DOI:10.15774/PPKE.ITK.2014.003

ABSTRACTION AND IMPLEMENTATION OF UNSTRUCTURED GRID ALGORITHMS ON MASSIVELY PARALLEL HETEROGENEOUS ARCHITECTURES

István Zoltán Reguly

A thesis submitted for the degree of Doctor of Philosophy

Pázmány Péter Catholic University

Faculty of Information Technology

Supervisors:

András Oláh Ph.D.

Zoltán Nagy Ph.D.

Budapest, 2014

DOI:10.15774/PPKE.ITK.2014.003

I would like to dedicate this work to my loving family.

DOI:10.15774/PPKE.ITK.2014.003

Acknowledgements

First of all, I am most grateful to my supervisors, Dr. András Oláh and Dr. Zoltán Nagy for their motivation, guidance, patience and support, as well as Prof. Tamás Roska for the inspiring and thought-provoking discussions that led me down this path. I thank Prof. Barna Garay for all his support, for being a pillar of good will and honest curiosity. I am immensely grateful to Prof. Mike Giles for welcoming me into his research group during my stay in Oxford, for guiding and supporting me, and for opening up new horizons.

I would like to thank all my friends and colleagues with whom I spent these past few years locked in endless debate; Csaba Józsa for countless hours of discussion, Gábor Halász, Helga Feiszthuber, Endre László, Gihan Mudalige, Carlo Bertolli, Fabio Luporini, Zoltán Tuza, János Rudan, Tamás Zsedrovits, Gábor Tornai, András Horváth, Dóra Bihary, Bence Borbély, Dávid Tisza and many others, for making this time so much enjoyable.

I thank the Pázmány Péter Catholic University, Faculty of Information Technology, for accepting me as a doctoral student and supporting me throughout, and the University of Oxford, e-Research Centre. I am thankful to Jon Cohen, Robert Strzodka, Justin Luitjens, Simon Layton and Patrice Castonguay at NVIDIA for the summer internship, while working on AmgX together I have learned a tremendous amount.

Finally, I will be forever indebted to my family, enduring my presence and my absence, and for being supportive all the while, helping me in every way imaginable.

DOI:10.15774/PPKE.ITK.2014.003

Kivonat

Lassan tíz éve már hogy a processzorok órajelének folyamatos növekedése - és ezzel a frekvencia-skálázódásnak köszönhető teljesítmény-növekedés - véget ért. Ez arra kényszerítette a gyártókat, hogy a számítási kapacitás elvárt növekedését más eszközökkel tartsák fenn. Megkezdődött a processzormagok számának növekedése, megjelentek a többszintű memória-architektúrák, a vezérlő áramkörök egyre bonyolultabbá váltak, új architektúrák, segédprocesszorok, programozási nyelvek és párhuzamos programozási technikák sokasága jelent meg. Ezek a megoldások a teljesítmény növelését, a hatékony programozhatóságot és az általános-célú felhasználást hivatottak biztosítani. A modern számítógépeken a számítási kapacitás ilyen módon való növelésének azonban a - tipikusan nagyméretű - adatállományok gyors mozgatása szab határt. Ugyancsak korlátozó tényező, hogy a tudományos számítások területén dolgozók még mindig az évtizedekkel ezelőtt bevezetett - jól bevált és tesztelt - programozási nyelveket és a bennük implementált módszereket kívánják használni. Ezektől a kutatóktól nem elvárható, hogy a saját szakterületükön kívül még az új hardverek és új programozási módszerek terén is mély ismeretekre tegyenek szert. Ugyanakkor ezek mélyreható ismerete nélkül egyre kevésbé aknázható ki az új hardverekben rejlő egyre magasabb potenciál. Az informatika-tudományok kutatói – köztük jelen dolgozat szerzője is – arra vállalkoznak, hogy az új hardverekkel és programozási módszerekkel kapcsolatos tapasztalataikat, eredményeiket olyan formába öntsék – illetve olyan absztrakciós szinteket definiáljanak – amely lehetővé teszi a más kutatási területeken dolgozók számára a folyamatosan növekedő számítási kapacitások hatékony kihasználását.

Hosszú évtizedek kutatásai ellenére a mai napig nincs olyan általános célú gépi fordító mely a régi, soros kódokat automatikusan képes lenne hatékonyan párhuzamosítani és a modern hardvereket hatékonyan kihasználva futtatni. Ezért az elmúlt időszak kutatásai egy-egy jól körülhatárolható probléma osztályt céloznak meg – sűrű és ritka lineáris algebra, strukturált és nem-strukturált térhálók, soktest problémák, Monte Carlo. Kutatásom céljának – a fenti törekvésekkel összhangban – a nem-strukturált térhálókon értelmezett algoritmusokat választottam. A nem-strukturált térhálókat rendszerint olyan parciális dif-

ferenciálegyenletek diszkretizált megoldásánál használják ahol az értelmezési tartomány szerkezete rendkívül bonyolult és a megoldás során különböző területeken különböző térbeli felbontásra van szükség. Tipikus mérnöki problémák – amelyek a nem-strukturált térhálók segítségével oldhatók meg – például a sekélyvízi szimulációk összetett partvonal mentén, vagy repülőgépek sugárhajtóművének áramlástani szimulációja a lapátok körül.

Kutatásomat a programozhatóság jelen kori nagy kihívásai motiválták: a párhuzamosság, az adatlokalitás, a teherelosztás és a hibatűrés. A disszertációm célja az, hogy bemutassa a nem-strukturált térhálókon értelmezett algoritmusokkal kapcsolatos kutatásaimat. Kutatásaim során különböző absztrakciós szintekről kiindulva vizsgálom a nem-strukturált térhálókkal kezelhető probléma megoldásokat. Ezen eredményeimet végül számítógépes kódra képeztem le, különös figyelmet szentelve a különböző absztrakciós szintek, a programozási modellek, a végrehajtási modellek és a hardver architektúrák összhangjának. Futtatási eredményekkel bizonyítom, hogy a modern heterogén architektúrákon jelenlévő párhuzamosság és memória-hierarchia hatékonyan kihasználható kutatási eredményeim felhasználásával.

Munkám első része a végeselem-módszert vizsgálja, amely egy magasabb absztrakciós szintet jelent a dolgozat többi részéhez viszonyítva. Ez lehetővé tette, hogy egy magasabb szinten alakítsam ezt a numerikus módszert a modern párhuzamos architektúrák adottságait figyelembe véve. Kutatásom megmutatja hogy a végeselem integráció különböző formulációival a memóriamozgatást és tárigényt csökkenteni lehet redundáns számításokért cserébe, mely az erőforrásokban szűk GPU-k esetén képes magas teljesítményt adni, még magasrendű elemek esetén is. Ezt követően megvizsgálom az adatstruktúrák problémáját, és megmutatom hogy egy, a végeselem-módszerekre felírt struktúra, hogyan használható fel a számítások különböző fázisaiban, valamint levezetem a memória-mozgatás mennyiségét, és mérések segítségével bizonyítom, hogy a GPU architektúrán magasabb teljesítményt képes elérni. Végül a ritka mátrixok és vektorok szorzatának párhuzamos architektúrákra való leképzését vizsgálom, megadok egy heurisztikát és egy gépi tanulási eljárást mely az optimális paramétereket képesek megtalálni általános ritka mátrixok esetén is.

A második részben – az első rész aránylag szűk kutatási területét kibővítettem, és – azokat az általános nem-strukturált térhálókon értelmezett algoritmusokat vizsgáltam, melyek az OP2 *Domain Specific Language (DSL)* [16] által definiált absztrakcióval leírhatóak. Ez az absztrakciós szint alacsonyabb, mint az első részben a végeselem-módszer esetében alkalmazott absztrakciós szint, ezért a numerikus módszerek magasabb szintű átalakítására itt nincs lehetőség. Ugyanakkor az OP2 ezen absztrakciója által lefedett terület lényegesen szélesebb, így magában foglalja a végeselem-módszert, de emellett sok más algoritmust is, így például a véges térfogat módszert is. Kutatásom második része az OP2 absztrakciós szintjén definiált algoritmusok magas szintű transzformációival foglalkozik. Azt vizsgálja, hogy hogyan lehet a – tipikusan lineárisan megfogalmazott – számítási algoritmusok végrehajtását párhuzamos architektúrára transzformálni oly módon, hogy az adatlokalitás, a hibatűrés és az erőforrások kihasználtságának problémáját is megoldjuk. Ezen az absztrakciós szinten a kutatásaim még nem veszik figyelembe azt, hogy a végrehajtás milyen hardveren valósul meg. Ennek megfelelően a második rész eredményei megfelelő implementációs paraméterezéssel bármely végrehajtási környezetben felhasználhatóak. Elsőként egy teljesen automatizált algoritmust adok meg, mely képes megtalálni a végrehajtás során azt a pontot ahol az állapottér a legkisebb, elmenti azt, majd meghibásodás esetén képes attól a ponttól folytatni a számításokat. Ezek után általános megoldást adok a nem-struktúrált térhálókon értelmezett számítások olyan átszervezésére, mely az adatelérésekben lévő időbeli lokalitást javítja azzal, hogy egyszerre csak a probléma kisebb részén hajtja végre a számítások sorozatát - ú.n. "cache blocking" technikán alapulva. Megvizsgálom továbbá a mai heterogén rendszerek kihasználásának kérdését is, melyekben rendszerint több, eltérő tulajdonságokkal rendelkező hardver van. Megadok egy modellt a nem-struktúrált térhálón értelmezett algoritmusok heterogén végrehajtására, melyet az OP2-ben való implementációval igazolok.

Végül kutatásom harmadik része azzal foglalkozik, hogy az OP2 absztrakciós szintjén definiált algoritmusok miképp képezhetőek le különböző alacsonyabb szintű programozási nyelvekre, végrehajtási modellekre és hardver architektúrákra. Leképezéseket definiálok, amelyeknek segítségével a számításokat egymásra rétegzett párhuzamos programozási absztrakciókon keresztül a mai modern, heterogén számítógépeken optimálisan végre lehet hajtani, kihasználva az összetett memória-hierarchiát és a többszintű párhuzamosságot. Először bemutatom, hogy a végrehajtás miképp képezhető le GPU-kra, a CUDA programozási nyelv által, felhasználva a Single Instruction Multiple Threads (SIMT) modellt, valamint hogy hogyan lehet különböző optimalizációkat automatikusan kód generálás segítségével megvalósítani. Másodszor a CPU-szerű architektúrákra való leképzést mutatom meg, az egyes magokra OpenMP segítségével, valamint a magokon belül az AVX vektor utasításkészlet és a Single Instruction Multiple Data (SIMD) modell segítségével. A javasolt leképezések hatékonyságát áramlástani szimulációkon keresztül bizonyítom, többek közt a Rolls-Royce által repülőgép sugárhajtóművek tervezésére használt Hydra szoftver segítségével. Mindezt különböző hardvereken végzett mérések segítségével támasztom alá:

sokmagos CPU-kon, GPU-kon, az Intel Xeon Phi használatával valamint klasszikus és heterogén szuperszámítógépeken. Eredményeim azt bizonyítják, hogy a nem-strukturált térhálók probléma osztálya esetében a magas szintű absztrakciók módszere valóban képes a modern hardverekben rejlő teljesítmény automatizált kiaknázására és mindezek mellett hatékonyan leegyszerűsíti az algoritmikus problémák megfogalmazását és implementációját a felhasználók számára.

DOI:10.15774/PPKE.ITK.2014.003

Abstract

The last decade saw the long tradition of frequency scaling of processing units grind to a halt, and efforts were re-focused on maintaining computational growth by other means; such as increased parallelism, deep memory hierarchies and complex execution logic. After a long period of "boring productivity", a host of new architectures, accelerators, programming languages and parallel programming techniques appeared, all trying to address different aspects of performance, productivity and generality. Computations have become cheap and the cost of data movement is now the most important factor for performance, however in the field of scientific computations many still use decades old programming languages and techniques. Domain scientists cannot be expected to gain intimate knowledge of the latest hardware and programming techniques that are often necessary to achieve high performance, thus it is more important than ever for computer scientists to take on some of this work; to come up with ways of using new architectures and to provide a synthesis of these results that can be used by domain scientists to accelerate their research.

Despite decades of research, there is no universal auto-parallelising compiler that would allow the performance portability of codes to modern and future hardware, therefore research often focuses on specific domains of computational sciences, such as dense or sparse linear algebra, structured or unstructured grid algorithms, N-body problems and many others. For my research, I have chosen to target the domain of unstructured grid computations; these are most commonly used for the discretisation of partial differential equations when the computational domain is geometrically complex and the solution requires different resolution in different areas. Examples would include computational fluid dynamics simulations around complex shorelines or the blades of an aircraft turbine.

The aim of this dissertation is to present my research into unstructured grid algorithms, starting out at different levels of abstraction where certain assumptions are made, which in turn are used to reason about and apply transformations to these algorithms. They are then mapped to computer code, with a focus on the dynamics between the programming model, the execution model and the hardware; investigating how the parallelism and the

deep memory hierarchies available on modern heterogeneous hardware can be utilised optimally. In short, my goal is to address some of the programming challenges in modern computing; parallelism, locality, load balancing and resilience, in the field of unstructured grid computations.

The first part of my research studies the Finite Element Method (FEM), therefore starts at a relatively high level of abstraction, allowing a wide range of transformations to the numerical methods involved. I show that by differently formulating the FE integration, it is possible to trade off computations for communications or temporary storage; mapping these to the resource-scarce Graphical Processing Unit (GPU) architecture, I demonstrate that a redundant compute approach is scalable to high order elements. Then, data storage formats and their implications on the algorithms are investigated, I demonstrate that a FEM-specific format is highly competitive with classical approaches, deriving data movement requirements, and showing its advantages in a massively parallel setting. Finally, I parametrise the execution of the sparse-matrix product (spMV) for massively parallel architectures, and present a constant-time heuristic as well as a machine learning algorithm to determine the optimal values of these parameters for general sparse matrices.

Following the first part that focused on challenges in the context of the finite element method, I broaden the scope of my research by addressing general unstructured grid algorithms that are defined through the OP2 domain specific library [16]. OP2's abstraction for unstructured grid computations covers the finite element method, but also others such as the finite volume method. The entry point here, that is the level of abstraction, is lower than that of the FEM, thus there is no longer control over the numerical method, however it supports a much broader range of applications. The second part of my research investigates possible transformations to the execution of computations defined through the OP2 abstraction in order to address the challenges of locality, resilience and load balancing at a higher level, that is not concerned with the exact implementation. I present a fully automated checkpointing algorithm that is capable of locating the point during execution where the state space is minimal, save data, and in the case of a failure automatically fast-forward execution to the point of the last backup. Furthermore, I give an algorithm for the redundant-compute tiling, or cache-blocking, execution of general unstructured grid computations that fully automatically reorganises computations across subsequent computational loops in order to improve the temporal locality of data accesses. I also address the issue of utilisation in modern heterogeneous systems where different hardware with different performance characteristics are present. I give a model for such heterogeneous

DOI:10.15774/PPKE.ITK.2014.003

cooperative execution of unstructured grid algorithms and validate it with an implementation in OP2.

Finally, the third part of my research presents results on how an algorithm defined once through OP2 can be automatically mapped to a range of contrasting programming languages, execution models and hardware. I show how execution is organised on large scale heterogeneous systems, utilising layered programming abstractions, across deep memory hierarchies and many levels of parallelism. First, I discuss how execution is mapped to GPUs through CUDA and the Single Instruction Multiple Threads (SIMT) model, deploying a number of optimisations to execution patterns and data structures fully automatically through code generation. Second, I present a mapping to CPU-like architectures, utilising OpenMP or MPI for Simultaneous Multithreading as well as AVX vector instructions for Single Instruction Multiple Data (SIMD) execution. These techniques are evaluated on computational fluid dynamics simulations, including the industrial application Rolls-Royce Hydra, on a range of different hardware, including GPUs, multi-core CPUs, the Intel Xeon Phi co-processor and distributed memory supercomputers combining CPUs and GPUs. My work demonstrates the viability of the high-level abstraction approach and its benefits in terms of performance and developer productivity.

Abbreviations

AVX - Advanced Vector Extensions

AoS - Array of Structures

CG - Conjugate Gradient

CPU - Central Processing Unit

CSP - Communicating Sequential Processes

CSR - Compressed Sparse Row

CUDA - Compute Unified Device Architecture

DSL - Domain Specific Language

DDR - Double Data Rate (RAM type)

ECC - Error Checking & Correcting

FEM - Finite Element Method

GDDR - Graphics Double Data Rate (RAM type)

GMA - Global Matrix Assembly

GPU - Graphical Processing Unit

HPC - High Performance Computing

IMCI - Initial Many Core Instructions

LMA - Local Matrix Assembly

MPI - Message Passing Interface

NUMA - Non-Uniform Memory Access

OPlus - Oxford Parallel Library for Unstructured Grids

OP2 - The second version of OPlus

QPI - Quick Path Interconnect

RAM - Random Access Memory

RISC - Reduced Instruction Set Computing

SIMD - Single Instruction Multiple Data

SIMT - Single Instruction Multiple Threads

SMT - Simultaneous Multithreading

SM - Scalar Multiprocessor

SMX - Scalar Multiprocessor in Kepler-generation GPUs

SoA - Structure of Arrays

spMV - sparse Matrix-Vector multiplication

TBB - Threading Building Blocks

DOI:10.15774/PPKE.ITK.2014.003

Contents

DOI:10.15774/PPKE.ITK.2014.003

DOI:10.15774/PPKE.ITK.2014.003

DOI:10.15774/PPKE.ITK.2014.003

List of Figures

DOI:10.15774/PPKE.ITK.2014.003

DOI:10.15774/PPKE.ITK.2014.003

DOI:10.15774/PPKE.ITK.2014.003

List of Tables

DOI:10.15774/PPKE.ITK.2014.003

Chapter 1

Introduction

To understand the motivation behind this work, first we have to take a journey through the history of computing; both in terms of hardware and software. We have seen a staggering amount of innovation and an exponential growth in different aspects of hardware that are often difficult to keep track of, yet many of these technologies have reached and passed their climax, drawing the business-as-usual evolution we had enjoyed for two decades into question. At the same time, software has always been slower to evolve; therefore we must study its relationship to hardware and how this ever-widening chasm can be bridged - how the explosion in complexity and parallelism in modern hardware can be efficiently utilised for the purposes of scientific computations [1].

1.1 Hardware evolution

First, let us look at the evolution of hardware, and specifically changes to computer architecture, the memory system and how the two interact with each other. These are the key components that determine what computational throughput may be achieved and it offers an interesting insight into how certain parameters changed over time and what this meant to programming approaches, and it perhaps hints at what is to come.

1.1.1 Processing units

Microprocessor design has faithfully followed Moore's Law for the past forty years. While the number of transistors on a chip has been doubling approximately every two years, other characteristics have been undergoing dramatic changes. Processor voltage starting dropping from 5V in 1995, when constant field scaling (smaller feature sizes) enabled lower voltages and higher clock frequencies, but due to increasing leakage and practical power dissipation limitations, both have flattened out by 2005. This was an era of

DOI:10.15774/PPKE.ITK.2014.003

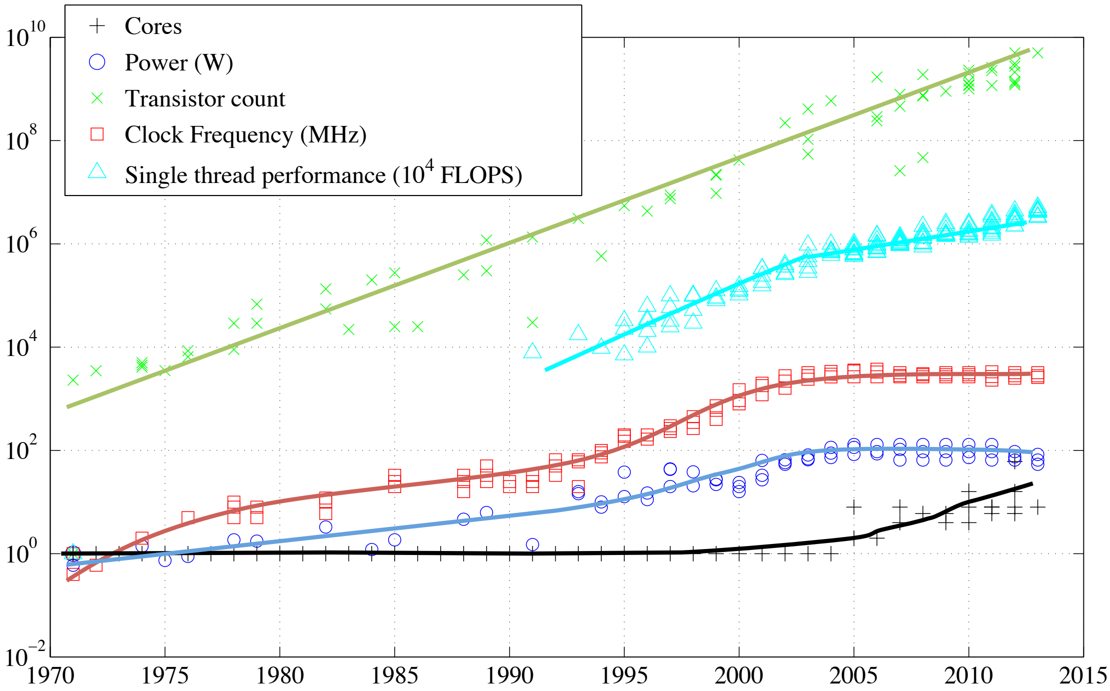


Figure 1.1. Evolution of processor characteristics

"free" scaling, with single thread performance increasing by 60% per year; supercomputer evolution at the time relied on flat parallelism and natural scaling from increased clock frequencies, with no more than a few thousand single-core, single-threaded processors.

By 2005, it had become clear that in order to maintain the growth of computational capacity, it would be necessary to increase parallelism; multi-core CPUs appeared and supercomputers went through a dramatic increase in processor core count. Due to market demands, some, such as Intel, keep manufacturing heavyweight architectures with high frequency, power-hungry cores and increasingly complicated logic in order to maintain some level of single thread performance scaling. Observe that Figure 1.1 still shows a 20% scaling per year after 2005, but this is at the expense of huge overheads to support out-of-order execution, branch prediction and other techniques aimed at reducing latency. If we were to ignore Intel's contributions to single thread performance figures, it would show a mere 13% increase per year. Other vendors, such as IBM with the Blue Gene, developed lightweight architectures with simpler processing cores that run at lower frequencies, thereby saving power; these are deployed at a massive scale. Alongside increasing core counts, there is a resurgence of vector processing; CPUs supporting the SSE, AVX or similar instruction sets are capable of operating on vectors, carrying out the same operation over the values packed into a 128-256 bit register. The emergence of accelerators took these trends to the extreme; GPUs and the Xeon Phi feature many processing cores that have very simplistic execution circuitry compared to CPUs, but feature much wider

DOI:10.15774/PPKE.ITK.2014.003

vector processing capabilities and support orders of magnitude higher parallelism.

Moore's law itself is in some danger as the limits of CMOS technology are likely to prohibit scaling beyond a few nanometers (1-5), at which point a transistor would only be 5-25 atoms across. One way to keep increasing the number of transistors is by stacking multiple chips on top of each other; however due to heat dissipation requirements and yield issues, this may be limited to a few layers (around 5). While the end of downscaling is still 10-15 years into the future, it is not clear what technology would succeed CMOS; there are several promising alternatives, such as graphene- or silicene-based technolgies, or carbon nanotubes [17], but most of them are still in their early stages.

1.1.2 Memory

While the economics of processor development has pushed them to gain increasingly higher performance, the economics of memory chip development favoured increasing capacity, not performance. This is quite apparent in their development; while in the 1980's memory access times and compute cycle times were roughly the same, by now there is at least two orders of magnitude difference, and accounting for multiple cores in modern CPUs, the difference is around a 1000×. Serially executed applications and algorithms therefore face the Von Neumann bottleneck; vast amounts of data have to be transferred through the high-latency, low-bandwidth memory channel, its throughput is often much smaller than the rate at which the CPU can work. While this is being addressed by adding multiple levels of caches and complex execution logic (such as out-of-order execution and branch prediction) onto CPUs, it also means that the vast majority of the transistors are dedicated to mitigating the effects of the Von Neumann bottleneck: to address the question of how to compute fast by reducing latency, as opposed to actually doing the computations.

It is possible to make faster memory chips, but it is expensive; SRAM, commonly used for fast on-chip caches, has a cell size that is 6 times larger than DRAM's. One of the main issues with these types of memory is that they are volatile; energy is required to maintain their contents. NAND flash memory on the other hand is non-volatile and has a small cell size, with a cell capable of storing multiple bits. However, it is currently not capable of supporting practically infinite read-write cycles ($> 10^{15}$ as SRAM and DRAM do), only on the order of $10^5 - 10^6$, and access speeds are comparatively slower as well, therefore it is not a viable candidate for replacing system memory. Although some current technologies address the latency and bandwidth issues, they are usually expensive to manufacture

DOI:10.15774/PPKE.ITK.2014.003

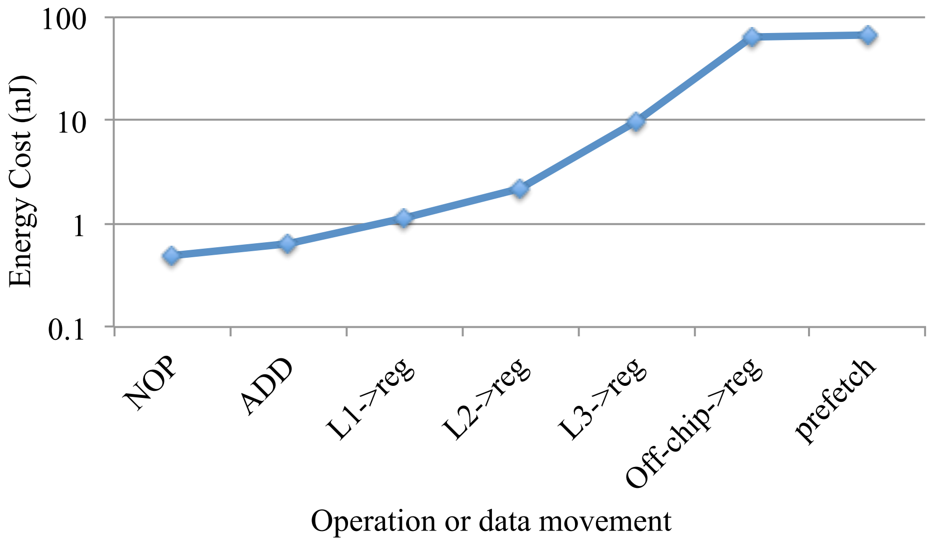


Figure 1.2. Cost of data movement across the memory hierarchy

and improvements in one area come with overheads in others; GDDR memory provides high bandwidth but at increased latency, reduced latency DRAM (RLDRAM) comes at the cost of increased die area. There is a considerable amount of research into emerging memory technologies; for example into Spin Transfer Torque RAM (STT-RAM), a type of magnetoresistive RAM, that promises fast, non-volatile memory rivalling DRAM in energy efficiency and area requirements and approaches SRAM in terms of performance. It is unclear how much performance potential is in these new technologies, but it is unlikely that they would deliver orders of magnitude improvements that would significantly change the balance between computational throughput and memory bandwidth/latency.

