be N^2. However, it should be clear that, since there are only N edges, only N such computations are required. A similar argument can be followed for the other geometric properties.

The `pre_process()` call of Listing 4.13 thus takes the geometric data from the initialisation phase (not shown here) and generates and stores any additional properties that are required as part of the computational process. In the case of the edge lengths, these take the form of double precision floating-point arrays that can be indexed using the edge number. In the case of the CUDA implementation, the arrays for the additional data are allocated in global device memory and `pre_process()` includes the allocation and transfer of all geometric data required in the computations.

In the CPU implementation, memory only needs to be allocated for the additional data and filled with calculated values. In both cases, a list of allocated pointers is maintained and the call to `cleanup()` in Listing 4.13 is responsible for the implementation-specific freeing of this memory. As such, the geometric data remains allocated – either on the host or on the device – for the duration of the RCS calculation. This allows for the calculated values to be reused for repeated calls to routines such as `calculate_Z_matrix()` and `calculate_V_vector()`. In the CUDA case, this serves to reduce the amount of data that has to be transferred to the device before computation can commence.

For the LU decomposition and the solution of the current coefficients (`solve_I_vector()`), neither `mom_solver_cpu.cpp` nor `mom_solver_cuda.cu` have a corresponding routine in `computational_kernel.h`. Instead, the functionality is provided by either a call to the CPU-based LAPACK routines, provided by the multi-core version of ACML, or a library containing the implementation of the MAGMA-panel-based LU decomposition as discussed in Section 3.3.3. In the case of the solution for the current coefficients, the `ZGETRS` and `CGETRS` routines used have no MAGMA implementations at this time and are thus only performed by calls to the relevanl ACML routines [8, 64].

The implementation procedures for the calculation of the excitation vector and the scattered field are almost identical to that of the matrix assembly. Here, routines `calculate_V_m()` and `calculate_E_m()`, that return a `double_complex` value and a `double_complex3` value, respectively, are implemented in `computational_kernel.h`. These routines provide the computational kernels for computing each of the excitation vector elements as in (4.12), and the contributions to the scattered field for each edge as in (4.20). In the case of the latter, these contributions are stored in a temporary array of type `double_complex3` and summed once all the elements have been computed.

The routines `calculate_V_vector()` and `calculate_E_s()` are implemented in both `mom_solver_cpu.cpp` and `mom_solver_cuda.cu`, and provide the compatible entry points into either the CPU or CUDA implementation. As is the case with the matrix assembly, the CPU implementation of `calculate_V_vector()` is parallelised using the `#pragma omp parallel` and `#pragma omp for` compiler directives for OpenMP. Since the expression for the scattered field in (4.20) contains summation, a simple OpenMP for loop is not sufficient and an OpenMP reduction operation is used [19].

The CUDA implementation consists of the respective entry function (`calculate_V_vector()` or `calculate_E_s()`), along with the CUDA kernels that wrap the computational kernels and allow them to be called by each thread. Since the structures being assembled are vectors, one-dimensional CUDA grids are used. For the present CUDA implementation, the summation of the temporary array of scattered field values is performed on the host, also making use of an OpenMP reduction. Once the total scattered field has been obtained, it is a straightforward process to obtain the scalar RCS of (4.1).

4.4 Verification results

Before continuing a discussion on the performance results of the methods implemented, the matter of verification is addressed. Since much of the formulation is taken from [10], it follows that the results presented therein are used as the first step for comparison.

One of the problems in [10] – and already introduced in Section 4.2.1 – is that of a square PEC plate in the xy-plane, measuring one wavelength on each side. An example mesh for such a plate is shown in Figure 4.9. Note that the direction of incidence of the electric field, $\hat{k} = -\hat{z}$, is normal to the surface of the plate and is parallel to the negative z-axis. As such, θ and ϕ (as defined in Figure 4.1) are both zero. The incident electric field, $\vec{E}_{inc}$, is also shown, and is defined as [10]

$$\vec{E}_{inc} = E_\theta \hat{\theta} + E_\phi \hat{\phi}, \tag{4.21}$$

with $E_\phi = 0$ in this case. The two cut-lines ,AA' and BB', are the same as those presented in [10], with the x-component of the surface current density – normalized with respect to the incident field – calculated along these cut lines. Figure 4.9 also shows a quiver plot of the calculated surface current density, with the distribution similar to [103].

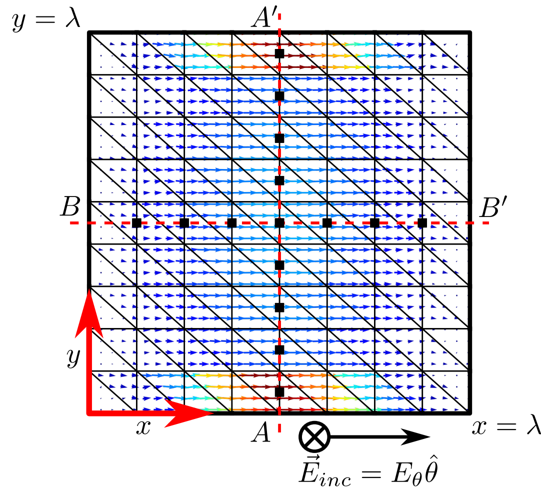


Figure 4.9: A figure showing the 1λ square plate (in the xy-plane) used as a sample problem. The mesh (with 199 degrees of freedom) used to obtain the verification results, the cut-lines AA' and BB', and the normally-incident electric field ($\vec{E}_{inc}$) as defined in [10] are also shown. The surface current density calculated using the implementation discussed is included as a quiver plot and the black squares (■) along the cut-lines indicate the points where the current is measured to produce Figure 4.10 and Figure 4.11.

Figure 4.10 shows the calculated current results as a function of displacement from the edge of the plate for the two cut-lines when using single precision floating-point computation and the mesh shown in Figure 4.9. Results from [10], as well as results computed using FEKO [92] are given for comparison. The CPU and CUDA results calculated here are indistinguishable from each other and agree well with the reference results.

The double precision equivalent of Figure 4.10 is shown in Figure 4.11. The same reference results from [10] and [92] have been included. As is the case for the single precision results, the CPU and CUDA results are indistinguishable, and agree well with the reference curves. Since the outputs from all the test systems (introduced in Section 3.4.1) were found to be indistinguishable, only one set of current density verification results – those of System 1 – have been presented.

The final verification step is performed by calculating the normalised monostatic RCS of the square plate of Figure 4.9, for the incident field polarisation as shown, but with the incident angle θ_i ranging from 0 to 90 degrees in steps of one degree. The method used to perform the calculation has been discussed in Section 4.2.2.

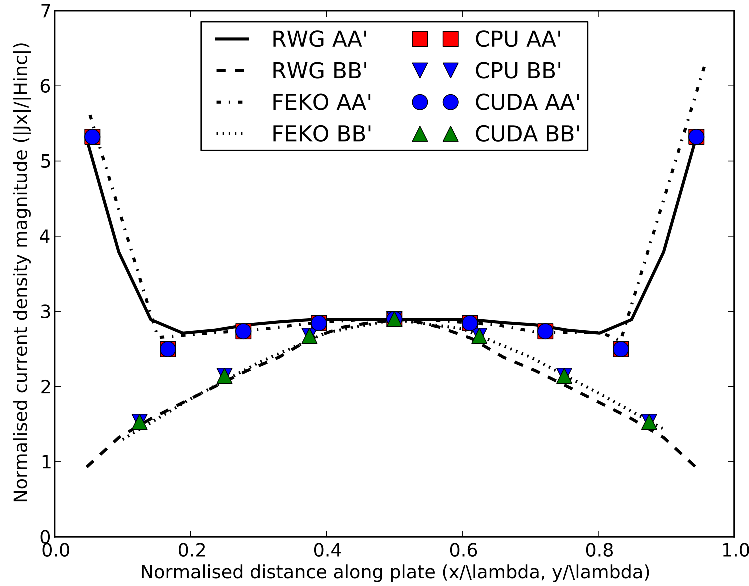


Figure 4.10: A recreation of Figure 6 from [10] showing the normalised magnitude of the x-component of the surface current density (calculated in **single precision**) along the two cut lines (and for the mesh) as shown in Figure 4.9. Results from CPU and CUDA implementations discussed here are given, with results read off the figure in [10] as well as results computed using FEKO [92] on a different $1\lambda \times 1\lambda$ mesh provided for comparison. The CPU and CUDA results are identical and agree well with the reference results.

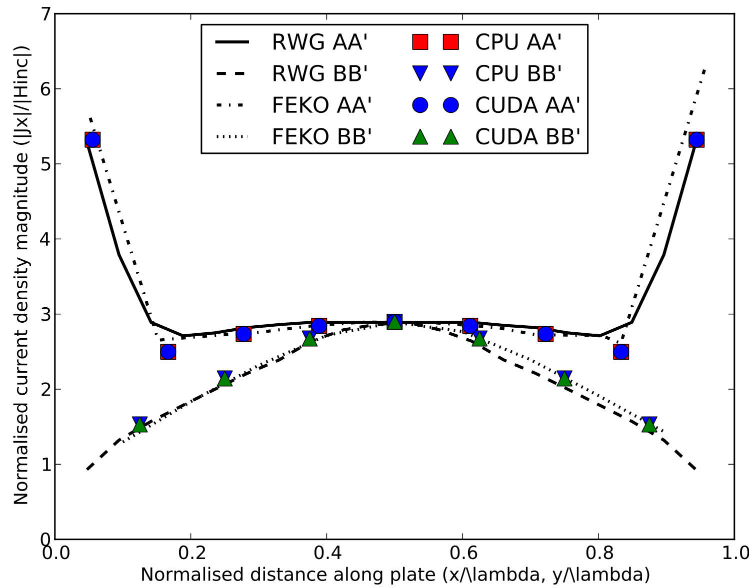


Figure 4.11: A recreation of Figure 6 from [10] showing the normalised magnitude of the x-component of the surface current density (calculated in **double precision**) along the two cut lines (and for the mesh) as shown in Figure 4.9. Results from CPU and CUDA implementations discussed here are given, with results read off the figure in [10] as well as results computed using FEKO [92] on a different $1\lambda \times 1\lambda$ mesh provided for comparison. The CPU and CUDA results are identical.

Figure 4.12 and Figure 4.13 show the normalised RCS computed for the mesh of Figure 4.9

on all the test systems for single and double precision calculations, respectively. Also shown are the RCS results computed using FEKO. Note that the FEKO mesh and the mesh in Figure 4.9 are not the same, with the FEKO mesh being non-uniform and consisting of more degrees of freedom (261 instead of 199), accounting for the differences. As with the current calculations, the CPU and CUDA results for all the systems are indistinguishable and agree well with the FEKO results.

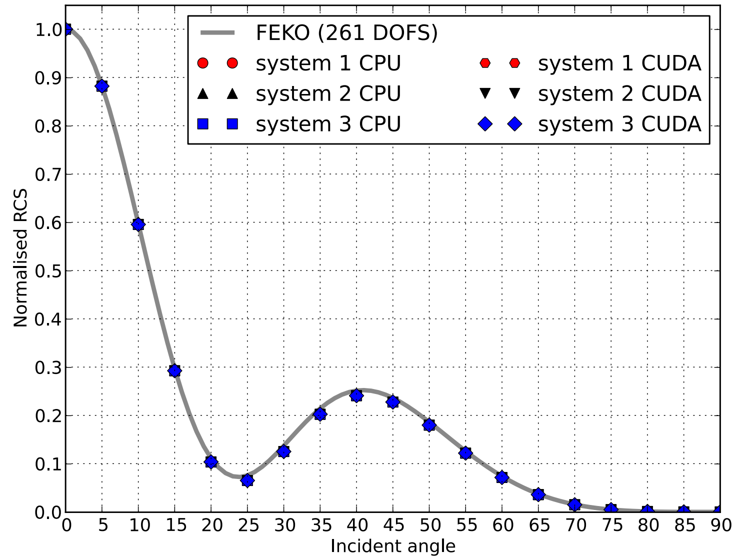


Figure 4.12: The computed normalised radar cross section of a 1λ square PEC plate as a function of the angle θ ($\phi = 0$) calculated in **single precision** using the mesh shown in Figure 4.9. Results are shown for both the CPU and CUDA implementations for the three test systems considered, with FEKO results being included as a reference. The results calculated on the three systems are indistinguishable.

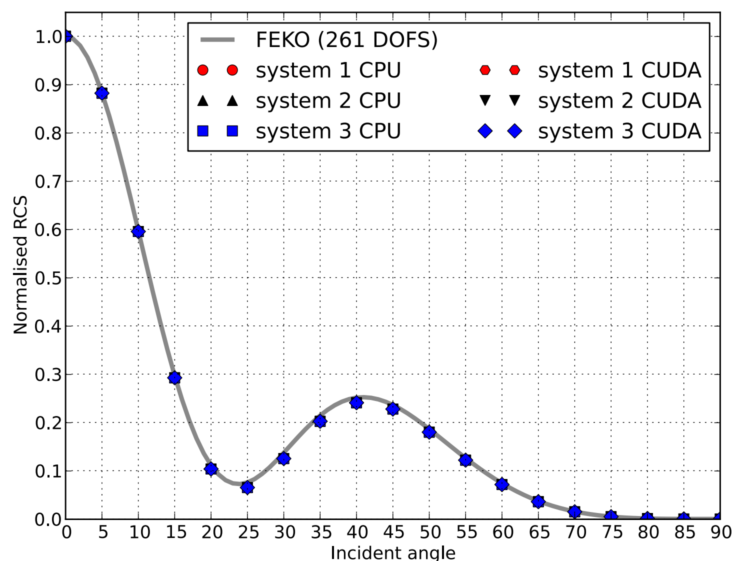


Figure 4.13: The computed normalised radar cross section of a 1λ square PEC plate as a function of the angle θ ($\phi = 0$) calculated in **double precision using** the mesh shown in Figure 4.9. Results are shown for both the CPU and CUDA implementations for the three test systems considered, with FEKO results being included as a reference. The results calculated on the three systems are indistinguishable.

4.5 Performance results

The performance results presented in this section are comprised of two parts. The first is a timing analysis of the various phases of the MOM process and the effect of their GPU-acceleration. The second set shows the speedups achieved through GPU acceleration. As was the case with the LU decomposition implementations discussed in Chapter 3, three test systems are used for the studies conducted here. In all cases, only the multi-core results using all the available cores in a system are presented as the CPU results. It should be noted that, in some cases, only the graphs for System 1 are presented in this chapter, with the equivalent charts for the other systems included as part of Appendix 6.2.

As for the LU decomposition performance results of Section 3.4.2, each of the curves shown give an indication of the problem size that equates to 1 GB and 4 GB, where applicable, of memory usage in terms of the size of the impedance matrix $[Z]$. This gives an indication of when the memory capacities of the CUDA devices used will be exceeded and shows whether or not the methods implemented are resilient with respect to limited memory resources.

4.5.1 Runtime contributions

In order to determine the importance of the acceleration of each of the phases of the MOM solution process, consider the graph depicted in Figure 4.14. This graph show the percentage contribution to the total runtime for the different phases for single precision CPU-based computation of the RCS (such as in Figure 4.12). It should be noted that, although all the other phases of the process, such as excitation vector calculation and post-processing – but not initialisation – are included (and grouped together), only the matrix assembly phase, LU decomposition phase, and back-substitution phase (to obtain the current coefficients) are explicitly labelled in the figure, since they contribute most significantly to the total runtime. Figure 4.14 also clearly shows that the dominant phase for the multi-core CPU-based computation of the RCS of the square plate is the assembly of the impedance matrix $[Z]$. It should be pointed out that although the phase of actually solving for the current coefficients contributes 8% of the computational time, this step is performed about 90 times – once for every angular step in the calculation of the monostatic RCS.

If only the matrix assembly phase is replaced by a CUDA implementation, the new distribution of computational time is given by Figure 4.15. Here, the contribution of the assembly phase is negligible, with even the other phases of the solution process not labelled explicitly contributing more significantly to the total runtime for smaller problems. This indicates that CUDA is able to provide a significant speedup to the matrix assembly phase, which is in agreement with previously published works as discussed in Section 4.1. With this acceleration, the LU decomposition phase is now the dominant phase of the solution process, with its contribution increasing as the size of the problem increases. Since the computational cost of the LU decomposition is $\mathcal{O}(N^3)$ and that of the solution of the currents and the matrix assembly phases are $\mathcal{O}(N^2)$, large enough problems of this type are expected to show a total dominance of the solution time by the LU decomposition.

The distribution for the complete CUDA-based implementation is given in Figure 4.16. This includes the CUDA acceleration of the matrix assembly, the LU decomposition, the excitation vector assembly, and post-processing phases. Finding the unknown currents by back-substitution is now the phase that now contributes most significantly to the total runtime, with a contribution in excess of 70%. The primary reason for its exclusion for CUDA-acceleration is the lack of a suitable routine as part of the MAGMA library at present. It should also be noted that in the case of an RCS calculation, the contribution of this phase is multiplied by the number of angular data points that are to be considered and as such the importance of the phase is heavily problem dependent.

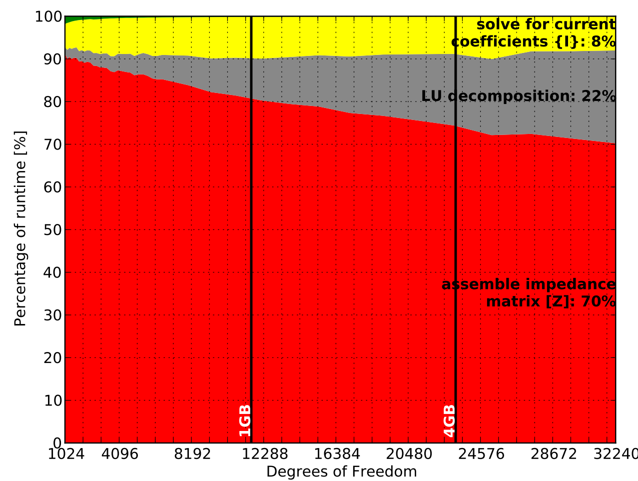


Figure 4.14: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.12. This figure shows the results for **single precision** calculations using the CPU-based implementation for **System 1**. The three phases that contribute most significantly to the total execution time are labelled, with the other phases (excluding initialisation) grouped together and just visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1 GB and 4 GB storage requirements (in terms of only the impedance matrix) for **single precision** complex values.

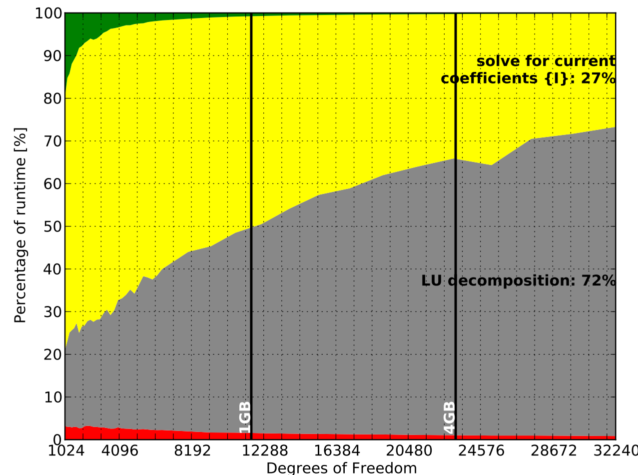


Figure 4.15: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.12. This figure shows the results for **single precision** calculations on **System 1** when only the matrix assembly phase is implemented in CUDA. The two phases that contribute most significantly to the total execution time are labelled, with the CUDA matrix assembly contribution shown at the bottom of the graph, and the other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1 GB and 4 GB storage requirements (in terms of only the impedance matrix) for **single precision** complex values.

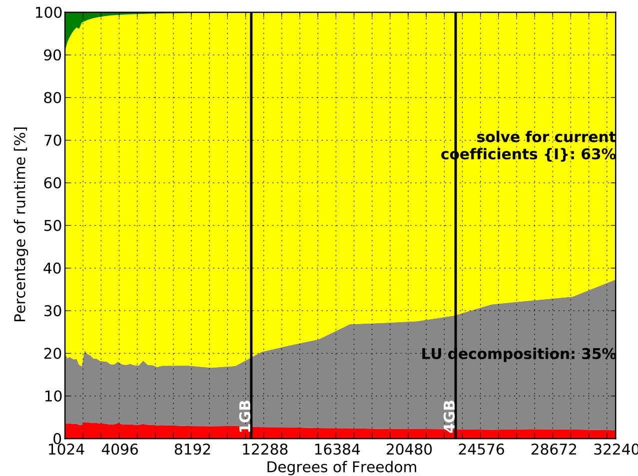


Figure 4.16: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.12. This figure shows the results for **single precision** calculations on **System 1** for the CUDA implementation. The two phases that contribute most significantly to the total execution time are labelled, with the matrix assembly contribution shown at the bottom of the graph and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1 GB and 4 GB storage requirements (in terms of only the impedance matrix) for **single precision** complex values. The phase to solve for the unknown current coefficients is not implemented in CUDA.

4.5.2 Speedups

With the investigation into the runtime contributions of the various phases in the MOM process in Section 4.5.1, it has already been indicated that the some of the phases of the MOM solution process have been successfully accelerated using CUDA. In this section the measured speedups for the three systems introduced in Section 3.4.1 are investigated for both single and double precision computation. The results shown are for the calculation of the RCS (as in Figure 4.12 and Figure 4.13) of the one wavelength square plate considered.

Single precision

The speedup results for the single precision complex case run on System 1 is given in Figure 4.17, with four speedup curves shown. The first of these are the speedup of the Method of Moments assembly phase, and the speedup of the LU decomposition phase, with the remaining curves showing two variants of the total solution speedup. The curve labelled **total solution speedup**, plotted in solid black, is the speedup for the solution when all the solution phases are accelerated using CUDA, and corresponds to the runtime distribution in Figure 4.16. The last curve is the solution speedup if only the matrix assembly phase is accelerated, as is represented by the runtime distribution plot of Figure 4.15. Here, it is clear that, although the assembly is accelerated by almost 300×, if the other phases are not also considered, the total system speedup decreases rapidly, with the attainable speedup limited by the runtime contribution of the other phases. The total solution speedup is measured at between 8× and 10× for most of the problem sizes considered. It should be noted that if only one incident angle is considered for the calculation – an effective speedup of the current coefficient solution phase of around 90× – the peak total solution speedup is expected to be about 50×.

The speedup results for System 2 are similar to those of System 1 and are given in Figure 4.18.

Here, the speedup for the matrix assembly phase is close to 300×. Furthermore, since the LU decomposition phase speedup is also similar to that of System 1, the total speedup obtained remains between 8× to 10× for the larger problems. The runtime distribution curves for this system are included in Appendix 6.2.

Figure 4.19 shows the single precision speedup results for System 3. Since the CPU-based performance of this system (as illustrated by the LU decomposition results of Section 3.4) is significantly higher than the other two systems, while the GPU offers similar performance, the measured speedups are not expected to be as high as for the other systems. This is indeed the case, with Figure 4.19 showing matrix assembly and LU decomposition speedups of around 40× and 2×, respectively. This translates to a total solution speedup of 4× to 5× times. As is the case for the System 2 results, the runtime distribution curves for System 3 are provided as part of Appendix 6.2.

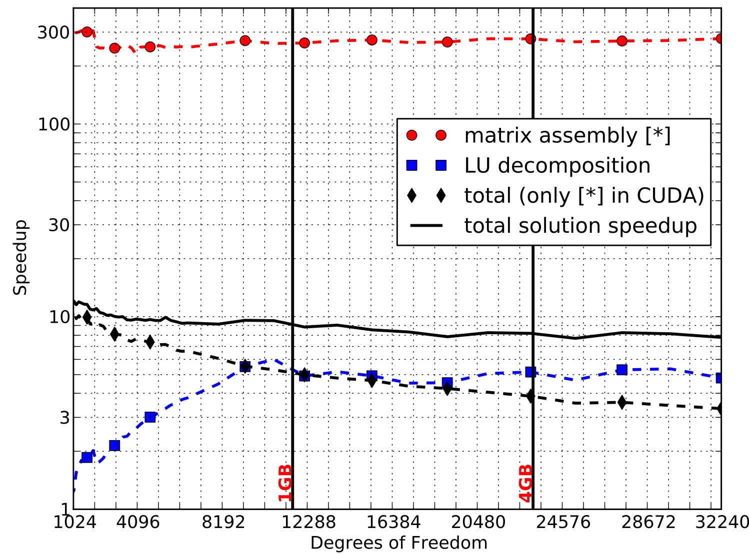


Figure 4.17: Measured speedups of the CUDA implementation over the CPU implementation on **System 1** in **single precision** as a function of the problem size (number of degrees of freedom). The phase-only speedups for the matrix assembly (●) and LU decomposition (■) are shown along with the total system speedup (–) and the system speedup if only the matrix assembly is accelerated using CUDA (♦). A logarithmic scale is used on the vertical axis for better comparison and vertical lines representing impedance matrix sizes corresponding to 1 GB and 4 GB storage requirements in **single precision** are also given.

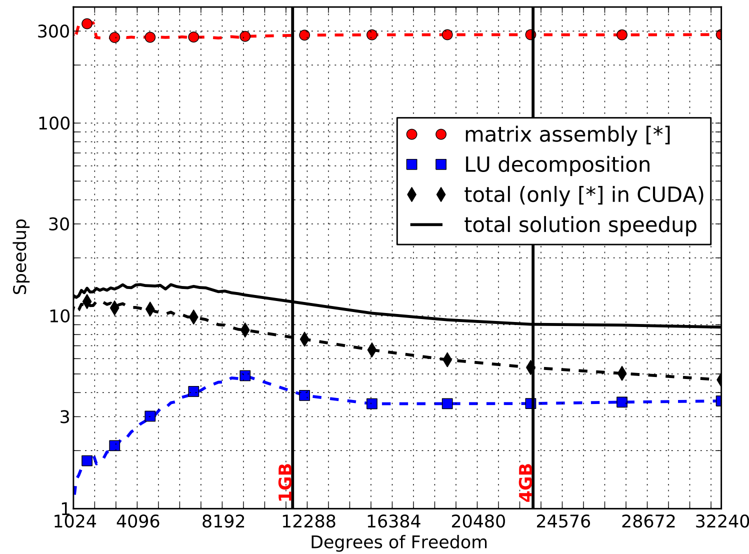


Figure 4.18: Measured speedups of the CUDA implementation over the CPU implementation on **System 2** in **single precision** as a function of the problem size (number of degrees of freedom). The phase-only speedups for the matrix assembly (●) and LU decomposition (■) are shown along with the total system speedup (–) and the system speedup if only the matrix assembly is accelerated using CUDA (♦). A logarithmic scale is used on the vertical axis for better comparison and vertical lines representing impedance matrix sizes corresponding to 1 GB and 4 GB storage requirements in **single precision** are also given.