1.1.3 Putting it together

The greatest challenge is architecting a computing system that can deal with such disparities between the performance of different subsystems and one that can be replicated and interconnected to create high performance computing systems. The single biggest problem we are facing is the cost of moving data; as illustrated in Figure 1.2, accessing registers and carrying out operations is cheap, costing less than a nanojoule, accessing data from on-chip caches is much more expensive, and depends on the actual distance it has to be moved. Moving data off-chip is yet another order of magnitude more costly, tens of nanojoules. While we can expect a 10× increase in computational power by 2018 (compared to computational power available in 2013, following the same growth trends), there is currently no technology in production that would allow the same decrease in the cost of data movement. From a programming perspective, all these limitations ultimately manifest themselves as low memory bandwidth and high latency. Studies carried out on a range of

DOI:10.15774/PPKE.ITK.2014.003

applications on traditional CPU hardware show that 18-40% of power consumption comes from data movement and 22-35% comes from advanced execution logic such as out of order execution [18].

There are several ways to combat bandwidth and latency issues using general-purpose hardware, usually applied in combination; (1) using hierarchical memory structures (caches and scratchpad memory) that reduce access latency, (2) complex execution logic to circumvent serial execution, such as out of order execution and branch prediction, (3) through parallelism, so memory latency can be hidden by executing instructions from other threads that have data ready. CPUs are traditionally pushing the first two approaches, because market demand is strong for low latencies and fast single-threaded execution, but ever since the introduction of Hyper Threading and multiple cores, the third approach is also being adopted. Having multiple cores however doesn't solve the problem if only one thread can be efficiently executed on a core, because it will stall until data arrives. Having a huge amount of active threads resident on a smaller number of cores is where accelerators have made great progress; by keeping the state of threads in dedicated hardware, context switching becomes very cheap. Intel's Xeon Phi co-processor supports up to 4 threads per core and the latest Kepler generation NVIDIA GPUs can have up to 2048 threads per 192 cores. Accelerators have to rely on parallelism and a high degree of spatial locality to hide latency because there is not enough die area for large caches, access to off-chip memory has a high latency (hundreds of clock cycles) and bandwidth can only be saturated by having a large number of memory requests in-flight. Trends point to future architectures having an even wider gap between computational throughput and memory latencies, the number of cores on a chip is bound to increase at a rate that cannot be matched by increasing memory performance, therefore an even higher level of parallelism will be required to hide latency.

Advances in systems architecture and interconnects aim to maintain or improve the compute-latency ratio in the future. An important issue plaguing accelerators, especially GPUs, is the PCI-express bottleneck, which limits the rate at which data and computations can be offloaded; trends show that traditional CPU cores and accelerators are getting integrated together; Intel's Knight's Landing will be a stand-alone device, and NVIDIA's Project Denver aims to integrate ARM cores next to the GPU. Unlike the way floating point units and vector units were integrated into the CPU, accelerators are likely to remain well separated from general-purpose cores, because they rely on deep architectural differences to deliver such high performance. Furthermore, there is a trend of moving

DOI:10.15774/PPKE.ITK.2014.003

DRAM closer to the processing cores by including them in the same package; already some of Intel's Haswell CPUs include 128MB eDRAM - this of course is more expensive and difficult to manufacture, but offers higher bandwidth than system memory. Similar techniques are going to be adopted by NVIDIA's Volta generation of GPUs and Knight's Landing, which will have up to 16 GBs of on-chip stacked high-speed memory. The rise of silicon photonics promises fast interconnects where lasers, receivers, encoders, decoders and others can be directly integrated onto the chip. Eventually, optical links would be developed between CPUs in different nodes, the CPU and system memory, and perhaps even between different parts of the same chip thereby speeding up data movement and reducing energy costs. A greater convergence of system components is expected to take place in the future, where processing, networking and memory are all integrated into the same package, enabled by 3D stacking that allows layers of silicon circuitry to be stacked on top of each other, interconnected by through silicon vias (TSVs).

As high performance computing systems grow in size and complexity, resilience becomes an important issue, due to several factors: (1) the increase in the number of system components, (2) near-threshold voltage operation increases soft-error rate, (3) as hardware becomes more complex (heterogeneous cores, deep memory hierarchies, complex topologies, etc.) (4) software becomes more complex too, hence more error-prone, (5) application codes also become more complex because workflows and underlying programming techniques have to address increasing latency, tolerate asynchronicity and reduce communications. At extreme scale (such as future exascale systems) this will have serious consequences, but even at smaller scales, and assuming that a lot of progress is made by hardware manufacturers, developers of the operating system and libraries such as MPI, more attention will have to be given to developing resilient applications (by e.g. using checkpointing) with additional focus on correctness.

From a hardware point of view it seems clear that the trend of increasing latency is going to continue, and more complicated approaches will be required to combat it; exploiting locality is going to be essential to reduce the cost of data movement, and to hide the latency when data does have to be moved, a growing amount of parallelism will be needed. To mitigate memory access latencies, the size and the depth of memory hierarchies is expected to keep growing, and parallelism in the hardware will continue to be exposed on many levels; this presents great challenges in terms of expressing locality and parallelism through software, a subject of intense research.

DOI:10.15774/PPKE.ITK.2014.003

Table 1.1. Amount of parallelism (number of “threads”) required to saturate the full machine

Machine	Parallelism
K-computer	5.120.000
Sequoia	25.165.824
Tianhe-1	103.104.512
Titan	403.062.784
Tianhe-2	90.624.000

1.2 Implications to Programming Models

Programming languages still being used today in scientific computing, such as C and Fortran, were designed decades ago with a tight connection to the execution models of the hardware that were available then. The execution model - an abstraction of the way the hardware works - followed a flat Von Neumann machine: a sequence of instructions for the Arithmetic Logic Unit (ALU) with a flat addressable memory space. Code written using these programming models was trivially translated to the hardware's execution model and then into hardware instructions. This ensured that the factor most important to performance, the sequence of instructions, was exposed the programmer, while others could be handled by the compiler, such as register management, or ignored and left to the hardware, such as memory movement. Over time, hardware and execution models have changed, but mainstream programming models remain the same, resulting in a disparity between the user's way of thinking about programs and what the real hardware is capable and suited to do. While compilers do their best to hide these changes, decades of compiler research has shown that this is extremely difficult.

The obvious driving force between the changes to execution models is the ever increasing need for parallelism; due to the lack of single-core performance scaling, the departmental, smaller-scale HPC systems in a few years will consist of the same number of processing elements as the world's largest supercomputers now. Table 1.1 show examples of the amount of parallelism required to fully utilise some of today's Top500 [19] systems; heroic efforts are required to scale applications to such systems and still produce scientifically relevant output. While improvements in hardware are expected to address some of these challenges by the time a similar amount of parallelism is going to be required on smaller-scale systems, many will have to be addressed by software; most important of all is parallelism. At the lowest level, in hardware, marginal improvements to instruction scheduling are still being made by using techniques such as out-of-order execution but above that level, in software, parallelism has to be explicit. Despite decades of research no universal parallelising compiler was ever demonstrated, therefore it is the responsibility of the programmer to expose parallelism - in extreme cases up to hundreds of million ways.

Handling locality and coping with the orders of magnitude cost differences between computing and moving data across the different levels of the memory hierarchy is second fundamental programming challenge. This is one that is perhaps more against conventional programming methodologies, because it has been almost completely hidden from the programmer, but just like there is no universal parallelising compiler, it is also very difficult to automatically improve the locality of computational algorithms. Most of the die area on modern CPUs is dedicated to mitigating the cost of data movement when there is locality, but this is something that cannot be sustained: the fraction of memory capacity to computational capacity is constantly decreasing [20, 1], on-chip memory size per thread is still on the order of megabytes for CPUs, but only kilobytes of the Xeon Phi and a few hundred bytes for the GPU. Once again, relying on lower-level hardware features (such as cache) is not going to be sufficient to guarantee performance scaling; locality will have to be exposed to the programmer. A simple example of this during a flow simulation would be the computation of fluxes across faces for the entire mesh - streaming data in and out with little data reuse - followed by a step where fluxes are applied to flow variables - once again, streaming data in and out with little reuse; which is the way most simulations are organised. Instead, if computational steps were to be concatenated for a small number of grid points, that would increase data-reuse and locality. However, this programming style is highly non-conventional and introduces non-trivial data dependencies that need to be handled via some form of tiling [21, 22], resulting in potentially very complex code.

Finally, huge parallel processing capabilities and deep memory hierarchies will inherently result in load balancing issues; the third fundamental obstacle to programmability according to [23]. As parallelism increases, synchronisation, especially global barriers, become more and more expensive, statically determined workloads for processing units or threads are vulnerable to getting out of sync with each other due to slight imbalances in resource allocation or scheduling. Therefore considerable research has to be invested into algorithms that avoid global barriers and at the same time the performance of local synchronisation will have to be improved. Furthermore, to tolerate latency due to synchronisation and instruction issue stalls, more parallelism can be exploited - this is what GPU architectures already do; by keeping around several times more threads than processing cores it is possible to hide latency due to stalls in some threads by executing others. This can be combined with dynamic load balancing strategies to ensure the number of idle threads is kept to a minimum. Load balancing strategies may be in conflict with the effort of trying to minimise data movement; especially in architectures with deep memory hierarchies it may lead to cache invalidation and trashing. For example, operating systems will move threads between different cores to improve resource utilisation, but with memory-intensive applications this often results in performance loss because locality is lost - thread pinning is advantageous in many cases. The idea of moving computations to data is receiving increasing attention, but as with the other programmability issues, this is currently not exposed by most programming abstractions and languages, and is contrary to the way of thinking about algorithms.

Traditionally, most popular programming abstractions and languages focus on the sequence of computations to be executed in-order. However, due to the growing disparity between the cost of computations and data movement, computations are no longer a scarce resource, studies show the benefits redundant computations as a means to reduce data movement [24]. Therefore we are experiencing a paradigm shift in programming; factors that affect performance, therefore factors that should be exposed to the programmer are changing, concurrency, locality and load balancing are becoming more important, computations less so.

While there is a growing number of programming languages and extensions that aim to address these issues, at the same time it is increasingly more difficult to write scientific code that delivers high performance and is portable to current and future architectures, because often in-depth knowledge of architectures is required, and hardware-specific optimisations have to be applied. Therefore there is a push to raise the level of abstraction; describing *what* the program has to do instead of describing *how* exactly to do it, leaving the details to the implementation of the language. Ideally, such a language would deliver generality, productivity and performance, but of course, despite decades of research, no such language exists. Recently, research into Domain Specific Languages (DSLs) [16, 25, 26, 27] applied to different fields in scientific computations has shown that by sacrificing generality, performance and productivity can be achieved. A DSL defines an abstraction for a specific application domain and provides an API that can be used to describe computational problems at a higher level. Based on domain-specific knowledge it can for example re-organise computations to improve locality, break up the problem into smaller parts to improve load-balancing, and map execution to different hardware, applying architecture-specific optimisations. The key challenge is to define an abstraction specific enough so that these optimisations can be applied, but sufficiently general to support a broad set of

DOI:10.15774/PPKE.ITK.2014.003

applications.

1.3 Motivations for this research

In light of these developments, an application developer faces a difficult problem. Optimising an application for a target platform requires more and more low-level hardware-specific knowledge and the resulting code is increasingly difficult to maintain. Additionally, adapting to new hardware may require a major re-write, since optimisations and languages vary widely between different platforms. At the same time, there is considerable uncertainty about which platform to target: it is not clear which approach is likely to "win" in the long term. Application developers would like to benefit from the performance gains promised by these new systems, but are concerned about the software development costs involved. Furthermore, it can not be expected of domain scientists to gain platform-specific expertise in all the hardware they wish to use. This is especially the case with industrial applications that were developed many years ago, often incurring enormous costs not only for development and maintenance but also for validating and maintaining accurate scientific outputs from the application. Since these codes usually consist of tens or hundreds of thousands of lines of code, frequently porting them to new hardware is infeasible.

Therefore it is the responsibility of researchers in computer science and associated areas to alleviate this burden on domain scientists; to explore how scientific codes can achieve high performance on modern day's architectures and then try and generalise these experiences, in essence to provide a methodology to domain scientists that can be understood and directly used in their own research thereby permitting them to focus on the problem being solved.

A study from Berkeley [25] determined 13 "dwarfs"; groups of algorithms that are widely used and very important for science and engineering. Algorithms in the same group tend to have similar computational and data access patterns, therefore they can be often reasoned about together. Their definition is at a very high level and they encompass a wide range of applications, but the observation is that while programs and numerical methods may change, the underlying patterns persist; as they have done for the past few decades. Therefore they are ideal targets for research, these dwarfs include: dense and sparse linear algebra, spectral methods, N-body problems, structured and unstructured grids, Monte Carlo and others. I have chosen unstructured grids as a target for my research since it is poses interesting challenges due to its irregularity and data-driven behaviour; and at the same time it is a very important domain in engineering due to its flexibility in describing highly complex problems.

With this in mind, the first part of my research focuses on a specific field in unstructured grid algorithms; the Finite Element Method, exposing algorithmic transformations relevant to today's challenges of parallelism and data locality, and to explore and discuss the mapping of these algorithms to the massively parallel architecture of Graphical Processing Units.

As a natural generalisation of the first part of my research, I got involved in the OP2 domain-specific "active library" for unstructured grids [16, 2], to study the abstraction and the implementation of generic unstructured grid algorithms with two main objectives; (1) to investigate what high-level transformations are possible between the abstract definition of an algorithm and its actual execution on hardware, and (2) to study how to map the abstraction to execution on modern architectures that are equipped with deep memory hierarchies and support many levels of parallelism.

Much research [27, 28, 29, 30, 31, 32] has been carried out on such high-level abstraction methods targeting the development of scientific simulation software. However, there has been no conclusive evidence so far, for the applicability of active libraries or DSLs in developing full scale industrial production applications, particularly demonstrating the viability of the high-level abstraction approach and its benefits in terms of performance and developer productivity. Indeed, it has been the lack of such an example that has made these high level approaches confined to university research labs and not a mainstream HPC software development strategy. My research also addresses this open question by detailing our recent experiences in the development and acceleration of such an industrial application, Rolls-Royce Hydra, with the OP2 active library framework.

1.4 The structure of the dissertation

The structure of this dissertation is slightly different from that of the theses in order to better separate abstraction from implementation and to clearly describe the background behind the second part of my research. Figure 1.3 illustrates the structure of this work, note that for Thesis I., the discussion of I.1 and I.2 is split into two parts; an abstract algorithmic part, and a GPU implementation and performance investigation part. Also, while I.3 stems from the study of the Finite Element Method, it also addresses a domain closely related to unstructured grid algorithms; sparse linear algebra. Each chapter that discusses my scientific contributions ends with a summary, which states my theses.

I start by briefly discussing the hardware platforms and the associated parallel pro-

DOI:10.15774/PPKE.ITK.2014.003

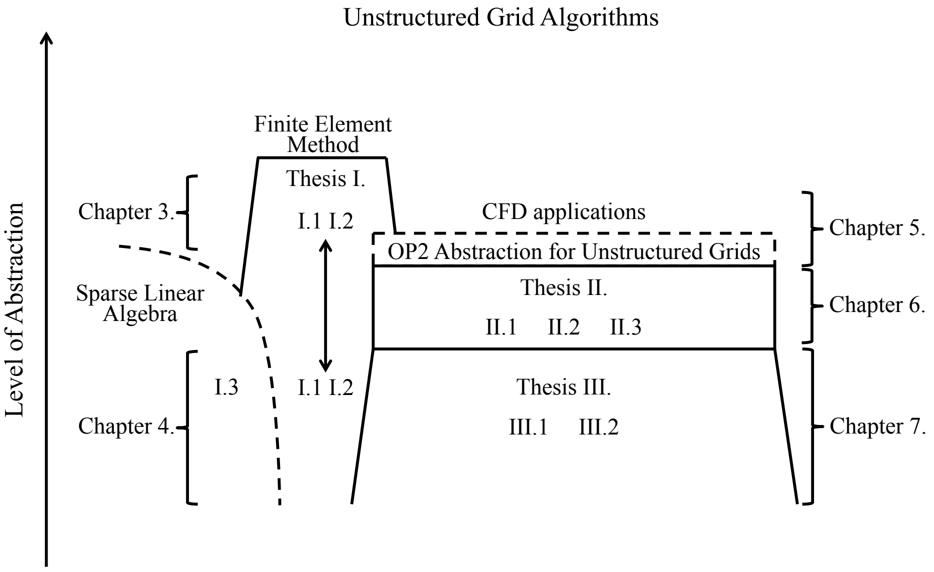


Figure 1.3. Structure of the dissertation

gramming abstractions in Chapter 2 that I use throughout this work. Chapter 3 describes the theoretical background of the Finite Element Method (Sections 3.1, 3.2) and then presents my research into algorithmic transformations (Section 3.2) and data structures (Section 3.3) related to the FEM at an abstract level, discussing matters of concurrency, parallelism and data movement. It also briefly describes the theoretical background of established sparse linear algebra algorithms that I utilise in the solution phase of the FEM (Section 3.4). Thus, Chapter 3 lays the abstract foundations for Thesis I.1 and I.2.

Chapter 4 takes these algorithms and data structures, and discusses my research into mapping them to GPU architectures. It begins with a description of GPU-specific implementation considerations that have to be taken into account when mapping algorithms to this massively parallel architecture (Section 4.1). Then, it follows the logic of the first chapter, describing the effect of algorithmic transformations on performance (Section 4.3) and then discussing how classical data structures (such as CSR and ELLPACK [33]) and the FEM-specific data structure I describe in Chapter 3 perform on the GPU, with various means of addressing data races and coordination of parallelism. This completes Thesis I.1 and I.2. Finally, I discuss the extension of my research to the intersection of unstructured grid algorithms and sparse linear algebra; novel mappings and parametrisations of the sparse matrix-vector product on GPUs are presented (Section 4.5), forming Thesis I.3.

Chapter 5 serves as an interlude between the discussion of my theses; it introduces the OP2 domain-specific library for unstructured grid computations and provides a background and setting to the second half of my research (Section 5.1). I have joined the work on this project in its second year, and while I have carried out research on a number of its components, I have only chosen subjects for Thesis groups II. and III. that are self-

DOI:10.15774/PPKE.ITK.2014.003

contained and pertinent to the over-arching theme of this dissertation. The chapter also discusses the computational fluid dynamics applications used to evaluate algorithms in later chapters; the Airfoil benchmark, the Volna tsunami simulation code (which I ported to OP2) and the Rolls-Royce Hydra industrial application (which my colleagues and I ported to OP2).

Chapter 6 presents my research into high-level transformations to computations defined through the OP2 abstraction, Thesis group II., starting with a novel and fully automated method of checkpointing that aims to improve resiliency and minimise the amount of data saved to disk (Section 6.1). Subsequently, an algorithm is presented that can improve locality through a technique called tiling [34], and presents my research into how dependencies across several parallel loops can be mapped out and how execution can be reorganised to still satisfy these dependencies but only work on parts of the entire problem domain that fit into fast on-chip memory (Section 6.2). Finally, I discuss the issue of hardware utilisation in modern day's heterogeneous systems and present my research into the execution of unstructured grid algorithms on such systems, modelling performance and discussing how this is facilitated in OP2 (Section 6.3).

Chapter 7 discusses my research into mapping the execution of unstructured grid algorithms to various hardware architectures, Thesis group III.; first to the Singe Instruction Multiple Thread (SIMT) model and GPUs (Section 7.2), second to classical CPU architectures and the Xeon Phi, using Simultaneous Multithreading (SMT) and Single Instruction Multiple Data (SIMD) (Section 7.3). The discussion on how execution scales on modern supercomputers with CPUs and GPUs is postponed to the Appendix, since the MPI distributed functionality of OP2 is not my contribution. This chapter serves to present my research into how an application written once through OP2 can be automatically mapped to a wide range of contrasting hardware, demonstrating that near-optimal performance can indeed be achieved at virtually no cost to the application programmer, delivering maintainability, portability and future-proofing.

Finally, Chapter 8 draws conclusions and Chapter 9 presents the summary of my theses.

DOI:10.15774/PPKE.ITK.2014.003

Chapter 2

Hardware Architectures Used

This chapter briefly presents the hardware architectures that I use throughout this work, and later on I will refer back to architectural specifics, execution models and best practices of programming for a given platform.

Today's hardware landscape is very diverse, ranging from classical CPUs, through low-power RISC architectures, general purpose GPUs, and many-core architectures, such as the Xeon Phi, to field programmable gate arrays. Each of these architectures has their own strengths and weaknesses, especially in terms of computational problems they are good at solving. Some of the hardware used in this work are outlined in Figure 2.1, clearly there are significant differences, therefore, when it comes to mapping unstructured mesh computations to them, different parallel programming approaches and optimisation techniques are required.

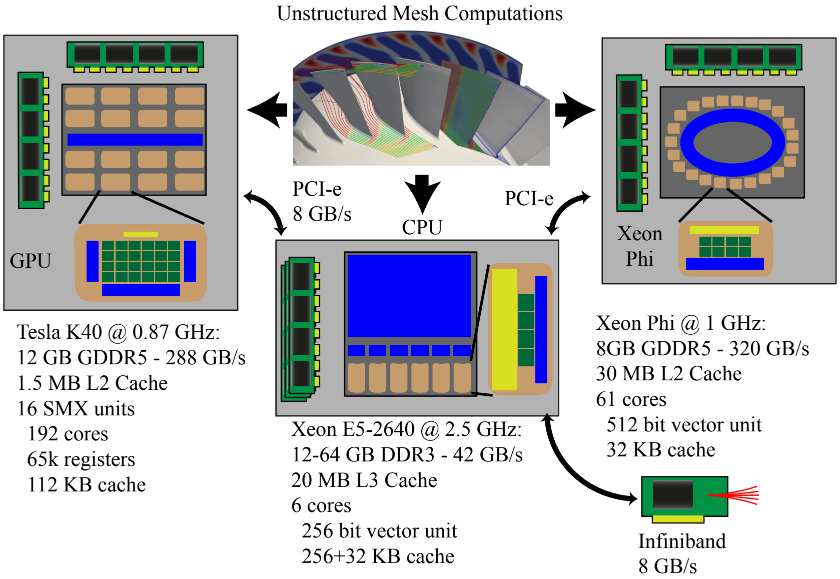


Figure 2.1. The challenge of mapping unstructured grid computations to various hardware architectures and supercomputers

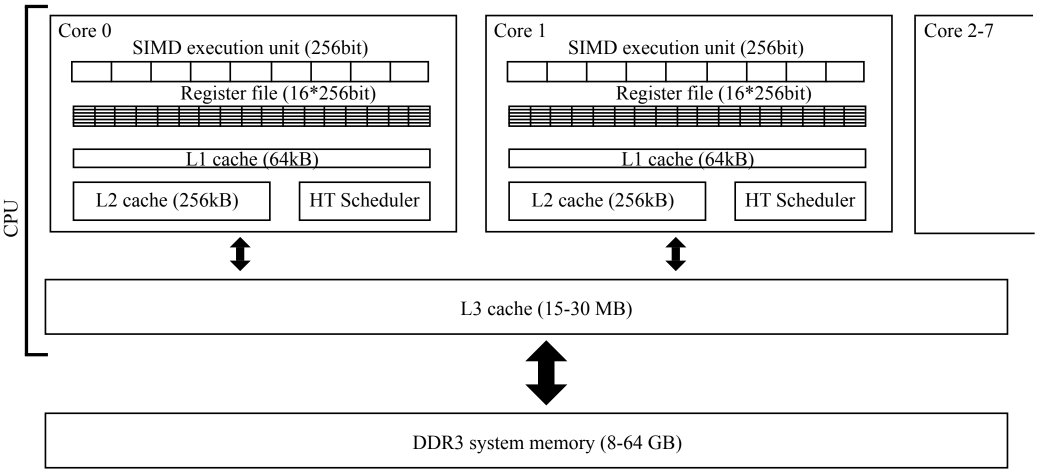

CPU
Core 0
SIMD execution unit (256bit)
Register file (16*256bit)
L1 cache (64kB)
L2 cache (256kB)
HT Scheduler
Core 1
SIMD execution unit (256bit)
Register file (16*256bit)
L1 cache (64kB)
L2 cache (256kB)
HT Scheduler
Core 2-7
L3 cache (15-30 MB)
DDR3 system memory (8-64 GB)

Figure 2.2. Simplified architectural layout of a Sandy Bridge generation server CPU

2.1 CPU architectures

CPUs have been constantly evolving for the past several decades, and they carry a burden of legacy; the need for backwards compatibility and the expectation that subsequent generations of hardware should deliver increasing single-thread performance and reduced latencies. With the end of frequency scaling this became much more difficult, as it has to be achieved through more complex execution logic and better caching mechanisms. Moore's law now pushes CPUs towards supporting increased parallelism; today, high-end CPUs feature up to 12 cores, each capable of supporting two hardware threads (without the need for expensive context switching), vector units that are up to 256-bit long (AVX), and caches up to 30 MB.

There are several excellent publications on CPU hardware and optimisations for achieving high performance, such as [35], here I discuss some of the specifics pertinent to my work. Figure 2.2 outlines a Sandy Bridge generation server CPU, note the low number of registers and the high amount of caches at different levels in the hierarchy. CPUs have cache coherency across cores, different levels of caches, and threads are allowed to be moved between different cores by the operating system, which is usually harmful to performance when carrying out data-intensive scientific computations, thus in practice communication and synchronisation between cores is considered expensive and within a core cheap, albeit the latter is implicit due to the SIMD nature of the hardware. Furthermore, in most servers there are multiple CPU sockets, which presents a NUMA (Non-Uniform Memory Access) issue [36]: when memory is accessed on one socket that is allocated to physical memory connected to the other socket, then the requests have to go through a QPI link between the two sockets and then to physical memory - this restricts the total bandwidth available, and increases latency. Unfortunately most Linux kernels allow threads moving between sockets, but in practice this can be prevented with tools like `numactl`, and it is common to use distributed memory programming to avoid this issue.