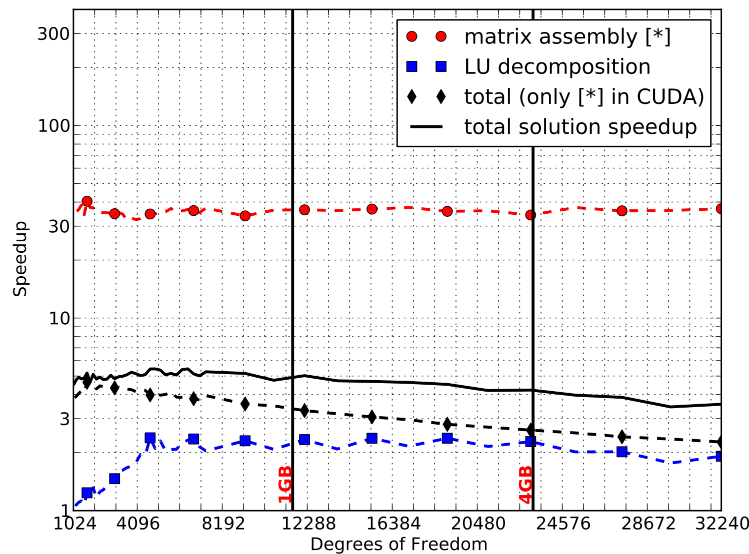


Figure 4.19: Measured speedups of the CUDA implementation over the CPU implementation on **System 3** in **single precision** as a function of the problem size (number of degrees of freedom). The phase-only speedups for the matrix assembly (●) and LU decomposition (■) are shown along with the total system speedup (–) and the system speedup if only the matrix assembly is accelerated using CUDA (♦). A logarithmic scale is used on the vertical axis for better comparison and vertical lines representing impedance matrix sizes corresponding to 1 GB and 4 GB storage requirements in **single precision** are also given.

As stated, one of the contributions of this research is the investigation of the use of GPUs in the extension of the usefulness (measured in terms of performance) of an older system such as System 1. To this end, the CPU and CUDA results for the various systems in single precision are summarised in Figure 4.20, where the performance of the various implementations are compared to the CPU results for System 1. Here, it can be seen that although the CPU results for System 2 and System 3 indicate that they are about 1.5× and 6× faster, respectively, all the CUDA results outperform the baseline by at least 8×. Thus, the addition of a GPU allows System 1 to perform better than the other systems used in the comparison. This performance advantage is removed, however, if GPU acceleration is used in each of the other systems.

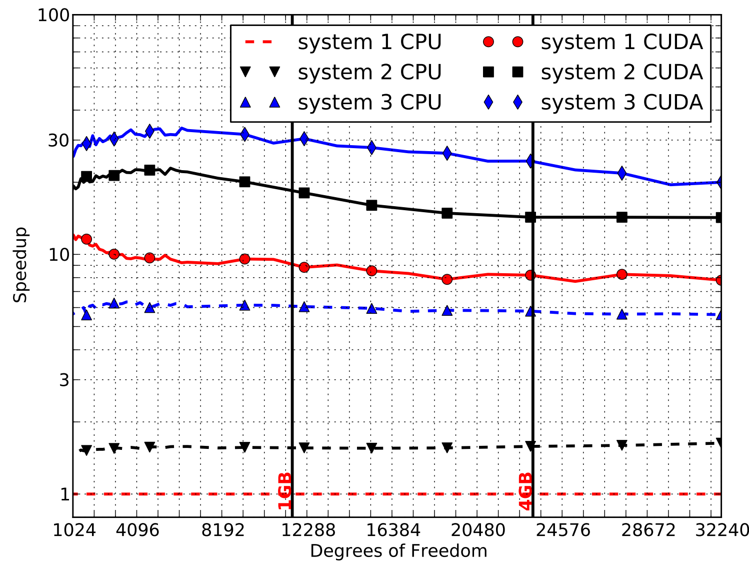


Figure 4.20: Total solution speedup relative to the **System 1** CPU results as a function of the problems size for for the various other CPU and CUDA implementations in **single precision**. The CPU speedups for **System 2** (▼) and **System 3** (▲) as well as the CUDA speedups for **System 1** (•), **System 2** (■), and **System 3** (⧫) are shown.

Double precision

The speedup results presented in the previous section for the single precision case are repeated here for double precision calculations, with much of the discussion also applicable. The results for System 1, System 2, and System 3 are shown in Figure 4.21, Figure 4.22, and Figure 4.23, respectively.

System 1 and System 2 shown similar results for the speedup of the matrix assembly phase, with a speedup of about 45× measured in each case. For the LU decomposition phase, however, the speedup in the case of System 1 is between 2× and 3× compared to a speedup of less than 2× for System 2. This is due to the difference in speedups when comparing the CPUs and GPUs of the two systems and is evident in the `ZGETRF` results of Section 3.4.2. For the `ZGETRF` routines – as also used here – the CPU speedup of System 2 over System 1 is about 4×, whereas System 2's GPU results are only about 50% faster than those of System 1. The total solution speedup for both systems is more than 3× over most of the range of problem sizes.

Although the CUDA matrix assembly for System 3 is around 10× faster than the multi-core CPU implementation running on 8 cores (as shown in the System 3 CUDA speedup curve of Figure 4.23), the total solution speedup decreases from about 3× for problems of around 1000 degrees of freedom to just more than unity for the largest problem considered (31621 DOFs). This mediocre performance is a result of the high CPU-based LU decomposition performance, as shown in Section 3.4.2, along with the poor double precision performance compared to the single precision performance of the GT200 device used (see Figure 3.9 and Figure 4.19 specifically).

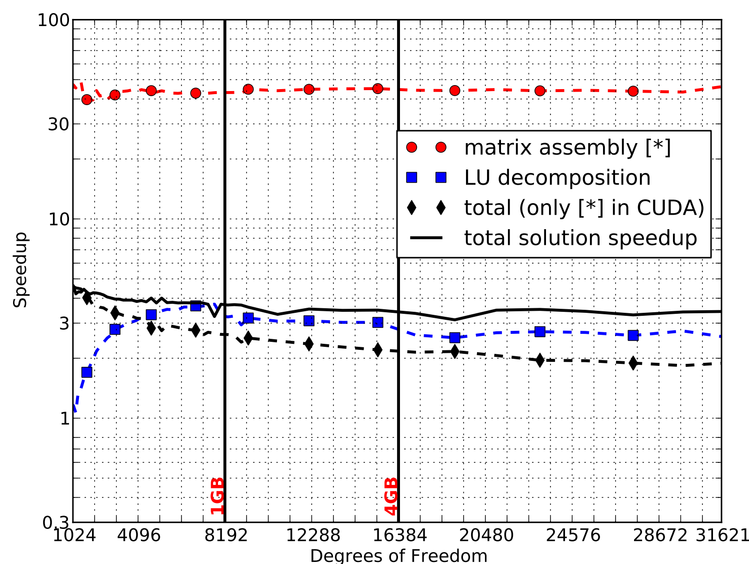


Figure 4.21: Measured speedups of the CUDA implementation over the CPU implementation on **System 1** in **double precision** as a function of the problem size (number of degrees of freedom). The phase-only speedups for the matrix assembly (•) and LU decomposition (■) are shown along with the total system speedup (–) and the system speedup if only the matrix assembly is accelerated using CUDA (♦). A logarithmic scale is used on the vertical axis for better comparison and vertical lines representing impedance matrix sizes corresponding to 1 GB and 4 GB storage requirements in **double precision** are also given.

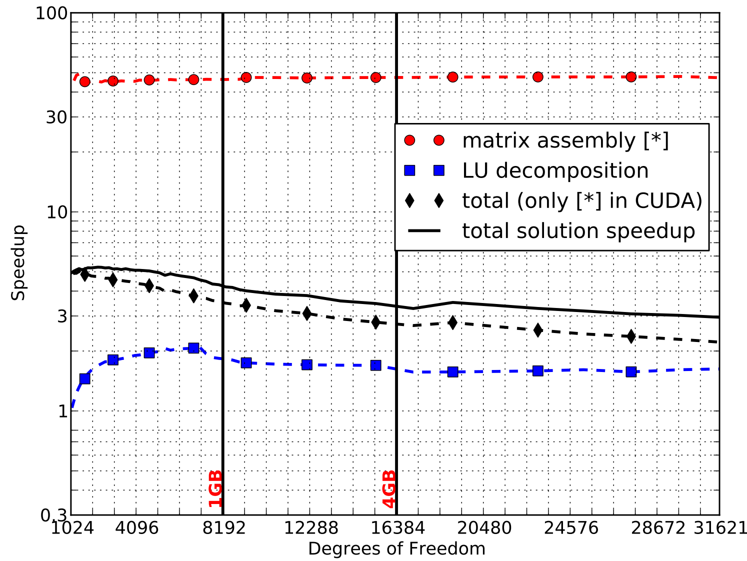


Figure 4.22: Measured speedups of the CUDA implementation over the CPU implementation on **System 2** in **double precision** as a function of the problem size (number of degrees of freedom). The phase-only speedups for the matrix assembly (•) and LU decomposition (■) are shown along with the total system speedup (–) and the system speedup if only the matrix assembly is accelerated using CUDA (♦). A logarithmic scale is used on the vertical axis for better comparison and vertical lines representing impedance matrix sizes corresponding to 1 GB and 4 GB storage requirements in **double precision** are also given.

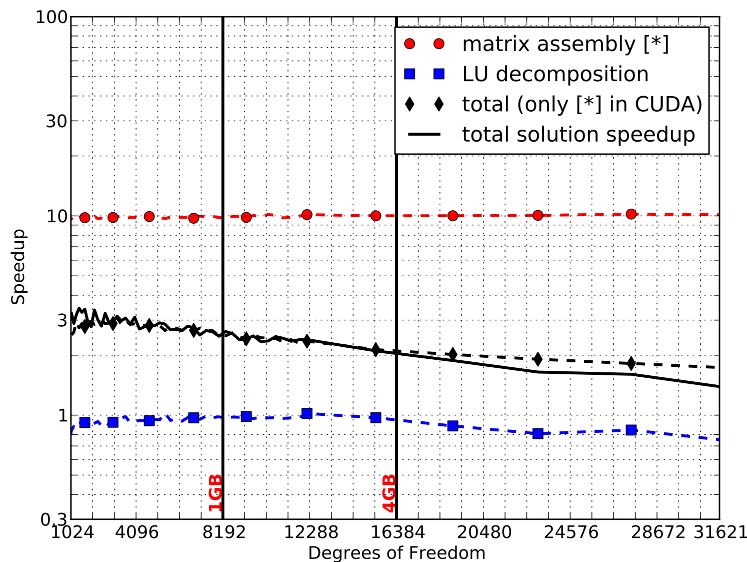


Figure 4.23: Measured speedups of the CUDA implementation over the CPU implementation on **System 3** in **double precision** as a function of the problem size (number of degrees of freedom). The phase-only speedups for the matrix assembly (•) and LU decomposition (■) are shown along with the total system speedup (–) and the system speedup if only the matrix assembly is accelerated using CUDA (♦). A logarithmic scale is used on the vertical axis for better comparison and vertical lines representing impedance matrix sizes corresponding to 1 GB and 4 GB storage requirements in **double precision** are also given.

Figure 4.24 shows the double precision equivalent to Figure 4.20, where the aim is to ascertain the performance of the CPU and CUDA implementations of the three systems relative to the CPU results of System 1 (indicated by the baseline at 1×). In this case, the CUDA-accelerated implementation on System 1 is not able to outperform the CPU-based implementation on System 3. It does, however, reduce the performance gap from about 5× to about 1.7×. The CUDA results of System 1 do perform better than the CPU-only results of System 2 by at least 50% for the range of problem sizes considered. Furthermore, although the CPU-only results of System 3 offer more than double the performance of those of System 2, the addition of a CUDA GPU allows System 2 to outperform System 3.

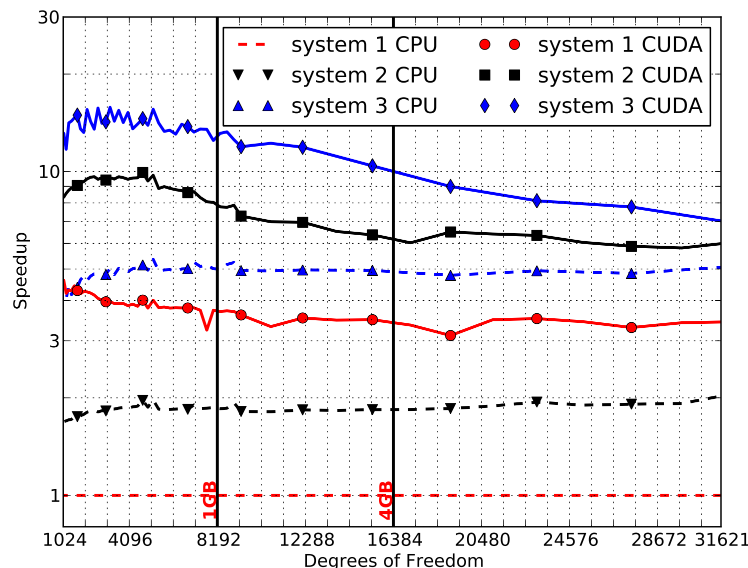


Figure 4.24: Total solution speedup relative to the **System 1** CPU results as a function of the problems size for for the various other CPU and CUDA implementations in **double precision**. The CPU speedups for **System 2** (▼) and **System 3** (▲) as well as the CUDA speedups for **System 1** (●), **System 2** (■), and **System 3** (♦) are shown.

4.5.3 Discussion of results

Before discussing the implications of the results presented thus far, the question of the performance improvement of the multi-core CPU implementation – used to obtain the CPU results – is addressed. Consider Figure 4.25, which shows the speedup of the single precision multi-core implementation (using OpenMP) over the same implementation running on a single core (of System 2). The same square plate RCS problem as for the performance results already presented is used, although the problem sizes considered are not as large. The multi-core speedups for the LU decomposition (implemented in ACML), and matrix assembly phases are presented, along with the total solution speedup. In all cases, the speedup is between 3× and 4× times, indicating that the OpenMP-based multi-core CPU implementation scales well with the number of processes.

The equivalent double precision results are shown in Figure 4.26, and follow much the same trend as the single precision results of Figure 4.25. The measured multi-core CPU speedups for both individual phases, as well as the total solution speedup are above 3×. These results indicate that the OpenMP implementation presented in Section 4.3.4 is relatively efficient.

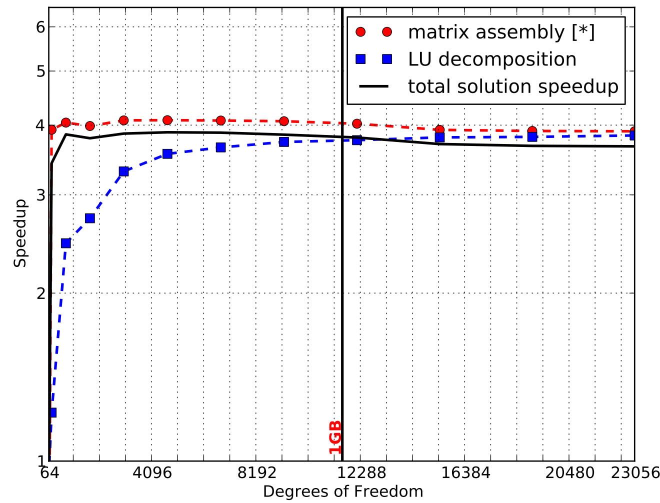


Figure 4.25: Measured speedups of the multi-core CPU implementation over a single core run on **System 2** in **single precision** as a function of the problem size (number of degrees of freedom). The phase-only speedups for the matrix assembly (•) and LU decomposition (■) are shown along with the total solution speedup (–). A logarithmic scale is used on the vertical axis and vertical lines representing impedance matrix sizes corresponding to 1 GB and 4 GB storage requirements in **single precision** are also given.

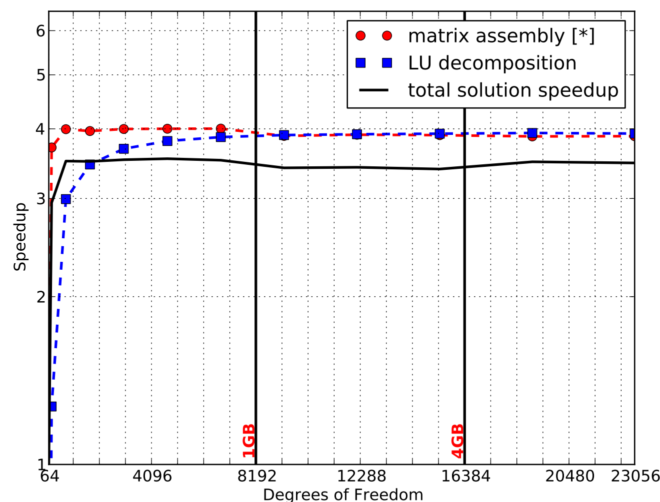


Figure 4.26: Measured speedups of the multi-core CPU implementation over a single core run on **System 2** in **double precision** as a function of the problem size (number of degrees of freedom). The phase-only speedups for the matrix assembly (•) and LU decomposition (■) are shown along with the total solution speedup (–). A logarithmic scale is used on the vertical axis and vertical lines representing impedance matrix sizes corresponding to 1 GB and 4 GB storage requirements in **double precision** are also given.

When the matrix assembly phase is considered alone, the CUDA implementation consistently outperforms the multi-core CPU implementation. Here speedups of up to 300× are measured in single precision. Even for System 3, a speedup of around 40× is measured. For the double precision case, the speedups are not as high, with 40–50× and 10× being measured for the first two systems and System 3, respectively. It should be added that these speedups are maintained even when the amount of memory available on the CUDA devices (measured in terms of impedance matrix storage size) has been exceeded. This indicates that the super-block CUDA implementation of Section 4.3.5 does not limit performance. The high performance of this particular phase of the solution process can be attributed to the limited communication required between the threads resulting in a highly parallel computational task, for which the CUDA GPUs are well suited. This argument also holds for the multi-core CPU implementation of the matrix assembly.

Since the performance of one of the other contributing phases, the LU decomposition, is discussed in detail in Section 3.4, this is not repeated here. It suffices to say that the speedups obtained for this phase are significantly lower than those of the matrix fill. However, even though the runtime contribution of this phase is increasing with problems size (since it is $\mathcal{O}(N^3)$ in computational cost compared to $\mathcal{O}(N^2)$ for the other phases), the total solution speedups have been measured as more than unity (break-even) for the range of problem sizes considered, with peak speedups of 12× and 5× obtained for single and double precision, respectively (both for System 2).

4.6 Conclusion

This chapter discussed the implementation and performance benchmarking of a complete MOM solution process for solving simple scattering problems. This has been successfully applied to the problem of computing the monostatic RCS of a square PEC plate. Although a number of prior works that tackle similar problems exist, this particular solution offers a number of novel additions.

One of these is the inclusion of native double precision support for the entire MOM solution process which extends the works in [89] and [95], where double precision is also used. In the former this is implemented as double-single precision computation [93] and not native double precision and applied to general Helmholtz problems such as acoustics. In [95], CUDA acceleration (in native double precision) is only applied to the LU decomposition phase for wire problems. Double precision results for the matrix assembly phase have been presented in [11], with this work an extension of that research. Here, the other phases of the solution process, as well as a multi-core CPU-based implementation are included, and this accounts for the difference in speedup results.

Staying on the topic of precision, this chapter presents both double and single precision results for a wide range of problem sizes benchmarked on three difference systems, each with its own strengths and weaknesses. This allows for a better understanding of the relative performance between the precisions. It it is clear that, in the GPU case, it is not as simple as a factor two improvement for single precision over double precision. This is the general consensus for CPU-based implementations and is indicated by the CPU results of Section 3.4.2. This comparative study further allows for the investigation of the extension of usability of older machines through GPU upgrades for a type of problem other than the LU decomposition discussed at length in Chapter 3. The results indicate that, at least for single precision, a comparatively cheap GPU upgrade to System 1 allows for performance that is at worst comparable to the newer systems considered. For double precision, the GPU-accelerated System 1 is able to outperform only System 2's CPU results and not those of System 3. The addition of a GPU or a Tesla 20-series compute card – built using the GF100 architecture and promising improved double precision

performance – may be able to further close the gap.

One of the primary aims of the panel-based LU decomposition, discussed in Chapter 3, is dealing with the limited amount of memory available on CUDA devices (especially the consumer-level devices used in System 1 and System 2). This has also been considered in this chapter, with the implementation of the super-block method (adapted from [91]) in the CUDA implementation of the matrix assembly phase. From the performance results presented in Section 4.5, it is clear that no degradation in the performance of the matrix assembly phase occurs if the impedance matrix size exceeds the available device memory. For the problem sizes considered, the memory required exceeds the available memory by a considerable factor for both single and double precision computation (up to eight and 16 times, respectively).

Much of the discussion in this chapter was related to the concurrent development of both a multi-core CPU and CUDA implementation of the solution process considered. Although one could argue that this could result in implementations that are non-optimal in either case, this is of secondary concern. The simple framework presented allows for the rapid development of stable, accurate implementations. Furthermore, since much of the code is shared its maintenance and extension – to support alternate formulations or integration schemes, for example – are both greatly simplified. The rapid development process also allows for the identification of critical routines (such as the LU decomposition and linear system solve in this case) whose acceleration is imperative to obtain improved total solution performance.

Once aspect that has not been addressed in this chapter is the use of multiple CUDA devices or the concurrent use of GPU and CPU computation. Due to the data-parallel nature of the computations and the framework implemented here, specifically the super-block based matrix assembly implementation, distributing the matrix assembly between multiple similar CUDA devices installed in the same host should not pose too much of a problem. If the aim is to obtain an implementation that makes full use of all the CPU and GPU resources available in a system, load-balancing issues could lead to sub-par performance. In this case, the use of a scheduling system such as StarPU [82] (discussed briefly pertaining to the LU decomposition in Chapter 3) would prove useful. This would be further helped by the fact that the independence of the entries in the impedance matrix, for example, make the process of specifying parallel computational tasks trivial.

Chapter 5

FEM waveguide analysis

One of the first applications of the Finite Element Method (FEM) to the field of electromagnetics, was its use in the analysis of hollow waveguide problems by Silvester in 1969 [109]. Since then, many more publications on the topic, including a number of books, have been produced [110, 111, 112, 113].

This chapter considers the application of the Finite Element Method to waveguide analysis and more specifically waveguide problems that require the solution of eigenvalue problems. These cutoff and dispersion problems have been considered in a number of other publications such as [88, 111, 113]. This chapter does not aim to provide an in-depth analysis of the formulation aspects of the FEM as applied to these problems, instead focussing on some of the details of the implementation.

With regards to the implementation, one of the contributions of this chapter is a discussion on the use of FEniCS in the analysis of electromagnetic waveguide problems. FEniCS is a collection of software for the automated modelling and solution of differential equations using the Finite Element Method [16]. Although it has been applied to electromagnetic problems (such as eddy current problems in [114]), this work represents the first time that it has been applied to waveguide analysis, and specifically waveguide eigenproblems.

The second implementation aspect considered relates to the solution of the eigensystems that result when applying the FEM to the analysis of waveguide cutoff or dispersion problems. As in Chapter 3 and Chapter 4, the use of CUDA in accelerating the solution process is considered. To this end, a standard eigenvalue problem solver that finds a small number of extreme eigenvalues (largest magnitude, for example) and their associated eigenvectors for dense matrices is implemented using ARPACK [14]. The conversion of generalised eigenvalue problems to standard eigenvalue problems is also considered. CUDA-accelerated implementations are presented and the relative performance of the CUDA and CPU implementations are considered for two of the systems introduced in Section 3.4.1.

It should be noted that, although the CUDA implementations of Chapter 3 and Chapter 4 both succeed (with varying levels of success) to overcome the inherent memory limitations associated with using CUDA devices, this is not considered here. Furthermore, the eigensolver implementation is aimed at general dense matrices even though finite element matrices are sparse and can be symmetric [88]. The switch to dense matrices is touched upon again at the end of this chapter, with the use of non-symmetric (general) matrix routines being motivated by the fact that this should make the implementation more widely applicable. This also allows for the reuse of the LU decomposition routines from Chapter 3.

As a start to the discussion, a summary of relevant proir research is presented in Section 5.1. This is followed by a brief discussion on the finite element formulation for the problems considered in Section 5.2, where the assembly of the various matrices required is discussed and the generalised eigenvalue problems addressed. Section 5.3 then presents the actual implementation

details, with the use of FEniCS discussed in-depth for the three example problems considered. The implementation of the generalised eigensolver – both the CPU and CUDA versions – is also addressed.

In terms of results, Section 5.4 shows the FEniCS and eigensolver implementations applied to three different waveguide examples. In each case, the computed results are compared to either analytical values, or results taken from other sources. This is followed by an analysis of the performance of the ARPACK-based standard eigenproblem solver and the generalised eigenproblem solver, of which it forms a part, in Section 5.5. Conclusions and future directions for the research are discussed in Section 5.6.

5.1 Related work

When considering the finite element analysis of waveguide structures, a multitude of publications exist on the topic. Most notable – and used for much of the formulations presented here, as well as reference results – are [111], [113], and [115], with the theoretical behaviour of the guides discussed in [13]. The FEM and its application to electromagnetic problems is also addressed in [88], [110], [116], [117], and [118].

The FEniCS Project, used here to implement the FEM formulations, represents a collection of free software designed to aid in the automation of computational mathematical modelling [16]. Traditionally, the development of the FEniCS set of packages has been driven from an applied mathematics point of view, with many users showing applications in computational fluid dynamics or structural mechanics. The use of FEniCS in electromagnetic modelling is limited, with examples such as [114] showing its use in eddy current simulations. This work represents the first, to the knowledge of the author, application of FEniCS to electromagnetic waveguide eigenanalysis.

For the CUDA acceleration of eigenvalue problems, a number of examples exist, including [119] and [120]. The former is limited to the solution of the standard eigenvalue problem for dense tri-diagonal symmetric matrices. The latter also only considers the standard eigenvalue problem, again for symmetric matrices using the Lanczos algorithm [63]. CUDA has been applied to other computational electromagnetic finite element problems in [121], where the acceleration of the Discontinuous Galerkin Finite Element Method is considered for the solution of driven, time-domain finite element problems. This contrasts with the implementation considered here, where the FEM is used in the frequency domain to solve resonance problems for bounded structures.

The eigensolver implementation presented here is the continuation of the research presented in [15], where the CUDA acceleration of the k-step Arnoldi factorisation [122] was considered. This factorisation forms the basis of methods such as the Implicitly Restarted Arnoldi Method, which can be seen as a generalisation of the Lanczos method to non-symmetric matrices and is used in the (non-symmetric) eigensolvers of ARPACK [14]. Although [15] also only considers the standard eigenvalue problem, this work extendeds it to the generalised eigenvalue problem, by making use of the shift-invert procedure, as suggested in the ARPACK literature [14]. The current implementation of this shift-invert process (discussed in Section 5.3.2) makes use of LAPACK routines and thus allows for the use of the MAGMA library – as in Chapter 3 and Chapter 4 – to provide CUDA acceleration, with an optimised CPU implementation being provided by ACML.

5.2 FEM formulation for cutoff and dispersion analysis

In this section, the formulation of two related waveguide analysis problems are briefly considered, namely cutoff and dispersion analysis. As a starting point, consider the representation of an arbitrary waveguide shown in Figure 5.1.

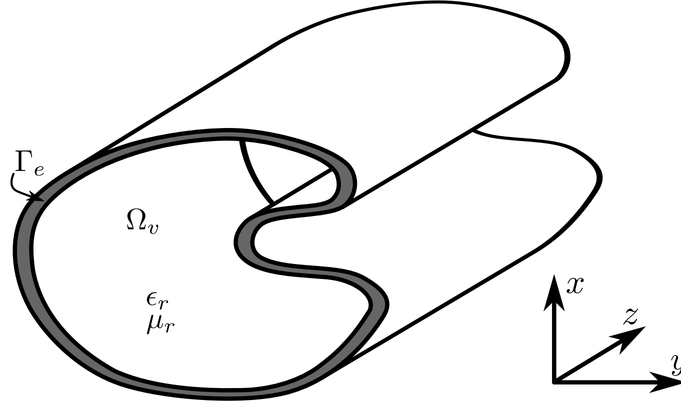


Figure 5.1: A long waveguide with an arbitrary cross-section aligned with the z-axis. Shown is the interior domain of the waveguide Ω_v, surrounded by a PEC surface modelled as an electric wall Γ_e. The relative permittivity and permeability in the guide are ϵ_r and μ_r. These are not necessarily constant over the guide cross-section.

In the full wave analysis of such guides, it is required to solve the vector Helmholtz equation [111, 115], along with the relevant boundary conditions. The time-independent versions of these are given by

$$\nabla \times \frac{1}{\mu_r} \nabla \times \vec{E} - k_o^2 \epsilon_r \vec{E} = 0 \quad \text{in } \Omega_v, \tag{5.1}$$

$$\hat{n} \times \vec{E} = 0 \quad \text{on } \Gamma_e, \tag{5.2}$$

$$\hat{n} \times \nabla \times \vec{E} = 0 \quad \text{on } \Gamma_m, \tag{5.3}$$

where Ω_v is the domain represented by the interior of the waveguide, and Γ_e and Γ_m are electric and magnetic walls, respectively. The parameters μ_r and ϵ_r are the relative permeability and permittivity, respectively, of the medium inside the guide and may be position dependent. The operating wavenumber k_o is related to the operating frequency (f_o) as [104]

$$k_o = \frac{2\pi f_o}{c_0}, \tag{5.4}$$

with c_0 the speed of light in free space.

It should be noted that, this boundary value problem can also be written in terms of the magnetic field [111]. Due to the similarity of the formulations, many of the discussions that follow are applicable to both cases and the magnetic field formulation is not considered further.

If the guide of Figure 5.1 is sufficiently long with the z-axis chosen parallel to the central axis as shown, the z-dependence of the electric field can be assumed to be of the form $e^{-\gamma z}$, with

$$\gamma = \alpha + j\beta, \tag{5.5}$$

a complex propagation constant [13, 113]. Using this assumed form and splitting the electric field into transverse ($\vec{E}_t$) and axial ($\hat{z}E_z$) components, the electric field of (5.1) is then given by

$$\vec{E}(x, y, z) = [\vec{E}_t(x, y) + \hat{z}E_z(x, y)]e^{-\gamma z}, \tag{5.6}$$

with x and y the Cartesian coordinates in the cross-sectional plane of the waveguide, and z the coordinate along the length of the guide. This z-dependence of the electric field permits the solution of the boundary value problem (BVP) defined by (5.1), (5.2), and (5.3). The two-dimensional domain Ω represents the cross-section of the waveguide in the xy-plane, as opposed to the interior volume of the guide Ω_v.

Using the definition of the electric field in (5.6), the continuous functional corresponding to the BVP can be written as [111, 113, 115]

$$F(\vec{E}) = \int_\Omega \frac{1}{\mu_r}(\nabla_t \times \vec{E}_t)\cdot(\nabla_t \times \vec{E}_t) - k_o^2\epsilon_r\vec{E}_t\cdot\vec{E}_t + \frac{1}{\mu_r}(\nabla_t E_z + \gamma\vec{E}_t)\cdot(\nabla_t E_z + \gamma\vec{E}_t) - k_o^2\epsilon_r E_z E_z d\Omega, \quad (5.7)$$

with

$$\nabla_t = \frac{\partial}{\partial x}\hat{x} + \frac{\partial}{\partial y}\hat{y}, \quad (5.8)$$

the transverse del operator. Note that the integration is over the two-dimensional cross-sectional domain, and not the interior volume Ω_v.

5.2.1 Cutoff analysis

One of the simplest cases to consider, and often a starting point when testing a new finite element implementation, is waveguide cutoff analysis. When a waveguide is operating at cutoff, the electric field is uniform along the z-axis, which corresponds with $\gamma = 0$ in (5.6) [13]. Substituting $\gamma = 0$ into (5.7) yields the following functional

$$F_c(\vec{E}) = \int_\Omega \frac{1}{\mu_r}(\nabla_t \times \vec{E}_t)\cdot(\nabla_t \times \vec{E}_t) - k_c^2\epsilon_r\vec{E}_t\cdot\vec{E}_t + \frac{1}{\mu_r}(\nabla_t E_z)\cdot(\nabla_t E_z) - k_c^2\epsilon_r E_z E_z d\Omega. \quad (5.9)$$

Here, the symbol for the operating wavenumber k_o has been replaced with k_c, indicating that the quantity of interest is now the cutoff wavenumber (the subscript c is also used for the functional itself). This wavenumber, in addition to the field distribution at cutoff, are the quantities of interest in of problems of this kind.

Using two-dimensional curl-conforming vector basis functions ($\vec{N}_i$), such as the basis functions from the Nédélec function space of the first kind [123], for the discretisation of the transverse field, and scalar basis functions (L_i) for the axial components [111, 113], the discretised field components (indicated by the prime) of (5.6) are given by [111, 113]

$$\vec{E}_t' = \sum_{i=1}^{N_N}(e_t)_i\vec{N}_i, \quad (5.10)$$

$$E_z' = \sum_{i=1}^{N_L}(e_z)_i L_i. \quad (5.11)$$

Here, $(e_t)_i$ and $(e_z)_i$ are the coefficients of the ith vector and scalar basis functions, respectively, while N_N and N_L are the total number of each type of basis function used in the discretisation. The letters N and L are chosen for the basis function names as a reminder that the basis functions come from a Nédélec function space and a Lagrange polynomial space, respectively.

Substituting the discretised field equations of (5.10) and (5.11) into the functional (5.9) and applying a minimisation procedure, results in the following matrix equation

$$\begin{bmatrix} S_{tt} & 0 \\ 0 & S_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix} = k_c^2 \begin{bmatrix} T_{tt} & 0 \\ 0 & T_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix}, \quad (5.12)$$

or simply

$$[S]\{e\} = k_c^2 [T]\{e\}. \quad (5.13)$$

The matrix equations of (5.12) and (5.13) are in the form of generalized eigenvalue problems, with the square of the cutoff wavenumber the unknown eigenvalue. The sub-matrices S_{oo} and T_{oo} (with $oo = tt$ or $oo = zz$) represent the stiffness and mass matrices common in finite element literature [88, 111]. The subscripts tt and zz indicate transverse and axial components, respectively, and the entries of the matrices of (5.12) are defined as [111, 113]

$$(S_{tt})_{ij} = \int_\Omega \frac{1}{\mu_r}(\nabla_t \times \vec{N}_i) \cdot (\nabla_t \times \vec{N}_j) d\Omega, \tag{5.14}$$

$$(T_{tt})_{ij} = \int_\Omega \epsilon_r \vec{N}_i \cdot \vec{N}_j d\Omega, \tag{5.15}$$

$$(S_{zz})_{ij} = \int_\Omega \frac{1}{\mu_r}(\nabla_t L_i) \cdot (\nabla_t L_j) d\Omega, \tag{5.16}$$

$$(T_{zz})_{ij} = \int_\Omega \epsilon_r L_i L_j d\Omega, \tag{5.17}$$

with $\int_\Omega \cdot d\Omega$ representing integration over the cross-section of the waveguide.

In (5.12), the possible cutoff wavenumbers k_c, are the square roots of the eigenvalues of the system, and the elements of the corresponding eigenvectors are the coefficient of the basis functions as in (5.10) and (5.11). As such, the solution of the eigensystem not only allows for the computation of the cutoff wavenumbers, but also for the visualisation of the fields associated with the modes by substituting the elements of the computed eigenvector into (5.10) and (5.11). It should be noted that transverse electric (TE) modes will have zeros as coefficients for the scalar basis functions ($\{e_z\} = 0$), whereas transverse magnetic (TM) modes will have $\{e_t\} = 0$, although this condition only holds at cutoff [13].

5.2.2 Dispersion analysis

In the case of cutoff analysis discussed in Section 5.2.1, the aim is to obtain the value of the cutoff wavenumber $k_o^2 = k_c^2$ for a given propagation constant γ, namely $\gamma = 0$. For most waveguide design applications, however, k_o is specified and the propagation constant is calculated from the resultant eigensystem [111, 113]. This calculation can be simplified somewhat by making the following substitution into the original functional of (5.7) [113]

$$\vec{E}_{t,\gamma} = \gamma \vec{E}_t, \tag{5.18}$$

which results in the modified functional

$$F_d(\vec{E}) = \int_\Omega \frac{1}{\mu_r}(\nabla_t \times \vec{E}_{t,\gamma}) \cdot (\nabla_t \times \vec{E}_{t,\gamma}) - k_o^2 \epsilon_r \vec{E}_{t,\gamma} \cdot \vec{E}_{t,\gamma} - \gamma^2 \left[\frac{1}{\mu_r}(\nabla_t E_z + \vec{E}_{t,\gamma}) \cdot (\nabla_t E_z + \vec{E}_{t,\gamma}) - k_o^2 \epsilon_r E_z E_z \right] d\Omega. \tag{5.19}$$

Using the same field discretisation as for cutoff analysis discussed in Section 5.2.1 (given in (5.10) and (5.11)), the matrix equation associated with the solution of the BVP presented at the start of this chapter is given by

$$\begin{bmatrix} A_{tt} & 0 \\ 0 & 0 \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix} = \gamma^2 \begin{bmatrix} B_{tt} & B_{tz} \\ B_{zt} & B_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix}, \tag{5.20}$$

or

$$[A]\{e\} = \gamma^2 [B]\{e\}, \tag{5.21}$$

with

$$\left[A_{tt}\right] = \left[S_{tt}\right] - k_o^2 \left[T_{tt}\right], \tag{5.22}$$
$$\left[B_{zz}\right] = \left[S_{zz}\right] - k_o^2 \left[T_{zz}\right]. \tag{5.23}$$

This is also in the form of a generalised eigenvalue problem. In this case, however, the eigenvalues correspond with the square of the complex propagation constant (γ).

The matrices $\left[S_{tt}\right]$, $\left[T_{tt}\right]$, $\left[S_{zz}\right]$, and $\left[T_{zz}\right]$ are those defined in Section 5.2.1 with entries given by (5.14), (5.15), (5.16), and (5.17) respectively. The entries of the other sub-matrices, B_{tt}, B_{tz}, and B_{zt}, are given by

$$(B_{tt})_{ij} = \int_\Omega \frac{1}{\mu_r} \vec{N}_i \cdot \vec{N}_j d\Omega, \tag{5.24}$$
$$(B_{tz})_{ij} = \int_\Omega \frac{1}{\mu_r} \vec{N}_i \cdot \nabla_t M_j d\Omega, \tag{5.25}$$
$$(B_{zt})_{ij} = \int_\Omega \frac{1}{\mu_r} \nabla_t M_i \cdot \vec{N}_j d\Omega. \tag{5.26}$$

For lossless waveguides, the square of the imaginary part of the propagation constant (β^2 in (5.5)) can be shown to be less than a real value (θ^2) dependent on the operating wavenumber as well as the relative permittivity and permeabilities of the medium in the guide [115]. The expression for θ^2 is [115]

$$\theta^2 = k_o^2 \mu_{r,max} \epsilon_{r,max}, \tag{5.27}$$

where k_o is the operating wavenumber of (5.4) and $\mu_{r,max}$ and $\epsilon_{r,max}$ are, respectively, the maximum relative permeability and permittivity in the guide cross-section. The relation between θ^2 and β^2 is given by

$$\beta^2 \leq \theta^2 \tag{5.28}$$

Note that from (5.5) and (5.28), it follows that if $\alpha = 0$ – as is the case for propagating modes in a lossless guide – then the relationship between θ^2 and γ^2 is given by

$$\begin{aligned} \gamma^2 &= (j\beta)^2 \\ &= -\beta^2 \\ &\geq -\theta^2. \end{aligned} \tag{5.29}$$

There is thus a lower bound ($-\theta^2$) on the eigenvalue of (5.20).

5.2.3 The generalised eigenvalue problem

As shown in Section 5.2.1 and Section 5.2.2, both cutoff and dispersion problems can be expressed as generalised eigenvalue problems of the form

$$\left[A\right] \left\{x\right\} = \lambda \left[B\right] \left\{x\right\}. \tag{5.30}$$

The eigenvectors $\left\{x\right\}$ are the coefficients of the basis functions used to discretise the electric field within the waveguide, and the eigenvalues λ are related to the associated cutoff wavenumbers or propagation constants, depending on the problem being considered. Furthermore, the solution of this eigensystem can contribute a considerable amount to the total computational time required to solve the problem. As mentioned in the previous sections, the matrices $\left[A\right]$ and $\left[B\right]$ are real valued in the case of lossless media, with the theory behind the solution of such systems discussed in much detail in [63] and [124].

For dense matrices, the LAPACK `xGGEV` routines can be used to solve (5.30). Since these routines calculate all the eigenpairs of the system, they are not very well suited to the solution of waveguide cutoff and propagation problems, where only a few of the eigenvalues are typically of interest. For this reason, iterative methods such as the Lanczos or Arnoldi methods (both Krylov-subspace methods) [63, 124] that converge to subset of the eigenpairs by the repeated application of matrix-vector products are often used.

A number of software packages exist that provide iterative eigensolver functionality, with a summary provided in [125] and [126]. The ARPACK software package, which is mentioned in both studies and based on the Implicitly Restarted Arnoldi Method [127], is used here. Although this is now considered legacy software [125], the relatively simple reverse-communication interface and its use in applications such as MATLAB and Octave, make it a popular choice with a large user-base.

It should be noted that although ARPACK provides routines to solve the generalised eigenvalue problem of (5.30), this is done by transforming the problem to a standard eigenvalue problem [14]

$$\left[C\right]\left\{x\right\} = \nu\left\{x\right\}, \tag{5.31}$$

with the new eigenvalue, ν, dependant on the transformation used and the associated eigenvectors unchanged. If the matrix $\left[B\right]$ in (5.30) is non-singular, for example, then $\left[C\right] = \left[B\right]^{-1}\left[A\right]$ and $\nu = \frac{1}{\lambda}$.

The more general method is to use a shift-invert process [14], where a real-valued scalar shift (σ) is used to convert the generalised eigenvalue problem of (5.30) to a standard one as

$$\begin{aligned}\left[A - \sigma B\right]^{-1}\left[B\right]\left\{x\right\} &= \nu\left\{x\right\},\\ \left[C\right]\left\{x\right\} &= \nu\left\{x\right\}.\end{aligned} \tag{5.32}$$

In this case, the eigenvalue of the new eigensystem is related to the original by [14]

$$\lambda = \frac{1}{\nu} + \sigma. \tag{5.33}$$

Rearranging (5.33) yields

$$\nu = \frac{1}{\lambda - \sigma}, \tag{5.34}$$

and as such, eigenvalues of the original system (λ) close to the shift (σ) will have the largest magnitude allowing iterative methods to converge to them rapidly [115]. The choice of σ is usually not clear cut, although in some cases the problem being considered gives some insight into a valid choice. Take the dispersion curve analysis as an example, where there exists a bound for the propagation constant that is calculated as the eigenvalue [115]. It can be shown that choosing a shift of $\sigma = -\theta^2$ (see (5.29) and Section 5.2.2) is equivalent to the scaling procedure discussed in [115].

Although the shifted eigensystem of (5.32) appears to require the solution of a matrix system, this is not necessarily the case. When the Arnoldi process implemented by ARPACK requests a matrix-vector multiplication by $\left[C\right]$ (using the reverse-communication interface), this can be performed in parts. Thus the matrix-vector product

$$\left\{y\right\} = \left[C\right]\left\{x\right\}, \tag{5.35}$$

with $\left\{x\right\}$ supplied by ARPACK, can be replaced by

$$\left\{w\right\} = \left[C\right]\left\{x\right\}, \tag{5.36}$$

$$\left\{y\right\} = \left[A - \sigma B\right]^{-1}\left\{w\right\} \tag{5.37}$$

with the final solution step begin performed by an iterative process such as GMRES [63], or making use of a pre-computed direct factorisation of $\left[A - \sigma B\right]$ as is the case in this chapter.

As mentioned, ARPACK does support the shift-invert process, with either multiplication by the matrix $\left[B\right]$ or $\left[C\right]$ being requested at each step of the iterative process. However, for matrices that are not positive semi-definite, as is the case here [115], it is required to construct the matrix $\left[C\right]$ explicitly and use the ARPACK functionality for solving the standard eigenvalue problem [14]. In this case, only multiplications by $\left[C\right]$ are required and the resultant eigenvalues must be adjusted according to (5.33) as a post-processing step.

5.3 Implementation

With the basics of the formulation aspects of non-driven waveguide analysis using the Finite Element Method discussed in Section 5.2, this section presents some details pertaining to the implementation of the formulations discussed. This is done in two parts. The first of these is an introduction to FEniCS [16], which is used to realise the assembly of the matrices mentioned in Section 5.2. The second part of the implementation covers the use of ARPACK in conjunction with BLAS, LAPACK, CUDA, and MAGMA in the implementation of a dense eigenvalue solver framework that not only offers performance improvements over the LAPACK `xGGEV` implementation, but allows for GPU acceleration.

5.3.1 FEniCS

The FEniCS Project is a set of software tools that allows for the rapid implementation of the expressions associated with the finite element analysis of problems from a wide array of disciplines. The main interface of the software system is DOLFIN, which provides both a C++ and Python front-end. The use of the Python front-end in modelling waveguide problems is considered here. DOLFIN provides built-in support for a number of finite element families, including the curl-conforming Nédélec (of the first and second kind [123, 128]) and Lagrange finite elements spaces, used in vector and scalar formulations, respectively [110].

As an introduction to the modelling of such problems using FEniCS, consider Listing 5.1, which shows a snippet from the FEniCS code for the waveguide cutoff problem. The listing shows the definition of both the vector (Nédélec) and nodal function spaces, and their use in the formation of a combined finite element space for the definition of the basis functions as discussed in Section 5.2. The variables `N_i` and `L_i` (`N_j` and `L_j`), are implementations of the basis functions $\vec{N}_i$ and L_i ($\vec{N}_j$ and L_j) used to define the matrix elements in (5.14), (5.16), (5.15), and (5.17). Here, the variables `vector_order` and `nodal_order` are used to specify the order of the Nédélec and Lagrange functions spaces, respectively, and `mesh` represents the DOLFIN mesh, which will be discussed when the example problems are considered in more detail.

The sample in Listing 5.1 clearly illustrates the high-level code that can be used to define the elements of the matrices `S` and `T`. This allows for an almost one-to-one mapping from the mathematical expressions given in the previous section to the Python code presented here, and facilitates the rapid development and testing of new formulations. The variables `e_r` and `u_r` represent the relative permittivity and permeability of the medium within the waveguide and may be simple floating-point values or more complex expressions. These will be discussed further in the subsequent sections.

The code for implementing the dispersion curve problems is similar to that of Listing 5.1, with the same function spaces and basis functions being used (as is also evident from Section 5.2). The construction of the matrices using the expressions from (5.22), (5.23), (5.24), (5.26), and (5.25) in DOLFIN are given in Listing 5.2.

Listing 5.1: The FEniCS DOLFIN Python code to assemble the matrices of (5.12). Shown are the initialisation of the function spaces and the vector and scalar basis functions themselves using DOLFIN classes. The variables `vector_order` and `nodal_order` specify the polinomial order of the Nédélec and Lagrange function spaces used respectively. Variables with names ending in `_ij` in this case represent forms, a FEniCS concept that can be rougly translated as an expression defining the elements of a finite element matrix. The variables `e_r` and `u_r` represent the relative permittivity and permeability for the calculation of the matrix element and `mesh` the mesh used in the spatial discretisation are all example specific.

```
# define the functions spaces
vector_space = FunctionSpace ( mesh, "Nedelec 1st kind H(curl)", vector_order )
nodal_space = FunctionSpace ( mesh, "Lagrange", nodal_order )
combined_space = vector_space * nodal_space
# define the test and trial functions from the combined space
# here N_v and N_u are Nedelec basis functions and L_v and L_u are Lagrange
    basis functions
(N_i, L_i) = TestFunctions ( combined_space )
(N_j, L_j) = TrialFunctions ( combined_space )

# define the forms (matrix elements) for cutoff analysis ito the basis functions
s_tt_ij = 1.0/u_r * dot ( curl_t(N_i), curl_t(N_j) )
t_tt_ij = e_r * dot ( N_i, N_j )
s_zz_ij = 1.0/u_r * dot ( grad(L_i), grad(L_j) )
t_zz_ij = e_r * L_i * L_j

# post-multiplication by dx will result in integration over the domain of the
    mesh at assembly time
s_ij = ( s_tt_ij + s_zz_ij ) * dx
t_ij = ( t_tt_ij + t_zz_ij ) * dx

# assemble the system matrices. DOLFIN automatically evaluates each of the forms
     for all the relevant test and trial function combinations. ie. all possible
     values of i and j
S = assemble ( s_ij )
T = assemble ( t_ij )
```

Besides the components already discussed with reference to Listing 5.1, Listing 5.2 contains a variable `k_o_squared`, which is an instance of the defined `OperatingWavenumberSquared` class – an extension of the DOLFIN `Expression` class as shown. This expression is used to specify k_o^2, as in (5.19), (5.22), and (5.23), when calculating the propagation constant for a given operating frequency f_o. The motivation for using this `Expression`, and not simply a floating-point value, is that DOLFIN caches the forms for matrix assembly. Using a floating-point variable, however, would require the forms to be recalculated for each frequency step when calculating dispersion curves over a frequency range.