The most commonly used parallel programming abstractions are the communicating sequential processes (CSP) and the simultaneous multithreading (SMT); the former completely isolates threads effectively simulating a distributed memory environment but the latter relies on the fact that they share caches and the system memory. In the case of both abstractions, communication and synchronisation are regarded expensive although for SMT only the synchronisation is explicit. Finally, vector units use a Single Instruction Multiple Data (SIMD) model, where instructions are carried out on vectors instead of scalars. The most commonly used parallel programming libraries are MPI for CSP and OpenMP for SMT, however for SIMD programming there is no established software approach; compilers have built-in capabilities to automatically vectorise scalar code, such as simple `for` loop with no irregular accessed or loop-carried dependencies, there are language extensions such as Cilk that are more explicit about data parallelism, thereby providing more opportunities for the compiler to automatically vectorise computations, and there are low-level vector intrinsics that have a one-to-one matching with machine-level assembly instructions, but these are compiler and architecture-specific, therefore their portability is poor, and their use results in very verbose code. In the latest vector instruction sets there is support for masked instructions, where it is possible to facilitate divergent execution by activating and deactivating different vector lanes.

2.2 The Intel Xeon Phi

Intel's approach to high-throughput, massively parallel computing is the Xeon Phi; it integrates a large number of Intel CPU cores onto a single chip that are much simpler compared to server CPU cores, since they are in-order cores with very simple execution logic, but they support four hardware threads and 512-bit vector units. Each of these cores has 128 vector registers divided between the four threads, 32KB of data and 32KB of instruction cache, and the cores themselves are arranged on a ring-bus, with 512 KB of L2 cache each, giving a total of 30MB shared cache for the 61 cores of the 5110P model.

One of the main selling points of the Xeon Phi is that it uses a Linux-based kernel and supports the same compiler tools that are used for regular CPU programming; however this does not mean that code that was running well on the CPU will run well on the Phi too. Many parallel constructs that used to work well on multicore CPUs (such as critical

DOI:10.15774/PPKE.ITK.2014.003

sections) do not scale well to 240-way parallelism, and control-intensive applications with complex logic also struggle on the logically much simpler cores of the Phi. However, it does have a high-bandwidth GDDR5 memory, therefore trivially data-parallel compute- or bandwidth-intensive applications may see significant benefits.

In a similar way to programming CPUs, the Phi supports CSP via MPI libraries and SMT via various threading libraries, such as OpenMP or Threading Building Blocks (TBB) and SIMD through compiler auto-vectorisation and vector intrinsics. There are clearly many differences between CPUs and the Phi, and therefore the same parallelisation strategies may not work well on the Phi; while on the CPU it is feasible to start an MPI process for each logical core, on the Phi's 240 logical cores this rarely leads to good performance. There are two execution strategies when using the hardware: either native execution, where the whole of execution takes place on the Phi, or offload mode, where a code running on the host system (on the CPU) offloads computations and data to the Phi. The new OpenMP 4 standard introduced the syntax and the semantics for these offload-type computations, and they are also supported by other libraries, such as OpenCL.

2.3 NVIDIA GPUs

NVIDIA was the first to introduce general-purpose computing capabilities in its Graphical Processing Units and they remain the largest vendor of GPUs for computations. The GPU compute architecture has close ties with its graphical processing origins; they are designed to apply the same transformations to different data with very high throughput - in the past this was restricted to graphics-type computations, but with the introduction of the Compute Unified Device Architecture (CUDA) [37], it now supports arbitrary computations.

Figure 2.3 depicts the simplified architectural layout of a Kepler-generation GPUs [38] - the first thing to note is its structural resemblance to CPUs, but also the huge differences in terms of actual parameters. While a CPU core has a vector unit up to 256 bits wide (in some sense 8 logical threads doing the same single precision computations), a GPU SMX has support for 192 logical threads executing the same instruction (in groups of 32) at the same time. Furthermore, the size of the caches is much smaller, especially relative to the number of threads executing at the same time. This leads to the fundamental difference in how GPUs and CPUs operate and what they are designed to do well: CPUs are good at low-latency computations with a handful of threads due to their large caches and their complex execution logic, but GPUs rely on a massive amount of parallelism (tens

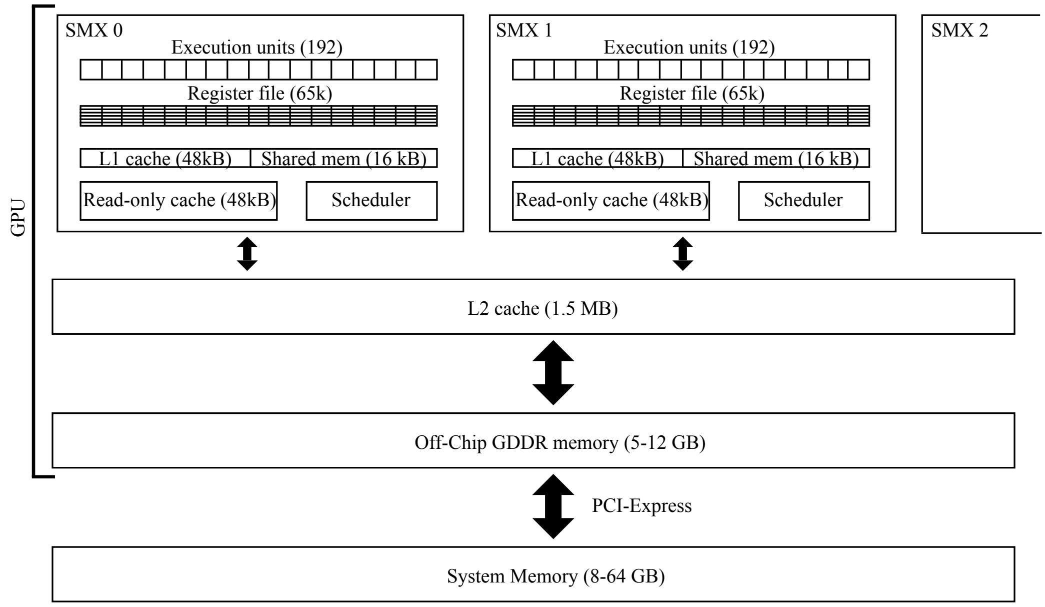


Figure 2.3. Simplified architectural layout of a Kepler-generation GPU

of thousands of threads) to hide memory latency; while some threads have outstanding memory requests, others can carry out their computations - this of course means that the latency in the execution of individual threads is comparatively worse, but the system-level throughput can be much higher than on CPUs.

NVIDIA has developed the CUDA ecosystem around their GPUs; this includes the CUDA language extensions to C, C++ and Fortran, based on the Single Instruction Multiple Threads parallel programming abstraction, as well as a number of numerical libraries to support computations common in scientific computing (such as FFTs or dense linear algebra routines). OpenCL [39] implements a similar programming abstraction and supports a much wider range of hardware, but its ecosystem is much smaller than that of CUDA. OpenACC [40] is a directive-based approach to programming GPUs (similar to OpenMP). In this work, I only utilise CUDA, as it permits the most low-level optimisation and the highest level of control over execution. Below, I present a summary of architectural specifics of Fermi and Kepler generation hardware and associated best practices that I use throughout this work. There are some architectural differences as well as more features supported in the later generation Kepler GPUs, NVIDIA's way of differencing between them is through the specification of the *compute capability*, which is 2.0 for Fermi, and 3.0 or 3.5 for Kepler. This was published in my paper [3] as a synthesis of various programming guides, examples and my own experience [37, 41, 42].

In the CUDA programming environment and in its SIMT abstraction there are two levels of parallelism; on the coarse level there is an execution grid, with a number of

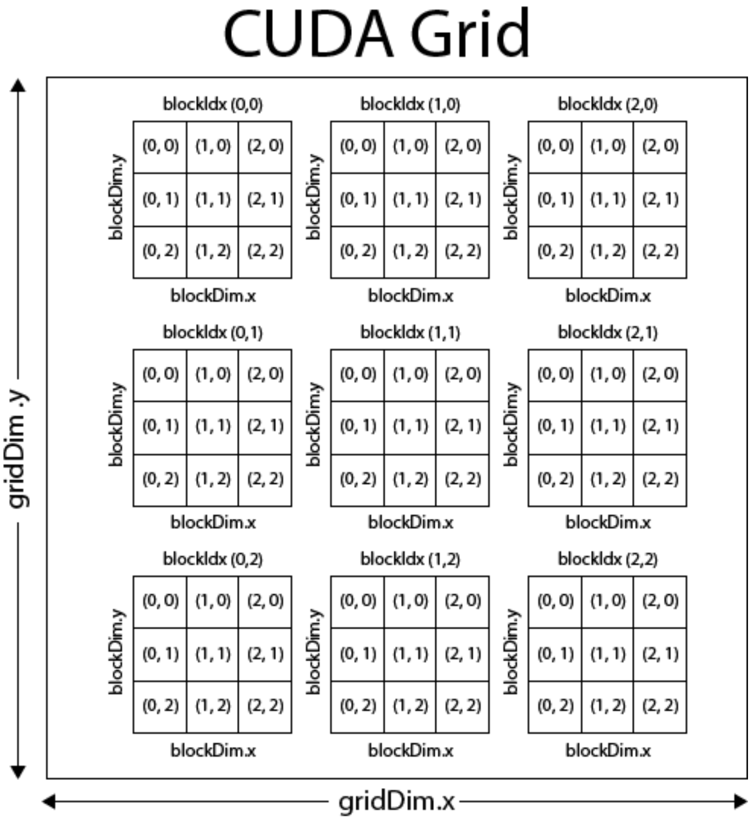


Figure 2.4. An example of a CUDA execution grid

thread blocks that execute independently with no guarantees as to how their execution is scheduled. On the fine level, each thread block consists of a number of threads, these execute simultaneously with access to a shared scratchpad memory and means for fast synchronisation. This execution model is illustrated in Figure 2.4.

When implementing algorithms in CUDA, the execution model of the hardware needs to be taken into account to apply optimisations, here I discuss the most important ones:

1. On NVIDIA GPUs, groups of 32 threads called warps are executed in a single instruction multiple data (SIMD) fashion. A collection of threads called a thread block is assigned to a Streaming Multiprocessor (SM or SMX); each SM can process up to 8(16) thread blocks at the same time, but no more than 1536(2048) threads on Fermi (Kepler) hardware. If there are too many thread blocks, then some execute only after others have finished.

2. The number of threads active at the same time (*occupancy*) depends on their resource usage: the number of registers and the amount of shared memory. Each Fermi SM has 32k 32-bit registers and each Kepler SMX has 65k, but each thread can only use up to 63(255). There is up to 48 KB of shared memory available on each SM.

3. The problem has to be decomposed into fairly independent tasks because collabora-

DOI:10.15774/PPKE.ITK.2014.003

tion is very limited. Threads in a thread block can communicate via on-chip shared memory. For threads that are not in the same thread block, communication is only possible indirectly via expensive operations through global memory, and even then synchronisation is only possible by starting a new kernel.

4. Memory bandwidth is very limited between the host and the GPU (PCI-Express) and has a high latency thus any unnecessary transfers should be avoided.

5. Memory bandwidth to off-chip graphics memory is highly dependent on the pattern of access and type of access. If threads in the same warp access memory locations adjacent to each other, than these memory transfers can be bundled together resulting in a so called *coalesced memory access* and the full width of the memory bus can be utilised. With the introduction of caching, coalesced access is no longer a strict requirement, however, for reasons described below, it is still desirable.

6. Implicit L1 and L2 caches were introduced with the Fermi architecture; each Streaming Multiprocessor (SM) has a 16k/48k L1 cache, and there is a single shared 768k L2 cache for the whole chip (1.5 MB for Kepler). The L1 cache size is comparable to the cache on the CPU (usually 32k), however while a CPU core executes only a few (1-2) threads, the GPU executes up to 2048 threads per SM. This results in very constrained cache size per thread: since a cache line (the unit of memory transaction when loading from/storing to off-chip memory) is 128 bytes long, even when using 48k L1 cache, only 384 lines can be stored. If all threads read or write in a non-coalesced way, only 384 of them can get cache hits when accessing that cache line, the others get a cache miss. Cache hits can greatly improve performance, but misses cause high latency and potentially cache trashing. It is therefore very important for threads in the same block to work on the same cache lines, to have so called "cache locality". On Kepler architectures there is also 48 KB of read-only non-coherent cache in each SMX that can be used for reading data.

7. Instruction throughput depends on both the number of threads per SM and the type of instructions: support for Fused Multiply-Add (FMA) enables high floating point throughput, however unstructured grids require a considerable amount of pointer arithmetic for which there are no separate integer units - unlike in CPUs.

8. Precision requirements are linked to all of the above as double precision floating point arithmetic requires more clock cycles for execution, and doubles the bandwidth requirements.

DOI:10.15774/PPKE.ITK.2014.003

Chapter 3

Finite Element Algorithms and Data Structures

In this chapter, I present the mathematical background of the Finite Element Method in Section 3.1, based on [43], to serve as a basis for the subsequent discussion of the corresponding integration algorithm in Section 3.2, where I also describe the algorithmic transformations at a high level, discussing issues of parallelism, concurrency and locality; these are going to be mapped to actual implementations in the next chapter. Section 3.3 presents the underlying data structures that are used by the parallel algorithms, with a focus on potential concurrency issues and the memory footprint of these structures. Finally, Section 3.4 discusses the algorithms involved in the sparse linear solution phase of the FEM, based on [44], adding my comments on matters of parallelism and concurrency, connecting the algorithms with the data structures.

3.1 The Finite Element Method

Unstructured meshes are used in many engineering applications as a basis for the discretised solution of PDEs, where the domain itself or the required accuracy of the solution is nonuniform. Variations of the Finite Element Method (FEM) can handle most types of partial differential equations, but to understand the algorithm, consider the following simple boundary value problem over the domain Ω [43]:

$$-\nabla \cdot (\kappa \nabla u) = f \text{ in } \Omega, \tag{3.1}$$

$$u = 0 \text{ on } \partial\Omega. \tag{3.2}$$

where ∇ is the gradient and κ may be a scalar of function and $f : \Omega \to \Re$. The solution

DOI:10.15774/PPKE.ITK.2014.003

is sought in the form of $u : \Omega \to \Re$. With the introduction of a test function v on Ω, the *variational form* of the PDE is as follows:

$$\text{Find } u \in V \text{ such that } \int\limits_{\Omega} \kappa \nabla u \cdot \nabla v \, dV = \int\limits_{\Omega} fv \, dV, \; \forall v \in V, \tag{3.3}$$

where V is the finite element space of functions which are zero on $\partial\Omega$, or as reformulated below:

$$u: \; a(u,v) = \ell(v), \; \forall v \in V, \tag{3.4}$$

$$a(u,v) = \int\limits_{\Omega} \kappa \nabla u \cdot \nabla v \, dV, \tag{3.5}$$

$$\ell(v) = \int\limits_{\Omega} fv \, dV. \tag{3.6}$$

The standard finite element method constructs a finite dimensional space $V_h \subset V$ of functions over Ω, and searches for an approximate solution $u_h \in V_h$. Let $\{\phi_{1 \dots N_v}\}$ be a basis for V_h, then:

$$u_h = \sum_i \bar{u}_i \phi_i \tag{3.7}$$

To find the best approximation to u, it is necessary to solve the system:

$$K\bar{u} = \bar{l}, \tag{3.8}$$

where K is the $n \times n$ matrix, usually called the *stiffness matrix*, defined by:

$$K_{ij} = a(\phi_i, \phi_j), \; \forall i,j = 1,2,\dots,N_v, \tag{3.9}$$

and $\bar{l} \in \Re^n$, usually called the *load vector*, is defined by:

$$\bar{l}_i = \ell(\phi_i), \; \forall i = 1,2,\dots,N_v. \tag{3.10}$$

If the underlying discretisation mesh has nodes $\bar{x}_i$, it is possible to choose a finite element space V_h with basis functions such that $\phi_i(\bar{x}_j) = \delta_{i,j}$. In this case u_h is determined by its values at $\bar{x}_i, \; i = 1,2,\dots N_v$. The mesh is a polygonal partitioning of the domain Ω into a set of disjoint elements $e_i \in E, \; i = 1 \dots N_e$. The basis functions are constructed so that ϕ_i is non-zero only over those elements e which have $\bar{x}_i$ as a vertex. This means that finite element basis functions ϕ_i have their support restricted to neighbouring elements, therefore the matrix K is going to be sparse. Basis functions are usually chosen to be piecewise polynomials, therefore in case of first-order elements, the basis functions are piecewise first-degree polynomials (in 2D this is going to be a bilinear), in case of second-order elements they are second-degree polynomials, etc.

In the case of an elasticity problem, the basis functions represent the unit displacement at different discretisation points (*nodes*), an entry K_{ij} of the stiffness matrix represents the force at node i due to the unit displacement at node j, keeping all the other nodes fixed. With elements of the load vector representing the external force, the solution of the linear system therefore yields the weights of basis functions - that is the absolute value of displacement at different discretisation points.

This formulation separates the FEM into two distinct parts: the assembly of the *stiffness matrix* K and the load vector $\bar{l}$, and the solution of a sparse linear system of equations $K\bar{u} = \bar{l}$, henceforth, they will be referred to as the *assembly* and *solution* phases.

3.2 Finite Element Assembly

In order to formulate a practical algorithm that could be later implemented, it is necessary to introduce some notations and data structures that describe the mesh; how elements and degrees of freedom (unconstrained discretisation points) are connected:

1. Global index for each node in the mesh $I = \{1 \dots N_v\}$.
2. Coordinate data for each node in the mesh $\bar{x}_i \in \Re^d$.
3. List of elements $E = \{1 \dots N_e\}$.
4. Element → node mapping M_e mapping from elements of E to a subset of I, where n is the number of *degrees of freedom - d.o.f.* in an element. This mapping is an ordered list of global node indices, for example in a counter-clockwise fashion.

Because the basis functions ϕ_i have their support restricted to neighbouring nodes, the bilinear form in (3.5) can be partitioned and constrained to be the sum of integrals over a few elements. In practice the stiffness matrix can be constructed by iterating through every element and adding up the contributions from integrals of non-zero basis functions $\phi_i,\ i \in M_e(e)$ over the current element Ω_e, as shown in Algorithm 1.

The resulting matrix K will be sparse because only neighbouring basis functions' products will be non-zero. The bilinear form $a(.,.)$ is symmetric, so K is symmetric too. This algorithm can be described in another way; as carrying out the integration and assembling dense *local matrices* K_e which contain the non-zeros that the current element contributes to the global matrix, then inserting these values to their place in the global matrix. Therefore one can further split the *assembly* phase into two logically distinct parts: *integration* and *insertion.*

Algorithm 1 Element by element assembly of the stiffness matrix and the load vector [43]

```
for each element $e \in E$ do
  for each degree of freedom $i \in M_e(e)$ do
    for each degree of freedom $j \in M_e(e)$ do
      $K_{ij} += \int_{\Omega_e} \kappa \nabla \phi_i \cdot \nabla \phi_j \, dV$
    end for
    $\bar{l}_i += \int_{\Omega_e} f \phi_i \, dV$
  end for
end for
```

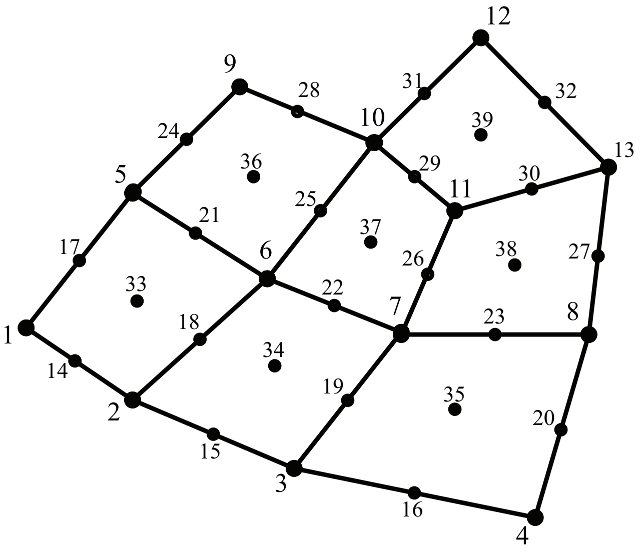


Figure 3.1. An example of a quadrilateral finite element mesh, with degrees of freedom defined for second-order elements

The algorithms in this work are based on quadrilateral elements - Figure 3.1 shows an example of a mesh composed of quadrilateral elements, with degrees of freedom defined such that (piecewise) second-degree polynomials will be used as basis functions. The calculation of the coefficients of the basis functions, the local quadrature points and the gradients are based on a transformation from the reference square. This bilinear transformation is calculated for every element and applied to the reference square [45].

3.2.1 Dirichlet boundary conditions

In general, essential boundary conditions constrain the solution u:

$$u = g \text{ on } \Gamma_{Dirichlet} \subset \partial\Omega, \tag{3.11}$$

for some function g on the constrained part of the boundary $\partial\Omega$. This means the nodes on the boundary have fixed values. This change of course has to appear in the linear system

$K\bar{u} = \bar{f}$. In my experiments I assume $g = 0$ and $\Gamma = \partial\Omega$ according to equation (3.2). There are two popular ways to implement this, either via pre- or post-processing:

1. Before assembly, renumber the nodes of the mesh in a way that constrained nodes are not included in the linear system, thereby eliminating them from further computations at the expense of having to look up different indices for accessing data on the nodes and accessing the linear system.

2. Assemble the stiffness matrix and load vector as if no nodes were constrained, then set each constrained node's row and column to zero, the diagonal element to one, and the entry in the right hand side vector to the prescribed value.

My test programs use the first approach, the non-constrained nodes are referred to as *free nodes* or *degrees of freedom* $I_f = \{1 \dots N_f\}$, $N_f \leq N_v$.

3.2.2 Algorithmic transformations

When increasing the degree of polynomials used as basis functions, both the number of degrees of freedom and the number of quadrature points increase as a square function of the degree. For 2D quadrilateral elements it is equal to: $(degree + 1)^2$. As Algorithm 1 shows, the number of computations also increases, and by accounting for the numerical integration, the total is on the order of $(degree+1)^3$. Since there is a huge growth with the increasing degree of polynomials, I investigate possible formulations of the FE integration.

The classical formulation of the integration is described in Algorithm 2 [43]. Here the outermost loop is over the quadrature points, computing the local quadrature point by mapping from the reference square and the corresponding Jacobian once. Then it iterates over pairs of degrees of freedom to compute the product of the gradients of the two basis functions, evaluated at the current quadrature point, and adding this contribution, multiplied by the quadrature weight, to the stiffness value. Observe, that this formulation carries out the minimum number of computations, and only stores the bilinear mapping locally, however it updates $K_{local}(i, j)$ repeatedly, once in each iteration of the outermost loop over the quadrature points. Therefore this algorithm has good computational efficiency, has a low local storage footprint, but has poor temporal locality in accessing the stiffness values.

My first transformation, the second formulation, of the integration is described by Algorithm 3. Here, the loop over quadrature points is now the innermost loop, therefore the stiffness values are updated with very good temporal locality, however a number of values are pre-computed and stored locally; the coordinates of local quadrature points and

Algorithm 2 Traditional formulation of FE integration [43]

```
$I_{local} \leftarrow M_e(e)$
generate bilinear *mapping* based on the four vertices
**for** each quadrature point $p$ **do**
  calculate local quadrature point for $p$
  calculate jacobian $J$ to map to $p$
  **for** $i = 1$ *to* $length(I_{local})$ **do**
    **if** $i$ is not constrained **then**
      **for** $j = i$ *to* $length(I_{local})$ **do**
        **if** $j$ is not constrained **then**
          $K_{local}(i,j) += weights(p) * \nabla\phi_i(p) * \nabla\phi_j(p)$
          $K_{local}(i,j) = K_{local}(j,i)$
        **end if**
      **end for**
      $\bar{l}_{local}(i) += weights(p) * \phi_i(p) * f(p)$
    **end if**
  **end for**
**end for**
```

the corresponding Jacobians. Since the number of quadrature points increases linearly with the degree of polynomials used, this can have large local storage requirements, especially at high degrees. Like the classical formulation, it also carries out the minimal number of computations.

Algorithm 3 Local storage formulation of FE integration

```
$I_{local} \leftarrow M_e(e)$
generate bilinear *mapping* based on the four vertices
pre-compute local quadrature points $quadPoints_{local}$
pre-compute Jacobians $J$ to map to each quadrature point
**for** $i = 1$ *to* $length(I_{local})$ **do**
  **if** $i$ is not constrained **then**
    **for** $j = i$ *to* $length(I_{local})$ **do**
      **if** $j$ is not constrained **then**
        **for** each quadrature point $p$ in $quadPoints_{local}$ **do**
          $K_{local}(i,j) += weights(p) * \nabla\phi_i(p) * \nabla\phi_j(p)$
        **end for**
        $K_{local}(i,j) = K_{local}(j,i)$
      **end if**
    **end for**
    **for** each quadrature point $p$ in $quadPoints_{local}$ **do**
      $\bar{l}_{local}(i) += weights(p) * \phi_i(p) * f(p)$
    **end for**
  **end if**
**end for**
```

The third formulation of the integrations is described by Algorithm 4, which is structurally similar to Algorithm 3, except that local quadrature points and Jacobians are re-computed in the innermost loop. Therefore this formulation achieves good temporal locality, it only stores the bilinear mapping locally, but performs redundant computations.

Algorithm 4 Redundant compute formulation of FE integration

```
$I_{local} \leftarrow M_e(e)$
generate bilinear *mapping* based on the four vertices
**for** $i = 1$ *to* $length(I_{local})$ **do**
  **if** $i$ is not constrained **then**
    **for** $j = i$ *to* $length(I_{local})$ **do**
      **if** $j$ is not constrained **then**
        **for** each quadrature point $p$ **do**
          calculate local quadrature point for $p$
          calculate Jacobian $J$ to map to $p$
          $K_{local}(i,j) += weights(p) * \nabla\phi_i(p) * \nabla\phi_j(p)$
        **end for**
        $K_{local}(i,j) = K_{local}(j,i)$
      **end if**
    **end for**
    **for** each quadrature point $p$ **do**
      calculate local quadrature point for $p$
      calculate Jacobian $J$ to map to $p$
      $\bar{l}_{local}(i) += weights(p) * \phi_i(p) * f(p)$
    **end for**
  **end if**
**end for**
```

These three formulations have a property highly relevant to properties of modern computer architectures; they trade off computations for local storage and temporal locality, Chapter 4 discusses how these map to hardware.

3.2.3 Parallelism and Concurrency

Algorithms 2, 3 and 4 describe the FE integration for individual elements that constitute the whole domain. Integration itself is "embarrassingly" parallel; there is no data dependency between the execution of different elements. However, during *insertion* there is a write-after-read data race when incrementing stiffness values in the sparse matrix; two elements with common degrees of freedom, such as elements sharing an edge or a vertex, may try to increment the same value. Therefore it is important to address these race conditions; either by carrying out the increment atomically, or by enforcing an ordering in the execution of potentially conflicting updates. The former may be supported by hardware, the latter has to be achieved through a modification to the parallel execution of elements; colouring.