Listing 5.2: The FEniCS DOLFIN Python code required to extend the cutoff formulation of Listing 5.1 to the problem of dispersion analysis for a specified operating frequency `f_o`. Note that the test and trial (basis) functions from Listing 5.1 are used here, as are the forms `s_tt_ij`, `t_tt_ij`, `s_zz_ij`, and `t_zz_ij`.

```
class OperatingWavenumberSquared ( Expression ):
  """ an extension of a DOLFIN Expression to allow for the inclusion of the
      cutoff wavenumber squared in the forms for the matrices [A] and [B] """
  def eval ( self, values, x ):
    """ evaluate return the operating wavenumber squared """
    values[0] = self.__k_o_squared
  def set_frequency ( self, f ):
    """ set the operating wavenumber squared """
    self.__k_o_squared = 2*pi*f/c0

# create a variable to store the operating wavenumber
k_o_squared = OperatingWavenumberSquared ()
k_o_squared.set_frequency ( f_o )
# define the matrix entry components (using the basis functions from Listing 5.1
b_tt_ij = 1.0/u_r * dot ( N_i, N_j )
b_tz_ij = 1.0/u_r * dot ( N_i, grad ( L_j ) )
b_zt_ij = 1.0/u_r * dot ( grad ( L_i ), N_j )
# define the matrix entries (using s_tt_ij, t_tt_ij, s_zz_ij, and t_zz_ij from
    Listing 5.1 with *dx resulting in integration over the domain of the mesh when
     assemble() is called
a_ij = ( s_tt_ij - k_o_squared * t_tt_ij ) * dx
b_ij = ( s_zz_ij - k_o_squared * t_zz_ij + b_tt_ij + b_tz_ij + b_zt_ij ) * dx
# assemble the matrices
A = assemble ( a_ij )
B = assemble ( b_ij )
```

Hollow rectangular waveguide

The first of the sample problems considered is a hollow waveguide with a rectangular cross-section, as depicted in Figure 5.2. The width $a = 1.0$m and height $b = 0.5$m of the guide as indicated. Since the guide is hollow, the relative permittivity and permeability of Listing 5.1

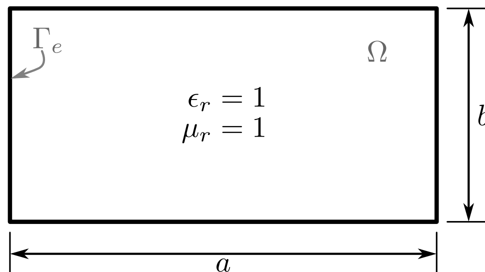


Figure 5.2: A diagram showing the cross section and dimensions of a hollow rectangular waveguide. The PEC boundary of the cross-sectional domain Ω is indicated by Γ_e, with the values for both the relative permittivity (ϵ_r) and permeability (μ_r) unity over the whole domain. For the problem considered here $a = 1.0$m and $b = 0.5$m.

and Listing 5.2 are both unity.

This structure has the advantage that analytical solution for both the cutoff and propagation problems are known [13], with the propagation constant (γ^2) and cutoff wavenumber (k_c^2) of Section 5.2 given by

$$\gamma^2 = k_c^2 - k_o^2, \tag{5.38}$$

and

$$k_c^2 = \left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2, \tag{5.39}$$

respectively. Here, k_o is the operating wavenumber as discussed with reference to Listing 5.2, and m and n represent the indices of the TE_{mn} and TM_{mn} waveguide modes. Valid values for m and n are any non-negative integers, with at least one of them non-zero in the case of the TE modes, and neither m nor n zero for the TM modes.

Although Listing 5.1 and Listing 5.2 discuss the assembly of relevant system matrices, no mention has been made the boundary conditions associated with the PEC wall surrounding the guide cross-section. This, along with the remaining implementation details, are addressed in Listing 5.3. The first of the details shown is the creation of the `mesh` variable, used in the initialisation of the function spaces in Listing 5.1, as an instance of a DOLFIN `Rectangle`. A plot of the mesh generated using the DOLFIN `Rectangle` class is shown in Figure 5.3. The relative permittivity and permeability (`e_r` and `u_r`) are also initialised to unity.

Listing 5.3 also shows the application of the Dirichlet boundary condition – as in (5.2) – to the assembled matrices. This boundary condition, `electric_wall`, is created as instance of a DOLFIN `DirichletBC` that is initialised using the combined functions space from Listing 5.1, a zero-valued `Expression`, and an instance of a DOLFIN `SubDomain` that is equivalent to the boundary of the mesh used. The `apply()` method called for each of the assembled matrices uses this information to apply the boundary condition to the assembled matrices for the degrees of freedom that correspond with the sub-domain defined by `PECWall`. Once the matrices have been assembled and the boundary conditions applied, the relevant eigenvalue problem (see Section 5.2) can be solved.

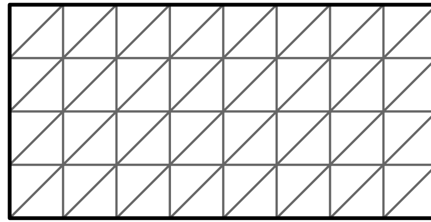


Figure 5.3: The mesh used for the solution of both the hollow and half-filled rectangular waveguide problems. This mesh is generated using the DOLFIN code `mesh = Rectangle ( 0, 0, 1.0, 0.5, 8, 4 )` as in Listing 5.3. The thick solid outer line represents the electric wall (the `PECWall` class) on which the Dirichlet boundary condition of (5.2) is applied.

Listing 5.3: The extension of the FEniCS DOLFIN cutoff and dispersion implementations in Listing 5.1 and Listing 5.2 to the case of the 1.0m × 0.5m hollow rectangular waveguide shown in Figure 5.2. The problem specific implementation details are the initialisation of the mesh as a DOLFIN `Rectangle`, the setting of the permittivity and permeability (both to unity), and the implementation of the Dirichlet boundary condition associated with an electric wall using the DOLFIN `DirichletBC` class and a sub-domain `PECWall` corresponding with the walls of the waveguide cross-section.

```
# the mesh used for the rectangular hollow guides
a = 1.0
b = 0.5
# create a rectangular mesh with origin (0,0) extending to (a,b) with 8 edges
    along the long side and 4 elements along the short side
mesh = Rectangle ( 0, 0, a, b, 8, 4 )
# specify the permittivity and permeability
e_r = 1.0
u_r = 1.0

# assemble the matrices
as in Listing 5.1 and Listing 5.2

# define the subdomain on which the boundary condition must be applied
class PECWall ( SubDomain ):
  def inside(self, x, on_boundary):
    return on_boundary;

# create the boundary condition using the combined function space, a zero
    Expression and the PECWall sub-domain
electric_wall = DirichletBC ( combined_space, Expression ( ("0.0", "0.0", "0.0")
     ) , PECWall() )
# apply the boundary condition to the assembled matrices:
# for the cutoff problem: Listing 5.1
electric_wall.apply ( S )
electric_wall.apply ( T )
# for the propagation problem: Listing 5.2
electric_wall.apply ( A )
electric_wall.apply ( B )

# solve the eigensystems (A, B)  or (S, T)
...
```

Half-filled rectangular waveguide

The second example problem considered is that of a rectangular waveguide which is partially filled with a dielectric material ($\epsilon_r > 1$). This lowers the cutoff frequency of the dominant mode, when compared to a hollow guide of the same dimensions, and is often employed in impedance matching or phase-shift waveguide components [13]. An example of such a waveguide, with its lower half filled with an $\epsilon_r = 4$ dielectric material, is shown in Figure 5.4. The dimensions of the guide are the same as for the hollow rectangular case, that is $a = 1.0$m and $b = 0.5$m. This

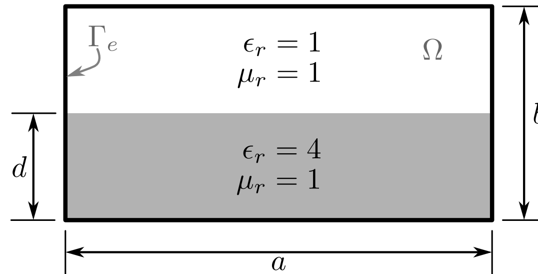


Figure 5.4: A diagram showing the cross section and dimensions of a half-loaded rectangular waveguide. The lower half of the guide is filled with an $\epsilon_r = 4$ dielectric material. For the problem considered here $a = 1.0$m, $b = 0.5$m, and $d = 0.25$m.

example is also used in [111], and allows for a comparison of results.

Due to the similarities in structure between the half-filled and hollow guides, much of the code of Listing 5.3 can be reused. Listing 5.4 shows the additional code required for the definition of the dielectric structure. This is done through the extension of the DOLFIN `Expression` class, overriding the `eval()` method. Note that the parameter `x` is passed to this method by DOLFIN as part of the matrix assembly process, and is an array of the coordinates at which the expression must be evaluated (`x[0]`$= x$ and `x[1]`$= y$). The `values` parameters is an array in which the desired value of the expression is to be stored.

The class `HalfFilledDielectric` provides an `eval()` method that evaluates as

$$\epsilon_r(x, y) = \begin{cases} 4 & \text{if } y < 0.25 \ , \\ 1 & \text{if } y \geq 0.25 \ , \end{cases} \tag{5.40}$$

and properly defines the dielectric shown in Figure 5.4. The `e_r` variable is then defined as an instance of this class and is used in the subsequent matrix element definitions (see Listing 5.1 and Listing 5.2). Note that the mesh and PEC boundary conditions (`mesh` and `electric_wall`) from Listing 5.3, and depicted in Figure 5.3, are reused as indicated.

Listing 5.4: The extension of the FEniCS DOLFIN cutoff and dispersion implementations in Listing 5.1 and Listing 5.2 to the case of the $1.0\text{m} \times 0.5\text{m}$ half-filled rectangular waveguide shown in Figure 5.4. The same mesh and boundary conditions as the hollow rectangular guide (implemented in Listing 5.3) are used. In this case the relative permittivity is defined as an extension of the DOLFIN Expression class with the desired behaviour implemented by overriding the `eval()` method.

```
# use the same mesh as the hollow rectangular guides
as in Listing 5.3

# Extend the expression class to define the permittivity
class HalfFilledDielectric ( Expression ):
  def eval ( self, values , x ):
    if ( x[1] < 0.25 ):
      values[0] = 4.0
    else:
      values[0] = 1.0

# create an instance of the permittivity class and set permeability to unity
e_r = HalfFilledDielectric ()
u_r = 1.0
# these are then used in the forms of Listing 5.1 and Listing 5.2

# assemble the matrices
as in Listing 5.1 and Listing 5.2

# define the boundary conditions and apply
as in Listing 5.3

# solve the eigensystems (A, B)  or (S, T)
...
```

Shielded microstrip line

The final sample problem is that of a shielded microstrip transmission line, and, although it is not always shielded, represents a widely used structure in microwave design [13]. The primary component of such a structure is a thin conducting strip of width w on a dielectric substrate with thickness d. Both are placed above a ground plane as shown in Figure 5.5, where the strip thickness t has been exaggerated. Also shown in the figure, is the shielding of the structure by a PEC box of dimensions $a \times b$, and a vertical symmetry line in the centre of the figure. The latter is modelled as a magnetic wall, and reduces the size of the computational domain, but only allows for even modes to be computed [113].

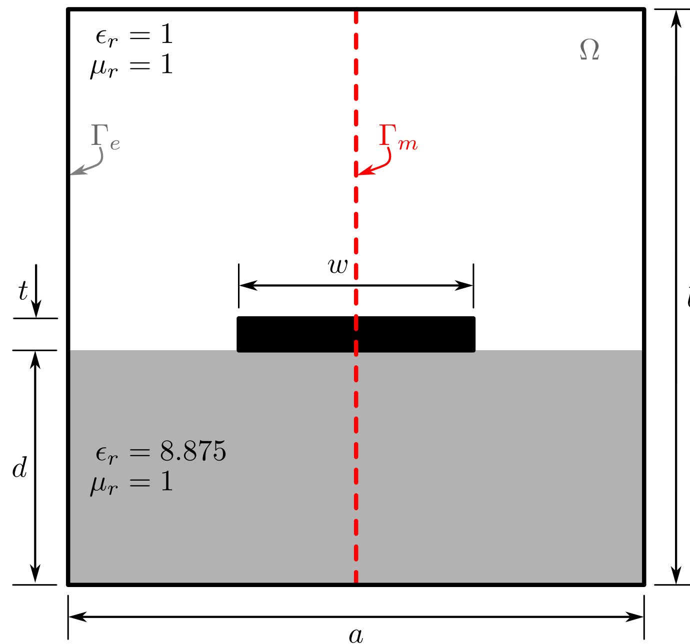


Figure 5.5: A diagram showing the cross section and dimensions of a shielded microstrip line with the thickness of the line exaggerated for effect. The microstrip is etched on a dielectric material with a relative permittivity of $\epsilon_r = 8.875$. The plane of symmetry is indicated by a dashed line (Γ_m) and is modelled as a magnetic wall in order to reduce the size of the computational domain. The dimensions used here are; $a = 12.7$mm, $b = 12.7$mm, $d = 1.27$mm, $w = 1.27$mm, and $t = 0.127$mm.

The FEniCS code for the implementation of the shielded microstrip example is given in Listing 5.5, with the definition of the boundary subdomain (similar to `PECWall` in Listing 5.3) given in Listing 5.6. For this example, the mesh is loaded from a file, `shielded_microstrip_mesh.xml`, which was generated in Gmsh [129] and converted to a DOLFIN mesh using a supplied DOLFIN utility. The mesh itself is shown in Figure 5.6, where the electric and magnetic walls (in heavy solid black and dashed red lines, respectively) are also indicated. Only half the structure shown in Figure 5.5 is meshed, with the triangulation of the dielectric ($\epsilon_r = 8.75$) layer and that of the free space ($\epsilon_r = 1$) shown in different colours.

Apart from the differences in the mesh, the dielectric definition, and the boundary subdomain, which returns true if the requested point `x` falls on one of the thick black lines of Figure 5.6 (as shown in Listing 5.6), the implementation for the shielded microstrip is identical to that of the other problems considered. This allows for the reuse of much of the code when performing the computations.

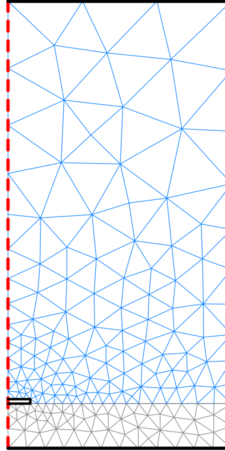


Figure 5.6: A mesh generated by Gmsh [129] and used for the computation of the dispersion curves for a shielded microstrip line. Only the right half the computational domain is meshed with the magnetic wall through the centre of the original domain indicated by a dashed red line and the PEC (electric wall) boundaries are shown in heavy black with the finite thickness strip not meshed.

Listing 5.5: The extension of the FEniCS DOLFIN dispersion implementation in Listing 5.2 to the case of shielded microstrip, as shown in Figure 5.5. In this case, the mesh is loaded from a file, `shielded_microstrip_mesh.xml`, and the Dirichlet boundary sub-domain is given in Listing 5.6. The relative permittivity is defined in a similar manner to that of the half-filled guide of Listing 5.4.

```
# load the mesh as shown in Figure 5.6
mesh = Mesh ( "shielded_microstrip_mesh.xml" )
# Extend the expression class to define the permittivity
class Dielectric ( Expression ):
  def eval ( self, values , x ):
    # x is the position in milimetres
    if x[1] <= 1.27:
      values[0] = 8.875
    else:
      values[0] = 1.0;

# create an instance of the permittivity class and set permeability to unity
e_r = Dielectric ()
u_r = 1.0
# these are then used in the forms of Listing 5.1 and Listing 5.2

# assemble the matrices
as in Listing 5.1 and Listing 5.2

# define the boundary sub-domain
see Listing 5.6

# create the boundary condition using the combined function space, a zero
   Expression and the ShieldedMicrostripPECWall sub-domain
electric_wall = DirichletBC ( combined_space , Expression ( ("0.0", "0.0", "0.0")
     ) , ShieldedMicrostripPECWall() )
# apply the boundary condition to the assembled matrices:
# for the propagation problem as in Listing 5.3

# solve the eigensystems (A, B)
...
```

Listing 5.6: PEC boundary sub-domain for the shielded microstrip example. `DOLFIN_EPS` is a DOLFIN constant that indicates machine precision.

```
class ShieldedMicrostripPECWall(SubDomain):
  def inside(self, x, on_boundary):
    # x is the position in milimetres
    if on_boundary:
      if x[0] >= DOLFIN_EPS:
        # x is on the PEC boundary but not on the left edge
        return True
      if x[1] <= DOLFIN_EPS:
        # point is at lower-left corner
        return True
      if ( x[1] >= (1.27 - DOLFIN_EPS) and x[1] <= ( 1.397 + DOLFIN_EPS ) ):
        # point is at the left corners of the microstrip
        return True
      if x[1] >= ( 12.7 - DOLFIN_EPS ):
        # point is at upper-left corner
        return True

    return False
```

5.3.2 Eigensolver implementations

As discussed in Section 5.2, the problems considered here take the form of generalised eigenvalue problems. Furthermore, the solution of these systems involves transforming the generalised eigenvalue problem to a standard one using the shift-invert process summarised in Section 5.2.3. This standard eigenvalue problem is then solved using ARPACK [14].

In this section, some of the implementation details of this ARPACK-based standard eigenvalue problem solver are presented. This solver is then incorporated into a framework that allows for the solution of the generalised eigenvalue problem using CPU-based routines such as those supplied by ACML, as well as a CUDA-accelerated implementation that makes use of both CUBLAS and MAGMA. These linear algebra libraries are discussed in more detail in Chapter 3, and limit the current implementation to use with dense matrices.

An overview of the implementations of the eigenvalue solvers is presented as a block diagram in Figure 5.7. Both the CPU and CUDA implementations result in shared libraries, `libgev_solver_cpu.so` and `libgev_solver_cuda.so`, allowing them to be included in the FEniCS solution process in Python using tools such as Ctypes [105]. Furthermore, since both the CPU and CUDA implementations share a common interface (`gev_solver_interface.h` in Figure 5.7), these libraries can be used interchangeably.

The code defining the common interface for the implementations is given in Listing 5.7. The interface declares three routines, `arpack_ssev_()`, `sgemv_wrapper_()`, and `dense_gev()`. The latter is the entry point into the generalised eigenvalue problem solver and takes as input the size of the (square) matrices `N`, the two single precision real matrices `A` and `B` that define the eigensystem as well as their leading dimension `LDMAT`. The parameters `NEV` and `shift` pass the number of requested eigenvalues and the shift (σ in (5.32)) to be used.

The calculated single precision complex eigenvalues, and their corresponding real eigenvectors, are returned in the arrays pointed to by the parameters `eigenvalues` and `eigenvectors`. The final two parameters `timing_data` and `int_data` are used to obtain information on the computational process, such as the timing of individual steps, and are thus only used for debugging. Of the other routines mentioned, `arpack_ssev_()`, is simply the declaration of the entry point into the ARPACK driver.

The final routine shown, `sgemv_wrapper_()`, is a wrapper for the implementation-specific matrix-

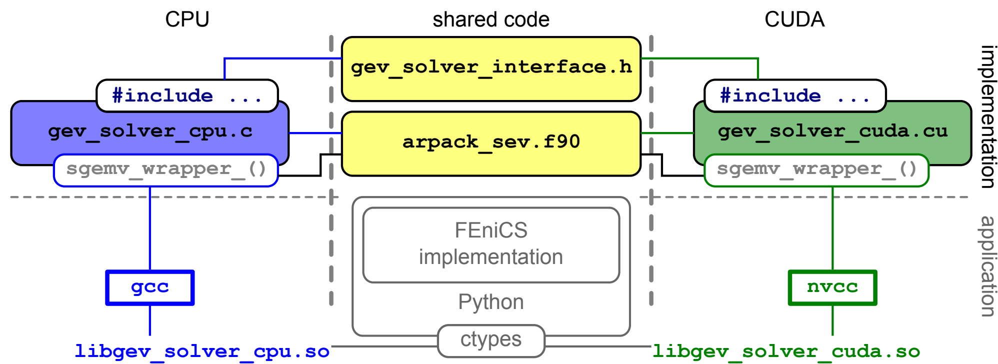


Figure 5.7: A block diagram showing the relationship of the CPU and CUDA implementations of a generalised eigenvalue problem (GEV) solver. The two implementations share a common interface and an ARPACK driver for the solution of the standard eigenvalue problem (implemented in `arpack_sev.f90`). This shared driver makes use of matrix-vector wrapper routines (`sgemv_wrapper_()`) supplied by each of the implementations. The CPU and CUDA versions are compiled with either `gcc` or `nvcc` (the NVIDIA C Compiler [29]) into shared libraries – `libgev_solver_cpu.so` and `libgev_solver_cuda.so` respectively – and included as part of the FEniCS solution process in Python using Ctypes.

Listing 5.7: A header file `gev_solver_interface.h` defining the common interface for the CPU- and CUDA-based generalised eigenvalue solver implementations of Figure 5.7.

```
#ifndef GEV_INTERFACE_H
#define GEV_INTERFACE_H

// define a macro for external functions depending on the compiler
#ifdef __cplusplus
  #define EXT extern "C"
#else
  #define EXT extern
#endif

// declaration of the ARPACK entry routine
EXT
void arpack_ssev_ ( int* N, void* DATA, int* NEV, int* NCV, float* eigenvalues,
    float* eigenvectors, float* residuals, char* which );
// declaration of the SGEMV wrapper that must be visible to the ARPACK driver
EXT
void sgemv_wrapper_ ( int* N, void** DATA, float* x, float* y );
// declaration for the routine to solve the generalised eigenvalue problem
EXT
int dense_gev ( int N, float* A, float* B, int LDMAT, int NEV, float shift,
    float* eigenvalues, float* eigenvectors, double* timing_data, int* int_data
    );

#endif // #ifndef GEV_INTERFACE_H
```

vector routines that are required by the ARPACK implementation. Recall from Section 5.2.3 that the ARPACK process requires the repeated calculation of a matrix-vector product as in (5.35). The parameter `N` of the `sgemv_wrapper_()` routine is a pointer to the size of the matrix $[C]$, with the pointers `x` and `y` representing $\{x\}$ and $\{y\}$ of (5.35), respectively. The final parameter, `DATA`, is a pointer to (a pointer to) an implementation-specific data structure that contains the information required to perform the matrix-vector product on the intended architecture, and includes the matrix $[C]$. A pointer to this data structure is also passed to the ARPACK driver

routine `arpack_ssev_()`. The exact nature of these data structures are discussed with reference to the specific implementations in two of the following sections.

Although the implementations differ slightly in each case, the computational process of the CPU and CUDA versions of the generalised eigensolvers discussed here are the same, with the steps given in Listing 5.8. Here, step 1 and step 3, are implemented in the target-specific files, `gec_solver_cpu.c` and `gec_solver_cuda.cu`, with the ARPACK driver implemented in `arpack_sev.f90`. The latter is adapted from the `sndrv1.f` example file supplied as part of the ARPACK source code. Modifications include rewriting the implementation as a `SUBROUTINE` with a parameter list as shown in Listing 5.7. This driver routine then calls the implementation-specific `sgemv_wrapper_()` routines (step 2.a), which are also implemented in `gec_solver_cpu.c` and `gec_solver_cuda.cu` for the CPU and CUDA versions, respectively.

Listing 5.8: The computational process for solving the generalised eigenvalue problem using the standard eigenvalue problem ARPACK driver.

```
1. transform the generalised eigensystem to the standard one as in (5.32)
2. call the ARPACK driver routine
  a. repeatedly call the matrix−vector product provided by sgemv_wrapper_()
3. transform the calculated eigenvalues to those of the original system as in
   (5.34)
```

For the transformation to the standard eigenvalue problem (step 1 in Listing 5.8), a shift funtion is implemented and LAPACK routines are used for the inversion step. Since the shift operation represents a large number of independent operations, OpenMP is used to parallelise the loops over the matrix elements. This is done in a similar way to that already discussed in Chapter 4. For the invert procedure, accelerated LAPACK implementations are used. That is to say, ACML is used for the CPU implementation, while MAGMA is used for the CUDA version. Specifically, the `SGETRF` and `SGETRS` routines are used.

The matrix-vector product wrapper

As discussed, the ARPACK driver makes use of a wrapper routine to perform the required matrix-vector products. Furthermore, from Listing 5.7, it follows that these routine adhere to the same interface. This serves to hide the implementation-specific details from the ARPACK driver, and allows for the CPU and CUDA versions to be used interchangeably.

The CPU implementation of the wrapper routine, `sgemv_wrapper_()`, is shown in Listing 5.9. Also shown in the listing is the definition of the data structure used to facilitate the passing of relevant pointers and other matrix information to the ARPACK driver routine and the matrix-vector wrapper. Note that for the CPU implementation of Listing 5.9, the standard BLAS `SGEMV` routine, as implemented by ACML, is used. In this case, the parameter `DATA` consists of a pointer to the matrix `C` in host memory, as well as the leading dimension of this matrix, `LDC`.

Since the CUDA version of the matrix-vector product wrapper has to take the movement of data between host and device memory into account, the implementation of the `sgemv_wrapper_()` routine is slightly more complex than that of Listing 5.9, with the source code for the routine shown in Listing 5.10. The parameter list of the two implementations is, however, identical. In the case of the CUDA implementation, a CUBLAS routine is used to perform the matrix-vector product. It should be noted, that although the vectors `x` and `y` are transferred to and from the device, respectively, there is no need to transfer the matrix to the device for each multiplication operation. As already mentioned, this is only done once, since the matrix remains constant throughout the eigensystem solution process.

Listing 5.9: The source code for the CPU implementation of the `sgemv_wrapper_()` routine as called from the ARPACK driver. Also shown is the casting of the `void**` pointer `DATA` to the `data_struct` type to allow access to the data needed by the routine.

```
struct data_struct {
  float* C;
  int LDC;
};
typedef struct data_struct data_struct;

void sgemv_wrapper_ ( int* N, void** DATA, float* x, float* y )
{
  data_struct* pDATA = (data_struct*) DATA;
  float* C = pDATA->C; int LDC = pDATA->LDC;
  float f_one = 1.0f; float f_zero = 0.0f; int i_one = 1;
  // calculate y <-- Cx using the ACML implementation of the SGEMV routine
  sgemv_ ( "N", N, N, &f_one, C, &LDC, x, &i_one, &f_zero, y, &i_one, 1 );
}
```

Listing 5.10: The source code for the CUDA implementation of the `sgemv_wrapper_()` routine as called from the ARPACK driver. Also shown is the casting of the `void**` pointer `DATA` to the `data_struct` type to allow access to the data needed by the routine. This data structure includes information on pre-allocated device pointers for the storage of the vectors `x` and `y` as well as the matrix information passed in the CPU implementation (although the matrix information here is related to a matrix in device memory and not host memory).

```
struct data_struct {
  float* pdev_C; float* pdev_x; float* pdev_y;
  int LDC;
};
typedef struct data_struct data_struct;

void sgemv_wrapper_ ( int* N, void** DATA, float* x, float* y )
{
  data_struct* pDATA = (data_struct*)DATA;
  float* pdev_C = pDATA->pdev_C; int LDC = pDATA->LDC;
  float* pdev_x = pDATA->pdev_x; float* pdev_y = pDATA->pdev_y;
  // transfer x to the device
  cudaMemcpy ( (void*)pdev_x, (void*)x, (*N)*sizeof(float),
      cudaMemcpyHostToDevice );
  // calculate {y} = [C]{x} using CUBLAS
  cublasSgemv ( 'N', *N, *N, 1.0, pdev_C, LDC, pdev_x, 1, 0.0, pdev_y, 1 );
  // transfer y from the device
  cudaMemcpy ( (void*)y, (void*)pdev_y, (*N)*sizeof(float),
      cudaMemcpyDeviceToHost );
}
```

Another difference between the CPU and CUDA implementations, is the definition of the `data_struct` structure. The CUDA implementation shows two additional pointers, `pdev_x` and `pdev_y`, which are pointers to arrays in device memory. These blocks of device memory are pre-allocated specifically to store the input and output vectors (`x` and `y`, respectively). This pre-allocation and storage of the pointers is an attempt to reduce the overhead involved in computing the matrix-vector product, as it is not necessary to allocate and free device memory for the vectors for each multiplication. The naming of the fields of the data structure also indicates that the matrix used in the multiplication is stored in device memory.