Colouring is done on two levels: blocks of elements and individual elements within the blocks. This is to allow coarse and fined grained parallelism; in some cases it is advanta-

DOI:10.15774/PPKE.ITK.2014.003

geous to have only a single block of elements, but in other cases, for hardware that support multiple levels of parallelism, it is useful to have many. Only blocks of elements with the same colour are processed at the same time, thereby making sure that no two blocks access the same data at the same time. Within these blocks further parallelism can be supported by colouring individual elements, and then serialising accesses colour-by-colour, with an explicit synchronisation between each one. This two-level approach is preferred over a single colouring of all elements, due to the fact that in such a case all data reuse would be lost.

3.3 Finite Element Data Structures

There are three principal factors that have to be taken into account when designing data structures for modern parallel hardware:

1. Memory footprint: how much memory is required to store all the data and auxiliary information on how to access it.
2. Memory movement: how much data has to be transferred when carrying out an operation involving the data.
3. Locality: what degree of spatial and temporal locality is achievable when carrying out operations.

Given that this research is carried out in the context of the Finite Element Method, I investigate two distinctly different approaches; inserting stiffness values during assembly, immediately after integration, into a single, globally consistent stiffness matrix (referred to as the Global Matrix Approach (GMA)), or alternatively to store the dense, local stiffness matrices for each element and postpone insertion in some form to the solution phase (referred to as the Local Matrix Approach (LMA)). This section discusses the layout of input data for the assembly phase and the layout of the sparse matrix that is used as both the output of the assembly and the input of the solution phase.

Input mesh data consists of the mapping M_e, the node coordinates, the mapping from nodes to the indices of the linear system and the state variables for each node. Since the algorithm iterates through elements, only the mapping data M_e is accessed directly. The rest of the input data in the assembly phase is accessed in an indirect fashion, based on the global indices of the degrees of freedom.

The layout of stiffness matrix data is one of the most important factors that will determine the performance of the finite element method on different hardware: in the

	col 0	col 1	col 2	col 3	col 4	col 5	col 6	col 7
row 0	0.3	1.4	0	0	0	0.5	0	0
row 1	0	1.1	0	3.7	0	0	0.6	7.1
row 2	1.0	0.1	0	3.2	0	0	0	0
row 3	0	0	0	8.3	1.4	4.5	0	2.7

Figure 3.2. Dense representation of a sparse matrix

assembly phase the amount of data written to the stiffness matrix is at least comparable to the amount of input mesh data, but for higher order elements it is many times higher. Since in the solution phase the sparse-Matrix Vector (spMV) product is evaluated repeatedly when using sparse iterative solvers, the matrix data will be read multiple times and hence it is very important to optimise its structure.

3.3.1 Global Matrix Assembly (GMA)

There are several different sparse matrix storage formats commonly used [33], here I discuss their structure and give examples based on the dense representation shown in Figure 3.2;

1. COO (COOrdinate storage) stores every non-zero, their row, and column indices separately. These vectors are not necessarily ordered, so during a spMV this may result in random accesses to the multiplicand and the result vector and in the latter case write conflicts may occur. It is also difficult to populate such a matrix on a parallel architecture. Because of these disadvantages I did not use this storage scheme.

2. CSR (Compressed Sparse Row) stores the non-zeros in a row-major order. As shown in Figure 3.3, this format uses two arrays storing every non-zero and their column indices, and an additional array with pointers to the first non-zero of every row and the total number of non-zeros at the end. The size of the first two arrays are the same as the number of non-zeros in the matrix, the size of the third array equals the number of rows plus one. The advantage of CSR is that it is very compact and easy to read, however it is very difficult to populate in a dynamic way; preprocessing is necessary to determine the locations of the non-zeros beforehand. This also results in a lot more input data in the assembly phase.

3. ELLPACK [33, 46] stores the sparse matrix in two arrays, one for the values and one

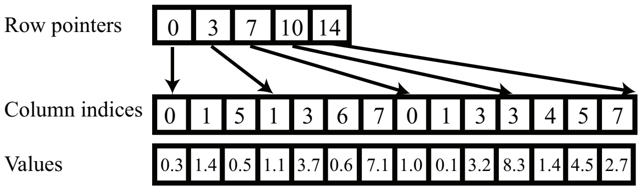
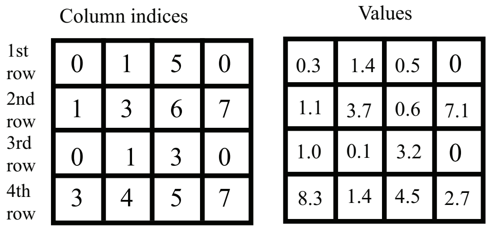


Figure 3.3. Memory layout of the Compressed Sparse Row (CSR) format on the GPU


Column indices
Values

	Column indices			
1st row	0	1	5	0
2nd row	1	3	6	7
3rd row	0	1	3	0
4th row	3	4	5	7

Values			
0.3	1.4	0.5	0
1.1	3.7	0.6	7.1
1.0	0.1	3.2	0
8.3	1.4	4.5	2.7

Figure 3.4. Memory layout of the ELLPACK format. Rows are padded with zeros

for the column indices. Both arrays are of size $number\ of\ rows * max\ row\ length$. Note that the size of all rows in these compressed arrays are the same because every row is padded as shown in Figure 3.4.

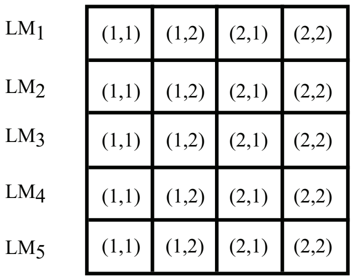
4. Hybrid ELLPACK [33] stores the elements up to a certain row length in the ELLPACK format and the rest in a COO format. This layout has smaller memory requirements than the ELLPACK, but it is more difficult to use because of the COO part. Due to the relatively well-known length of the rows in the stiffness matrix, I only use the ELLPACK format.

3.3.2 The Local Matrix Approach (LMA)

Many numerical methods to approximate the solution of the system of equations $K\bar{u} = \bar{l}$, such as the *conjugate gradient method* [44], do not explicitly require the matrix K during operations, such as the sparse matrix-vector multiplication. During the $\bar{y} = K\bar{x}$ multiplication the *local matrix approach* gathers the values of $\bar{x}$, performs the multiplication with the local matrices, and finally scatters the product values to $\bar{y}$ [47, 48]. Formally this can be described as follows:

$$\bar{y} = \mathcal{A}^T(K_e(\mathcal{A}\bar{x})), \tag{3.12}$$

where K_e is the matrix containing the local matrices in its diagonal and $\mathcal{A}$ is the local-to-global mapping from the local matrix indices to the global matrix indices. Formally this requires three sparse matrix-vector products, but the mapping matrix $\mathcal{A}$ does not have to be constructed, and the whole operation reduces to N_e dense matrix - vector products, the gathering of $\bar{x}$, and the scattering of products to $\bar{y}$ based on the element-node mapping



LM1	(1,1)	(1,2)	(2,1)	(2,2)
LM2	(1,1)	(1,2)	(2,1)	(2,2)
LM3	(1,1)	(1,2)	(2,1)	(2,2)
LM4	(1,1)	(1,2)	(2,1)	(2,2)
LM5	(1,1)	(1,2)	(2,1)	(2,2)

Figure 3.5. Memory layout of local matrices (LM) for first order elements. Stiffness values of each element are stored in a row vector with row-major ordering

M_e.

In the LMA approach I store the dense element matrices, which are of size $(\#d.o.f.\ per\ element)^2$. These matrices are stored in vectors, each one containing the local matrix in a row-major order as shown in Figure 3.5. These vectors are then stored row-by-row in memory.

3.3.3 The Matrix-Free Approach (MF)

The logic of the local matrix approach can be taken a step further; never storing local stiffness matrices, but recalculating them every time the matrix-vector product is performed. This approach can save a high amount of memory space and bandwidth by not moving stiffness data to and from memory. This method has a different data transfer versus computation ratio than the others, thus makes an interesting case when exploring performance bottlenecks of the finite element algorithm.

3.3.4 Estimating data transfer requirements

I have already given an overview of the trade-offs in computations and communications for assembly algorithms, but the exact ratios depend on the actual implementation. However, it is possible to give a good estimation for the data movement requirements in the assembly and the solution phases, therefore here I work out these values, these will be used later to predict and understand performance.

To provide an estimate of the amount of data to be transferred in both phases, consider a 2D mesh with quadrilateral elements where the average degree of vertices (in the graph theoretical sense - the average number of edges connected to vertices) is four, and the number of elements is N_e. Let p denote the degree of polynomials used as basis functions.

The total number of degrees of freedom is the sum of $d.o.f.$s on the vertices, the edges and inside the elements, not counting some on the boundaries, their number is approximately:

$$\begin{aligned} N_{vertex} &= N_e, \\ N_{edge} &= 2 * N_e(p-1), \\ N_{inner} &= N_e(p-1)^2. \end{aligned} \tag{3.13}$$

In the assembly phase every approach has to read the coordinates of the four vertices, the mapping M_e, write the elements of the stiffness matrix and the load vector. Additionally, global assembly approaches have to read indexing data to determine where to write stiffness values, which involves at least as many reads per element as there are values in the local stiffness matrices. Thus, for every element,

$$T_{assembly,LMA} = 2*4 + (p+1)^2 + (p+1)^4 + (p+1)^2, \tag{3.14}$$

$$T_{assembly,GMA} = T_{assembly,LMA} + (p+1)^4, \tag{3.15}$$

where T_{LMA}, T_{GMA} denote the units of data transferred per element for the local and the global matrix approaches. It is clear that the local matrix approach moves significantly less data than the global assembly approaches when p is large.

In the sparse matrix-vector multiplication the matrix values, row indices, column indices and the values of the multiplicand and result vectors have to be moved. The local matrix approach has to access all of the local stiffness matrices, the mapping M_e to index rows and columns and the corresponding values of the two vectors for every element. Thus, the amount of data moved is:

$$T_{spMV,LMA} = N_e(p+1)^4 + 3 * N_e(p+1)^2. \tag{3.16}$$

For global assembly approaches, the matrix is already assembled when performing the spMV, the length of any given row depends on whether the *d.o.f.* corresponding to that row was on a vertex, an edge, or inside an element:

$$\begin{aligned} L_{vertex} &= (2p+1)^2, \\ L_{edge} &= (2p+1)(p+1), \\ L_{inner} &= (p+1)^2, \end{aligned} \tag{3.17}$$

based on (3.13), the total number of stiffness values plus the column indices and the values of the multiplicand and result vectors:

$$\begin{aligned} T_{spMV,GMA} = 3 * (&N_{vertex} * L_{vertex} \\ &+ N_{edge} * L_{edge} + N_{inner} * L_{inner}) \\ &+ N_{vertex} + N_{edge} + N_{inner}. \end{aligned} \tag{3.18}$$

In addition to this, the CSR format has to read indexing data for rows as well, the size of which is $N_{vertex} + N_{edge} + N_{inner}$.

Table 3.1 shows the relative amount of data moved by global matrix approaches compared to the local matrix approach for different degree of polynomials used at single and

DOI:10.15774/PPKE.ITK.2014.003

Table 3.1. Ratio between data moved by local and global matrix approaches during the spMV in single and double precision.

Degree	1	2	3	4
ELLPACK/LMA single	1.0	1.81	2.25	2.49
CSR/LMA single	1.04	1.85	2.28	2.51
ELLPACK/LMA double	0.9	1.58	1.93	2.12
CSR/LMA double	0.92	1.6	1.95	2.13

double precision. Observe, that based on these calculations it would be always worth doing LMA except for first degree elements at double precision. However these figures do not take the effects of caching nor the atomics/colouring employed by LMA into account, so I expect the actual performance to be somewhat different. Similar calculations can be carried out for the three dimensional case as well, with the global matrix approaches moving less data in both precisions at first degree elements and significantly more at higher degrees.

3.4 Solution of the Sparse Linear System

While it is possible to solve sparse linear systems in the form of $Ku = l$ with direct methods, these are of $O(N^3)$ complexity, therefore only efficient on small matrices, and due to the fill-in they generate in sparse matrices they are highly non-trivial to implement. Iterative methods are hugely popular since they are only of $O(N^2)$ complexity, there is a lot of literature on subject [44]. Since I am interested in the type of operations that are carried out during the iterative solution, I chose the well-known preconditioned conjugate gradient iteration [44], described by Algorithm 5, the convergence of which is guaranteed for symmetric positive definite problems.. Note that every step in the algorithm consists of either operations between dense vectors, or a sparse matrix-vector multiplication (spMV). Parallelisation of the former is trivial, for the latter though, depending on the data structure and the parallelisation approach, it may be necessary to handle race conditions when updating the result vector. In practice, it is the spMV that is the most time-consuming step, and it is a key operation in most iterative solvers, therefore I investigate it in more detail.

When using global matrix assembly approaches, it is possible to parallelise over all non-zeros in the matrix, in which case race conditions between updates with the same row index have to be handled. Alternatively, one can parallelise over rows of the sparse matrix, in which case no race conditions exist - since most data structures store the matrix row by row, this is the most popular parallelisation approach. However, in a multi-level parallelism

Algorithm 5 The Preconditioned Conjugate Gradient Method for $Ku = l$ [44]

```
$r_0 = l - Ku_0$
$z_0 = M^{-1}r_0$
$p_0 = z_0$
for $k = 0$ until convergence do
    $\alpha_k = \frac{r_k^T z_k}{p_k^T K p_k}$
    $u_{k+1} = u_k + \alpha_k p_k$
    $r_{k+1} = r_k - \alpha_k K p_k$
    $z_{k+1} = M^{-1} r_{k+1}$
    $\beta_k = \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$
    $p_{k+1} = z_{k+1} + \beta_k p_k$
    $k = k + 1$
end for
```

setting, it may be worthwhile to map the coarse grained parallelism to independent chunks of work (e.g. rows) when communication and synchronisation are expensive, and use fine level parallelism over non-zeros in the same row, where communication and synchronisation are cheap. Since the sparsity pattern of matrices may be arbitrary, it is clear that there are trade-offs in the parallelisation approaches that have an important effect on the amount of parallelism and the locality. These are investigated in more detail in Chapter 4, where they are applied to GPU hardware, and evaluated on a number of matrices with different characteristics.

When using the local matrix approach, it is possible to parallelise over elements, degrees of freedom within elements or even pairs of degrees of freedom within elements, but it is always necessary to handle race conditions; there may be multiple stiffness entries in different elements for the same pair of degrees of freedom. However, while during assembly $(degree+1)^4$ race conditions exist for each element, when using the global matrix assembly approach, when using the local matrix assembly approach there are only $(degree + 1)^2$ race conditions during one iteration of the solve phase. Furthermore, locality is improved when parallelising over elements, since during the dense matrix multiplication between an element matrix and the multiplicand vector, $(degree + 1)^2$ values of the latter are reused $(degree + 1)^2$ times.

When solving ill-conditioned problems with a high condition number, one usually relies on preconditioners to ensure and accelerate convergence during the iterative solution phase. This consists of the insertion of a step in the iterative algorithm that performs the preconditioning:

$$z = M^{-1}r, \tag{3.19}$$

Algorithm 6 Coloured SSOR preconditioning [49]

```
Compute $z$ defined by:
$(L+D)D^{-1}(L+D)^T z = r$
Sub-step: solve $(L+D)y = r$ for y:
for colour $c = 1...ncolours$ do
    for each row $i$ with colour $c$ in parallel do
        $y_i = (1/D_i)(r_i - \sum_{columns\ j\ with\ colour<c} K(i,j)y_j)$
    end for
end for
Sub-step: solve $D^{-1}(L+D)^T z = y$ for z:
for colour $c = ncolours...1$ do
    for each row $i$ with colour $c$ in parallel do
        $z_i = y_i - (1/D_i) \sum_{columns\ j\ with\ colour>c} K(i,j)z_j)$
    end for
end for
```

where M is the preconditioning matrix. These preconditioners can range from simple diagonal (Jacobi), popular for its ease of use, through SSOR , Cholesky and LU factorisation, to the most powerful (and most expensive) multigrid-based preconditioners [44]. The Jacobi preconditioner is defined as follows:

$$M_{ij} = \begin{cases} K_{ij} \text{ if } i = j \\ 0 \text{ if } i \neq j \end{cases} \tag{3.20}$$

where K is the stiffness matrix, and the Symmetric Successive Over-Relaxation (SSOR) for symmetric matrices is defined as:

$$M = (L+D)D^{-1}(L+D)^T, \tag{3.21}$$

where L is the strictly lower triangular part of K and D is the diagonal. The Jacobi preconditioner is trivially parallel, however the SSOR enforces an ordering during the solution of the lower and the upper triangular systems, thus I use a red-black variant in conjunction with a full graph colouring algorithm that ensures that no two nodes that are connected via an edge are updated at the same time, and provides an ordering for the upper and lower triangular solves [49]. The algorithm is described in Algorithm 6.

These preconditioners are easily formulated for global matrix approaches, but for SSOR and the local matrix approach it is necessary to add an additional level of colouring in order to ensure race-free updates when parallelising over elements.

3.5 Summary and associated theses

In this chapter I presented a brief description of the Finite Element Method based on [43] and then moved on to present my research into high-level algorithmic transformations of the FE integration that trade off computations for local storage and memory movement [3]. Thus, I have laid the foundations of Thesis I.1: *By applying transformations to the FE integration that trade off computations for communications and local storage, I have designed and implemented new mappings to the GPU, and shown that the redundant compute approach delivers high performance, comparable to classical formulations for first order elements, furthermore, it scales better to higher order elements without loss in computational throughput.*

Furthermore I have discussed data structures relevant to the FEM, briefly introducing CSR and ELLPACK [33] and presented the Local Matrix Approach (LMA) discussing its properties in terms of memory footprint, memory movement and potential locality, and shown how algorithms involved in the solution of the sparse linear system can be adopted to use LMA [3]. This forms the basis of Thesis I.2: *I introduced a data structure for the FEM on the GPU, derived storage and communications requirements, shown its applicability to both the integration and the sparse iterative solution, and demonstrated superior performance due to improved locality.* At the same time, I have introduced the sparse linear algebra algorithms based on [43, 44] that I use; the conjugate gradient algorithm, the Jacobi, and the SSOR preconditioners.

This chapter presented an abstract study of the algorithms and data structures involved, not specific to any target architecture, investigating matters of parallelism, concurrency, data locality and data movement, these will serve as guidelines when carrying out the actual implementation and they will help understand performance.

DOI:10.15774/PPKE.ITK.2014.003

Chapter 4

The FEM on GPUs

In the previous chapter, I have introduced a number of algorithms and data structures; in this chapter I investigate how these map to the CUDA programming abstraction and the execution model of GPU hardware, and how they influence performance when applied to a Poisson problem. Section 4.1 describes the implementation considerations and the mapping to GPUs, and Section 4.2 presents the experimental setup. Section 4.4 presents and analyses the performance of different data structures and algorithms on the GPU. Finally, Section 4.5 describes my research into generic sparse matrix-vector multiplications on a single GPU and across multiple GPUs.

4.1 Implementation of algorithms for GPUs

When working on unstructured grids, the biggest problem is the gather-scatter type of memory access, and much depends on the ordering of elements and their vertices. In the assembly phase the vertex data (coordinates and state variables) have to be gathered for each degree of freedom in an element. This data is accessed indirectly via the M_e mapping, which usually results in an uncoalesced memory access. After assembling the local matrices, their elements have to be scattered to populate the global stiffness matrix. The latter operation poses a data hazard, as neighbouring elements have to write to overlapping segments of memory in the global matrix. This problem can be solved either by colouring the elements and executing the writes colour by colour [50], or by the use of atomic operations that are guaranteed to carry out the operation uninterrupted, thereby ensuring data-race-free execution - however these are not yet available for double precision variables. The problem can be avoided altogether by choosing either non-zeros or whole lines of the stiffness matrix as a unit of work to be assigned to threads [51]. These ap-

DOI:10.15774/PPKE.ITK.2014.003

proaches only involve gather type operations, however the amount of computations and data movement required is much higher since the algorithm has to iterate through all the elements connected to the given degree of freedom.

For the CUDA implementations Algorithms 2, 3 and 4 map directly to CUDA threads, the quadrature points, weights, basis functions and their gradients evaluated at the quadrature points on the reference triangle are kept in constant memory for fast access. The mapping M_e, the ELLPACK storage and the LMA storage of stiffness values are transposed on the GPU to support coalesced accesses when neighbouring threads access data associated with neighbouring elements. For the example of M_e, to access the n^{th} degree of freedom (out of N) in an element e (out of N_e), normally data would be accessed with the following index $e * N + n$, after transposition this becomes $n * N_e + e$, thus if adjacent threads are assigned adjacent e elements, they would be accessing subsequent memory locations when executing in lockstep, granting good spatial locality.

4.1.1 Related work

Due to the high demand for accelerating finite element methods (FEM) several studies have investigated FEM implementation on GPUs. The method can be divided into two phases: the assembly of a matrix, then the solution of a linear system using that matrix. In general the solution phase is the more time consuming one, specifically the sparse matrix-vector product between the assembled matrix and vectors used by iterative solvers. Because this is a more general problem, several studies focused on the performance evaluation of this operation on the GPU [33, 52, 46] and in the FEM context [53]. Finite element assembly has also been investigated both in special cases [52, 54, 55, 56] and in more general cases to show the alternative approaches to matrix assembly for more general problems [51, 47, 57, 58]. Their results show a speedup of 10 to 50 compared to single threaded CPU performance in the assembly phase, arguably an unfair comparison, and only very limited speedup in the iterative solution phase.

4.2 Experimental setup

4.2.1 Test problem

Since my goal was to investigate the relative performance of the finite element method using different approaches on different hardware I chose a simple Poisson-problem with a

DOI:10.15774/PPKE.ITK.2014.003

known solution:

$$-\Delta u(\bar{x}) = sin(\pi x_1) \cdot sin(\pi x_2), \tag{4.1}$$

$$u(\bar{x}) = 0 \; on \; \partial\Omega, \tag{4.2}$$

where $\Delta = \nabla \cdot \nabla$, is the Laplace operator.

The underlying two dimensional grid consists of quadrilateral elements with up to 16 million vertices and the order of elements ranges from 1 to 4. The tests run on grids that have the same number of degrees of freedom, so the number of elements in a fourth degree test case is one-sixteenth of the number of elements in a first degree test case.

4.2.2 Test hardware and software environment

The performance measurements were obtained on a workstation with two Intel Xeon X5650 6-core processors, clocked at 2.67GHz with 12Mb shared L3 cache, 256kB L2, and 32kB L1 cache per core, 24GBytes of system memory running Ubuntu 10.10 with Linux kernel 2.6.35. The system had 2 NVIDIA Tesla C2070 graphical processors installed, both with 6GB global memory clocked at 1.5GHz and 364-bit bus width, 448 CUDA cores in 14 streaming multiprocessors (SMs) clocked at 1.15GHz. The CPU codes were compiled with Intel's C Compiler 11.1, using SSE 4.2, Intel's OpenMP library and the *O2* optimisation flag. The GPU codes were compiled with NVIDIA's nvcc compiler with the CUDA 4.2 framework and the `-use_fast_math` flag.

For accurate timing of GPU computations I used CUDA Events as described in Section 8.1.2 of the CUDA C Best Practices Guide. CPU timings were obtained by using the `clock_gettime(CLOCK_MONOTONIC)` function in the standard linux *time.h* header, called outside of any OpenMP parallel region.

4.2.3 Test types

Several tests were performed that aim at investigating different aspects of the finite element algorithm. These tests are analysed from the viewpoint of different performance metrics such as speed and limiting factors, and also their place in the whole algorithm.

1. Stiffness matrix assembly: different approaches to assembly were tested, using the CSR and the ELLPACK sparse storage format, the LMA and the Matrix-free approach. Assembly approaches trading off computations for communications (discussed in Section 3.2.2) are evaluated, but only the best ones are shown.

DOI:10.15774/PPKE.ITK.2014.003

2. Conjugate gradient iteration: the spMV product is the most time-consuming part of the conjugate gradient method and the only one that depends on the matrix layout. Performance is displayed on the basis of number of Conjugate Gradient (CG) iterations per second. The performance of different storage and calculation schemes was analysed.

3. Data conflicts: race conditions can be handled in two ways: with the use of atomic operations or colouring. The former is straightforward and does not require any special implementation considerations. Optimal colouring on the other hand is an *NP-hard* problem, but suboptimal colouring algorithms are fast. The GPU algorithm uses two levels of colouring: thread and block colouring. Blocks with different colours are executed after each other by a kernel call. Memory writes for threads with different colours are separated by a $\_\_syncthreads()$ call. To decrease the number of synchronisations, these writes were buffered and grouped.

4. CPU and GPU: the algorithms are implemented as CPU codes using OpenMP threads executing Algorithm 2 by blocks of elements. In the GPU version one element is assigned to each thread.

5. Preconditioning: a simple Jacobi or an SSOR preconditioner are included in the conjugate gradient iteration, and their impact on performance is analysed.

4.2.4 CPU implementation

Comparing single-core CPU performance to the performance of GPUs with hundreds of processing cores is not realistic since modern systems have multiple CPU cores. To provide a fair comparison, I implemented the assembly and the solution phase using OpenMP. My test system included a 2 socket Intel Xeon X5650 processor, with a total of 12 physical cores, 24 with hyper-threading enabled.

The CPU implementations use a block-colouring approach to avoid race conditions; each thread works on a block of quadrilaterals and no two blocks with the same colour have neighbouring quadrilaterals. During assembly in these implementations I do not do any redundant computations since I follow the commonly used Algorithm 2, discussed in the previous chapter. To further optimise performance, threads were pinned to CPU cores through the `KMP_AFFINITY=compact` environment variable to avoid migration between sockets. To minimise the effects of non-uniform memory access (NUMA), memory is initialised from within OpenMP loops corresponding to further computational loops so

DOI:10.15774/PPKE.ITK.2014.003

as to make sure that memory most frequently used by different threads during the assembly and the solution process is allocated to physical memory attached to the appropriate CPU socket (this relies on the first touch page allocation policy of the operating system). Every CPU figure below shows performance using 24 threads.

The benefit from having more threads is clear in both phases, using 24 threads there is a 7 to 12 times speedup over a single thread during assembly, and a speedup between 5 and 8 during solve.

4.3 Performance of algorithmic transformations of the FE integration

As the GPU has significantly less resources per thread than the CPU, it is important to investigate the balance of computations and communications when it is possible to trade one for the other. As described in Section 3.2.2, I have implemented three different approaches, the first that interchanges the loop over quadrature points with the loop over pairs of degrees of freedom, performing the least amount of computations but resulting in increased memory traffic because the stiffness values, residing in global memory, are updated repeatedly (labeled as *Global reuse*, Algorithm 2), the second using local memory to precompute the coordinates of local quadrature points to save on computations in the innermost loop (labeled as *Local reuse*, Algorithm 3), and the third which recomputes these coordinates and hence it is the most computationally intensive (labeled as *Redundant compute*, Algorithm 4). Figure 4.1 shows that the first approach on the GPU is not a viable option, however for low degree polynomials the L1 cache can contain the register spillage resulting from the increased memory footprint of the second approach making it faster than the redundant compute version. For higher degree polynomials this is no longer true, the precomputed values are spilled to global memory resulting in a dramatic drop in instruction throughput and performance. The third approach scales very well with the increasing degree of polynomials, using the same amount of registers and showing a steady instruction throughput rate.

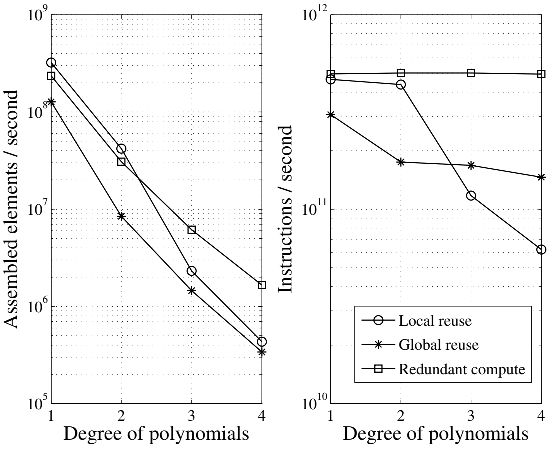


Figure 4.1. Number of elements assembled and instruction throughput using assembly strategies that trade off computations for communications. Values are stored in the LMA format

4.4 Performance with different data structures

4.4.1 The CSR layout

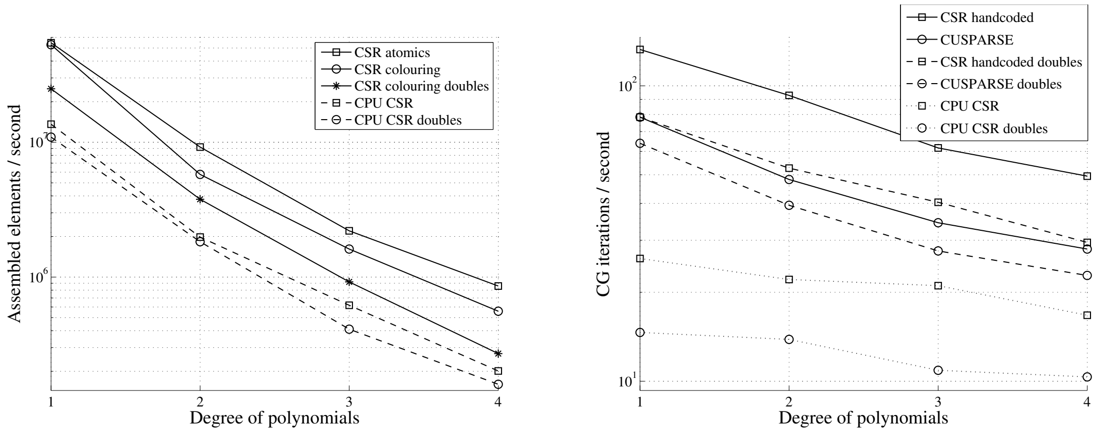
The compressed sparse row format (CSR) is one of the most widely used sparse storage formats, supported by several libraries such as NVIDIA's CUSPARSE [59]. In the case of the finite element method the matrix layout can be calculated from the mapping M_e and used in the assembly phase to find the location of a non-zero within its row. My tests use this approach but it is out of scope of this work to optimise this precomputing phase; in practice I do this on the CPU and use it as an input to the algorithm.

Assembly phase

In the assembly phase, the global memory address of K_{ij} has to be determined by accessing the row and column pointers for the current local matrix element, then writing the value while avoiding write conflicts either via atomic operations or colouring. Only the column pointer lookup can be coalesced due to the unstructured nature of the grid. The impact of this overhead decreases with the increasing computational requirements of higher order elements. Figure 4.2a shows a speedup of up to 4.5 times over the CPU.

Solution phase

There are two approaches to evaluate the matrix-vector product: assign one thread per row and add up the products with the multiplicand or to assign multiple threads per row thereby achieving coalesced memory access to the values in the same row. The partial sums



(a) Number of elements assembled and written to the global stiffness matrix

(b) Number of CG iterations per second on a 4 million row matrix. The spMV was performed by a hand-coded kernel and by CUSPARSE

Figure 4.2. Performance using the CSR storage format

in the latter case are added up by a binary-tree reduction algorithm [8, 33]. Figure 4.2b shows the performance of only the optimal versions of my CSR spMV kernel compared to CUSPARSE and CPU versions. By always evaluating and assigning the optimal number of threads to process the rows of the matrix, my algorithm, compared to CUSPARSE, shows a speedup of up to 2 times in single and 1.4 in double precision and between 3 to 5 times over the CPU. A heuristic decision algorithm for the number of threads assigned to process each row is discussed in Section 4.5, it was submitted to NVIDIA, and it is now included in CUSPARSE.

4.4.2 The ELLPACK layout

The ELLPACK storage format has gained popularity for use on the GPUs because of its aligned rows as shown in Figure 3.4. The population of the matrix can be done in a similar way to CSR - precomputing the matrix layout and writing to those memory locations in a thread-safe way. The other possibility is to allocate an empty matrix with an extra field for each row that stores the current length of that row. The maximum length of the rows can be upper-bounded by the number of degrees of freedom in each element and the maximum degree of vertices in the mesh. Then in the assembly phase each new value and its column index are appended to the end of the row. This of course requires more memory space allocated for the matrix, but when computing the local matrices row by row, each row can be written to global memory safely by increasing the row length only once with the number of new values to be written. After the assembly, these rows can be consolidated by sorting them by column indices and adding up values with the same index. This approach will result in the same matrix as if the layout was precomputed.

The other approach I call 'lazy ELLPACK' is to leave the rows as they are, which will still produce a valid result in the spMV phase, but the rows will be longer. The relative number of multiple entries for the same column index in a row decreases as the order of elements increases, making this overhead smaller.

Assembly phase

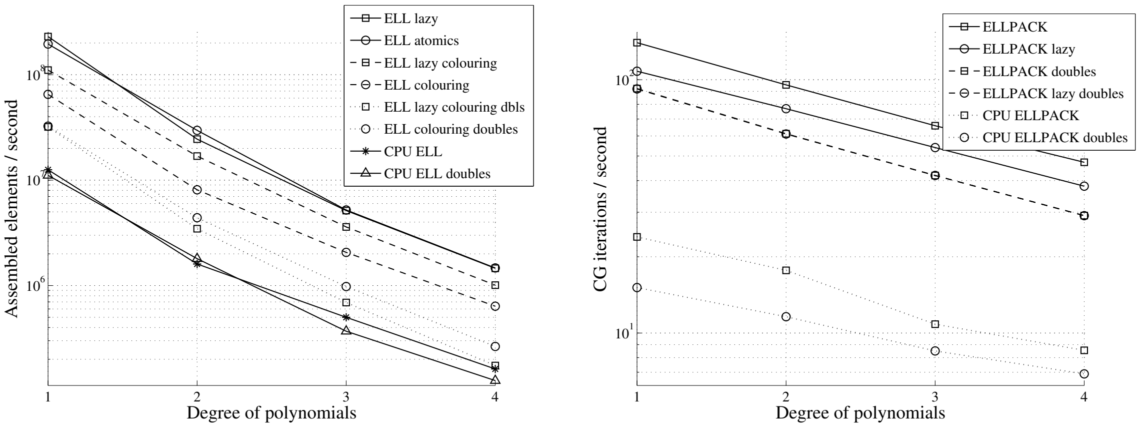
Computationally the assembly phase is exactly like the others, only the memory access patterns differ. In the lazy scheme less memory input is required as only one pointer (the row length) per local matrix row is necessary to determine where to write its values in global memory, unlike the precomputed version, where a pointer is required for each local matrix element - but this information can be laid out in memory so the access to it will be coalesced. When writing data to memory the lazy scheme has to store column indices for every value, in the precomputed version this is already known. Figure 4.3a shows that the two schemes perform almost the same when using atomics to avoid race conditions, but when using colouring the lazy version outperforms the precomputed one as it requires less synchronisation. The atomic and lazy approaches have a speedup of up to 20 times over the CPU in single precision and up to 3 times in double precision mode.

Solution phase

In my tests one thread is assigned to each row of the matrix, and since the matrix is transposed in GPU memory, these threads are reading the values and column indices of the matrix in a coalesced way - that is if the rows had the same length. Due to caching in Fermi GPUs, even then whole cache lines will be loaded resulting in excess use of bandwidth. The lazy scheme has similar issues, but the imbalance between the length of the rows is worse - when a degree of freedom belongs to more elements, its row in the stiffness matrix will have more values with the same column index. As shown in Figure 4.3b, the GPU is up to 6 times faster than the CPU in single and 5 times in double precision.

4.4.3 The LMA method

The local matrix assembly method is based on the observation that in the iterative solver, the stiffness matrix K is not required explicitly. This approach circumvents the sparse matrix issues that arise when dealing with unstructured grids by storing stiffness values on a local matrix basis. This enables completely coalesced access to stiffness values in both the assembly and the spMV phase - at the expense of postponing the handling



(a) Number of elements assembled and written to the global stiffness matrix. The lazy version does not write to predetermined memory locations, but appends every new value to the end of the row

(b) Number of CG iterations per second on a 4 million row matrix. The rows of the lazy version are not sorted and may have multiple values for the same column index

Figure 4.3. Performance using the ELLPACK storage format

of race conditions. Contributing stiffness values related to degrees of freedom that are on edges or vertices connecting elements are stored in the local matrices of those elements. This overhead decreases as the order of elements increases.

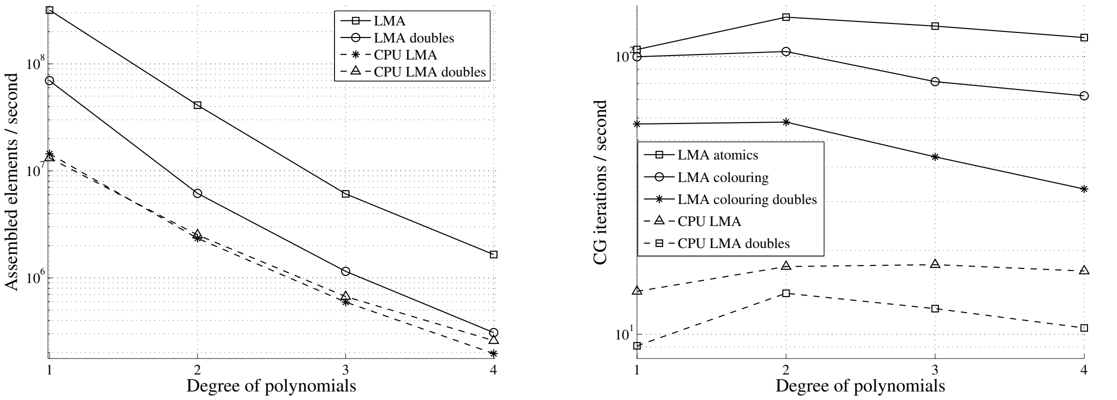
Assembly phase

The assembly phase requires the mapping from elements to node indices and the node coordinates to be read from global memory - the only part of the LMA method that cannot be coalesced, all writes to global memory are aligned by the thread index. Also an important aspect of LMA assembly is that there are no write conflicts as each thread writes to its own memory space. As shown in Figure 4.4a, the speedup of the GPU over the CPU is up to 20 times in single and 5 times in double precision.

Solution phase

The spMV computation is slightly different from the others as it is based on elements and not the rows of the matrix. Given these small dense local matrices, it is possible to efficiently exploit the symmetric nature of the stiffness matrix. In the implementation of the multiplication this would mean a nested loop with the bounds of the inner loop depending on the outer loop. However, the compiler is not able to create an efficient machine code from such a nested loop, so I unrolled the whole dense matrix-vector multiplication by hand. While CSR and ELLPACK spMV multiplications have to read a column index for each value in the stiffness matrix, the LMA method uses the mapping M_e to get row and

DOI:10.15774/PPKE.ITK.2014.003



(a) Number of elements assembled and written to the global memory

(b) Number of CG iterations per second on a grid with 4 million degrees of freedom

Figure 4.4. Performance using the LMA storage format

column indices. Thus, as I have shown in Section 3.3.4, at higher degrees the data moved by LMA is significantly less then the data moved by global matrix approaches.

In this scenario however, more threads can increment the same value of the product vector; thread safety is guaranteed by the use of atomics or colouring. Figure 4.4b clearly shows the penalty incurred by the synchronisation between colours. As the spMV phase is bandwidth limited, and memory accesses which may conflict (i.e. writing to the product vector) are probably not coalesced anyway, a separate memory transaction has to be issued for each atomic operation and each coloured write alike. This makes the overhead of synchronisation between colours a key limiting factor. The GPU achieves speedups of up to 8 times in single and 4 times in double precision over the CPU.

4.4.4 The Matrix-free method

The matrix free method is very similar to the LMA method, but instead of writing the local matrices to global memory, it uses them to perform the matrix-vector product, thereby fusing the assembly and the spMV phase. This of course means that the local matrices have to be recalculated every time the spMV multiplication is performed. The matrix-free approach has the advantage of not having to move the stiffness matrix to and from memory, which saves a lot on bandwidth. On the other hand, calculating local stiffness matrices is increasingly more computationally expensive with higher order elements. As a result the matrix-free approach can only be better than the other methods, if the computation of the local matrices is faster than moving them from global memory to the chip. Figure 4.5 shows a clearly steeper decline in performance compared to the spMV phase of other methods, which means that at higher degree elements the matrix-

DOI:10.15774/PPKE.ITK.2014.003

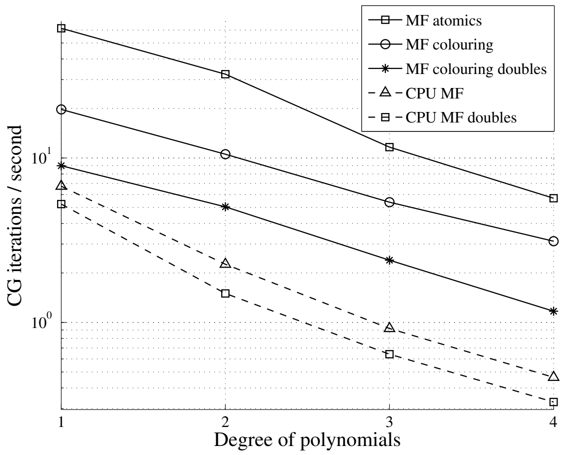


Figure 4.5. Number of CG iterations per second on a grid with 4 million degrees of freedom using the matrix-free method

free method is more compute limited than the other methods are bandwidth limited. On current hardware the gap between memory transactions and computations is not wide enough to support such a high amount of redundant computations.

4.4.5 Bottleneck analysis

When considering avenues to improve performance on different hardware, it is first important to determine whether they are compute or bandwidth limited. The NVIDIA Tesla C2070 has a theoretical maximum bandwidth of 144 GB/s, and a compute capacity of 1,03 TFLOPS for single and 515 GFLOPS for double precision calculations [60], although these numbers assume fully coalesced memory accesses and purely fused multiply-add floating point operations, which have a throughput of one instruction per clock cycle, but are counted as two operations. Applications rarely achieve more than two-thirds of the theoretical peak performance. Unstructured grids deal with scattered data access and a high amount of pointer arithmetics, both of which make the utilisation of GPU resources difficult. The introduction of L1 and L2 implicit caches with Fermi helps achieve better memory bandwidth, but in some cases it can degrade performance: cache trashing can result in a high fraction of the moved data being unused. As a comparison, the Intel Xeon X5650 system has a theoretical performance of around 120 GFLOPS when SSE vectorisation is fully utilised, which amounts to 10 GFLOPS per core.

On the computation side, there are certain operations that are a natural fit for the GPU and provide high throughput, such as floating point multiply and add (32 operations per clock cycle per multiprocessor). Some integer operations are more expensive: 32 operations per cycle for addition but only 16 for multiplication and compare. The most expensive operations are floating point division and special functions such as square root, sine, cosine, and exp, which have a throughput of 4 operations per cycle per multiprocessor.

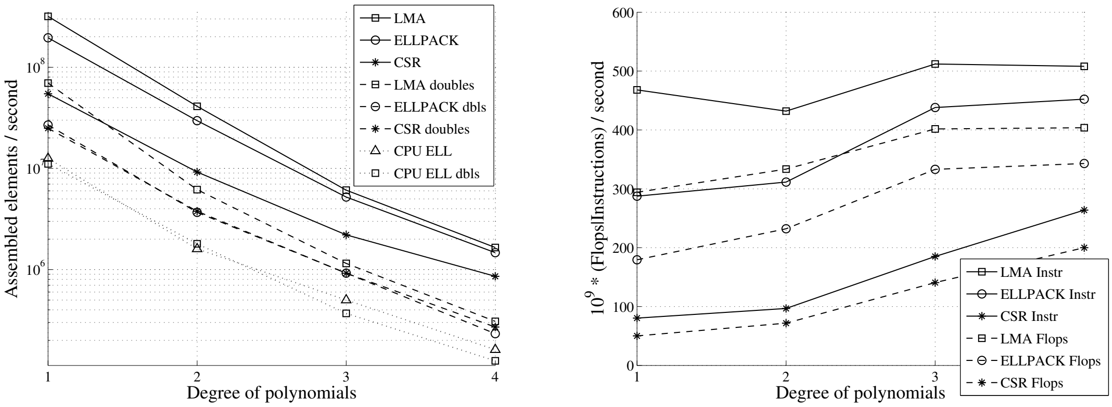
From a single thread perspective the algorithm is a sequence of memory operations and compute operations. To utilise the maximum amount of bandwidth there have to be enough threads and enough compute operations so that while some threads are waiting for data from memory, others can execute compute instructions. In theory, for the Tesla C2070 to operate at maximum efficiency from both the memory and compute perspective there have to be 28 floating point multiply-add operations for every single precision floating number loaded from global memory. In practice because of control overhead, other kinds of operations, and caching this number is significantly less (around 10).

The NVIDIA Visual Profiler gives very useful hints to decide whether a GPU kernel is compute or memory limited, but it also displays metrics such as ratio of divergence, cache statistics etc. which can be very helpful when analysing non-trivial aspects of an algorithm.

The assembly phase

Both the assembly and the spMV phases have to move the entire stiffness matrix to or from memory, and this transfer makes up the bulk of all memory traffic in both phases. Looking at the performance degradation in Figure 4.6a, as the order of elements increase, it is apparent that the assembly becomes much slower than the spMV. This means that while increasing the order of the elements results in having to move more data to and from memory, it also requires more computation in the assembly phase because the number of quadrature points also increases as a square function (in 2D) of the degree of polynomials used. Figure 4.6b shows a slight increase in instruction throughput for the LMA and ELLPACK approaches, and Figure 4.7a shows quickly decreasing bandwidth in the assembly phase. These factors indicate that the assembly phase is compute limited. CSR throughput figures on the other hand show an increasing tendency, while its bandwidth utilisation is the same as that of the other two approaches. The reason for this is the high percentage of cache misses; while threads writing to LMA and ELLPACK data layouts work on a small set of cache lines at the same time, thanks to their transposed data layout, threads writing to the CSR layout do not.

Based on these observations, it can be stated that the assembly kernel is increasingly compute limited with higher order elements. According to the Visual Profiler's output and my own calculations the instruction throughput of the LMA approach is around $500 * 10^9$



(a) Number of elements assembled and written to global memory

(b) Number of general and floating point instructions executed in the assembly phase

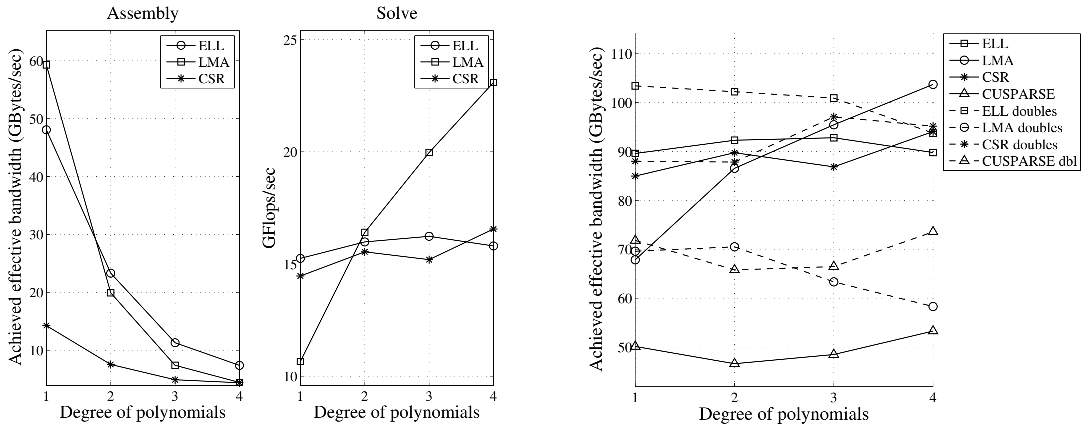
Figure 4.6. Throughput of the FEM assembly when using different storage formats on a grid with 4 million degrees of freedom

instructions per second, which is a good proportion of theoretical 1TFLOPS throughput considering the amount of integer arithmetic and control overhead - as a comparison, the CUBLAS [61] dense matrix-matrix multiplication benchmark reports 630 GFLOPS in single and 303 GFLOPS in double precision. It is also interesting to see the performance penalty for colouring (2 to 4 times) - that is the cost of synchronisation within thread blocks and the multiple kernel calls for different block colours.

When moving to double precision, it can be seen in Figure 4.6a that the LMA assembly performance is only a fourth of its single precision version. Double precision ELLPACK and CSR have to use colouring due to the lack of native support for atomic operations with doubles. In the case of the CPU, the compute limits are again apparent, even though the CPU versions have to perform less computations at the expense of having to store local quadrature points and their Jacobians. Regardless of the memory access pattern, all versions perform similarly.

Using LMA, the GPU's speedup over the CPU is between a factor of 10 and 30 in single precision using atomics, and between 2.5 and 7 in double precision. The speed difference in the actual calculations and in the theoretical performance of the GPU versus the CPU are very close.

The number of assembled quadrilateral elements per second outperforms triangular assembly shown in [51, 47] by a factor of up to 10.



(a) Achieved effective bandwidth in the assembly phase and number of floating point instructions in the spMV phase by different approaches

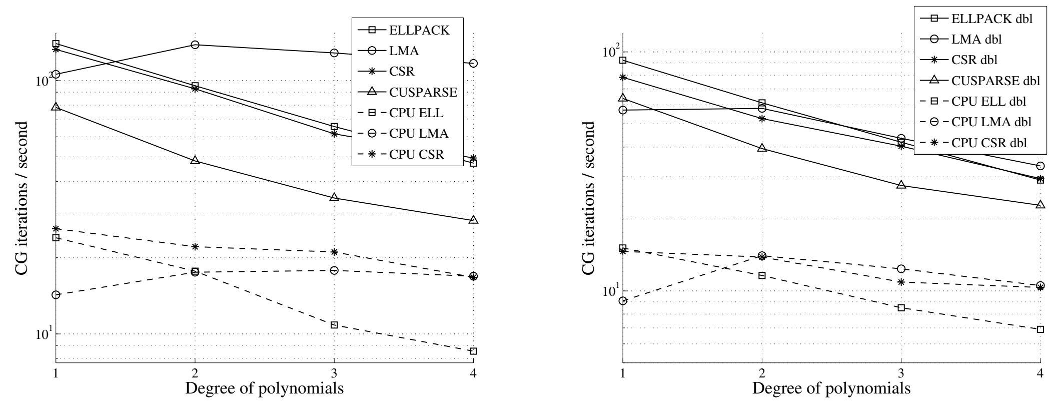
(b) Achieved effective bandwidth in the spMV phase by different approaches

Figure 4.7. Absolute performance metrics during the assembly and solve phases on a grid with 4 million degrees of freedom

The spMV phase

The sparse matrix-vector product is commonly known to be a bandwidth limited operation [33]. In fact it has to move just a little less data than the assembly phase, but the number of operations is significantly less: approximately one fused multiply-add for each non-zero element in the matrix. Figure 4.7a clearly shows a low instruction throughput compared to the theoretical maximum. Using the LMA approach incurs an overhead of having to avoid race conditions using atomics or colouring. The LMA approach offers fully coalesced access to the elements of the stiffness matrix, and both ELLPACK and CSR use optimisations to improve bandwidth efficiency. However, the access to the elements of the multiplicand vector is not coalesced, but caching can improve performance. Global matrix approaches using CSR store the least amount of stiffness data, but they have to read a column index for every non-zero of the sparse matrix. The LMA approach uses the mapping M_e to get row and column indices. As shown in Section 3.3.4 the local matrix approach has to move significantly less data, especially at higher degree polynomials.

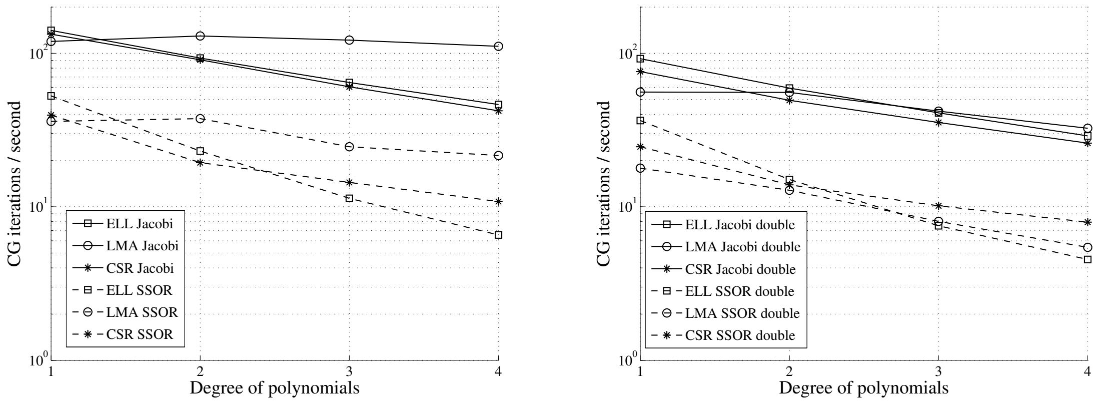
Figure 4.8a shows that global matrix approaches have very similar performance, but the less data-intensive local matrix approach provides the best performance for higher order elements. As expected, the difference increases with the increasing degree of elements. This also supports the conclusion that the sparse matrix-vector product is bandwidth limited. Furthermore Figure 4.7b shows the bandwidth utilisation of different approaches. The ELLPACK layout shows the best bandwidth utilisation, but since it has to move more data it is still up to 50% slower than the LMA layout. Although using either the CSR or the ELLPACK layout results in having to move the same amount of useful data, the



(a) Single precision

(b) Double precision

Figure 4.8. Number of CG iterations per second with different storage formats



(a) Single precision

(b) Double precision

Figure 4.9. Number of CG iterations per second with different preconditioners and storage formats

transposed layout of ELLPACK provides up to 10% higher effective bandwidth. The zeros padding the rows of ELLPACK are not factored into these figures. Figure 4.8b shows that in double precision, the performance of the LMA approach falls back because colouring has to be used to avoid race conditions.