5.4 Verification results

As for the Method of Moment results presented in Section 4.4 and Section 4.5, the results presented in this chapter have two objectives. These are, the verification of the implementations by comparison to analytical or previously published results, and the analysis of the performance of the implementations discussed here. The former is considered in this section, and addressed the FEniCS implementation discussed in Section 5.3.1, as well as the CPU and CUDA implementations of the generalised eigensolver based on ARPACK of Section 5.3.2. The performance benefits of the CUDA-based implementation are considered in Section 5.5.

In the verification of the implementations presented here, the three structures introduced in Section 5.3.1 are considered. As mentioned, the same mesh – shown in Figure 5.3 – is used for both the hollow and half-filled rectangular guides, while the shielded microstrip problem uses the mesh as shown in Figure 5.6. The order of the vector and nodal functions spaces (`vector_order` and `nodal_order` in Listing 5.1) are taken as 2 and 3 respectively. This equates to 549 DOFs in the case of the hollow rectangular and half-filled waveguides, and 3082 DOFs for the shielded microstrip line.

In the case of the hollow and half-filled rectangular guides, both the cutoff and dispersion analysis is performed, with the visualisation of the modes (and cutoff wavenumbers) for the TE_{10} and TM_{11} modes considered in each case. The propagation curves for these modes and the modes that occur between them are also presented and compared to either analytical results, in the case of the hollow rectangular waveguide, or to previously published results for the half-filled guide.

For the shielded microstrip problem, only the dispersion analysis is considered and compared to results presented in [113]. In all cases, the results obtained using both the CPU and CUDA implementations of the generalised eigensolver discussed in Section 5.3.2 are compared to each other.

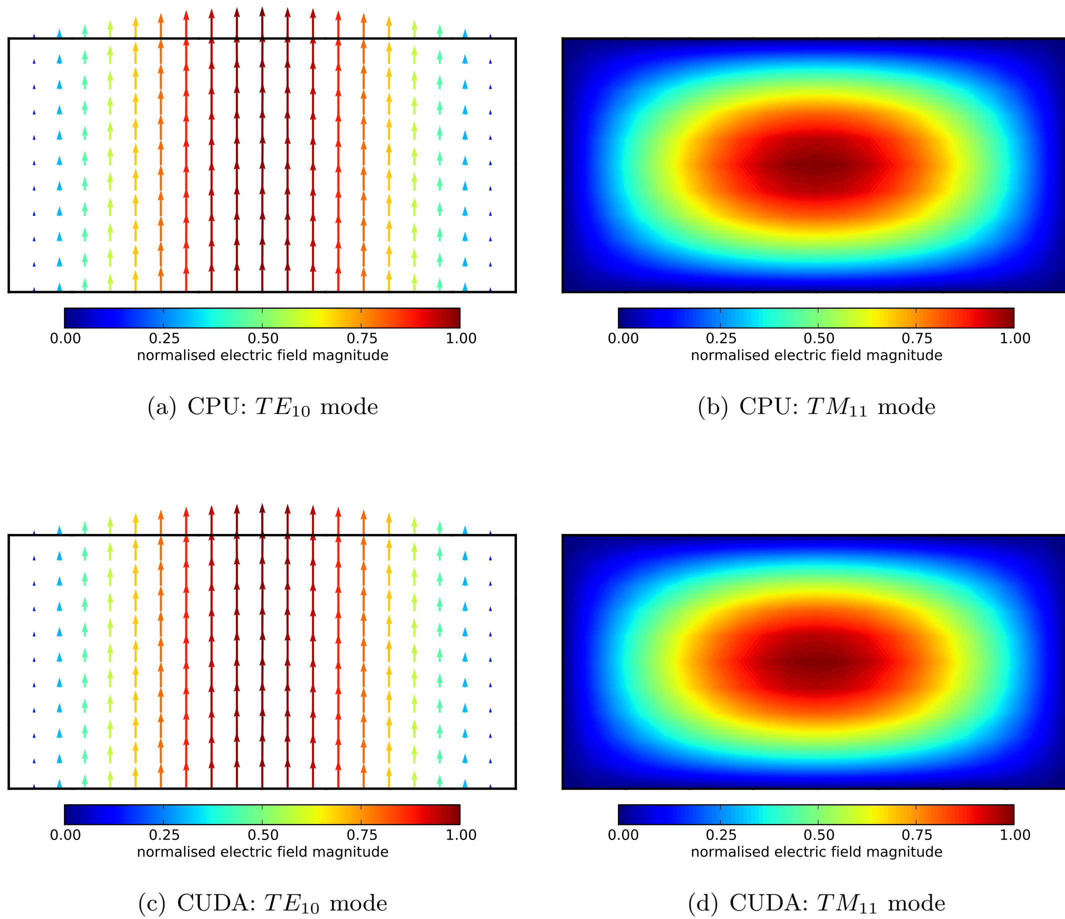
Hollow rectangular waveguide

Since the vector Helmholtz equation of (5.1) can be solved analytically for the hollow rectangular waveguide, it is often used as a starting point for the verification and validation of new finite element implementations. It is also covered in a number of electromagnetic texts, both from a computational and analytical point of view [13, 88, 111, 113].

Figure 5.8 shows two of the computed cutoff modes for the hollow rectangular waveguide with dimensions 1.0m $\times$ 0.5m. As mentioned, and should be evident from the images, the modes considered are the TE_{10} and TM_{11} modes that represent the lowest frequency transverse electric and transverse magnetic modes, respectively. Since the TE_{10} mode consists of both an x- and y-component, it is displayed as a quiver plot. Here the colour is mapped to the normalised – with respect to the calculated maximum – magnitude of the field vector $\vec{E}_t(x, y)$. For the TM_{11} mode at cutoff ($\gamma = 0$), only the z-component of the electric field ($E_z(x, y)$) in the guide is non-zero [13]. As such, E_z can be plotted as a contour surface, with the colour indicating the magnitude of the vector – again normalised with respect to the maximum.

The figures indicate that there is no visually discernible difference between the CPU and CUDA eigenvalue results. This is further supported by a comparison of the computed eigenvalues to each other and to analytical results presented in Table 5.1. Here, the relative error for the two modes is at least $\mathcal{O}(10^{-4})$ for the two modes shown.

The dispersion results for the first five modes of the hollow rectangular waveguide are shown in Figure 5.9, where the normalised propagation constant squared ($\frac{|\gamma|^2}{k_o^2}$) is plotted as a function of operating frequency. Note that, in the case of the propagating modes shown, the propagation constant is imaginary with $\gamma = j\beta$. From (5.38), it follows that at cutoff ($k_c = k_o$), the



(a) CPU: TE_{10} mode (b) CPU: TM_{11} mode

(c) CUDA: TE_{10} mode (d) CUDA: TM_{11} mode

Figure 5.8: Computed results for the normalised TE_{10} (x and y-components) and TM_{11} (z-component) cutoff modes plotted on the cross-section of the hollow rectangular guide of Figure 5.2 for the FEniCS implementation of Listing 5.3 using both CPU (top row) and CUDA (bottom row) eigensolvers.

propagation constant is zero, and thus the cutoff frequencies can be read off the plot as the intersection of the respective curves with the f_o-axis.

Note that since (5.38) and (5.39) hold for both TE and TM modes, it follows that a number of these modes will have the same cutoff frequency and propagation constant if their indices (mn) are the same. This is the case for the TE_{11} and TM_{11} curves shown. Furthermore, since

Table 5.1: Comparison of the computed and analytical cutoff wavenumbers squared for the TE_{10} and TM_{11} modes of a hollow rectangular guide considered here. Shown are the calculated cutoff wavenumbers squared as well as the calculated absolute relative errors between the calculated and expected value for both the CPU and CUDA eigensolvers.

mode name	$\boldsymbol{TE_{10}}$	$\boldsymbol{TM_{11}}$
analytical [m^{-2}]	$1.0\pi^2$	$5.0\pi^2$
CPU [m^{-2}]	9.86950397	49.35956955
relative error	1.0175×10^{-5}	2.3400×10^{-4}
CUDA [m^{-2}]	9.86937523	49.35956573
relative error	2.3220×10^{-5}	2.3392×10^{-4}

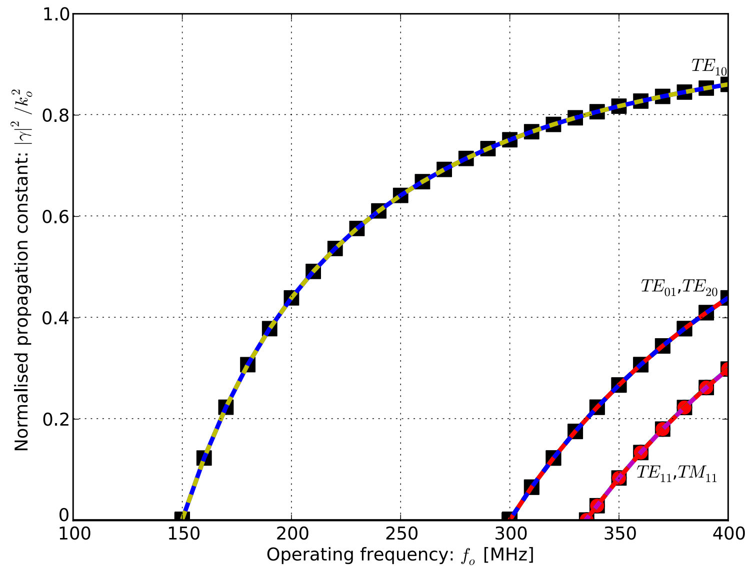


Figure 5.9: Comparison of calculated and analytical and computed (using both the CPU and CUDA eigensolvers) normalised propagation constants (corresponding to the imaginary part of γ in (5.5)) for first five modes of a 1.0m $\times$ 0.5m hollow rectangular waveguide as in Figure 5.2. The curves are labelled to show their relevant modes and the CPU and CUDA eigensolver results are indistinguishable.

$a = 2b$ for the guide considered here, the TE_{01} and TE_{20} modes are also degenerate [88].

Half-filled rectangular waveguide

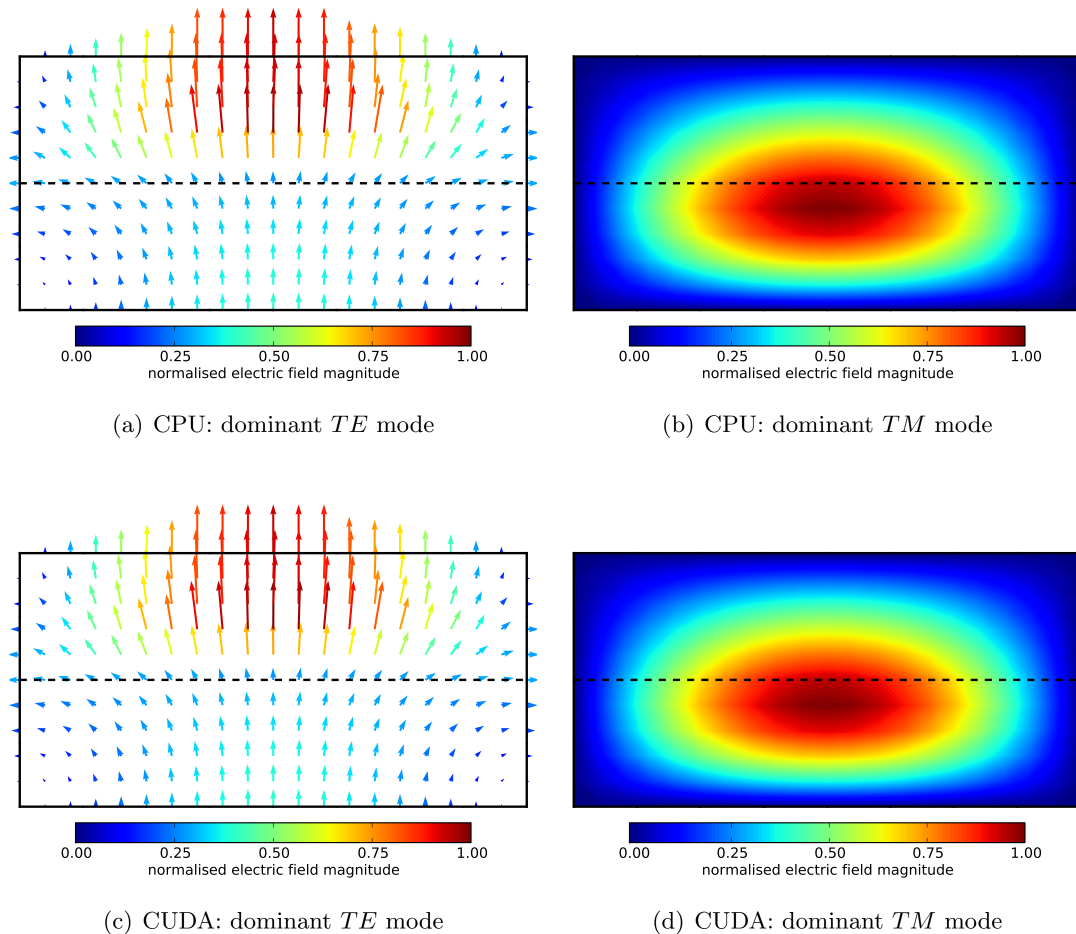
The results for the half-filled rectangular guide follow the same approach as those of the hollow guide just discussed. The visualisation of the first TE and TM modes – obtained using both the CPU and CUDA eigensolver – are shown in Figure 5.10, where there is once again good agreement between the result of the two implementations. The effect of the dielectric in the lower half of the guide is clearly evident, with the electric field of the dominant TE mode now concentrated into the upper half of the guide.

In this case, no analytical results are presented and as such, only the computed eigenvalues and the relative differences between the CPU and CUDA eigensolver results are given in Table 5.2. The measured differences are $\mathcal{O}(10^{-6})$, which is close to the attainable numeric accuracy of the single precision floating-point computation used [130].

Although analytical results are not presented in Table 5.2, the dispersion results of Figure 5.11 show good agreement with the results from [111] for the two modes shown in Figure 5.10. The other two modes computed are also close to the reference results, although in this case there are some frequency points where they do deviate slightly. This may be as a result of an error

Table 5.2: Comparison of the computed cutoff wavenumbers squared for the TE_{10} and TM_{11} modes of the half-filled rectangular guide considered here. Shown are the calculated cutoff wavenumbers squared and the absolute relative difference (taking the CPU values as the reference) between the CPU and CUDA results.

mode name	$\boldsymbol{TE_{10}}$	$\boldsymbol{TM_{11}}$
CPU $[m^{-2}]$	5.04175949	17.01956558
CUDA $[m^{-2}]$	5.04171276	17.01948929
relative difference	9.2687×10^{-6}	4.4827×10^{-6}



(a) CPU: dominant TE mode (b) CPU: dominant TM mode

(c) CUDA: dominant TE mode (d) CUDA: dominant TM mode

Figure 5.10: Computed results for the normalised dominant TE (x and y-components) and dominant TM (z-component) cutoff modes plotted on the cross-section of the half-filled rectangular guide of Figure 5.4 for the FEniCS implementation of Listing 5.4 using both CPU (top row) and CUDA (bottom row) eigensolvers. The edge of the dielectric is shown as a horizontal dashed line in each of the figures.

in reading the data off the figure in [111]. As is the intention of the half-filled design [13], the propagations curves of Figure 5.11 show that the cutoff frequency of the dominant mode has been decreased by about 45 MHz. As was the case for the hollow rectangular guide, both the CPU and CUDA propagation curves are shown in Figure 5.11. These are indistinguishable from each other.

Shielded microstrip line

Since the shielded microstrip line consists of two conductors, it supports a dominant transverse electromagnetic (TEM) mode that has no axial component of the electric or magnetic field [13]. Such a mode has a cutoff wavenumber of zero, and thus propagates for all frequencies [111, 113]. The cutoff analysis of this structure is not considered here, with only the dispersion problem considered. As is the case with the previous two examples, the cutoff wavenumbers for the higher order modes (which are hybrid TE-TM modes [13]) can be determined from the propagation curves from the intersection of a curve with the f_o-axis, corresponding to a propagation constant of zero.

The normalized propagation constant as a function of the operating frequency is shown in

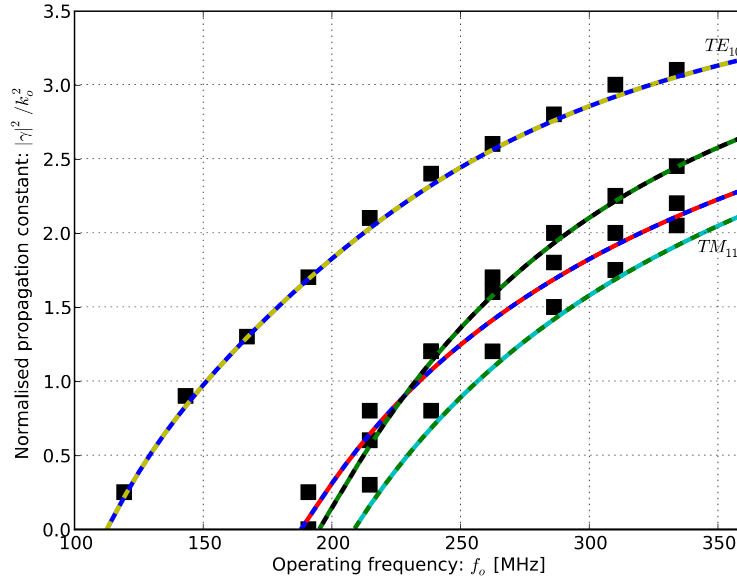


Figure 5.11: The normalised propagation curves as a function of frequency for the first four modes of the half-filled rectangular guide shown in Figure 5.4. The black squares show results for the same structure from [111] as reference results. The CPU and CUDA results are plotted on the same figure and are indistinguishable and both show good agreement with the reference results, especially for the dominant mode. The dominant TE and TM modes for which the cutoff results are given in Figure 5.10 and Table 5.2 are labelled as such.

Figure 5.12, with reference values from [113] also given. The computed results for both the CPU and CUDA eigensolvers (plotted over each other), used with the FEniCS implementation of Listing 5.5, agree well with the reference results.

It should be noted that the complex modes shown in [113], corresponding to conjugate-pair eigenvalues of the eigensystem in (5.32) are not visible in Figure 5.12. This is due to the fact that only the first four modes are considered here, whereas the complex modes are associated with modes six and seven in [113].

the timing results for the dispersion analysis of the shielded microstrip line are considered. That is the computational time required for the production of the results for the graph given in Figure 5.12.

Table 5.3 shows a summary of the timing and speedup results corresponding to the dispersion curve of Figure 5.12. This serves as an introduction to Section 5.5, where the performance results of the CPU- and CUDA-based implementations of the eigenvalue solver are presented. The FEniCS implementation of Listing 5.5 is used, with `vector_order = 2` and `nodal_order = 3`, resulting in 3082 degrees of freedom. The eigenvalue computation of Section 5.3.2 is performed in single precision, with four eigenvalues requested.

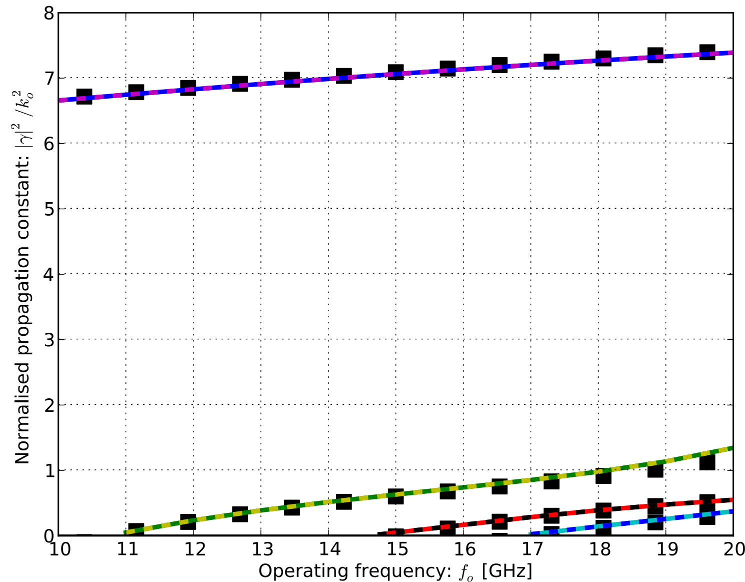


Figure 5.12: The normalised propagation curves as a function of frequency for the first four modes of the shielded microstrip line shown in Figure 5.5. The black squares show results for the same structure from [113] as reference results. The CPU and CUDA results are plotted on the same figure and are indistinguishable and both show good agreement with the reference results.

Table 5.3: Summary of timing and speedup results for the CPU and CUDA eigensolvers of Section 5.3.2 as applied to the dispersion analysis of the shielded microstrip. The results are shown for **System 1** and **System 2** (introduced in Section 3.4.1), with the time required for the FEniCS part of the implementation (Listing 5.5) as well as the eigensolver discussed in Section 5.3.2 given. The time measurements are in seconds and represent the combined time required for the evaluation of 11 frequency points between 10 and 20 GHz (including) in each case. The percentage contribution of each part of the solution to the total solution time is also shown.

		FEniCS [s]	**Eigensolver** [s]	**Total** [s]
System 1	**CPU**	22 (8%)	253 (92%)	275
	CUDA	22 (43%)	29 (57%)	51
	CUDA speedup		**8.7×**	**5.4×**
System 2	**CPU**	10 (9%)	98 (91%)	108
	CUDA	10 (30%)	23 (70%)	33
	CUDA speedup		**4.3×**	**3.3×**
System 1 CUDA *vs* **System 2 CPU**			**3.4×**	**2.1×**

5.5 Performance results

From the results in Table 5.3, it is clear that the solution of the eigenvalue problem is by far the most costly part of the dispersion analysis process. This is expected to be the case for the cutoff mode computations as well. The relative performance of the CPU and CUDA implementations of the eigensolver presented in Section 5.3.2 are now considered.

The performance results presented here show runtime contribution or speedup as a function of problem size for the eigenproblem solution phase of Section 5.3.2. The hollow rectangular waveguide cutoff problem – as implemented in Listing 5.3 – is considered with the problem size varied by changing the density of the mesh of Figure 5.3 in conjunction with the order of the functions spaces used. Note that for these results, only the calculation of the first seven TE modes is considered, and thus only `vector_order` of Listing 5.1 is changed. The performance results are shown for only System 1 and System 2 from Section 3.4.1, with the performance of System 3 not considered. The results are also limited to single precision calculations.

5.5.1 Runtime contribution

The results of Table 5.3 indicate that solving the generalised eigenvalue problem contributes most significantly to the total solution time for the dispersion (and by extension cutoff) problems. In this section, the contributions of the individual steps in the process (see Listing 5.8) are considered.

Figure 5.13 shows the runtime contributions, as a percentage of the total runtime, of the steps of the generalised eigensolver for the CPU implementation run on System 1. Here, it is evident that the shift-invert process contributes most significantly to the total eigensolver time, while the computation of the matrix vector product accounts for almost all of the ARPACK standard eigenvalue solution time. The results for System 2 are similar and are included in Appendix 6.2.

In the case of the CUDA implementation (again for System 1), the contributions of each of the phases are given in Figure 5.14. Here the shift-invert operation is also the dominant component of the total runtime. In this case, the contributions of the different parts of the shift-invert phase – the back-substitution step (implemented as a call to the MAGMA `magma_sgetrs_gpu()` routine), and the combined shift and LU decomposition step (implemented as a call to the MAGMA `magma_sgetrf_gpu()` routine) – are indicated by a dashed line and shows that the back-substitution step is most costly.

For the ARPACK standard eigenproblem solver phase, comparing Figure 5.13 to Figure 5.14 indicates that some speedup has been achieved. Furthermore, the change in contribution of the matrix-vector product, and the fact that the ARPACK driver is identical for both the CPU and CUDA versions, indicates that this speedup is due to the acceleration of the matrix vector product. As is the case for the CPU plot of Figure 5.13, the equivalent figure for System 2 is included in Appendix 6.2.

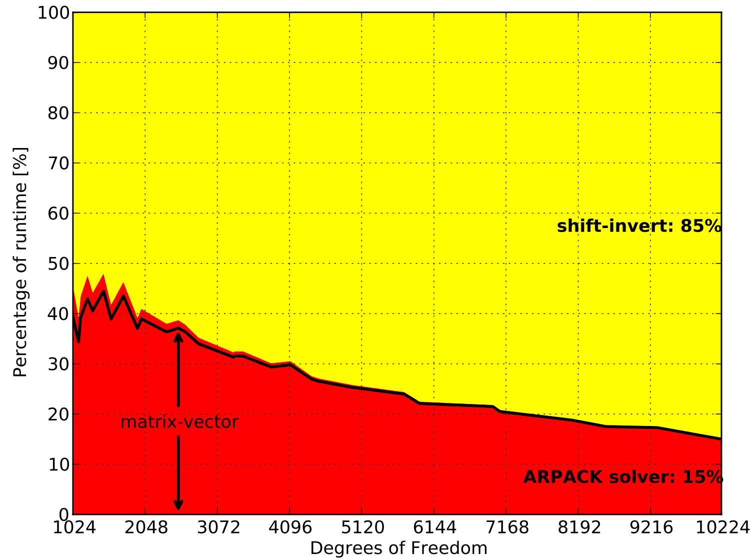


Figure 5.13: A graph showing the relative distribution of the execution times of the phases of the generalised eigensolver (discussed in Section 5.3.2) as a function of problem size. This figure shows the results of **single precision** calculations using the CPU implementation of the solver on **System 1**. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution for the largest problem considered) with the other phases grouped together but not visible due to their negligible contribution. The contribution of the matrix-vector product (as part of the ARPACK solver step) is also shown as a solid black line.