The GPU's speedup over the CPU using global matrix approaches is up to 5 in single precision and 3 in double precision. Local matrix approaches outperform the CPU by up to 7 times in single and 5 in double precision.

4.4.6 Preconditioning

The diagonal preconditioning adds very little overhead to the conjugate gradient iteration; once the diagonal values are extracted from the stiffness matrix, it is a matter of a simple element-wise vector-vector multiplication, and so it has virtually no impact (less than 10%) on performance when compared to performance shown in Figures 4.8a and 4.8b. The coloured SSOR on the other hand involves several kernel launches, for each colour, in both the forward and the backward solution steps. In theory, the amount of useful computations carried out during preconditioning is almost the same as during a single sparse-matrix vector product. Performance figures are shown in Figures 4.9a and 4.9b. It can be observed that even for global matrix assembly approaches, the performance penalty is much higher than two times, which is due to the coloured execution: no two degrees of freedom in the same element can have the same colour, resulting in at least $(degree + 1)^4$ colours.

The CSR format is least affected by the coloured execution, because multiple threads are assigned to compute the product between any given row of the matrix and the multiplicand vector, which ensures some level of coalescing in memory accesses. ELLPACK on the other hand stores data in a transposed layout that gives good memory access patterns during an ordinary spMV, however during coloured execution neighbouring rows do not have the same colour thus neighbouring threads processing rows of the same colour are accessing rows that are far apart, resulting in very poor memory access patterns due to big strides and low cache line utilisation. Because of these issues, the performance of the ELLPACK layout on the SSOR falls rapidly with the increasing degree of polynomials used. When executing SSOR using the LMA layout, threads are still processing neighbouring elements, however there is no opportunity of data reuse within an element because by definition all its degrees of freedom have a different colour. Still, LMA's data layout enables more efficient memory access patterns, granting it an edge over global assembly methods in single precision, but the requirement for a two-level colouring approach in double precision reduces its performance below that of CSR.

Even though ELLPACK is very well suited for simple operations such as the spMV, it does not scale well to larger degree polynomials. LMA performs well on simple operations and single precision, but the lack of support for atomics in double precision and the complexity of the implementation for preconditioners like SSOR do not make it the obvious best choice. Furthermore LMA restricts the choice of preconditioners: for example an incomplete LU factorisation with fill-in cannot be applied, because the layout cannot accommodate the additional non-zeros. However, a geometric or p-Multigrid scheme [62] could be a good choice in combination with a Jacobi smother; the LMA format has the geometric information readily available and an element structure can be maintained on the coarser levels with appropriate restriction schemes. The implementation of such a multigrid preconditioner is out of scope of this work and will be addressed by future research.

DOI:10.15774/PPKE.ITK.2014.003

4.5 General sparse matrix-vector multiplication on GPUs

The sparse matrix-vector multiplication is an integral part of a host of sparse linear algebra algorithms, in many cases it is the single most time-consuming operation in an iterative linear solver [44]. Therefore it is an ideal target for performance optimisation, however, sparsity patters very greatly from one matrix to an other, since they can represent random graphs, electronic circuits, social interactions, discretisations of partial differential equations on structured or unstructured grids and a range of other problems. In this part of my research, I study the parametrisation of the sparse matrix-vector product operation in order to determine the best possible mapping to the GPU hardware, depending on properties of different matrices. Previous studies have addressed some of these issues but they lack the dynamism to adapt execution parameters to different matrices [63, 33, 64, 65, 66]

My main focus is to minimise data transfer by maximising data reuse. When performing the sparse matrix-vector product, each non-zero element of the matrix is only used once, and their column and row indices are used to address the multiplicand and result vectors. Depending on the storage format these indices can be compressed to an extent. More importantly, the access pattern to the multiplicand vector depends on the structure of the matrix and, in the general case, very few assumptions can be made about it. This is why caching mechanisms can greatly improve performance - or in some extreme cases decrease it. Due to the relatively small amount of cache per thread on the GPU, it is essential to understand and optimise for these caching mechanisms.

In general, sparse matrices can greatly differ in the number of rows and columns, in the average and variance of the length of the rows, and in the distribution of non-zeros within individual rows. Therefore there is no “best” storage format or fixed algorithm for SpMV multiplication. CSR has widespread usage in the scientific community and it is the most commonly used format in CPU based algorithms. It is one of the most efficient formats in terms of storing non-zero values because no padding are redundant storage is required. Blocked matrix formats store small parts of the matrix in a dense format, thereby cutting back on the amount of indexing data required. However, their use is limited to the blocked subclass of sparse matrices, otherwise the fraction of zeros in the dense sub-matrices would make the storage inefficient. Aligned formats like ELLPACK which are better suited for data access in vector architectures do not require the explicit storage of row indices, because all rows have the same length. This results in having to pad shorter rows with zeros up to the size of the longest row, which means that the memory required

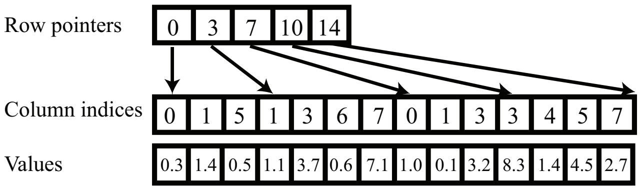


Figure 4.10. Memory layout of the Compressed Sparse Row (CSR) format on the GPU

for storing both the column indices and non-zero values is higher than absolutely necessary. In case of matrices with a high standard deviation in the length of the rows this would be infeasible; to tackle this problem the hybrid format stores rows up to a certain length in the ELLPACK format, and the rest of the non-zeros are stored along with their row and column indices in the coordinate (COO) format. However, it is difficult to handle matrices in the hybrid format, especially when trying to access structural properties of the matrix. Because of these compromises and the fact that only CSR has widespread support by other libraries I chose the CSR format for this research.

Many scientific algorithms require repeated execution of sparse matrix-vector products using either the same matrix or matrices with a very similar structure. A well-known example is the iterative solution of a system of linear equations where the number of equations make the use of direct solvers infeasible. Similarly, during Newton-Raphson iterations, a matrix is constructed, then a linear system is solved, but the matrix itself has the same structure in each iteration, just different values. In these cases, a sub-optimal multiplication algorithm can be tuned, and I will show that the overall performance can be greatly improved.

During the multiplication of a sparse matrix with a vector, each element of a given row has to be multiplied with the corresponding element from the multiplicand vector, these products are added up and written to the result vector. The CSR format stores matrix data in three vectors as shown in Figure 4.10 (a copy of Figure 3.3). The first vector `rowPtrs` points to the first element of each row, and its last element is the total number of non-zeros. The second and third vectors, `colIdxs` and `values`, store the column indices and the values for each non-zero element in the matrix. The total number of rows is `dimRow` and the total number of non-zeros is `nnz`. The simplest serial version of the multiplication $y = Ax$ using the CSR format is described by Algorithm 7 [33].

Based on Algorithm 7 a multiplication and addition is performed for each non-zero in the matrix, thus the total number of floating-point operations is:

$$2 * \texttt{nnz}. \tag{4.3}$$

DOI:10.15774/PPKE.ITK.2014.003

Algorithm 7 Sparse matrix-vector multiplication using CSR storage [33]

```
for i = 0 to dimRow-1 do
  $row = rowPtrs[i]$
  $y[i] = 0$
  for j = 0 to rowPtrs[i+1]-row-1 do
    $y[i] += values[row + j]*x[colIdxs[row + j]]$
  end for
end for
```

This formula will be used to calculate the instruction throughput.

Similarly, the number of bytes moved to and from off-chip memory without considering caching, is for each non-zero its value, column index and corresponding value from the multiplicand vector x and for every row the pointer to its first element and the write of the row sum to the result vector y. If there are no empty rows, the amount of data to be moved:

$$\begin{gathered}\texttt{nnz} * (2 * sizeof(float|double) \ + sizeof(int)) + \\ \texttt{dimRow} * (sizeof(float|double) + sizeof(int)).\end{gathered} \tag{4.4}$$

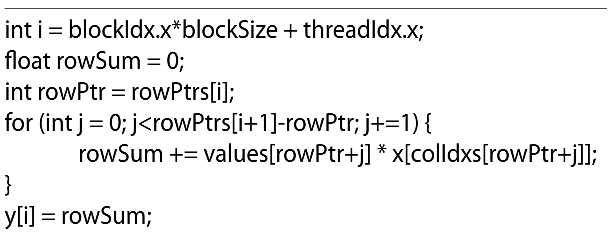
Dividing this quantity by the execution time gives what is called the *effective* bandwidth, and this is the formula the figures in [33] are based on, thus I use it for the purpose of comparability. On a caching architecture this corresponds to each value on each cache line being read while it is in the cache, but only once.

The worst case scenario occurs when only one value of each cache line is used, thus in fact a lot more data is moved to the chip, but most of it is not read. In single precision a cache line holds 32 floats or integers, thus if each memory access loads an entire cache line, but reads or writes only one value on it before removing it from the cache, the actual amount of data moved to and from global memory:

$$\begin{gathered}32 * (\ \texttt{nnz} * (2 * sizeof(float) + sizeof(int)) + \\ \texttt{dimRow} * (sizeof(float) + sizeof(int))\ ).\end{gathered} \tag{4.5}$$

In the best case scenario, with 100% data reuse, each element of the multiplicand vector has to be loaded from global memory only once, thus the lower bound on all data transfers considering perfect cache efficiency is:

$$\begin{gathered}\texttt{nnz} * (sizeof(float|double) + sizeof(int)) + \\ \texttt{dimRow} * (2 * sizeof(float|double) + sizeof(int)).\end{gathered} \tag{4.6}$$



```
int i = blockIdx.x*blockSize + threadIdx.x;
float rowSum = 0;
int rowPtr = rowPtrs[i];
for (int j = 0; j<rowPtrs[i+1]-rowPtr; j+=1) {
        rowSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
}
y[i] = rowSum;
```

Figure 4.11. Naive CUDA kernel for performing the SpMV [33].

The naive CUDA implementation

The simplest implementation of the CSR SpMV kernel in CUDA assigns one thread to each row, defines the block size (`blockSize`) to be a multiple of the warp size (currently 32), e.g. 128, and calculates the number of blocks (`gridSize`) accordingly. The sketch of the CUDA code is described by Figure 4.11.

If the matrix has several elements per row, this implementation suffers from a high cache miss rate because the cache is not big enough to hold all of the cache lines being used. Moreover, if the number of non-zeros per row has a high variance over adjacent rows, some threads run for more iterations than others in the same warp, thus warp divergence may also become an issue.

Thread cooperation

To reduce the number of cache lines used when accessing the arrays `values` and `colIdxs`, multiple threads can be assigned to work on the same row [33]. The cooperating threads access adjacent elements of the row, perform the multiplication with the elements of the multiplicand vector x then add up their results in shared memory using parallel reduction. Finally, the first thread of the group writes the result to the vector y. Because threads in the same warp are executed in lockstep, the synchronisation between cooperating threads is implicit - as long as their number is a power of 2, and no more than the warp size. From here on, the number of cooperating threads assigned to each row is indicated by `coop`.

This technique can greatly improve the efficiency of caching; the same cache lines from the `values` and `colIdxs` arrays are used by the cooperating threads, thus there are more cache lines left for storing the values of the multiplicand vector. If the length of the rows is not a factor of 32 then this algorithm may also suffer from branch divergence, resulting in loss of parallelism and decreased performance.

Because of the assignment of subsequent rows to subsequent groups of threads, the worst case scenario data transfer described by formula (4.5) cannot happen, because either

DOI:10.15774/PPKE.ITK.2014.003

reading non-zeros or writing to the result vector uses the same cache line. With `coop`=1 and long rows, the cache lines storing non-zeros and their column indices may be removed from the cache after just one read, however writing to the elements of the result vector will use the same cache line. On the other hand, with `coop`=32, the cache lines storing non-zeros and column indices are fully utilised (unless row length is not a factor of 32), but writing to the elements of the result vector may force loading a new cache line every time. The effects of this trade-off will be seen in the performance evaluation.

Granularity

It is possible for a cooperating thread group to process more than one row, thereby I can control the work granularity. Thus, a thread block processes `repeat*blockSize/coop` contiguous rows.

For a fixed value of `repeat`, `blockSize` and `coop`, the total number of blocks is the following:

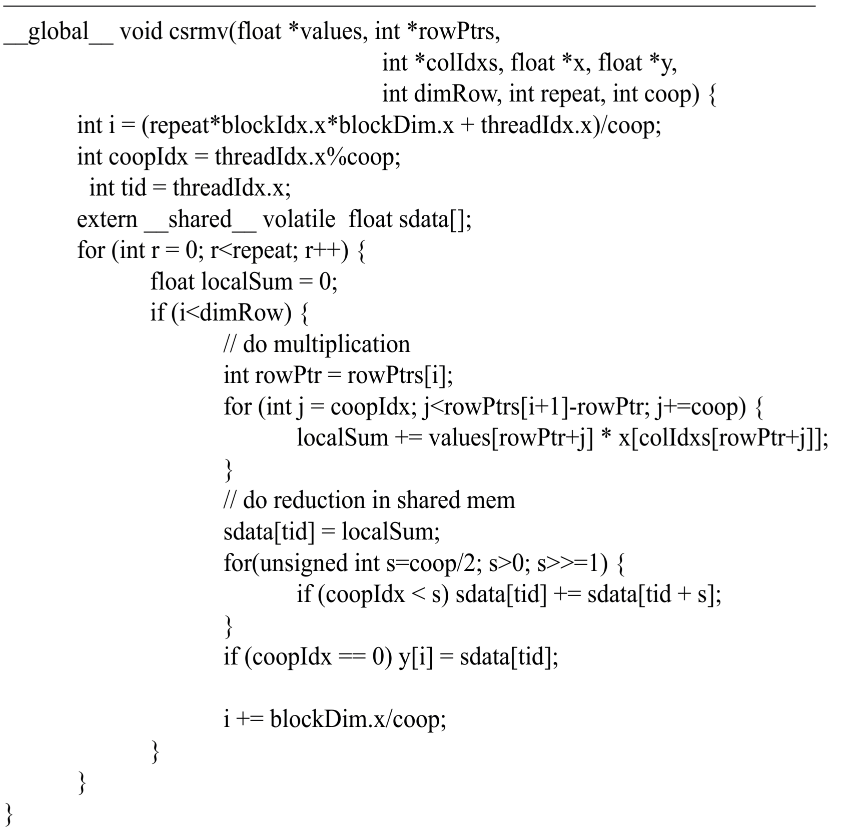
$$gridSize = 1 + (dimRow * coop - 1)/(repeat * blockSize). \tag{4.7}$$

If `repeat` is small, the algorithm is *fine grained* and if `repeat` is big, the algorithm is *coarse grained*. To have sufficient occupancy and load balancing on the GPU, the `gridSize` should be at least a few hundred, which may limit small matrices to execute in a fine grained way. However, for larger matrices, the difference in the number of blocks using the two approaches may be large.

Granularity is closely related to the efficiency of caching, or *cache blocking*; if for example the matrix has a diagonal structure, i.e. the rows access a contiguous block of the multiplicand vector, then the data reuse is improved by coarse grain processing because most of the values used are already in the cache. The optimal granularity depends on the structure of the matrix, which is not known in the general case.

The fully parametrised algorithm

The full parameter space of the algorithm is described by the following parameters: the number of threads per block (`blockSize`), the number of cooperating threads per row (`coop`) and the number of rows processed by each cooperating thread group (`repeat`). The total number of blocks (`gridSize`) is uniquely defined by these parameters according to equation (4.7). The complete algorithm is described by Figure 4.12. For the sake of brevity, the parallel reduction is also parametrised with `coop`, whereas in my real implementation there is a different kernel for each different value of `coop`, thus the reduction is unrolled.



```
__global__ void csrmv(float *values, int *rowPtrs,
                      int *colIdxs, float *x, float *y,
                      int dimRow, int repeat, int coop) {
    int i = (repeat*blockIdx.x*blockDim.x + threadIdx.x)/coop;
    int coopIdx = threadIdx.x%coop;
     int tid = threadIdx.x;
    extern __shared__ volatile  float sdata[];
    for (int r = 0; r<repeat; r++) {
        float localSum = 0;
        if (i<dimRow) {
            // do multiplication
            int rowPtr = rowPtrs[i];
            for (int j = coopIdx; j<rowPtrs[i+1]-rowPtr; j+=coop) {
                localSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
            }
            // do reduction in shared mem
            sdata[tid] = localSum;
            for(unsigned int s=coop/2; s>0; s>>=1) {
                if (coopIdx < s) sdata[tid] += sdata[tid + s];
            }
            if (coopIdx == 0) y[i] = sdata[tid];

            i += blockDim.x/coop;
        }
    }
}
```

Figure 4.12. Parametrised algorithm for sparse matrix-vector multiplication.

Note that there is no use of *___syncthreads()* thread synchronisation due to the implicit warp synchronisation discussed in section 4.5.

4.5.1 Performance Evaluation

The test matrices were taken from the University of Florida Sparse Matrix Collection [67] based on the test cases of the CUSPARSE 4.0 library [59]. 15 of these matrices were used to train and tune my algorithms, and a total of 44 matrices were used to evaluate them and calculate performance figures [1]. A short summary of the training matrices is shown in table 4.1. These matrices represent a range of structured and unstructured matrices with different sizes, different averages and standard deviations in the length of the rows and different distributions of non-zeros.

In the following sections I describe the performance of the sparse matrix-vector multiplications when adjusting the parameters of the multiplication algorithm. These performance results are interpreted in light of the matrix structure to describe the underlying bottlenecks. For the sake of clarity, these parameters are discussed one by one, however it

[1] List of matrices not shown in table 4.1: amazon0505, cont1_l, filter3D, msdoor, troll, ct20stif, halfb, parabolic_fem, shipsec1, vanbody, BenElechi1, dc1, pkustk12, StocF-1465, bmw3_2, delaunay_n22, large-basis, poisson3Db, stomach, xenon2, boneS01, mc2depi, qcd5_4, tmt_sym, consph, F2, memchip, rma10, torso3

DOI:10.15774/PPKE.ITK.2014.003

Table 4.1. Description of test matrices, avg. is the average row length and std. dev. is the standard deviation in row length.

Name	`dimRow`	`nnz`	avg.	std. dev.	Description
atmosmodd	1270432	8814880	6.9	0.24	diagonal, CFD problem
cage14	1505785	27130349	18	5.36	directed weighted graph
cant	62451	4007383	64.2	14.05	narrow diagonal, FEM
cop20k_A	121192	2624331	21.6	13.8	very wide, 2D problem
F1	343791	26837113	39.5	40.86	AUDI engine, FEM
mac_econ_fwd500	206500	1273389	6.2	4.43	macroeconomics problem
pdb1HYS	36417	4344765	60.2	31.93	protein 1HYS
scircuit	170998	958936	5.6	4.39	wide, circuit simulation
shallow_water1	81920	327680	4	0	structured CFD problem
webbase-1M	1000005	3105536	3.1	25.34	web connectivity
nd24k	72000	28715634	398	76.9	ND problem set
crankseg_2	63838	14148858	221	95.8	wide, structural problem
pwtk	217918	11524432	52.8	5.44	pressurised wind tunnel
ldoor	952203	42493817	44.6	14.7	structural problem
2cubes_sphere	101492	1647264	16.2	2.65	electromagnetics simulation

is shown that they are not independent.

Number of cooperating threads

The number of cooperating threads (`coop`) can be a power of 2, up to the warp size. Figure 4.13a shows the performance on different test matrices for different values of `coop`, with `blockSize = 128` and `repeat = 8`; the performance difference between the best and worst choice can be more than an order of magnitude. The figure also shows the square root of the average row length of the matrices; note how the optimal value of `coop` is close to this value. Table 4.2 lists the L1 cache hit rates and warp divergence reported by the CUDA Profiler as a function of the number of cooperating threads. It is important to note that the cache hit rate usually increases with the increasing number of cooperating threads because multiple threads access the same cache lines for the matrix data, leaving space for more cache lines to store the values of the multiplicand vector. However too high a value of `coop` results in a high fraction of inactive threads: e.g. shallow_water has four non-zeros in every line, thus assigning 8 threads to process each row would result in half of the threads being inactive all the time.

Level of granularity

The number of rows processed by each cooperating thread group is the most important factor in determining the total number of blocks (`gridSize`). As Figure 4.13b shows, for larger matrices the difference in performance is relatively small (around 10%), however for

Table 4.2. Cache hit rates and warp divergence for test matrices at different values of `coop`, based on the CUDA Visual Profiler.

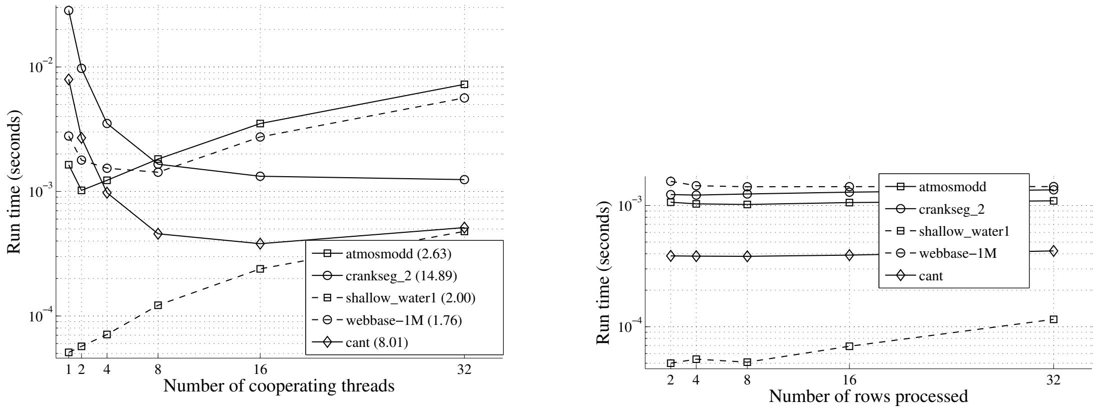
`coop`	1	2	4	8	16	32
atmosmodd						
L1 hit rate (%)	54.5	77.5	79.1	75.3	86.0	90.6
Divergence (%)	2.6	13.3	13.79	15.9	16	16
crankseg_2						
L1 hit rate (%)	20.4	48.4	56.1	62.0	60.6	64.4
Divergence (%)	1.8	4.1	5.6	4.8	5.1	5.3
shallow_water1						
L1 hit rate (%)	71.8	60.8	59.0	75.7	87.0	92.1
Divergence (%)	0.0	0.0	0.0	16.0	16.0	16.0
webbase-1M						
L1 hit rate (%)	72.8	81.2	73.6	80.6	86.9	92.1
Divergence (%)	9.5	9.2	16.8	16.55	16.1	15.9
cant						
L1 hit rate (%)	19.56	50.9	60.9	66.9	74.7	78.6
Divergence (%)	8	12.8	7.2	9.9	5.1	6.7

smaller matrices like *shallow_water1* high values of `repeat` results in low occupancy - in this case at `repeat = 32` there are only 20 blocks. In CUDA, the user is not in control of the execution scheduling of blocks, thus for larger structured matrices higher values of `repeat` offer marginally better data reuse. On the other hand, fine grained processing (more blocks) offers better load balancing across the SMs.

Number of threads in a block

The number of threads in a block (`blockSize`) plays an important role in determining occupancy: the maximum number of threads per SM (in Fermi) is 1536, and the maximum number of blocks per SM is 8. Thus a small block size leads to low occupancy, 33% with `blockSize=64` ($64 * 8 = 512$). My algorithm also uses shared memory, storing one floating point number for each thread, which does not limit occupancy; since a double precision number is 8 bytes long, with 100% occupancy the total amount of shared memory used is 12.2kB. The optimal block size however is usually smaller than what is required for full occupancy to balance parallelism and cache size per thread. Double precision storage also effectively halves the number of values held in the L1 cache. For this reason the optimal value of the parameter is often even smaller in double precision. Figure 4.14 shows the impact of `blockSize` on performance.

DOI:10.15774/PPKE.ITK.2014.003



(a) Number of cooperating threads. `blockSize = 128` and `repeat = 8`. The square root of the average row length is displayed in brackets after the name of the matrix.

(b) Number of rows per thread group, `blockSize = 128` and `coop` is fixed at its optimal value according to figure 4.13a.

Figure 4.13. Performance as a function of the number of cooperating threads and the number of rows processed per cooperating thread group.

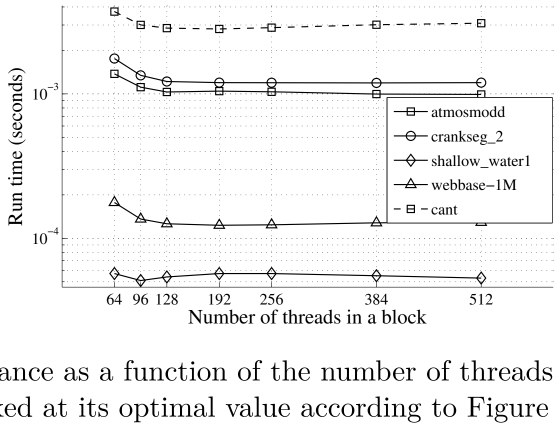
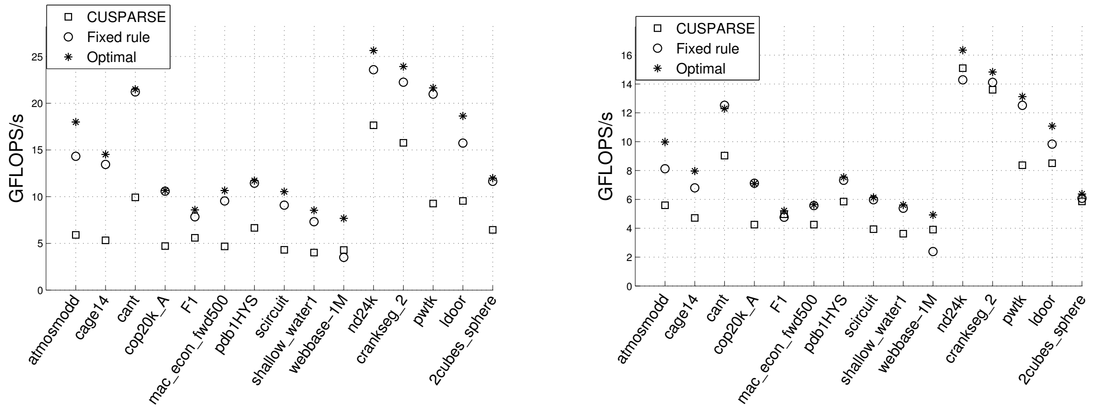


Figure 4.14. Performance as a function of the number of threads in a block. `repeat = 4` and `coop` is fixed at its optimal value according to Figure 4.13a and 4.13b.



(a) Single precision

(b) Double precision

Figure 4.15. Floating point operation throughput of the sparse matrix-vector multiplication.

The fixed rule

For general-purpose SpMV code, a constant time *fixed rule* is required to decide the input parameters for the multiplication algorithm. For this purpose I first implemented an exhaustive search algorithm that tests a wide range of parameters and ran the training matrices through it. Based on this data, I fine-tuned my fixed rule to find the optimal parameter values that provide the highest average performance. Since this rule is intended for use by a general-purpose library, this average is not a weighted average, performance is equally important for smaller and larger matrices.

The fixed rule defines the parameters as follows:

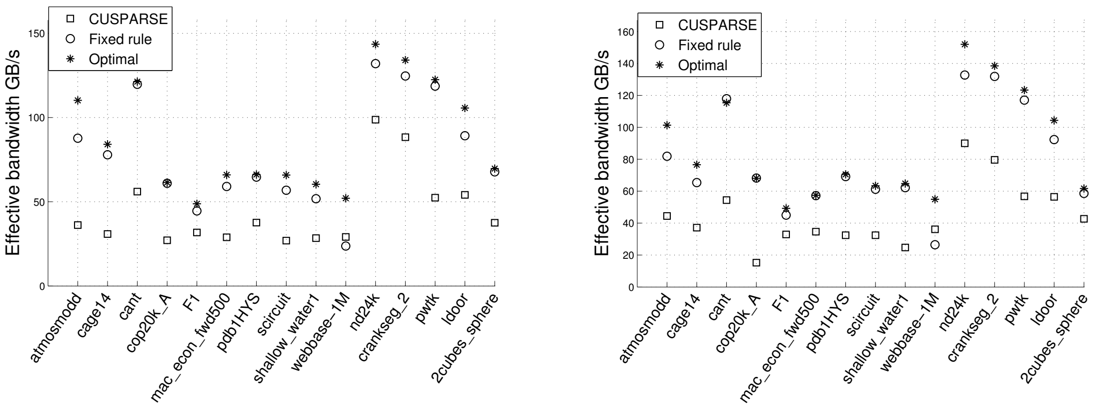
1. `blockSize = 128`.

2. `coop` is the smallest power of two which is larger than the square root of the average row length, up to a maximum limit of the warp size (32).

3. `gridSize` and `repeat` is calculated in a way that ensures there are at least 1500 blocks.

The calculation of these parameters incurs virtually no overhead, since only the number of rows and the number of non-zeros is required.

Figures 4.15a and 4.16a describe performance in single precision and figures 4.15b and 4.16b in double precision. All bandwidth numbers show effective bandwidth as described by equation (4.4). Since instruction throughput is directly proportional to the number of non-zeros processed per second (equation (4.3)), it accurately describes the relative efficiency of data reuse. Note that the bandwidth of the matrix (the distance of non-zeros



(a) Single precision.

(b) Double precision.

Figure 4.16. Effective bandwidth of the sparse matrix-vector multiplication

from the diagonal) is not directly related to throughput. For example *crankseg_2* has non-zeros everywhere in the matrix but the non-zeros in *pwtk* lie very close to the diagonal, and yet their performance is very similar because they are both structured matrices, thus non-zeros in consecutive rows have similar column indices which enables efficient caching. The average row length and the relative value of the standard deviation of the length of the rows has some effect on the performance, however the most important factor is the “structuredness” of the matrix which is difficult to describe.

Estimating bandwidth and caching

Block and warp scheduling on GPUs is out of the control of the programmer, and the exact parameters of Fermi’s L1 cache, like associativity, are not public knowledge, thus exact modelling of the cache is extremely difficult and out of the scope of this work. It is however possible to estimate the number of cache lines loaded by each block. I make the following assumptions: (1) cache lines are not shared between blocks, (2) all cache lines loaded by a block stay in the cache until the execution of that block finishes. These assumptions only affect the caching of the multiplicand vector significantly, because the access pattern to other data structures does not depend on matrix structure. Due to the semi-random assignment of thread blocks to SMs and because the L1 cache is probably not fully associative, I argue that the first assumption does not overestimate the actual amount of data moved by too much. The second assumption however implies unconstrained cache size per block, thus underestimates the actual amount of data moved. I argue that this error is only significant if the temporal difference between accesses to same cache lines is high; for structural matrices this is usually low, for non-structural matrices data reuse is

DOI:10.15774/PPKE.ITK.2014.003

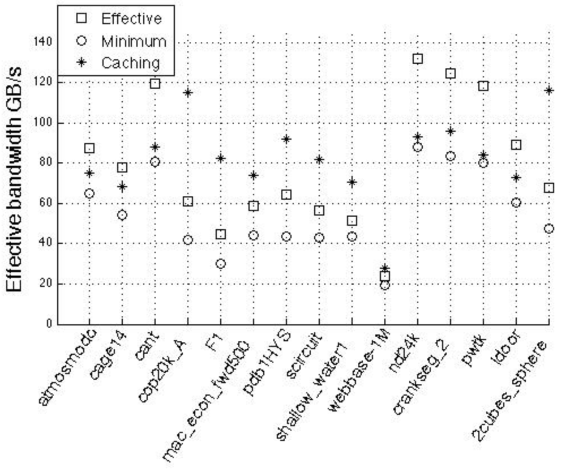


Figure 4.17. Values of the minimum, the effective and the caching bandwidth in single precision.

low anyway. The effects of the L2 cache are ignored.

Figure 4.17 shows calculated bandwidth estimates when using the fixed rule, according to formulas (4.4) and (4.6), and based on the number of cache lines touched by each thread block. Note that bandwidth in some cases is over 80 GB/s even according to the minimum formula that assumes perfect data reuse. This is a very high fraction of the theoretical peak (144 GB/s) considering that in practice a bandwidth over 100 GB/s is rarely achieved. It is important to observe that for structured matrices data reuse between subsequent rows is high, thus caching bandwidth falls between the minimum and the effective bandwidth. In the case of non-structural matrices, several values of the cache lines holding elements of the multiplicand vector are not read by that block at all, thus caching bandwidth is even higher than the effective bandwidth, because part of the data moved is not used. Caching bandwidth has a mean of 83 GB/s with a relatively low variation compared to other bandwidth figures, which hints at the average hardware utilisation for the problem of sparse matrix-vector multiplication. In some extreme cases, caching bandwidth exceeds 120 GB/s, probably due to cache lines being in fact shared between blocks.

4.5.2 Run-Time Parameter Tuning

The performance of the SpMV multiplication using the fixed rule is in many cases close to the optimal, however there is room for improvement in some other cases. An extreme example is *webbase-1M*, where the fixed rule only achieves 50% of the optimal performance, but matrices like *atmosmodd, cage14, F1* and *ldoor* can also benefit from improved parameters. In a general-purpose library it is not feasible to perform off-line parameter tuning because it may have a significant overhead. It is also unknown how many times a sparse matrix-vector multiplication routine is called from the user's code,

DOI:10.15774/PPKE.ITK.2014.003

and whether those calls use the same matrix or different matrices.

I argue that if subsequent calls to the SpMV routine use the same pointers to matrix data and have the same amount of rows and non-zeros then the matrix structure is not changing significantly across multiple executions. This hypothesis enables the library to test slightly different parameters when performing the multiplication in order to find the best set of parameters. This of course may result in having to run a few multiplications with worse performance than the fixed rule, but if the number of iterations is large enough, then this overhead is compensated by the improved overall performance. It is important to note that in such situations there is no statistical data available, thus system noise may affect measured performance.

I propose an empirical algorithm that is based on my experience with different parameter settings and the parameters found by exhaustive search. The main concepts of the algorithm are the following:

1. As the first step, double the number of blocks by halving `repeat`. If the change is higher than 5%, proceed optimising `repeat` (step 3). If the change is less than 5%, double `coop` and increase `blockSize` to 192 (step 2).

2. If doubling `coop` increased performance then try increasing it further, if it did not, start decreasing it.

3. Depending on whether halving `repeat` increased or decreased performance, further decrease or increase it.

4. Finally the value of `blockSize` is changed in increments (or decrements) of 32, but no more than 512 and no less than 96 in single and 64 in double precision.

Figure 4.18 shows the impact of tuning on performance in the single precision case. The fixed rule achieves 84% of the optimal performance on average over the 15 training matrices, and 73% over all 44 test matrices. Figure 4.19 shows the improvement of performance over subsequent iterations due to the run-time tuning. The performance surpasses the 95% threshold after just 5 iterations, and 98% after 8 iterations. Similar figures apply in double precision.

Performance analysis on the full test set

To provide a fair assessment of my algorithm I ran it through a set of 44 test matrices and calculated average instruction throughput, bandwidth and speedup figures. I

DOI:10.15774/PPKE.ITK.2014.003

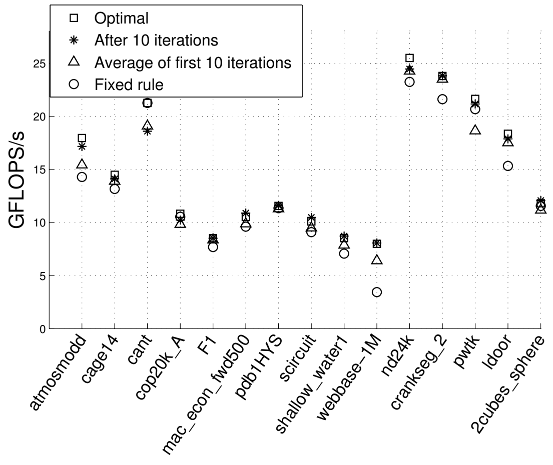


Figure 4.18. Single precision floating point instruction throughput after 10 iterations of run-time tuning.

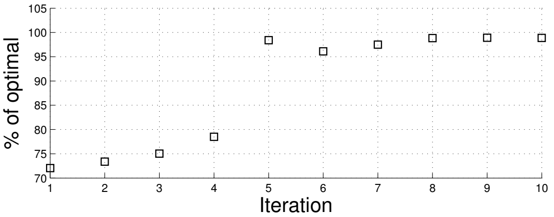
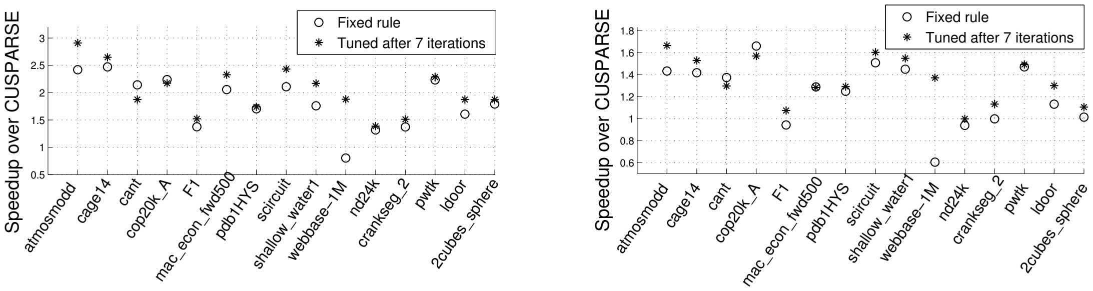


Figure 4.19. Relative performance during iterations compared to optimal, averaged over the 44 test matrices.



(a) Single precision

(b) Double precision

Figure 4.20. Speedup over CUSPARSE 4.0 using the fixed rule and after run-time tuning

Table 4.3. Performance metrics on the test set.

	CUSPARSE	Fixed rule	Tuned
Throughput single GFLOPS/s	7.0	14.5	15.6
Throughput double GFLOPS/s	6.3	8.8	9.2
Min Bandwidth single GB/s	28.4	58.9	63.7
Min Bandwidth double GB/s	38.7	54.0	56.8
Speedup single over CUSPARSE	1.0	2.14	2.33
Speedup double over CUSPARSE	1.0	1.42	1.50

found one matrix for which the CSR format and the standard multiplication algorithm proved unsuitable (*dc1*): not even the exhaustive search found parameters that would have brought performance above 4 GFLOPS/s. This particular matrix has an average row length of 6.5, however there are two very long adjacent rows (114200 and 47190 non-zeros) which results in severe load imbalance that radically decreases performance. Using the standard SpMV algorithm would be clearly inappropriate in this case; the simplest solution is to process those two rows separately with multiple blocks performing two parallel reductions and the rest is processed by the standard algorithm. Results from the standard algorithm on *dc1* are omitted. Performance metrics in table 4.3 clearly indicate that there is a wider performance gap between the fixed rule and the tuned version, since the fixed rule itself was trained on the first 15 matrices.

4.5.3 Distributed memory spMV

In most large-scale applications it is necessary to employ distributed memory parallelism because the problem may not fit in the memory of a single machine. The prevalent programming model and environment is the Message Passing Interface (MPI), which is used to explicitly communicate between remote processes by means of messages. Depending on the hardware setup and the topology, communication may have significant latency and usually a much lower bandwidth compared to intra-node communication. Therefore, attention has to be given to how computations and communications are coordinated so that they don't get in each others way.

The distributed memory parallelisation of the sparse matrix-vector product is conceptually simple and well known [68]; following the decomposition of the graph described by the sparse matrix, I have a number of subgraphs that are connected by edges. The multiplication of a row of the sparse matrix with the multiplicand vector can be interpreted as a weighted sum of values on vertices that the current row's vertex is connected to via edges. This introduces a data dependency where edges cross from one subgraph to the other, therefore before the multiplication can be carried out on a *boundary* vertex, data

DOI:10.15774/PPKE.ITK.2014.003

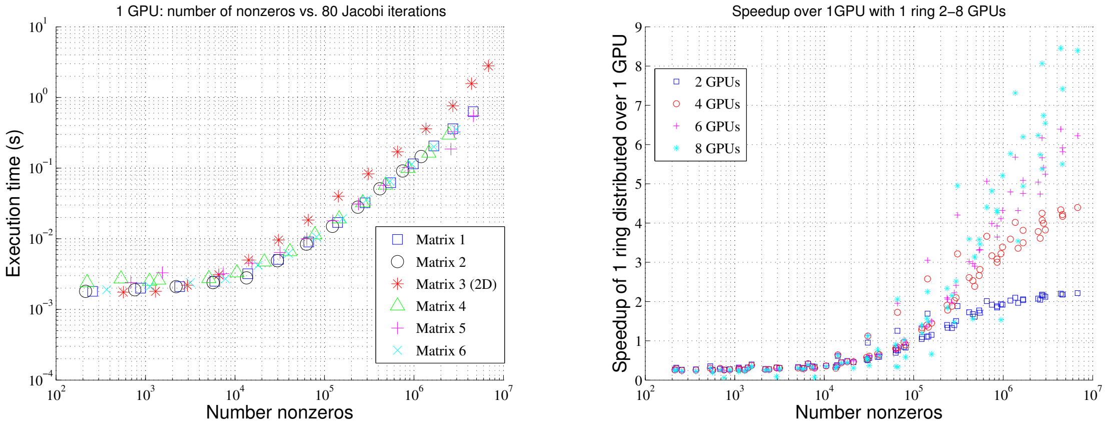
on vertices it is connected to need to be up to date - in the MPI model, values on these boundary vertices are gathered and sent in a message to other subgraphs where they are duplicated as *halo* vertices, so that their values can be updated. Furthermore, each subgraph, consisting of the *owned* vertices, can be partitioned into two classes of vertices; boundary vertices that have a data dependency from different subgraphs and *interior* vertices that do not. Thus it is possible to perform the communications and the computations on the interior at the same time, thereby trying to hide the latency of communications with computations [69].

When investigating the performance of distributed memory algorithms, strong scaling is often of interest; how fast can the same problem be solved, given more resources. This pushes the limits in terms of the latency of communications, because in such a situation the size of the subgraphs decrease, but the relative number of boundary vertices increases - computations over the interior are increasingly unable to hide the latency of communications. This situation also surfaces when considering Multigrid-type techniques to solve a system of linear equations [70]: on subsequent multigrid levels the graph has fewer and fewer vertices, and if these are further partitioned into the same number of subgraphs on each level, then this can be considered a strong scaling problem. Since GPUs are especially sensitive to latency due to memory copy operations, kernel launches and underutilisation of the hardware, addressing these issues is paramount to achieving high performance. In extreme cases of over-partitioning this may result in the distributed computation being slower than carrying out the same computations on a single device.

To address the negative impact of latency, following the idea of tiling [21] I have proposed a communication-avoiding algorithm that is capable of reducing the number of messages at the expense of redundant computations. By carrying out a dependency analysis, it is possible to map out several *halo rings*, the first consisting of vertices that boundary vertices are connected to, the second consists of vertices that are connected the first ring and do not belong to either the first ring or the boundary, and so on. If only one ring of halo is updated, then only one sparse matrix-vector multiplication can be carried out (over the owned vertices) that will still yield globally consistent results. By updating two rings of halo, it is possible to carry out the two operations, first over the owned vertices and the first ring and then a second over the owned vertices and still achieve globally consistent results, but only communicating once. Clearly, this can be extended to an arbitrary number of rings.

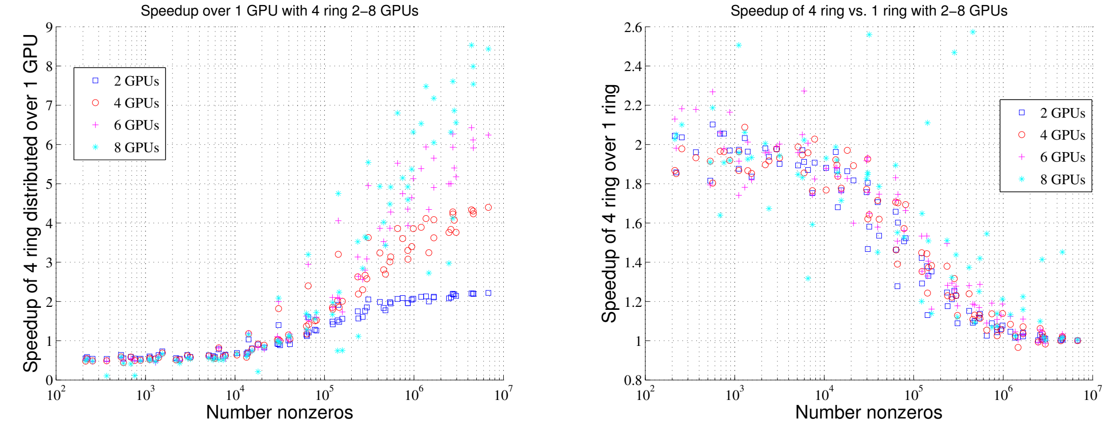
Having implemented this algorithm for GPUs, I carry out a set of benchmarks on a

DOI:10.15774/PPKE.ITK.2014.003



(a) Performance scaling on a single GPU (b) Performance scaling over multiple GPUs

Figure 4.21. Performance on different size sparse matrices on a single GPU and on multiple GPUs



(a) Performance scaling over multiple GPUs (b) Improvement over 1 ring approach

Figure 4.22. Improvements by applying the communication-avoiding algorithm

DOI:10.15774/PPKE.ITK.2014.003

cluster of 8 NVIDIA K20X GPUs, partitioning a set of 6 sparse matrices, retrieved from ANSYS Fluent's CFD simulations, fed into AmgX, and dumped at each multigrid level, yielding a total of 72 matrices with increasingly smaller sizes. Strong scaling is investigated when computing 80 iterations of a Jacobi smoother, consisting of an spMV operation and a vector-vector operation. Figure 4.21a shows performance on a single GPU as a function of the number of non-zeros in the sparse matrix; it is clear that the GPU does not perform well below 10^4 non-zeros due to underutilisation. Figure 4.21b shows the speedup over the single GPU performance when employing distributed memory parallelism using 2-8 GPUs, the cost of communications is obvious; below 10^5 non-zeros it is actually slower to use multiple GPUs.

By employing the communication-avoiding algorithm, it is possible to lower the threshold where multi-GPU computations become excessively slow; Figure 4.22b shows that by using 4 rings of halo, performance scaling is maintained up to 3×10^4, a more than three-fold improvement over the naive 1 ring approach. Finally, Figure 4.22a shows the speedup of the 4 ring approach over the 1 ring approach, clearly at high number of non-zeros latency is sufficiently hidden by a large interior, but as matrices get smaller, the advantage of the communication avoiding approach becomes clear. Thus, by employing this technique, it is possible to deliver strong-scalability to general spMV-type operations and scalability to multigrid solvers.

4.6 Summary and associated theses

In this chapter I provided a brief introduction to GPUs and discussed some of the architectural specifics than influence implementation and help understand performance. After describing the experimental setup and the reference CPU implementation, I investigated the performance of the three formulations of FE integration on the GPU, discussing the effect of trade-offs, showing that the redundant compute approach delivers a sustained high performance even at high order elements [3]; this completes Thesis I.1: *By applying transformations to the FE integration that trade off computations for communications and local storage, I have designed and implemented new mappings to the GPU, and shown that the redundant compute approach delivers high performance, comparable to classical formulations for first order elements, furthermore, it scales better to higher order elements without loss in computational throughput.*

I have presented an in-depth study of different data structures, their implementation and performance on the GPU, both in the assembly and the solution phase, discussed how data races are resolved and what impact that has on parallel efficiency. I have demonstrated that near-optimal performance in absolute terms is achieved on the GPU, outperforming a fully utilised dual-socket server CPU up to 10 times. I have shown that the LMA delivers superior performance in most cases, especially in single precision where the atomics are available [3]. This completes Thesis I.2: *By applying transformations to the FE integration that trade off computations for communications and local storage, I have designed and implemented new mappings to the GPU, and shown that the redundant compute approach delivers high performance, comparable to classical formulations for first order elements, furthermore, it scales better to higher order elements without loss in computational throughput.*

Finally, I have connected my research with the field of sparse linear algebra by investigating the parametrisation and mapping of the sparse matrix-vector product on GPUs; I have studied the effect of changing the balance of multi-level parallelism and presented a novel constant-time heuristic for determining near-optimal parameters for execution. Furthermore, I have introduced a run-time parameter tuning algorithm, and I have shown that these outperform the state-of-the-art by up to 2.3 times. My research on communication-avoiding algorithms for the sparse matrix-product also resulted in up to 2 times improvement over classical approaches [3, 8, 12]. Thus, as Thesis I.3 states: *I have parametrised the mapping of sparse matrix-vector products (spMV) for GPUs, designed a new heuristic and a machine learning algorithm in order to improve locality, concurrency and load balancing. Furthermore, I have introduced a communication-avoiding algorithm for the distributed execution of the spMV on a cluster of GPUs. My results improve upon the state of the art, as demonstrated on a wide range of sparse matrices from mathematics, computational physics and chemistry.*

DOI:10.15774/PPKE.ITK.2014.003

Chapter 5

The OP2 Abstraction for Unstructured Grids

In this chapter I discuss the motivation behind OP2 and the abstraction it defines for unstructured grid computations [16, 4] to serve as a basis for the next two chapters where I present my research within the framework of OP2. Furthermore, this chapter introduces the suite of unstructured grid applications and briefly presents my work translating the Volna and the Hydra application to the OP2 abstraction.

5.1 The OP2 Domain Specific library

Following the 13 dwarfs [25], researchers began to work on domain specific libraries and languages, and there are a number of projects involved in structured grid computations [29, 30, 31], unstructured grid computations [27, 16] and a group at Stanford is involved in the generalisation of these ideas in a *meta-DSL framework* [26]. The group at the University of Oxford, led by Prof. Mike Giles is amongst the first ones to publish research on such DSLs [16, 2], work began in late 2010 in collaboration with Prof. Paul Kelly's group at Imperial College, with the main goal of creating a DSL for unstructured grid computations, eventually to be used by Rolls-Royce plc. for the simulation of turbomachinery. It also became part of FEniCS, a Finite Element simulation software package, through research carried out at Imperial College with a python-based variant [71].

I joined the project in its second year (2011), with a functional abstraction and single-node implementations, and while I have contributed to several areas, in this dissertation I focus on research that is sufficiently self-contained that it can be called my own. As such, the contributions in Chapter 6 are mine, but in Chapter 7 I make use of existing

DOI:10.15774/PPKE.ITK.2014.003

functionality in OP2, such as the construction of the execution plan for multi-level parallelism described in Section 7.1 or the MPI implementation, these serve as a basis for my own research. One of my over-arching contributions is porting Rolls-Royce Hydra to OP2, in itself not considered a scientific contribution, but by carrying out this work, I was able to show that Domain Specific Languages are indeed applicable to large-scale industrial applications, delivering maintainability, performance and future-proofing - something that to our knowledge has not been demonstrated so far for any DSL targeting scientific computations.

To provide an introduction to that work, in this chapter I discuss the OP2 domain specific library, as well as the applications used to assess the library and its performance. These I do not consider as my scientific contributions in this dissertation, they serve as a basis for understanding the next two chapters.

The OP2 abstraction involves breaking down the description of the mesh and the computations into four distinct parts:

1. Sets of elements, such as vertices, edges or cells.
2. Mappings between sets that explicitly describe the connectivity of the mesh, for example specifying for each edge which vertices it connects.
3. Data on sets, representing coordinates, flow variables etc.
4. Parallel loops over sets, accessing data on the set or via at most one level of indirection.

An important restriction of the abstraction is that it assumes that the order in which elements are executed during a parallel loop does not affect the results within machine precision. This assumption enables it to orchestrate parallelism and resolve data dependencies in a consistent manner.