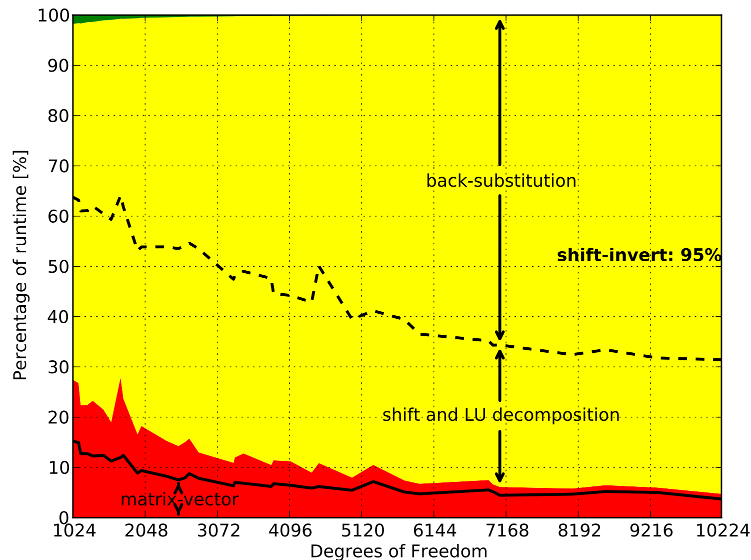


Figure 5.14: A graph showing the relative distribution of the execution times of the phases of the generalised eigensolver (discussed in Section 5.3.2) as a function of problem size. This figure shows the results of **single precision** calculations using the CUDA implementation of the solver on **System 1**. The contribution of the individual parts of the shift-invert process are divided by a dashed line and labelled accordingly with the contribution of the matrix-vector product (as part of the ARPACK solver step at the bottom of the graph) shown as a solid black line. In the case of the CUDA implementation considered here, there is some additional overhead that is not included in the individual timings an is just visible as a green section at the top left of the figure.

5.5.2 Speedup

Although Figure 5.13 and Figure 5.14 indicate that the ARPACK standard eigenvalue solver step of the solution process has been accelerated by using CUDA, they do not readily allow for the calculation of the attained speedups. The measured speedups of the CUDA implementation over the CPU implementation for System 1 are given in Figure 5.15. This figure shows a measured speedup of up to 22× for the ARPACK solution of the standard eigenvalue problem when using CUDA. When the shift-invert step is considered alone, the measured speedup when using the CUDA-accelerated MAGMA routines is a maximum of 6×. The CPU results are obtained using multi-core ACML routines. The combination of speedups results in a total measured speedup of almost 7× for the solution of the generalised eigenproblem resulting from the FEM cutoff analysis of the hollow rectangular waveguide.

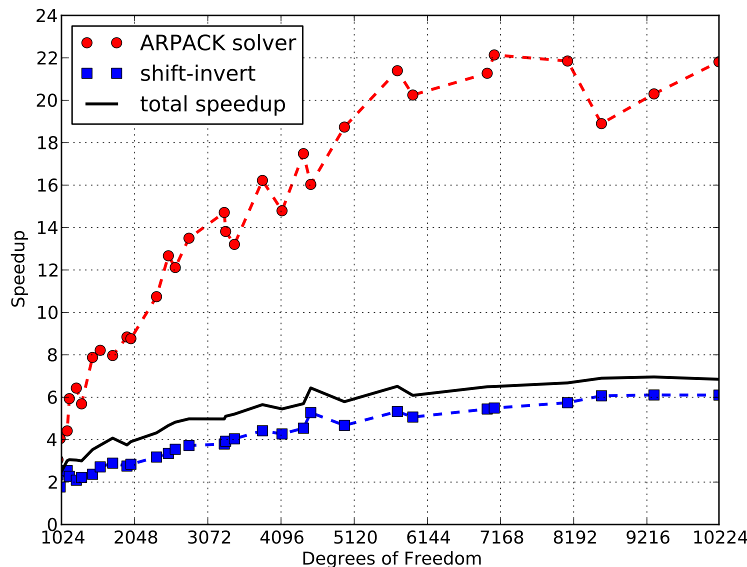


Figure 5.15: Measured speedups of the CUDA implementation over the CPU implementation on **System 1** as a function of problem size. The speedups for the ARPACK solver step (●) and shift-invert step (■) are shown along with the total solution speedup (–).

For System 2, the measured CUDA speedups – as shown in Figure 5.16 – are less than those attained in the case of System 1 (Figure 5.15). This can be attributed to an increase in processor performance over System 1 (also seen in the results of Chapter 3 and Chapter 4) in conjunction with similar single precision performance for the two CUDA devices.

Although the speedups are lower than those of System 1, Figure 5.16 still shows measured speedups of around 2× and 7× for the shift-invert and ARPACK solver steps, respectively. This translates to a total speedup of just more than 2×. It should be noted that the difference in total speedups for the two systems is comparable to the differences observed in Table 5.3 for a similar number of degrees of freedom.

Figure 5.17 provides a direct comparison of the total solution speedup results for the two systems considered here. As is the case for the Method of Moments in Section 4.5, the CPU results of System 1 are taken as the baseline and indicated on the graph by a unity speedup. Figure 5.17 shows that the CPU implementation on System 2 is about 2.5× faster than the System 1 CPU implementation, whereas both CUDA implementations attain a speedup of 6–7× relative to the System 1 CPU results. It should also be noted that the CUDA implementation on System 1 is more than 2× faster than the CPU implementation on System 2.

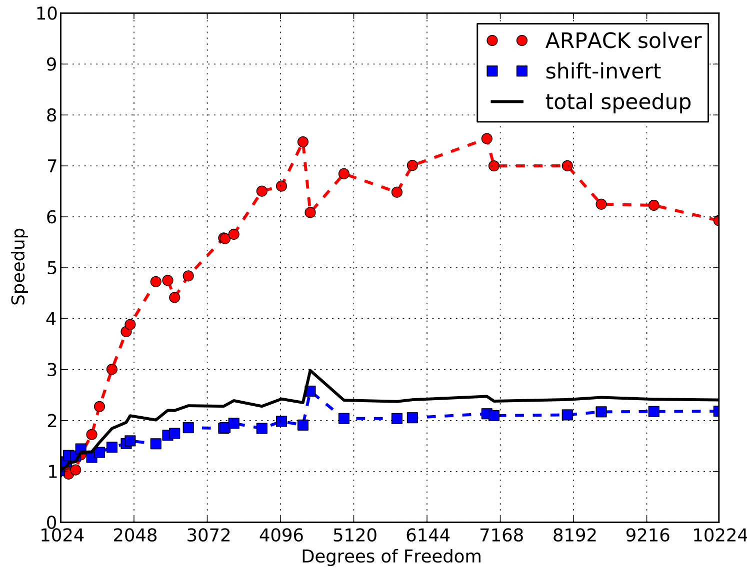


Figure 5.16: Measured speedups of the CUDA implementation over the CPU implementation on **System 2** as a function of problem size. The speedups for the ARPACK solver step (●) and shift-invert step (■) are shown along with the total solution speedup (–).

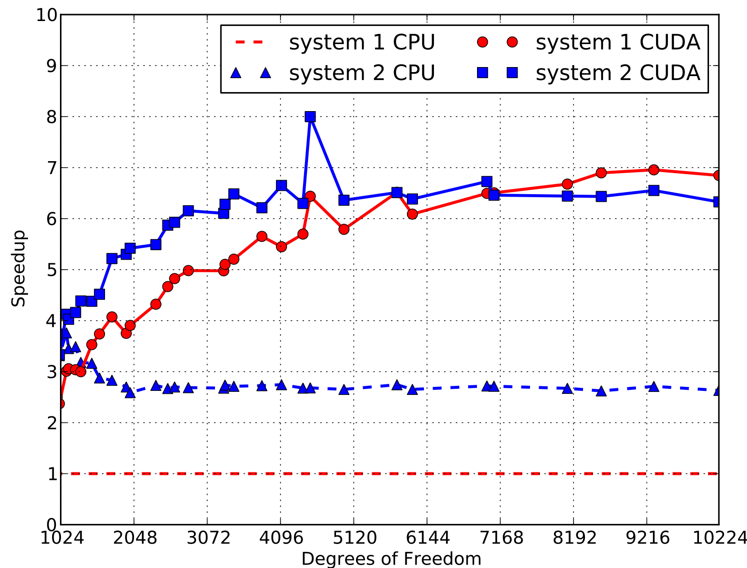


Figure 5.17: Total eigensolver speedup relative to the **System 1** CPU results as a function of problems size. The CPU speedups for **System 2** (▲) as well as the CUDA speedups for **System 1** (●) and **System 2** (■).

5.5.3 Discussion of results

The results shown in this section clearly show the advantage of the CUDA implementation over the CPU-based version of the generalised eigensolver implemented. Although not considered in detail, the timing results for the dispersion analysis of the shielded microstrip presented at the start of this section indicate that the amount of time required for the FEniCS portion of the solution process – assembling the finite element matrices – is far less than that required to obtain the solution to the resultant eigenproblems.

In terms of the use of an iterative eigensolver, of which ARPACK is an example, consider Figure 5.18 which shows the measured speedup of the CPU implementation of Section 5.3.2 over the LAPACK routine `SGGEV`. This routine computes the eigenvalues of the generalised eigenproblem of (5.30) using a direct dense method and is implemented in ACML. The speedups are shown for both the systems considered. In both cases, the ARPACK implementation greatly outperform the LAPACK routine.

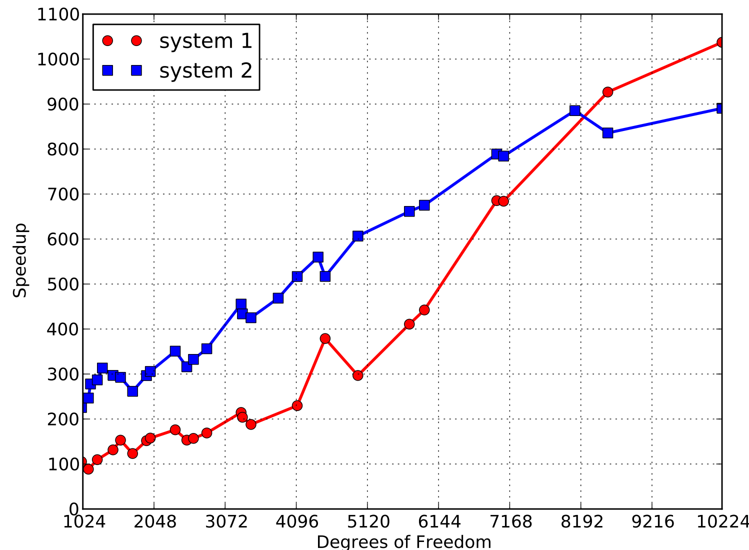


Figure 5.18: Total eigensolver speedups relative to a dense LAPACK eigensolver as a function of problems size. The CPU speedups for **System 1** (●) and **System 2** (■) are shown relative to a LAPACK implementation (using ACML) on each of the systems.

The large performance difference between the two versions is influenced by a number of factors. Firstly, an investigation of the CPU utilisation during the computation seems to indicate that the ACML implementation of `SGGEV` is not using multiple cores. Then, the LAPACK routine calculates all the eigenvalues and eigenvectors of the system, while the ARPACK-based implementation only calculates a small fixed number. This also explains why the speedup increases as the number of DOFs increase.

For both the CPU and CUDA implementation, the most costly part of solving the generalised eigenvalue problem is applying the shift-invert process to transform it to a standard eigenvalue problem as in (5.32). This is due to the fact that this process is $\mathcal{O}(N^3)$, where N is the number of degrees of freedom, in terms of the computational cost. This quickly outweighs the $\mathcal{O}(N^2)$ matrix-vector product shown to dominate the computational requirements of the ARPACK standard eigenproblem solver.

When considering the CUDA implementation specifically, the back-substitution is the most costly step, contributing about 70% of the total runtime for the largest problem considered (see Figure 5.14). Although both this routine (`SGETRS`) and the LU decomposition (`SGETRF`) are implemented for CUDA devices in MAGMA, the back-substitution does not offer the same performance improvement as the LU decomposition. An investigation of the MAGMA source code indicates that, for the present MAGMA implementation, the right-hand side vectors are transferred from the device, the row pivots from the LU decomposition are applied in host memory, and the pivoted matrix returned to the device, before applying the two triangular system solves. This additional data transfer greatly affects the performance of the routine.

As is the case for single precision computation in the LU decomposition (Section 3.4.2) and the Method of Moments (Section 4.5.2), the CUDA implementation on System 1 is able to best the CPU implementation on System 2 by a considerable margin. The CUDA results on System 2 show similar performance benefits. Although the System 3 results are not shown, it is clear that the CUDA implementation for System 1 and System 2 will at least narrow the performance advantage that the CPU results of System 3 would exhibit.

5.6 Conclusion

In this chapter two implementation aspects of the FEM analysis of waveguide structures have been discussed. The first of these is the use of FEniCS – an open-source software collection for the solution of partial differential equations – in the modelling and analysis of the cutoff and dispersion characteristics of these structures. A quick comparison of the equations in the problem formulations and the FEniCS code samples given, show that the process of going from formulation to implementation is greatly simplified when using high-level tools such as DOLFIN, which is included as part of FEniCS.

The ease with which this can be achieved allows for the rapid development and testing of new formulations. In addition, the object orientated approach followed by DOLFIN allows for the reuse of a large percentage of the code. This is demonstrated in the implementations presented, where only the problem specifics – such as material properties and boundary conditions – need to be redefined in each case. Although not discussed explicitly, DOLFIN also improves the ease with which results can be post-processed and visualised, with the results shown indicating that the solutions are of a high quality.

The second contribution of this chapter was the introduction of a CUDA-accelerated solver for generalised eigenvalue problems. In keeping with the co-development strategy of Chapter 4, both a CPU and CUDA framework were developed. The CUDA implementation shows a marked improvement, in terms of performance, over the CPU implementation, which in turn is significantly faster than using a dense direct generalised eigensolver as is supplied by LAPACK. Furthermore, the results also show that the use of a realtively cheap CUDA device in System 1 – the oldest system used for testing – allows for better performance than a multi-core implementation on a more current system. This is in agreement with the results of both Chapter 3 and Chapter 4.

In contrast to the results presented in Chapter 3 and Chapter 4, the results presented here are limited to single precision computation on System 1 and System 2, with the performance of System 3 not considered for administrative reasons. Although the limitation to single precision is an aspect that requires attention – and will be the focus of continued research – the availability of double precision implementations for all the CUBLAS and MAGMA routines used, means that this addition should be straightforward. Since the shift-invert step was found to be the dominant contributor to the eigenproblem solution time, performance results similar to those of Chapter 3 are expected.

As with most research, there are a number of of aspects to the work presented here that still require attention. The most important of these is the extension of the methods to support sparse matrices. In terms of the ARPACK-based standard eigenproblem solver, there should be no problem in this regard, as already demonstrated for the Lanczos process in [120]. Much research as gone into the CUDA acceleration of sparse matrix-vector products with notable contributions in [131] and [132]. The importance of sparse-matrix vector operations is also indicated by the recent release of the CUSPARSE library as part of version 3.2 of the CUDA toolkit [54], allowing for a high-level library-based approach to be used. Since the performance of these sparse methods are typically lower than the dense variants [131], it is expected that the speedups for the ARPACK standard eigenvalue phase will decrease if they are used.

The extension of the generalised eigenvalue problem to a sparse implementation poses more of a problem – especially for the matrices considered here. Since the matrices here are not positive semi-definite, the shifted matrix of (5.32) is explicitly constructed, with the suggestion that a direct sparse factorisation of the shifted matrix be computed and used in the matrix-vector product [14]. Although a number of CPU libraries such as SuperLU [133] exist that provide this functionality, there are no CUDA options that are mature and stable enough. Mention should be made of CUSP [134] and the work in [135], where CUDA acceleration is added to an existing sparse-matrix factorisation library with some success.

These sparse direct factorisations also often exhibit a large amount of fill-in, dramatically decreasing the advantage (in terms of memory requirements) of sparse matrices over dense matrices. The effect of fill-in can, however, be reduced by applying matrix bandwidth reduction and reordering techniques such as discussed in [136] and [137].

If the direct factorisation route is not followed, one option is to use an iterative solver such as GMRES [63] as part of the matrix-vector multiplication step in the ARPACK process. The use of such solvers for driven electromagnetic problems using the finite elment method has been demonstrated in [138]. In the case of the eigenvalue problems considered here, the matrix-vector product, and thus the solution of the linear system, would have to be applied a large number of times. This, and the typically large condition number of the finite element matrices for high order basis functions and fine meshes [116, 139, 140], means that the number of iterations for the convergence of the iterative solver will typically increase dramatically with an increase in problem size [116, 141]. This has been confirmed by initial testing.

The limit placed on solveable problem size by the amount of available GPU memory – as is addressed for the CUDA implementations of Chapter 3 and Chapter 4 – has not been addressed here. For the standard eigenvalue problem solver step, the use of sparse matrices will dramatically increase the size of the problems that can be addressed. It should be noted that, due to the $\mathcal{O}(N^2)$ computational and data storage requirements of the matrix-vector product, and the imbalance between the computational capability of CUDA devices and the bandwidth of the host-device link, an implementation that requires the matrix $\left[C\right]$ to be transferred to the device multiple times is expected to show a significant decrease in performance.

Despite the limitations of the eigensolver implemented here, the results in this chapter provide further evidence of the usefulness of CUDA acceleration in the field of computational electromagnetics, with the CUDA implementation out-performing the CPU implementation on both of the test systems used. Furthermore, the indication is also that the addition of a GPU to an existing system can be considered as a viable upgrade path to extend its usefulness. In terms of modelling, the use of FEniCS in the realisation of the formulations as presented has a large advantage in terms of ease of use over self-implemented codes. This is especially true if the primary focus of an implementation is the investigation of the effect of various formulation options.

Chapter 6

Conclusion and future work

Although this dissertation has covered quite a wide variety of topics, it can be stated that the overarching goal of investigating the use of GPU acceleration in matrix-based CEM methods has been achieved. It should be noted that even though the term investigation does not necessarily have to imply tangible results, a number of notable contributions have been made.

The first of these is the successful implementation of two CUDA-based LU decompositions, the panel-based and MAGMA-panel-based hybrid implementations of Chapter 3. These not only provide significant speedup in all but the double precision results of the fastest (and most expensive) CPU system, but do so while overcoming the restrictions placed on the problem size by the amount of memory available on a device. These results – although in a condensed form – have already been published in [9].

In the case of the Method of Moments, the CUDA-based LU decomposition was used as part of the solution process, with the acceleration of the other phases also considered. Here the matrix assembly phase was found to also contribute significantly to the total runtime of the solver. Due to the inherently data-parallel nature of the computations in this phase, speedups of up to 45× and 300× for double and single precision implementations, repectively, over multi-core CPU implementations were attained when only considering the matrix assembly phase.

The MOM matrix assembly implementation presented in Chapter 4 is also resilient with respect to the amount of memory available. This, when combined with the MAGMA-panel-based hybrid LU decomposition, allows for an accelerated solution process that was shown to solve problems that require up to 16 times as much memory as is available on the CUDA device. This is done while still achieving a total solution speedup. Early results for these accelerated processes have been published in [11] and [12].

Although the finite element implementation and results show the most scope for future research and improvements, a number of meaningful contributions are presented in Chapter 5. The first of these is the implementation of a GPU-accelerated iterative generalised eigenvalue problem solver. This solver is built around a CUBLAS-enhanced ARPACK implementation of a standard eigenvalue solver that represents the continuation of the work in [15]. At present this solver is limited to use with dense matrices.

In addition to the eigensolver implementation, that shows a total measured speedup of between 2× and 7× for the two systems considered, Chapter 5 also investigates the application of FEniCS to the modelling of finite element waveguide problems [16]. It is found that the high-level descriptive language provided greatly simplifies the implementation of new finite element formulations. The examples used to generate the results of Chapter 5 will also be included as demo scripts in the FEniCS code repository.

6.1 Research observations

Apart from the specific contributions already mentioned, a number of observations can be made pertaining to the research and some of the objectives discussed in Section 1.1.

Although the speedups measured indicate success (according to the criteria of a speedup of more than 1×, mentioned in Section 1.1), the comparative performance data also provides valuable insights in themselves. The results, for example, show that the 5 year old System 1 is able to perform competitively in most of the test cases with the addition of a relatively cheap CUDA GPU. This illustrates the feasibility of, or at very least the need for further investigation into, the use of GPUs as upgrades for existing systems to improve their computational usefulness.

One caveat, is that performance gains will only be attained for software that is implemented to take advantage of this massively parallel hardware and problems that are sufficiently parallel. However, as motivated in Chapter 1, there is a definite trend towards parallelisation and as such, the re-evaluation and extension of existing software to make use of parallel computing hardware should be a top priority.

In the MOM and FEM implementation sections of Chapter 4 and Chapter 5, the sharing of code between CPU and CUDA implementations was considered with varying levels of complexity. Chapter 4, specifically, showed how it is possible to implement a complex code for parallel execution on both a multi-core CPU using OpenMP and a CUDA device. Although this implementation may be superseded by compilers such as [6], or even properly designed OpenCL implementations, the exercise of developing such a parallel code is invaluable in designing systems that target vastly different platforms while being extensible and easy to maintain. This experience will also prove useful when developing code for other targets, such as OpenCL.

6.2 Future work

Since each of the results chapters provide a detailed discussion on possible future avenues of research specific to their results, the recommendations given here are kept relatively general in nature. These then serve to guide longer term research objectives, with the chapter-specific recommendations typically representing short-term project ideas.

The research and results presented here are limited to single multi-core machines making use of a single GPU. Although this is a perfectly valid everyday use-case, one of the factors that should receive further attention is the use of multiple GPUs installed in a single host to further improve performance. For many of the algorithms considered here, such as the MOM matrix assembly, and the matrix-vector product that is performed as part of the ARPACK standard eigensolver process, the additional complexity required is minimal and thus such an implementation should results in an almost linear speedup in the number of devices used. As with all parallel implementations, however, this is subject to a number of constraints, including that the problems being considered be sufficiently large.

In the case of more complex operations such as the LU decomposition, the use of a runtime environment such as StarPU [82] should be considered. As discussed in Chapter 3, StarPU in particular has already demonstrated very promising results using both multiple GPUs and multi-core CPUs to achieve significant speedups in the various matrix factorisations including the LU decomposition.

The appearance of libraries and tools such as MAGMA [8] and StarPU are commonplace in a field that is as rapidly evolving as GPU-computing. Given a few more months to a year, the feature set of each of these may have already been extended to such a point so as to provide all the desired functionality out-the-box. That said, a wait-and-see approach is not recommended, if only for the experience that is picked up while struggling with a particularly difficult implementation.

While on the topic of the LU decomposition and MAGMA, one of the sets of routines identified in both the MOM and FEM solution process as contributing significantly to the total computational time are the LAPACK `xGETRS` routines. Although the single precision real implementation is accelerated in MAGMA [8], there still appears to be much room for improvement. The issue of direct sparse factorisations for use in generalised eigensolvers dealing with sparse matrices also needs urgent attention.

A final question that can be raised is the relative performance, for the problems considered here and in general, of CUDA and OpenCL. This can then be extended to the comparative performance of AMD and NVIDIA GPUs and which of these presents a better value option. One of the most attractive factors of OpenCL is its cross-platform nature, which will be much more of an advantage once platforms such as Fusion by AMD [5] become commonplace. As briefly discussed in Chapter 2, the OpenCL specification also includes support for a heterogeneous programming model which allows for the use of multiple compute devices. If implemented correctly, this would results in functionality similar to that exhibited by StarPU for multi-core and multi-GPU systems.

List of References

[1] Sutter, H.: The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, vol. 30, no. 3, March 2005.
Available at: `http://www.gotw.ca/publications/concurrency-ddj.htm` (Cited on pages 1 and 7.)

[2] Meuer, H., Strohmaier, E., Dongarra, J. and Simon, H.: Top 500 Supercomputer Sites. 2010.
Available at: `www.top500.org` (Cited on pages 1, 6, and 9.)

[3] NVIDIA Corporation: NVIDIA GeForce 8800 GPU architecture overview. Technical Brief TB-02787-001_v01, November 2006. (Cited on pages 1, 8, 10, 11, and 17.)

[4] NVIDIA Corporation: CUDA Zone – The resource for CUDA developers. 2010.
Available at: `http://www.nvidia.com/cuda` (Cited on pages 1 and 29.)

[5] Advanced Micro Devices: AMD Fusion™ Family of APUs: Enabling a Superior, Immersive PC Experience. Tech. Rep. 48423B, March 2010. (Cited on pages 2, 11, 29, and 132.)

[6] The Portland Group: PGI to Develop Compiler Based on NVIDIA CUDA C Architecture for x86 Platforms. 2010.
Available at: `http://www.pgroup.com/about/news.htm#42` (Cited on pages 2, 29, and 131.)

[7] NVIDIA Corporation: CUDA CUBLAS Library. User guide PG-05326-032_V01, August 2010. (Cited on pages 3, 22, 30, 35, and 67.)

[8] Tomov, S., Nath, R., Du, P. and Dongarra, J.: MAGMA Library. User guide version 0.2, November 2009.
Available at: `http://icl.cs.utk.edu/magma/` (Cited on pages 3, 23, 30, 32, 36, 37, 40, 41, 42, 45, 46, 56, 81, 131, and 132.)

[9] Lezar, E. and Davidson, D.: GPU-based LU Decomposition for Large Method of Moments Problems. *Electronics Letters*, vol. 46, no. 17, pp. 1194–1196, 19 August 2010.
Available at: `http://link.aip.org/link/?ELL/46/1194/1` (Cited on pages 3, 30, 37, 42, 60, and 130.)

[10] Rao, S., Wilton, D. and Glisson, A.: Electromagnetic scattering by surfaces of arbitrary shape. *IEEE Transactions on Antennas and Propagation*, vol. 30, no. 3, pp. 409–418, 1982. (Cited on pages 3, 5, 58, 59, 60, 61, 63, 64, 65, 67, 68, 81, 82, and 83.)

[11] Lezar, E. and Davidson, D.: *GPU Acceleration of Method of Moments Matrix Assembly using Rao-Wilton-Glisson Basis Functions*, vol. 1, pp. 56–60. Proceedings of the 2010 International Conference on Electronics and Information Engineering (ICEIE 2010), Kyoto, Japan, August 2010. (Cited on pages 3, 60, 95, and 130.)

[12] Lezar, E. and Davidson, D.: GPU-Accelerated Method of Moments by Example: Monostatic Scattering. *Antennas and Propagation Magazine, IEEE*, December 2010. In press. (Cited on pages 3, 60, and 130.)

[13] Pozar, D.M.: *Microwave Engineering*. 3rd edn. John Wiley & Sons, Inc., 2005. (Cited on pages 3, 98, 99, 100, 101, 107, 109, 111, 117, and 120.)

[14] Lehoucq, R., Sorensen, D. and Yang, C.: *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. Siam, 1998. (Cited on pages 4, 97, 98, 103, 104, 113, and 129.)

[15] Lezar, E. and Davidson, D.: *GPU-based Arnoldi factorisation for accelerating finite element eigenanalysis*. Proceedings of the 11th International Conference on Electromagnetics in Advanced Applications - ICEAA'09, Torino, Italy, September 2009. (Cited on pages 4, 35, 98, and 130.)

[16] The FEniCS Project. 2010.
Available at: `www.fenicsproject.org` (Cited on pages 4, 97, 98, 104, and 130.)

[17] Culler, D., Singh, J. and Gupta, A.: *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc, San Francisco, CA, 1999. (Cited on pages 6, 7, 8, 20, 21, and 26.)

[18] Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L. and Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pp. 97–104. Budapest, Hungary, September 2004. (Cited on pages 6 and 73.)

[19] OpenMP Architecture Review Board: OpenMP Application Program Interface. Programmer's reference Version 3.0, May 2008.
Available at: `http://www.openmp.org/mp-documents/spec30.pdf` (Cited on pages 7, 11, 73, 74, and 81.)

[20] Flynn, M.J.: Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, September 1972.
Available at: `10.1109/TC.1972.5009071` (Cited on page 7.)

[21] Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S. *et al.*: The Landscape of Parallel Computing Research: A View from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. (Cited on page 8.)

[22] Tarricone, L. and Esposito, A.: *Grid Computing for Electromagnetics*. Artech House, Norwood, MA, 2004. (Cited on page 8.)

[23] England, J.: A system for interactive modeling of physical curved surface objects. In: *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pp. 336–340. ACM New York, NY, USA, 1978. (Cited on page 8.)

[24] Potmesil, M. and Hoffert, E.: The pixel machine: a parallel image computer. *ACM SIGGRAPH Computer Graphics*, vol. 23, no. 3, pp. 69–78, 1989. (Cited on page 8.)

[25] Rhoades, J., Turk, G., Bell, A., Neumann, U. and Varshney, A.: Real-time procedural textures. In: *Proceedings of the 1992 symposium on Interactive 3D graphics*, pp. 95–100. ACM New York, NY, USA, 1992. (Cited on page 8.)

[26] NVIDIA Corporation: NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™. Whitepaper 1.1, 2009. (Cited on pages 8, 10, and 11.)

[27] Kirk, D.B. and Hwu, W.W.: *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann, Burlington, 2010. (Cited on pages 8, 10, 11, 13, 18, 19, 20, 21, 29, and 68.)

[28] Advanced Micro Devices: AMD Close to Metal™Technology Unleashes the Power of Stream Computing. 2006.
Available at: http://amd.com/us/press-releases/Pages/Press_Release_114147.aspx (Cited on pages 8 and 10.)

[29] NVIDIA Corporation: NVIDIA CUDA™. NVIDIA CUDA C Programming Guide Version 3.2, Santa Clara, CA, October 2010. (Cited on pages 9, 11, 13, 14, 15, 16, 17, 18, 19, 20, 22, 27, 70, 71, 75, 80, and 114.)

[30] Chu, M.M.: GPU Computing: Past, Present and Future with ATI Stream Technology, March 2009.
Available at: http://developer.amd.com/gpu_assets/GPUComputing-PastPresentandFuturewithATIStreamTechnology.pdf (Cited on page 9.)

[31] Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. and Dubey, P.: Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, vol. 38, pp. 451–460, June 2010. ISSN 0163-5964.
Available at: http://doi.acm.org/10.1145/1816038.1816021 (Cited on page 9.)

[32] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E. and Purcell, T.J.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007. (Cited on page 9.)

[33] GPGPU: General-Purpose Computation Using Graphics Hardware. 2008.
Available at: http://www.gpgpu.org/ (Cited on page 9.)

[34] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J. and Phillips, J.: GPU Computing. *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008. ISSN 0018-9219. (Cited on page 10.)

[35] NVIDIA Corporation: NVIDIA GeForce GTX 200 GPU Architectural Overview. Technical Brief TB-04044-001_v01, May 2008. (Cited on pages 10, 11, and 18.)

[36] Advanced Micro Devices: ATI Stream SDK v2.2 with OpenCL 1.1 Support. 2010.
Available at: http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx (Cited on pages 10 and 29.)

[37] Advanced Micro Devices: A Brief History of General Purpose (GPGPU) Computing. 2010.
Available at: http://amd.com/us/products/technologies/stream-technology/opencl/Pages/gpgpu-history.aspx (Cited on page 10.)

[38] Khronos Group: OpenCL. 2009.
Available at: http://www.khronos.org/opencl/ (Cited on pages 10 and 26.)

[39] Advanced Mirco Devices: ATI Stream Computing: OpenCL™. Programming Guide 1.05, August 2010. (Cited on pages 10, 23, 24, 25, and 28.)

[40] Khronos OpenCL Working Group: The OpenCL. Specification 1.1, September 2010. Available at: `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf` (Cited on pages 10, 23, 26, 27, and 28.)

[41] Kapasi, U., Rixner, S., Dally, W., Khailany, B., Ahn, J., Mattson, P. and Owens, J.: Programmable stream processors. *Computer*, pp. 54–62, 2003. (Cited on pages 11 and 27.)

[42] Sanders, J. and Kandrot, E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison Wesley, Upper Saddle River, NJ, 2010. (Cited on pages 11 and 29.)

[43] Rixner, S.: *Stream processor architecture*. Kluwer Academic Pub, 2001. (Cited on pages 12 and 27.)

[44] NVIDIA Corporation: NVIDIA CUDA. Reference manual, August 2010. (Cited on page 16.)

[45] Barrachina, S., Castillo, M., Igual, F., Mayo, R., Quintana-Orti, E. and Quintana-Orti, G.: Exploiting the capabilities of modern GPUs for dense matrix computations. Tech. Rep., Technical Report ICC 01-11-2008, Universidad Jaime I (Spain), 2008. (Cited on page 16.)

[46] Kanter, D.: NVIDIA's GT200: Inside a Parallel Processor. September 2008. Available at: `http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242` (Cited on pages 18 and 20.)

[47] Fujimoto, N.: Faster matrix-vector multiplication on GeForce 8800GTX. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8. 2008. ISBN 1530-2075. (Cited on pages 18 and 35.)

[48] NVIDIA Corporation: NVIDIA GF100: World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism. Whitepaper 1.5. (Cited on page 19.)

[49] Wong, H., Papadopoulou, M., Sadooghi-Alvandi, M. and Moshovos, A.: Demystifying GPU microarchitecture through microbenchmarking. In: *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235–246. IEEE, March 2010. (Cited on page 20.)

[50] NVIDIA Corporation: NVIDIA CUDA™: CUDA C Best Practices Guide. Tech. Rep. 3.2, August 2010. (Cited on pages 20 and 22.)

[51] NVIDIA Corporation: CUDA CUFFT Library. Tech. Rep. PG-05327-032_V01, August 2010. (Cited on page 22.)

[52] Ziemer, R. and Tranter, W.: *Principles of communications: systems, modulation, and noise*. Wiley, 2002. ISBN 9780471392538. Available at: `http://books.google.com/books?id=6R0fAQAAIAAJ` (Cited on page 22.)

[53] Lawson, C.L., Hanson, R.J., Kincaid, D.R. and Krogh, F.T.: Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, 1979. ISSN 0098-3500. (Cited on pages 22, 30, 31, 32, 36, and 39.)

[54] NVIDIA Corporation: CUDA CUSPARSE Library. Tech. Rep. PG-05329-032_V01, August 2010. (Cited on pages 23 and 128.)

[55] NVIDIA Corporation: CUDA CURAND Library. Tech. Rep. PG-05328-032_V01, August 2010. (Cited on page 23.)

[56] EM Photonics, Inc: CULA Programmer's Guide. User guide Release 2.1, August 31 2009. Available at: `http://www.culatools.com/files/docs/2.1/CULAReferenceManual_2.1.pdf` (Cited on pages 23, 34, and 36.)

[57] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A. and Sorensen, D.: *LAPACK Users' Guide*. 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999. ISBN 0-89871-447-8 (paperback). (Cited on pages 23, 30, 31, and 36.)

[58] NVIDIA Corporation: OpenCL Programming Guide for the CUDA Architecture. Tech. Rep., August 2010. (Cited on pages 23 and 28.)

[59] Advanced Mirco Devices: ATI Stream Computing: Compute Abstraction Layer (CAL). Programming Guide rev2.01, March 2010. (Cited on page 25.)

[60] Advanced Micro Devices: AMD Core Math Library for Graphic Processors: User Guide for Version 1.1. Tech. Rep. 31208, July 2010. (Cited on pages 25 and 35.)

[61] Institute for Microelectronics, TU Wien: Viennacl. 2010.
Available at: `http://viennacl.sourceforge.net/index.html` (Cited on page 25.)

[62] Dongarra, J., Hammarling, S. and Walker, D.W.: Key Concepts for Parallel Out-of-Core LU Factorization. *Computers and Mathematics with Applications*, vol. 35, no. 7, pp. 13–31, 1998. (Cited on pages 30, 31, 32, 38, 39, 40, and 41.)

[63] Golub, G.H. and van Loan, C.F.: *Matrix Computations*. 3rd edn. The John Hopkins University Press, Baltimore, 1996. (Cited on pages 31, 32, 33, 36, 37, 45, 98, 102, 103, 104, and 129.)

[64] Advanced Micro Devices: AMD Core Math Library (ACML). User Guide Version 4.4.0, 2010. (Cited on pages 31, 35, and 81.)

[65] Whaley, R. and Petitet, A.: Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, 2005. ISSN 1097-024X. (Cited on pages 32, 34, and 35.)

[66] Intel: Intel® Math Kernel Library. User's Guide 314774-009US, March 2009.
Available at: `http://www.intel.com/software/products/` (Cited on page 34.)

[67] EM Photonics: CULA Tools - Performance. 2010.
Available at: `http://www.culatools.com/features/performance/` (Cited on page 34.)

[68] MathWorks: MATLAB - The Language Of Technical Computing. 2010.
Available at: `http://www.mathworks.com/products/matlab/` (Cited on page 35.)

[69] Eaton, J.W.: *GNU Octave Manual*. Network Theory Limited, 2002. ISBN 0-9541617-2-6. (Cited on page 35.)

[70] Nath, R., Tomov, S. and Dongarra, J.: An Improved MAGMA GEMM for Fermi GPUs. LAPACK Working Note 227 UT-CS-10-655, September 15 2010.
Available at: `http://www.netlib.org/lapack/lawnspdf/lawn227.pdf` (Cited on pages 35, 36, and 56.)

[71] Li, Y., Dongarra, J. and Tomov, S.: A Note on Auto-tuning GEMM for GPUs. Lapack Working Note 212 UT-CS-09-635, January 12 2009.
Available at: `http://www.netlib.org/lapack/lawnspdf/lawn212.pdf` (Cited on page 35.)

[72] Tomov, S., Nath, R., Ltaief, H. and Dongarra, J.: Dense linear algebra solvers for multicore with GPU accelerators. Tech. Rep., Innovative Computing Laboratory, University of Tennessee, 2009. (Cited on pages 36 and 41.)

[73] Galoppo, N., Govindaraju, N., Henson, M. and Manocha, D.: LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In: *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 3. IEEE Computer Society, 2005. ISBN 1595930612. (Cited on page 36.)

[74] Hall, J.D., Carr, N.A. and Hart, J.C.: Cache and bandwidth aware matrix multiplication on the gpu. Tech Report UIUCDCS-R-2003-2328, University of Illinois, Department of Computer Science, March 2003.
Available at: `http://graphics.cs.uiuc.edu/~jch/papers/gpumatrixmul.pdf` (Cited on page 36.)

[75] Fatahalian, K., Sugerman, J. and Hanrahan, P.: Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '04, pp. 133–137. ACM, New York, NY, USA, 2004. ISBN 3-905673-15-0.
Available at: `http://doi.acm.org/10.1145/1058129.1058148` (Cited on page 36.)

[76] Zee, F.G.V.: *libflame The Complete Reference*. r4556, draft edn. 2010. (Cited on page 36.)

[77] Castillo, M., Chan, E., Igual, F., Quintana-Ortı, E., Quintana-Ortı, G., van de Geijn, R. and Van Zee, F.: Making programming synonymous with programming for linear algebra libraries. FLAME Working Note #31 TR-08-20, The University of Texas at Austin, Department of Computer Sciences, May 9 2008.
Available at: `http://www.cs.utexas.edu/users/flame/pubs/flawn31.pdf` (Cited on page 36.)

[78] Fogué, M., Igual, F.D., Quintana-Ortí, E. and van de Geijn, R.: Retargeting PLAPACK to Clusters with Hardware Accelerators. FLAME Working Note #42 Technical Report TR-10-06, The University of Texas at Austin, Department of Computer Sciences, February 11 2010.
Available at: `http://www.cs.utexas.edu/users/flame/pubs/FLAWN42.pdf` (Cited on page 36.)

[79] Igual, F.D., Quintana-Ortí, G. and van de Geijn, R.: Level-3 BLAS on a GPU: Picking the Low Hanging Fruit. FLAME Working Note #37 Technical Report DICC 2009-04-01, Universidad Jaume I, Depto. de Ingenieria y Ciencia de Computadores, May 27 2009.
Available at: `http://www.cs.utexas.edu/users/flame/pubs/FLAWN37.pdf` (Cited on page 36.)

[80] Marqués, M., Quintana-Ortí, G., Quintana-Orti, E.S. and van de Geijn., R.: Solving Large Dense Matrix Problems on Multi-Core Processors and GPUs. FLAME Working Note #36 Technical Report ICC 01-01-2009, Universidad Jaume I, Depto. de Ingenieria y Ciencia de Computadores., January 7 2009.
Available at: `http://www.cs.utexas.edu/users/flame/pubs/FLAWN36.pdf` (Cited on page 36.)

[81] Joffrain, T., Quintana-Ortí, E. and van de Geijn, R.: Rapid development of high-performance out-of-core solvers. In: Dongarra, J., Madsen, K. and Wasniewski, J. (eds.), *Applied Parallel Computing*, vol. 3732 of *Lecture Notes in Computer Science*, pp. 413–422. Springer Berlin / Heidelberg, 2006.
Available at: `http://dx.doi.org/10.1007/11558958_49` (Cited on pages 36, 38, 40, and 45.)

[82] Augonnet, C., Thibault, S. and Namyst, R.: StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Technical Report 7240, INRIA, March 2010.
Available at: `http://hal.archives-ouvertes.fr/inria-00467677` (Cited on pages 37, 57, 96, and 131.)

[83] Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S. and Tomov, S.: Faster, Cheaper, Better a Hybridization Methodology to Develop Linear Algebra Software for GPUs. Lapack Working Note 230 UT-CS-10-658, September 15 2010.
Available at: `http://www.netlib.org/lapack/lawnspdf/lawn230.pdf` (Cited on pages 37 and 57.)

[84] Pennycook, S., Hammond, S., Mudalige, G. and Jarvis, S.: Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark. In: *1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 10) held in conjunction with IEEE/ACM Supercomputing 2010 (SC'10)*. New Orleans, LA, USA, 2010.
Available at: `http://eprints.dcs.warwick.ac.uk/110/1/uw22-9-2010-2.0.pdf` (Cited on page 37.)

[85] Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R.C.: *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. ISBN 0-89871-397-8 (paperback). (Cited on page 38.)

[86] Silberschatz, A., Galvin, P., Gagne, G. and Silberschatz, A.: *Operating System Concepts*. Seventh edn. John Wiley & Sons, Inc., Hoboken, NJ 07030, 2005. ISBN 0-471-69466-5. (Cited on page 38.)

[87] Volkov, V. and Demmel, J.: LU, QR and Cholesky factorizations using vector capabilities of GPUs. LAPACK Working Note 202 UCB/EECS-2008-49, May 2008 2008. (Cited on page 41.)

[88] Davidson, D.B.: *Computational Electromagnetics for RF and Microwave Engineers*. 2nd edn. Cambridge University Press, Cambridge, 2011. (Cited on pages 58, 60, 61, 62, 63, 97, 98, 101, 117, and 119.)

[89] Takahashi, T. and Hamada, T.: GPU-accelerated boundary element method for Helmholtz' equation in three dimensions. *International Journal for Numerical Methods in Engineering*, vol. 80, pp. 1295–1321, 2009. (Cited on pages 58, 59, and 95.)

[90] Chen, R., Xu, K. and Ding, J.: Acceleration of MoM Solver for Scattering Using Graphics Processing Units (GPUs). In: *Wireless Technology Conference*. Oriental Institute of Technology, Taipei, 2008. (Cited on pages 58 and 59.)

[91] Peng, S. and Nie, Z.: Acceleration of the Method of Moments Calculations by Using Graphics Processing Units. *IEEE Transactions on Antennas and Propagation*, vol. 56, no. 7, pp. 2130–2133, 2008. (Cited on pages 58, 59, 76, and 96.)

[92] EM Software & Systems-S.A. (Pty) Ltd: FEKO. 2010.
Available at: `www.feko.info` (Cited on pages 58, 82, and 83.)

[93] Bailey, D.H.: DSFUN: A Double-Single Floating Point Computation Package. March 2005.
Available at: `http://crd.lbl.gov/~dhbailey/mpdist/dsfun90.tar.gz` (Cited on pages 59 and 95.)

[94] Killian, T., Faircloth, D. and Rao, S.: Acceleration of TM cylinder EFIE with CUDA. In: *Antennas and Propagation Society International Symposium, 2009. APSURSI '09. IEEE*, pp. 1 –4. 2009. ISSN 1522-3965. (Cited on page 59.)

[95] Inman, M.J., Elsherbeni, A.Z. and Reddy, C.J.: *CUDA Based GPU Solvers For Method of Moment Simulations*. 26th Annual Review of Progress in Applied Computational Elctromagnetics - ACES2010, Tampere, Finland, April 2010. (Cited on pages 59, 60, and 95.)

[96] Virk, B.: *Implementing Method of Moments on a GPGPU using NVIDIA CUDA*. Masters Thesis, Georgia Institute of Technology, Atlanta, GA, May 2010.
Available at: `http://smartech.gatech.edu/handle/1853/33980` (Cited on pages 59 and 60.)

[97] De Donno, D., Esposito, A., Monti, G. and Tarricone, L.: Parallel efficient method of moments exploiting graphics processing units. *Microwave and Optical Technology Letters*, vol. 52, no. 11, pp. 2568–2572, August 2010. ISSN 1098-2760. (Cited on pages 59 and 60.)

[98] Tarricone, L., Mongiardo, M. and Cervelli, F.: A Quasi-One-Dimensional Integration Technique for the Analysis of Planar Microstrip Circuits via MPIE/MoM. *IEEE Transactions on Microwave Theory and Techniques*, vol. 49, no. 3, p. 517, 2001. (Cited on page 60.)

[99] Inman, M. and Elsherbeni, A.: Programming video cards for computational electromagnetics applications. *Antennas and Propagation Magazine, IEEE*, vol. 47, no. 6, pp. 71 –78, 2005. ISSN 1045-9243. (Cited on page 60.)

[100] De Donno, D., Esposito, A. and L. Tarricone, L.C.: Introduction to GPU computing and CUDA programming: a case study on FDTD. *IEEE Antennas and Propagation Magazine*, vol. 53, no. 3, June 2010. In press. (Cited on page 60.)

[101] Gumerov, N. and Duraiswami, R.: Fast multipole methods on graphics processors. *Journal of Computational Physics*, vol. 227, no. 18, pp. 8290–8313, 2008. ISSN 0021-9991. (Cited on page 60.)

[102] Jenn, D.C.: *Radar and Laser Cross Section Engineering*. 2nd edn. American Institute of Aeronautics and Astronautics, Inc., Reston, Verginia, 2005. (Cited on pages 61, 62, and 66.)

[103] Makarov, S.N.: *Antenna and EM Modeling with MATLAB*. John Wiley & Sons, Inc., New York, 2002. (Cited on pages 62, 63, 64, 65, 66, and 82.)

[104] Smith, G.S.: *An Introduction to Classical Electromagnetic Radiation*. Cambridge University Press, 1997. (Cited on pages 64, 65, 66, and 99.)

[105] Kloss, G.: Automatic C Library Wrapping–ctypes from the Trenches. *The Python Papers*, vol. 3, no. 3, p. 5, 2008. (Cited on pages 67, 72, 73, and 113.)

[106] Dunavant, D.A.: High degree efficient symmetrical gaussian quadrature formulas for the triangle. *International Journal for Numerical Methods in Engineering*, vol. 21, pp. 1129–1148, 1985. (Cited on page 68.)

[107] Khayat, M. and Wilton, D.: Numerical evaluation of singular and near-singular potential integrals. *Antennas and Propagation, IEEE Transactions on*, vol. 53, no. 10, pp. 3180–3190, 2005. ISSN 0018-926X. (Cited on page 68.)

[108] Khayat, M., Wilton, D. and Fink, P.: An improved transformation and optimized sampling scheme for the numerical evaluation of singular and near-singular potentials. *Antennas and Wireless Propagation Letters, IEEE*, vol. 7, pp. 377–380, 2008. ISSN 1536-1225. (Cited on page 68.)

[109] Silvester, P.P.: Finite-element solution of homogeneous waveguide problems. *Alta Frequenza*, vol. 38, pp. 313–317, 1969. (Cited on page 97.)

[110] Silvester, P.P. and Ferrari, R.L.: *Finite elements for electrical engineers*. 3rd edn. Cambridge University Press, 1996. (Cited on pages 97, 98, and 104.)

[111] Jin, J.: *The Finite Element Method in Electromagnetics*. 2nd edn. John Wiley & Sons, Inc., 2002. (Cited on pages 97, 98, 99, 100, 101, 109, 117, 119, 120, and 121.)

[112] Volakis, J.L., Chatterjee, A. and Kempel, L.C.: *Finite Element Method Electromagnetics: Antennas, Microwave Circuits, and Scattering Applications*. IEEE Press, 1998. (Cited on page 97.)

[113] Pelosi, G., Coccioli, R. and Selleri, S.: *Quick Finite Elements for Electromagnetic Waves*. Artech House, 1998. ISBN 0890068488. (Cited on pages 97, 98, 99, 100, 101, 111, 117, 120, 121, and 122.)

[114] Henrotte, F., Heumann, H., Lange, E. and Hameyer, K.: Upwind 3-D Vector Potential Formulation for Electromagnetic Braking Simulations. *IEEE Transactions on Magnetics*, vol. 46, no. 8, pp. 2835–2838, August 2010. (Cited on pages 97 and 98.)

[115] Lee, J.-F., Sun, D.-K. and Cendes, Z.J.: Full-wave analysis of dielectric waveguides using tangential vector finite elements. *IEEE Trans. Microwave Theory Tech.*, vol. 39, no. 8, pp. 1262–1271, August 1991. (Cited on pages 98, 99, 100, 102, 103, and 104.)

[116] Salazar-Palma, M., Sarkar, T.K., Garćia-Castillo, L.-E., Roy, T. and Djordjević, A.: *Iterative and Self-Adaptive Finite-Elements in Electromagnetic Modeling*. Artech House Inc., 1998. (Cited on pages 98 and 129.)

[117] Bossavit, A.: *Computational Electromagnetics: Variational Formulations, Complementarity, Edge Elements*. Academic Press, 1998. (Cited on page 98.)

[118] Monk, P.: *Finite Element Methods for Maxwell's Equations*. Oxford University Press, Oxford, UK, 2003. (Cited on page 98.)

[119] Volkov, V. and Demmel, J.: Using GPUs to Accelerate the Bisection Algorithm for Finding Eigenvalues of Symmetric Tridiagonal Matrices. Technical Report UCB/EECS-2007-179, Electrical Engineering and Computer Sciences, University of California at Berkley, 2007. Available at: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-179.html` (Cited on page 98.)

[120] Gamillscheg, R.: An Implementation of a Lanczoz Eigenvalue-Solver in CUDA, January 2009. Available at: `http://www.uni-graz.at/~haasegu/Lectures/GPU_CUDA/WS09/CUDAlanczos.pdf` (Cited on pages 98 and 128.)

[121] Goedel, N., Warburton, T. and Clemens, M.: GPU accelerated discontinuous Galerkin FEM for electromagnetic radio frequency problems. In: *Antennas and Propagation Society International Symposium, 2009. APSURSI'09. IEEE*, pp. 1–4. IEEE, 2009. ISSN 1522-3965. (Cited on page 98.)

[122] Dongarra, J.J., Duff, I.S., Sorensen, D.C. and van der Vorst, H.A.: *Numerical Linear Algebra on High-Performance Computers*. 2nd edn. Society for Industrial Mathematics, 1987. ISBN 0898714281. (Cited on page 98.)

[123] Nédélec, J.C.: Mixed finite elements in $\Re^3$. *Numerische Mathematik*, vol. 35, pp. 315–341, 1980. (Cited on pages 100 and 104.)

[124] Saad, Y.: *Numerical Methods for Large Eigenvalue Problems*. 2nd edn. Manchester University Press, Manchester, UK, 2011. Available at: `http://www-users.cs.umn.edu/~saad/eig_book_2ndEd.pdf` (Cited on pages 102 and 103.)

[125] Hernández, V., Román, J., Tomás, A. and Vidal, V.: A survery of software for sparse eigenvalue problems. SLEPc Technical Report STR-6, Universidad Politécnica de Valencia, February 2009. Available at: `http://www.grycap.upv.es/slepc` (Cited on page 103.)

[126] Lehoucq, R. and Scott, J.: An Evaluation of Software for Computing Eigenvalues of Sparse Nonsymmetric Matrices. Technical Report CRPC-TR96712, Center for Research on Parallel Computation, Rice University, January 1996. (Cited on page 103.)

[127] Sorensen, D.: Implicitly Restarted Arnold/Lanczos Methods for Large Scale Eigenvalue Calculations. Tutorial, Department of Computational and Applied Mathematics, Rice University, October 1995. Available at: `http://www.caam.rice.edu/software/ARPACK/DOCS/tutorial.ps.gz` (Cited on page 103.)

[128] Nédélec, J.C.: A new family of mixed finite elements in $\Re^3$. *Numerische Mathematik*, vol. 50, pp. 57–81, 1986. (Cited on page 104.)

[129] Geuzaine, C. and Remacle, J.: Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities. *International Journal for Numerical Methods in Engineering*, vol. 79, no. 11, pp. 1309–1331, 2009. ISSN 1097-0207. (Cited on pages 111 and 112.)

[130] Goldberg, D.: What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, vol. 23, pp. 5–48, 1991. (Cited on page 119.)

[131] Bell, N. and Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008. (Cited on page 128.)

[132] Dehnavi, M., Fernandez, D. and Giannacopoulos, D.: Finite-Element Sparse Matrix Vector Multiplication on Graphic Processing Units. *Magnetics, IEEE Transactions on*, vol. 46, no. 8, pp. 2982–2985, 2010. ISSN 0018-9464. (Cited on page 128.)

[133] Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S. and Liu, J.W.H.: A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, vol. 20, no. 3, pp. 720–755, 1999. (Cited on page 129.)

[134] Bell, N. and Garland, M.: Cusp : Generic parallel algorithms for sparse matrix and graph computations. 2010.
Available at: `code.google.com/p/cusp-library` (Cited on page 129.)

[135] Christen, M., Schenk, O. and Burkhart, H.: General-purpose sparse matrix building blocks using the NVIDIA CUDA technology platform. In: *First Workshop on General Purpose Processing on Graphics Processing Units*. Boston, MA, October 2007. (Cited on page 129.)

[136] Esposito, A., Catalano, M.S.F., Malucelli, F. and Tarricone, L.: A new matrix bandwidth reduction algorithm. *Operations Research Letters*, vol. 23, no. 3-5, pp. 99 – 107, 1998. ISSN 0167-6377.
Available at: `http://www.sciencedirect.com/science/article/B6V8M-3VY0016-4/2/25f186c3954b8eb0cc3216613d32f6ae` (Cited on page 129.)

[137] Liu, W. and Sherman, A.: Comparative Analysis of the Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices. *SIAM Journal on Numerical Analysis*, vol. 13, no. 2, pp. 198–213, 1976. ISSN 0036-1429. (Cited on page 129.)

[138] Dziekonski, A., Lamecki, A. and Mrozowski, M.: On Fast Iterative Solvers with GPU Acceleration for Finite Elements in Electromagnetics. In: *The 10th International Workshop on Finite Elements for Microwave Engineerin*. Mill Falls, NH, USA, October 12–13 2010. (Cited on page 129.)

[139] Ainsworth, M. and Coyle, J.: Conditioning of hierarchic p-version Nédélec elements on meshes of curvilinear quadrilaterals and hexahedra. *SIAM Journal on Numerical Analysis*, vol. 41, no. 2, pp. 731–750, 2003. (Cited on page 129.)

[140] Stupfel, B.: A study of the condition number of various finite element matrices involved in the numerical solution of Maxwell's equations. *Antennas and Propagation, IEEE Transactions on*, vol. 52, no. 11, pp. 3048–3059, 2004. ISSN 0018-926X. (Cited on page 129.)

[141] Sun, D.-K., Lee, J.-F. and Cendes, Z.: Construction of nearly orthogonal Nédélec bases for rapid convergence with multilevel preconditioned solvers. *SIAM Journal on Scientific Computing*, vol. 23, no. 4, pp. 1053–1076, 2001. (Cited on page 129.)

Additional MOM performance results

This appendix contains a number of additional plots pertaining to method of moments performance results presented in Section 4.5.

System 1 double precision

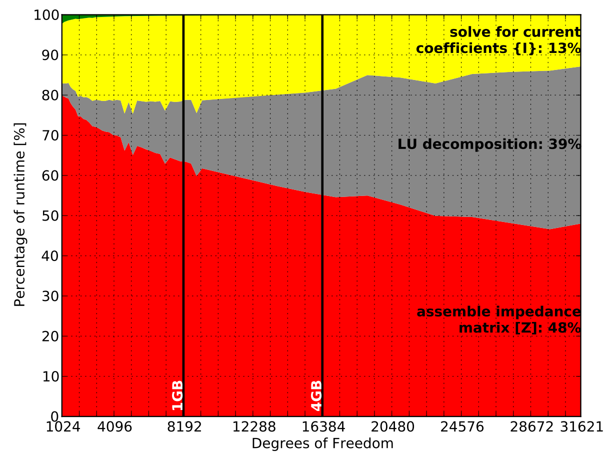


Figure 1: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.13. This figure shows the results for **double precision** calculations using the CPU-based implementation for **System 1**. The three phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the other phases (excluding initialisation) grouped together and just visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **double precision** complex values.

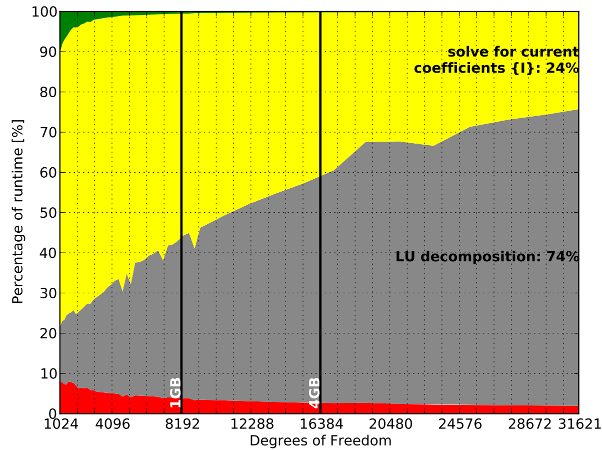


Figure 2: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.13. This figure shows the results for **double precision** calculations on **System 1** when only the matrix assembly phase is implemented in CUDA. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the CUDA matrix assembly contribution shown at the bottom of the graph (as is the case for the CPU assembly phase in Figure 4.14) and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **double precision** complex values.

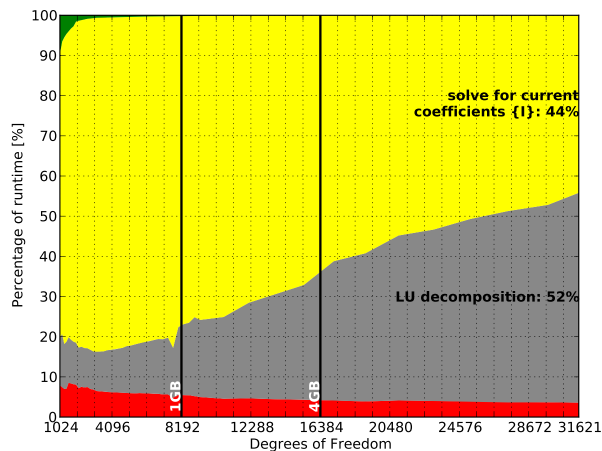


Figure 3: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.13. This figure shows the results for **double precision** calculations on **System 1** for the CUDA implementation. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the matrix assembly contribution shown at the bottom of the graph and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **double precision** complex values. The phase to solve for the unknown current coefficients is not implemented in CUDA.

System 2 single precision

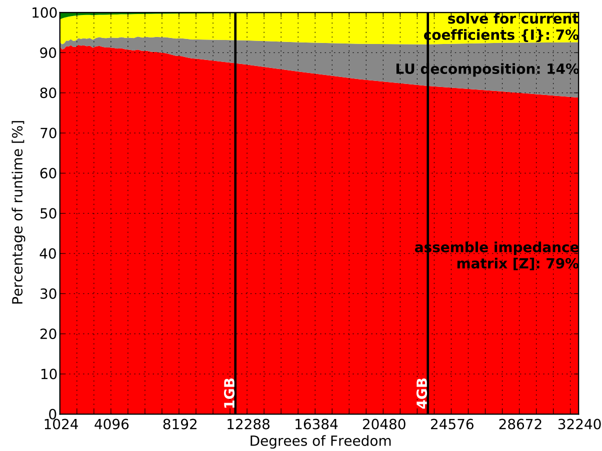


Figure 4: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.12. This figure shows the results for **single precision** calculations using the CPU-based implementation for **System 2**. The three phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the other phases (excluding initialisation) grouped together and just visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **single precision** complex values.

System 2 double precision

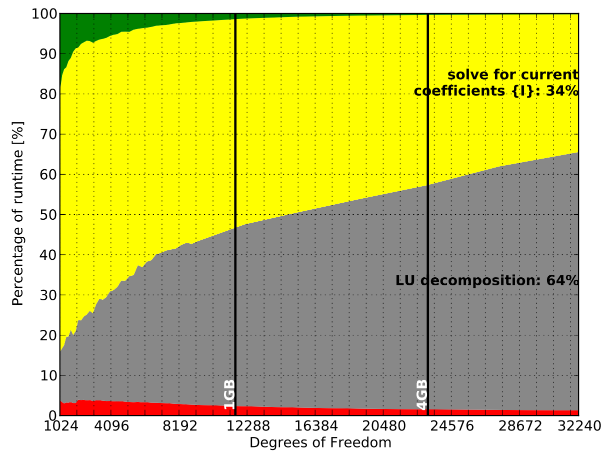


Figure 5: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.12. This figure shows the results for **single precision** calculations on **System 2** when only the matrix assembly phase is implemented in CUDA. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the CUDA matrix assembly contribution shown at the bottom of the graph (as is the case for the CPU assembly phase in Figure 4.14) and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **single precision** complex values.

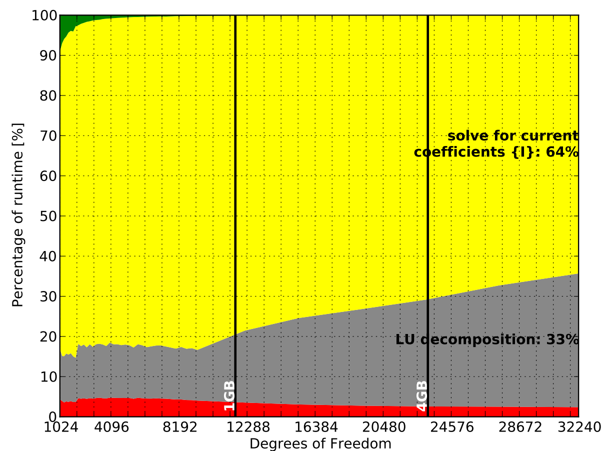


Figure 6: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.12. This figure shows the results for **single precision** calculations on **System 2** for the CUDA implementation. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the matrix assembly contribution shown at the bottom of the graph and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **single precision** complex values. The phase to solve for the unknown current coefficients is not implemented in CUDA.

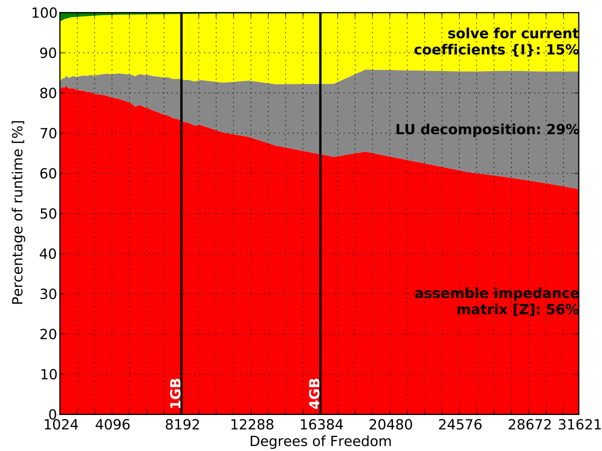


Figure 7: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.13. This figure shows the results for **double precision** calculations using the CPU-based implementation for **System 2**. The three phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the other phases (excluding initialisation) grouped together and just visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **double precision** complex values.

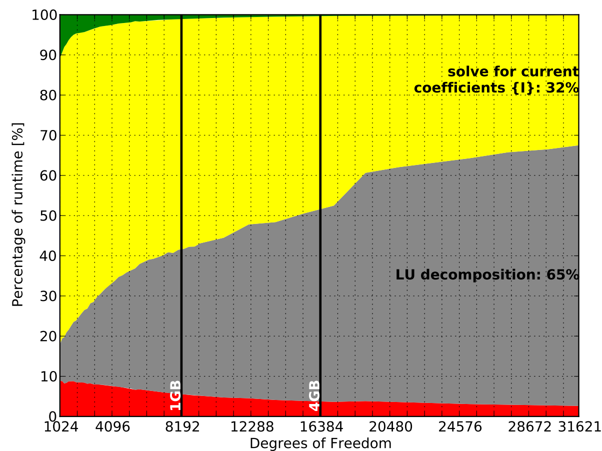


Figure 8: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.13. This figure shows the results for **double precision** calculations on **System 2** when only the matrix assembly phase is implemented in CUDA. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the CUDA matrix assembly contribution shown at the bottom of the graph (as is the case for the CPU assembly phase in Figure 1) and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **double precision** complex values.

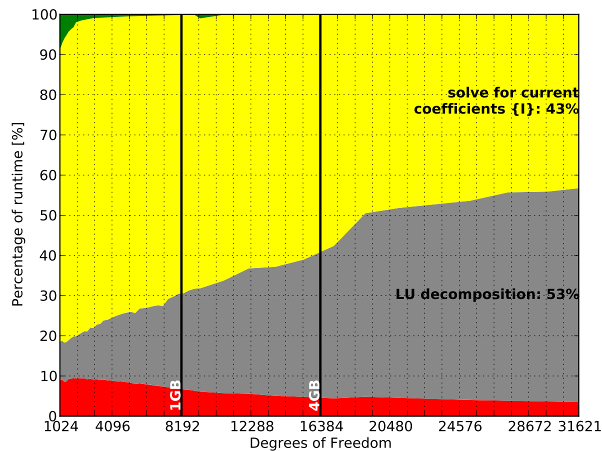


Figure 9: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.13. This figure shows the results for **double precision** calculations on **System 2** for the CUDA implementation. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the matrix assembly contribution shown at the bottom of the graph and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **double precision** complex values. The phase to solve for the unknown current coefficients is not implemented in CUDA.

System 3 single precision

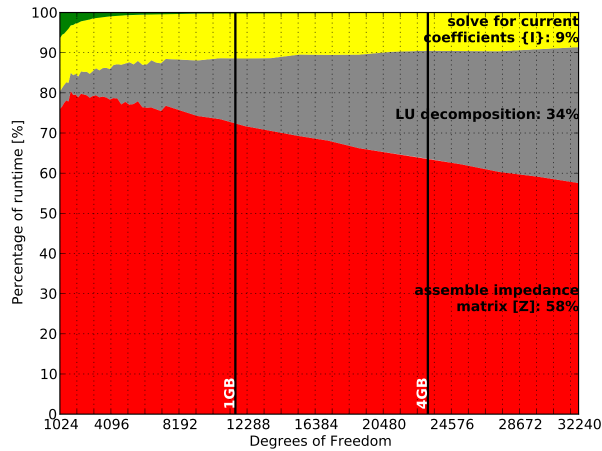


Figure 10: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.12. This figure shows the results for **single precision** calculations using the CPU-based implementation for **System 3**. The three phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the other phases (excluding initialisation) grouped together and just visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **single precision** complex values.

System 3 double precision

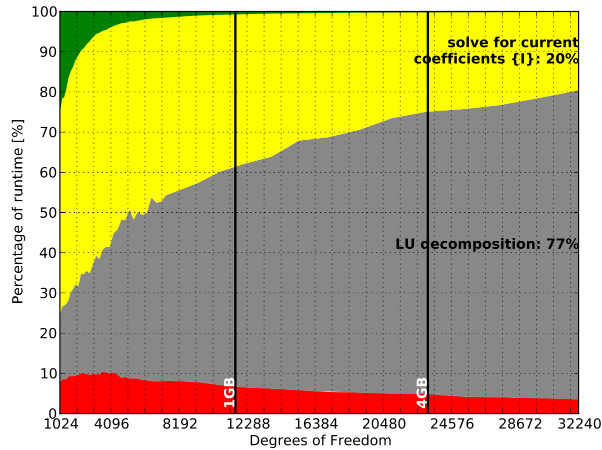


Figure 11: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.12. This figure shows the results for **single precision** calculations on **System 3** when only the matrix assembly phase is implemented in CUDA. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the CUDA matrix assembly contribution shown at the bottom of the graph (as is the case for the CPU assembly phase in Figure 4.14) and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **single precision** complex values.

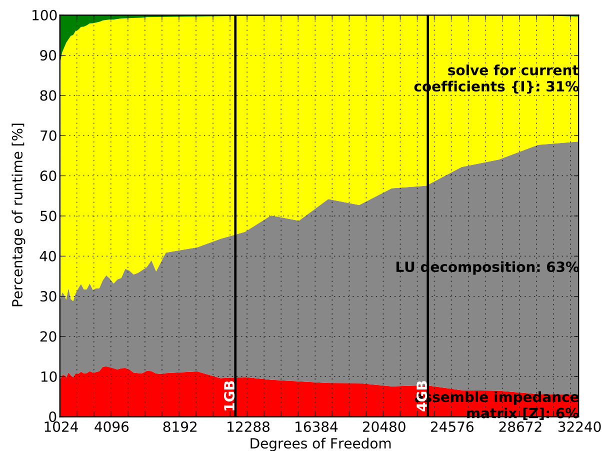


Figure 12: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.12. This figure shows the results for **single precision** calculations on **System 3** for the CUDA implementation. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the matrix assembly contribution shown at the bottom of the graph and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **single precision** complex values. The phase to solve for the unknown current coefficients is not implemented in CUDA.

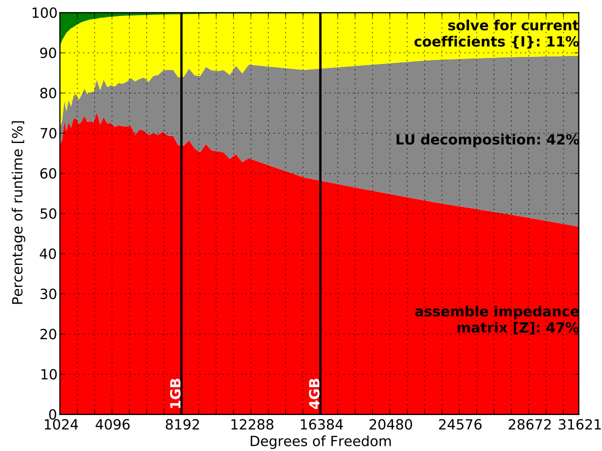


Figure 13: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.13. This figure shows the results for **double precision** calculations using the CPU-based implementation for **System 3**. The three phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the other phases (excluding initialisation) grouped together and just visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **double precision** complex values.

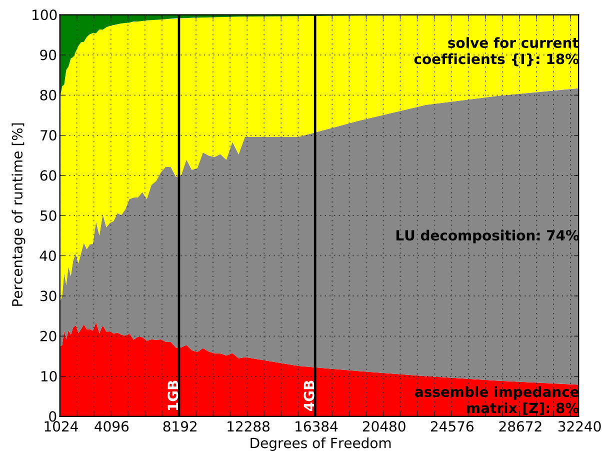


Figure 14: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.13. This figure shows the results for **double precision** calculations on **System 3** when only the matrix assembly phase is implemented in CUDA. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the CUDA matrix assembly contribution shown at the bottom of the graph (as is the case for the CPU assembly phase in Figure 1) and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **double precision** complex values.

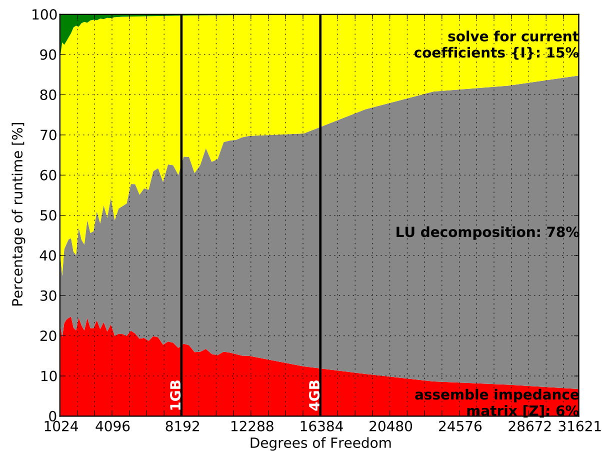


Figure 15: A graph showing the relative distribution of the execution times of the phases of the MOM computational process as a function of problem size (degrees of freedom) when calculating an RCS curve such as is shown in Figure 4.13. This figure shows the results for **double precision** calculations on **System 3** for the CUDA implementation. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution to the total solution time for the largest problem considered) with the matrix assembly contribution shown at the bottom of the graph and other phases (excluding initialisation) grouped together and visible in the upper left corner of the plot. Also shown are vertical lines corresponding to 1GB and 4GB storage requirements (in terms of only the impedance matrix) for **double precision** complex values. The phase to solve for the unknown current coefficients is not implemented in CUDA.

Additional FEM performance results

This appendix contains additional plots pertaining to finite element method performance results presented in Section 5.5.

System 2 single precision

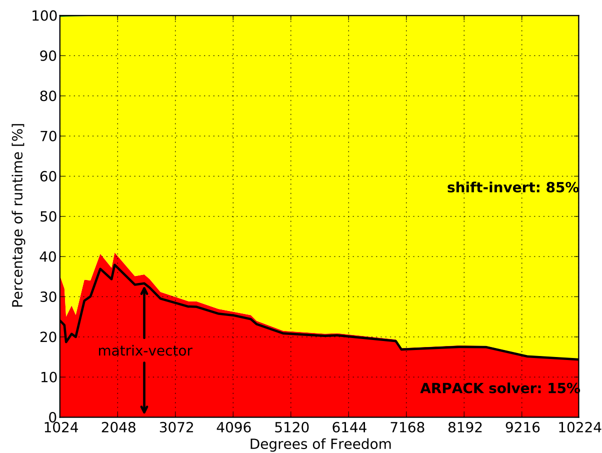


Figure 16: A graph showing the relative distribution of the execution times of the phases of the generalised eigensolver (discussed in Section 5.3.2) as a function of problem size. This figure shows the results of **single precision** calculations using the CPU implementation of the solver on **system 2**. The two phases that contribute most significantly to the total execution time are labelled (including their percentage contribution for largest problem considered) with the other phases grouped together but not visible due to their negligible contribution. The contribution of the matrix-vector product (as part of the ARPACK solver step) is also shown as a solid black line.

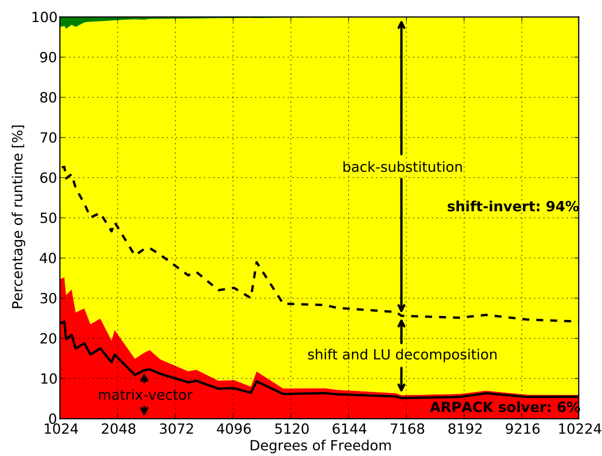


Figure 17: A graph showing the relative distribution of the execution times of the phases of the generalised eigensolver (discussed in Section 5.3.2) as a function of problem size. This figure shows the results of **single precision** calculations using the CUDA implementation of the solver on **system 2**. The contribution of the individual parts of the shift-invert process are divided by a dashed line and labelled accordingly with the contribution of the matrix-vector product (as part of the ARPACK solver step at the bottom of the graph) shown as a solid black line.