OP2 was designed from the outset to be a general high-level *active library* framework to express and parallelise unstructured mesh based numerical computations. OP2 retains the OPlus abstraction [72] but provides a more complete high-level API (embedded in C/C++ and Fortran) to the application programmer which for code development appears as an API of a classical software library. However, OP2 uses a source-to-source translation layer to transform the application level source to different parallelisations targeting a range of parallel hardware. This stage provides the opportunity to provide the necessary implementation specific optimisations. The code generated for one of the platform-specific

DOI:10.15774/PPKE.ITK.2014.003

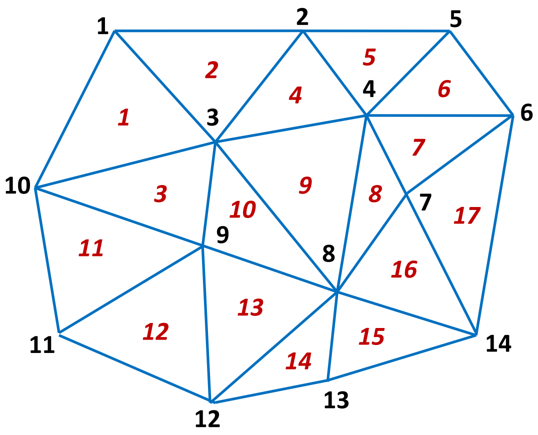


Figure 5.1. An example unstructured mesh

```
//define sets
op_set nodes = op_decl_set(num_nodes,"nodes");
op_set edges = op_decl_set(num_edges,"edges");
op_set cells = op_decl_set(num_cells,"cells");
//define mappings
int *e2n = ...; int *e2c = ...;
op_set edge2node= op_decl_map(edges,nodes,
                              2,e2n,“edge2node");
op_set edge2cell= op_decl_map(edges,cells,
                              2,e2c,“edge2cell");

//define datasets
double *x=...; double *q=...; double *res=...;
op_dat p_x = op_decl_dat(nodes,2,"double",
                              x,"coords");
op_dat p_q = op_decl_dat(edges,4,"double",
                              q,“statevars");
op_dat p_res=op_decl_dat(cells,4,"double",
                              res,“residual");
```

(a) Declaring sets, maps and datasets

```
//in res_calc.h
void res_calc(const double *x1, const double *x2,
              const double *q,
              double *res1, double *res2 ){
  //Computations, such as:
  res1[0] += q[0]*(x1[0]-x2[0]);
}
//in main program file
op_par_loop(res_calc,“res_calc",edges,
   op_arg_dat(p_x,   0,edge2node, 2,"double",OP_READ),
   op_arg_dat(p_x,   1,edge2node, 2,"double",OP_READ),
   op_arg_dat(p_q,  -1,OP_ID,     4,"double",OP_READ),
   op_arg_dat(p_res, 0,edge2cell, 4,”double”,OP_INC ),
   op_arg_dat(p_res, 1,edge2cell, 4,”double”,OP_INC));
```

Dataset name
Indirection index
Mapping
Data arity
Data type
Access type

(b) A parallel loop

Figure 5.2. The OP2 API

parallelisations can be compiled using standard C/C++/Fortran compilers to generate the platform specific binary executable.

Consider the unstructured mesh illustrated in Figure 5.1. The mesh consists of three sets - nodes (vertices), edges and triangular cells. There are `num_nodes = 14` nodes and `num_cells = 17` cells. The OP2 API allows to declare these sets and connectivity between the sets together with any data held on the sets (see Figure 5.2a). Note that data may be “multi-dimensional”; storing multiple state variables for each set element.

Any loop over the sets in the mesh is expressed through the `op_par_loop`, API call. OP2 enforces a separation of the per set element computation aiming to de-couple the declaration of the problem from the implementation [28]. Thus, an example of a computational loop can be expressed as detailed in Figure 5.2b using the OP2 API.

The `res_calc()` function is called a *user kernel* in the OP2 vernacular. Simply put, the `op_par_loop` describes the loop over the set `edges`, detailing the per set element computation as an outlined “kernel” while making explicit indication as to how each argument

DOI:10.15774/PPKE.ITK.2014.003

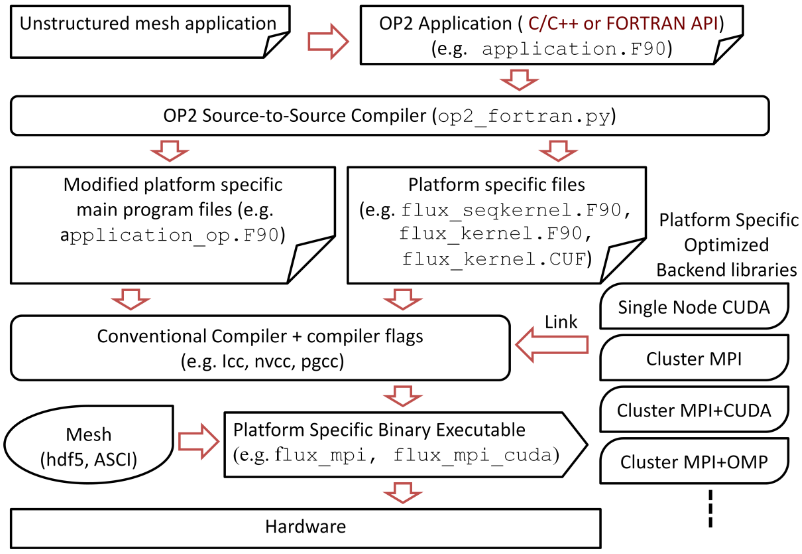

Unstructured mesh application
OP2 Application (C/C++ or FORTRAN API)
(e.g. application.F90)
OP2 Source-to-Source Compiler (op2_fortran.py)
Modified platform specific main program files (e.g. application_op.F90)
Platform specific files (e.g. flux_seqkernel.F90, flux_kernel.F90, flux_kernel.CUF)
Platform Specific Optimized Backend libraries
Single Node CUDA
Cluster MPI
Cluster MPI+CUDA
Cluster MPI+OMP
Link
Conventional Compiler + compiler flags (e.g. Icc, nvcc, pgcc)
Mesh (hdf5, ASCI)
Platform Specific Binary Executable (e.g. flux_mpi, flux_mpi_cuda)
Hardware

Figure 5.3. OP2 build hierarchy

to that kernel is accessed (OP_READ - read only, OP_INC - increment, OP_WRITE - write only, OP_RW - read and write) and the mapping, `edge2cell`, `edge2node` (edges to cells and edges to nodes) with the specific indices are used to indirectly accessing the data (`p_x, p_res`) held on each cell or node. With this separation the OP2 design gives a significantly larger degree of freedom in implementing the loop with different parallelisation strategies.

An application written with the OP2 API in the above manner can be immediately debugged and tested for accuracy by including OP2's "sequential" header file (or its equivalent Fortran module if the application is written in Fortran). This, together with OP2's sequential back-end library, implements the API calls for a single threaded CPU and can be compiled and linked using conventional (platform specific) compilers (e.g. gcc, icc, ifort) and executed as a serial application. Once the application developer is satisfied with the validity of the results produced by the sequential application, parallel code can be generated. The build process to obtain a parallel executable is detailed in Figure 5.3. In this case the API calls in the application are parsed by the OP2 source-to-source translator which will produce a modified main program and back-end specific code. These are then compiled using a conventional compiler (e.g. gcc, icc, nvcc) and linked against platform specific OP2 back-end libraries to generate the final executable. The mesh data to be solved is input at runtime. The source-to-source code translator is written in Python and only needs to recognise OP2 API calls; it does not need to parse the rest of the code.

DOI:10.15774/PPKE.ITK.2014.003

Table 5.1. Properties of Airfoil kernels; number of floating point operations and numbers transfers

Kernel	Direct read	Direct write	Indirect read	Indirect write	FLOP	FLOP/byte DP(SP)	Description
save_soln	4	4	0	0	4	0.04(0.08)	Direct copy
adt_calc	4	1	8	0	64	0.57(1.14)	Gather, direct write
res_calc	0	0	22	8	73	0.3(0.6)	Gather, coloured scatter
bres_calc	1	0	13	4	73	0.5(1.01)	Boundary
update	9	8	0	0	17	0.1(0.2)	Direct, reduction

Table 5.2. Airfoil mesh sizes and memory footprint in double(single) precision

Mesh	cells	nodes	edges	memory
small	720000	721801	1438600	94(47) MB
large	2880000	2883601	5757200	373(186) MB

5.2 The Airfoil benchmark

The basic benchmark that I use throughout my work is Airfoil, a non-linear 2D inviscid airfoil code that uses an unstructured grid [73]. It is a much simpler application than the Hydra [74] production CFD application used at Rolls-Royce plc. for the simulation of turbomachinery, but acts as a forerunner for testing the OP2 library for many core architectures. The code consists of five parallel loops: `save_ soln`, `adt_calc`, `res_calc`, `bres_calc` and `update`. The number of computations carried out per set element, the amount of data moved and a short description is shown in Table 5.1. For the domain, I use two meshes with different resolutions as described on Table 5.2. This is the benchmark I use to develop and validate code generation techniques and to provide an in-depth analysis of performance.

5.3 Volna - tsunami simulations

Volna is a finite volume, unstructured grid tsunami-simulation code that was first presented in [75]. It solves the shallow-water equations using a 2.5D grid - a triangular meshing of the domain with the bathymetry (distance from the ocean floor) stored as a state variable. The outline of the code is as shown in Algorithm 8; based on an input configuration file, during startup the mesh is read in from *gmsh* files, and state variables, such as the bathymetry that starts a tsunami, are initialised from text files. For the main time-marching loop, a variable timestep second-order Runge-Kutta method is used. In each iteration, events can be triggered; such as applying changes to the bathymetry or outputting simulation data to VTK files for visualisation.

In a similar way to the description of Airfoil's computational kernels, Table 5.3 de-

DOI:10.15774/PPKE.ITK.2014.003

Algorithm 8 Code structure of Volna

```
Read and set up mesh
Initialise state variables
while $t < t_{final}$ do
    Apply pre-iteration events
    Second-order Runge-Kutta time stepper
    Compute state variables on cell faces, apply boundary conditions
    Compute fluxes across cell faces and timestep
    Apply fluxes and bathymetric source terms to state variables on cells
    Apply post-iteration events
end while
```

Table 5.3. Properties of Volna kernels; number of floating point operations and numbers transfers

Kernel	Direct read	Direct write	Indirect read	Indirect write	FLOP	FLOP/byte	Description
RK_1	8	12	0	0	12	0.6	Direct
RK_2	12	8	0	0	16	0.8	Direct
sim_1	4	4	0	0	0	0	Direct copy
compute_flux	4	6	8	0	154	8.5	Gather, direct write
numerical_flux	1	4	6	0	9	0.81	Gather, reduction
space_disc	8	0	10	8	23	0.88	Gather, scatter

scribes the details of Volna's computational kernels. `RK_1`, `RK_2` and `sim_1` are direct kernels, due to their similarity `sim_1` is dropped from further analysis. `compute_flux` is a numerically intensive kernel which gathers data, and directly writes data. `numerical_flux` indirectly gathers data, and directly writes data as well as carrying out a reduction to find the minimum timestep. Finally, `space_disc` does both indirect gathering and scattering of data.

I ported this code to use the OP2 abstraction, and it is now being used at the University College London by Serge Guillas's group for the uncertainty quantification of tsunamis. During my experiments I use Volna to verify that optimisations tested on Airfoil are applicable to other applications as well, and whether improvements in performance are consistent. For Volna, I use a real-world mesh with 2.5M cells, describing the north-western coast of North America and the strait leading to Vancouver and Seattle, simulating a hypothetical tsunami originating in the Pacific Ocean.

5.4 Hydra - turbomachinery simulations

The aerodynamic performance of turbomachinery is a critical factor in engine efficiency of an aircraft, and hence is an important target of computer simulations. Significant computational resources are required for the simulation of these highly detailed three-dimensional meshes. Usually the solution involves iterating over millions of elements

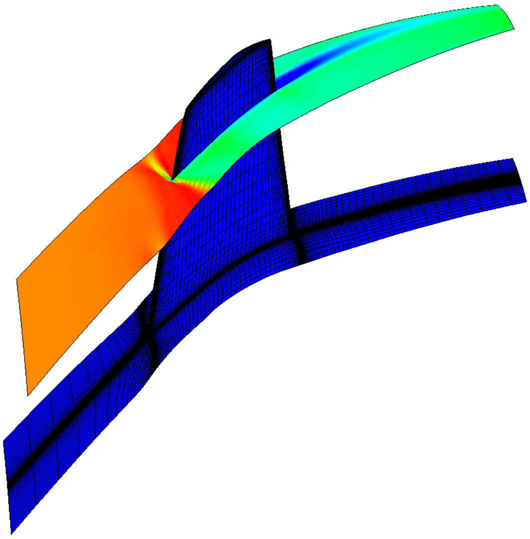


Figure 5.4. Mach contours for NASA Rotor 37.

(such as mesh edges and/or nodes) to reach the desired accuracy or resolution.

Rolls Royce's Hydra CFD application is such a full-scale industrial application developed for the simulation of turbomachinery. It consists of several components to simulate various aspects of the design including steady and unsteady flows that occur around adjacent rows of rotating and stationary blades in the engine, the operation of compressors, turbines and exhausts as well as the simulation of behaviour such as the ingestion of ground vortexes. The guiding equations which are solved are the Reynolds-Averaged Navier-Stokes (RANS) equations, which are second-order PDEs. By default, Hydra uses a 5-step Runge-Kutta method for time-marching, accelerated by multigrid and block-Jacobi preconditioning [76, 77, 78, 79]. The usual production meshes are in 3D and consist of tens of millions of edges, resulting in long execution times on modern CPU supercomputers.

Hydra was originally designed and developed over 15 years ago at the University of Oxford and has been in continuous development since, it has become one of the main production codes at Rolls-Royce. Hydra's design is based on a domain specific abstraction for unstructured mesh based computations, OPlus [80], a classical software library, which provided a concrete distributed memory parallel implementation targeting clusters of single threaded CPUs. OPlus essentially carried out the distribution of the execution set of the mesh across MPI processes, and was responsible for all data movement while also taking care of the parallel execution without violating data dependencies. Hydra consists of over 300 computational loops, and a wide array of functionality that can be enabled and disabled at runtime. When porting Hydra to OP2 all of these requirements needed to be taken into account and the results had to be validated. Further details of the deployment can be found in [4].

The problem used to benchmark the application is the standard CFD test NASA Rotor 37, which is a transonic axial compressor rotor widely used for validation in CFD. Figure 5.4 shows a representation of the Mach contours for this application on a single blade passage, the mesh consists of 2.5 million edges.

My main goal in studying Hydra is to demonstrate that Domain Specific Languages, such as OP2, are indeed applicable to large-scale simulations, delivering maintainability, portability, performance and future proofing; something that to my knowledge no other study has done before.

DOI:10.15774/PPKE.ITK.2014.003

Chapter 6

High-Level Transformations with OP2

Having detailed the OP2 abstraction for unstructured grids in Chapter 5, in this Chapter I present my research involving high-level transformations of unstructured grid algorithms that rely on domain-specific knowledge and information provided through the OP2 API to alter execution patterns. Section 6.1 discusses a novel checkpointing algorithm, Section 6.2 presents a redundant-compute tiling approach to executing unstructured grid algorithms in order to improve locality, and Section 6.3 demonstrates my research into enabling collaborative execution on radically different, heterogeneous hardware at the same time.

6.1 Checkpointing to improve resiliency

Due to the increasing number of components in modern computer systems, the mean time for failure is decreasing to the point where it may be smaller than the execution time of a large scale simulation. More complex systems inherently require more complex software as well, on all levels; the operating system, parallel libraries and applications, which increases the probability of software errors. One of the advantages of using Domain Specific Languages, such as OP2, is that the complexity in orchestrating parallelism is hidden from the application developer, and relies only on parallel programming experts developing and maintaining the library; this helps minimise the number of bugs. Regardless of what level the error occurs at, or whether it is a hard error (causing the application to crash) or a soft error (that might go unnoticed for a while), it is crucial to provide means to correct these errors and finish running the application. The detection of soft errors is

DOI:10.15774/PPKE.ITK.2014.003

the target of intense research: on the lowest, hardware level, Error Checking & Correcting (ECC) circuitry is added to memory to detect bit flips due to e.g. cosmic radiation. Errors occurring during computations are more difficult to detect, hardware redundancy has been proposed, but duplication of circuitry would be very expensive [81]. Due to the complexity of detecting soft errors, here I only consider the case where the error was already detected by some external (to OP2) means.

Traditionally in scientific computing, developers had to manually implement some sort of checkpointing system, saving the entire state space to disk and in case of failure, relaunch the application and provide some means of fast-forwarding to the point during execution where the last checkpoint was. Here, I show that this process can be automated in its entirety, furthermore that it is possible to find a point during execution where the state space is minimal and only save the necessary data. I demonstrate three approaches - one that makes a “locally optimal” decision, one that attempts to make a “globally optimal” decision by assuming some form of periodic execution pattern, and the third which given a test-run tells the user where the optimal point during execution is, so that the checkpointing can be triggered manually.

The basic idea comes from the checkpointing of transactions in database systems [82]; execution using OP2 can be considered as a sequence of parallel loops, each accessing data, describing the type of access. This information can be used to reason about what data is required for any given loop; for example if a loop reads A and writes B, then the prior contents of B are irrelevant, thus if a checkpoint were to be created, B would not have to be included in it. Such a transactional view of OP2 is valid because once OP2 takes control of data, the user can only access it via API calls which go through the library, therefore the checkpointing algorithm can take them into account when working out the state of different datasets.

To facilitate checkpointing, each dataset has a “dirty” flag to keep track of modifications to it. A high-level description of the checkpointing algorithm is as follows:

1. If a dataset was never modified (as it might be the case with e.g. coordinates), then it is not saved at all.

2. Variables passed to loops through `op_arg_gbl` that are not read-only are saved for every occurrence of the loop because data returned after a loop is out of the hands of OP2, and may be used for control decisions.

3. When checkpoint creation is triggered due to a timeout, then start looking for a loop

with at least one `OP_WRITE` argument, if found, then enter "checkpointing mode", and before executing that loop:

   (a) Save datasets used by the loop that are not `OP_WRITE`.

   (b) Drop datasets that are `OP_WRITE` in the loop from the checkpoint.

   (c) Flag datasets that are not used by this loop, but do not save them yet.

4. When already in "checkpointing mode" (previous point), start executing subsequent loops to determine whether datasets that were not yet saved nor dropped (i.e. are flagged) would have to be saved:

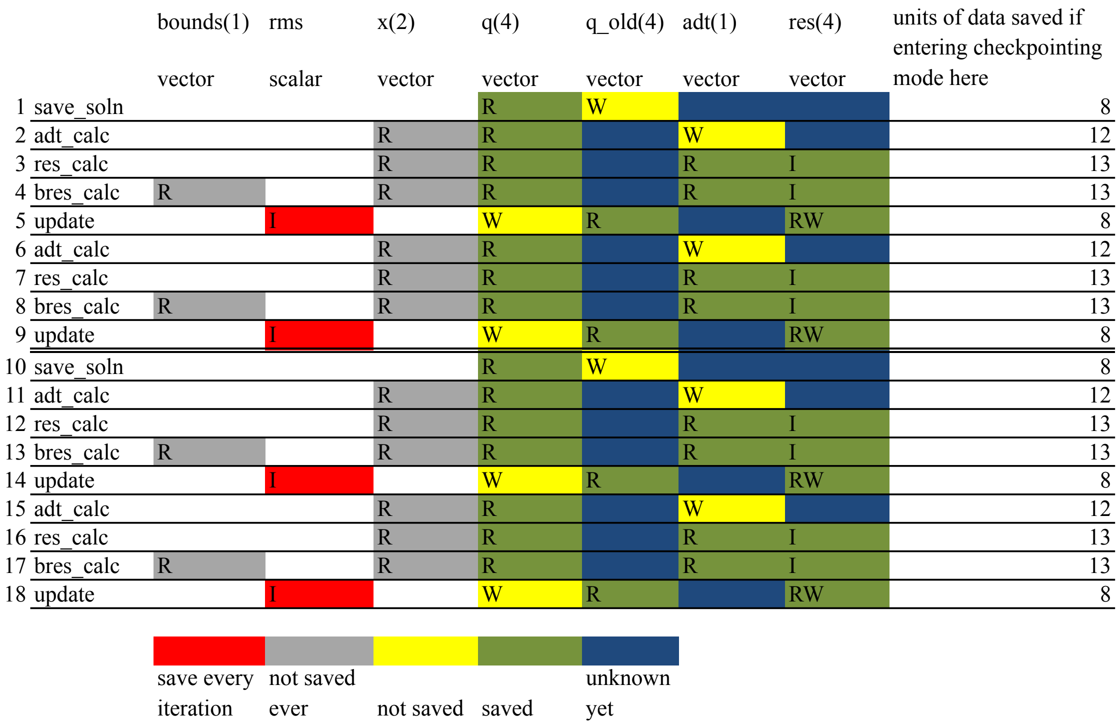
   (a) If a flagged dataset is encountered, save it if it's not `OP_WRITE`, otherwise drop it and remove the flag.

   (b) If a flagged dataset is not encountered within a reasonable timeframe allocated for the "checkpointing mode", then save it.

The immediate benefit of this algorithm is that a checkpoint does not necessarily depend on the state of the collection of all datasets at any particular point during the execution; the "checkpointing mode" can cover several loops and gather information based on different datasets being accessed by different loops.

The "locally optimal" algorithm starts looking for the first loop to enter "checkpointing mode" at, based on the criteria that it has at least one `OP_WRITE` argument, but this may be a suboptimal decision if for example the next loop would write two other arguments and read the one written by the first loop. However, OP2 cannot definitively predict what loops would come next because control flow is in the hands of the user, therefore this kind of decision will result in a checkpoint that may have a suboptimal size. There are two possible ways to improve upon this decision; the first is for OP2 to carry out a "dry run" where it looks for potential places during execution where a globally minimal checkpoint could have been created, reports this to the user and the user places a call in the code that will trigger checkpointing in the right place. The other possibility is to record the sequence of parallel loop calls and carry out a subsequence alignment analysis; assuming that some periodicity is found, it can determine the optimal location to begin checkpointing at, therefore entering the "checkpointing mode" can be postponed until that point is encountered again.

These algorithms were implemented in the OP2 library, available at [83], it currently supports single-node checkpointing into HDF5. There are some important practical con-



		bounds(1) vector	rms scalar	x(2) vector	q(4) vector	q_old(4) vector	adt(1) vector	res(4) vector	units of data saved if entering checkpointing mode here
1	save_soln				🟩 R	🟨 W	🟦	🟦	8
2	adt_calc			▨ R	🟩 R	🟦	🟨 W	🟦	12
3	res_calc			▨ R	🟩 R	🟦	🟩 R	🟩 I	13
4	bres_calc	▨ R		▨ R	🟩 R	🟦	🟩 R	🟩 I	13
5	update		🟥 I		🟨 W	🟩 R	🟦	🟩 RW	8
6	adt_calc			▨ R	🟩 R	🟦	🟨 W	🟦	12
7	res_calc			▨ R	🟩 R	🟦	🟩 R	🟩 I	13
8	bres_calc	▨ R		▨ R	🟩 R	🟦	🟩 R	🟩 I	13
9	update		🟥 I		🟨 W	🟩 R	🟦	🟩 RW	8
10	save_soln				🟩 R	🟨 W	🟦	🟦	8
11	adt_calc			▨ R	🟩 R	🟦	🟨 W	🟦	12
12	res_calc			▨ R	🟩 R	🟦	🟩 R	🟩 I	13
13	bres_calc	▨ R		▨ R	🟩 R	🟦	🟩 R	🟩 I	13
14	update		🟥 I		🟨 W	🟩 R	🟦	🟩 RW	8
15	adt_calc			▨ R	🟩 R	🟦	🟨 W	🟦	12
16	res_calc			▨ R	🟩 R	🟦	🟩 R	🟩 I	13
17	bres_calc	▨ R		▨ R	🟩 R	🟦	🟩 R	🟩 I	13
18	update		🟥 I		🟨 W	🟩 R	🟦	🟩 RW	8

🟥	▨	🟨	🟩	🟦
save every iteration	not saved ever	not saved	saved	unknown yet

Figure 6.1. Example checkpointing scenario on the Airfoil application

siderations that have to be taken into account when implementing this for general unstructured grid computations; if an indirect write exists in a loop then it has to be checked whether the mapping includes every element of the target set, therefore the whole of the target dataset would be written to. I also assume that every field of a multidimensional dataset is written to when it appears in a parallel loop.

6.1.1 An example on Airfoil

Figure 6.1 shows an example of how the checkpointing algorithm work when applied to the Airfoil benchmark application. It shows for each loop, which datasets are read (R), written (W), incremented (I) or read-and-written (RW) as well as the dimensionality, or number of components, for each dataset. It also indicates how many units of data would have to be saved if checkpointing mode was entered at the current loop. Colours are used to indicate when a dataset is saved (green), dropped (yellow) or flagged for further decision (blue). If checkpointing were to be triggered right before the execution of `adt_calc`, then checkpointing mode would be entered upon reaching `adt_calc`, saving `q` and dropping `adt` immediately, and then subsequently `res` would be saved when reaching `res_calc` and `q_old` when reaching `update`. Since `bounds` and `x` were never modified, they are not saved, and the values of `rms` are saved whenever `update` has executed. Clearly, entering checkpointing mode at kernel `adt_calc` is still better than entering it before either res_calc or bres_calc, but it does not give as much data savings as if checkpointing mode were to be entered before either save_soln or update. Therefore either the user can add a call to trigger checkpointing before these two loops, or OP2 can apply the "speculative" algorithm and recognise that there is likely a periodic execution because the sequence of kernels 1-9 repeats, thus it can wait with entering checkpointing mode until either save_soln or update are reached.

In the event of a failure, the application is simply restarted, but until reaching the point in execution where the checkpoint was made, the op_par_loops do not carry out any computations, only set the value of op_arg_gbl arguments, if any. This ensures that the same execution path is followed when fast-forwarding, once the checkpoint is reached, all state is restored from the saved data, and normal execution is resumed.

In summary, this algorithm can deliver checkpointing to any application that uses OP2 by only relying on domain specific knowledge and the high-level specification of parallel loops. This is completely opaque to the application programmer, thereby providing resiliency to unstructured grid computations at no effort from the user.

6.2 Tiling to improve locality

Trends in hardware development show an increasing gap between computational capacity of a chip and the amount of bandwidth to off-chip memory. To address this issue, the size of on-chip caches have been growing, in the latest Intel CPUs, there is up to 30MB last level cache. The purpose of these caches is to improve spatial and temporal locality; the assumption is that if a memory location is referenced, then with a high probability adjacent memory locations will also be referenced (spatial locality) and that the same memory address will be referenced again (temporal locality). This is completely hidden from the programmer, caching is fully automated by the hardware, but there is only so much it can do - if the programmer is not conscious of locality in the way code and data structures are implemented, then performance may even be worse than without caching; each memory transaction loads an entire cache line, if only part of it is used and the rest is not, then it's just wasted bandwidth. In most cases in scientific computations, spatial locality is given by the structure of the problem: for example structured grid algorithms tend to utilise the full cache lines, and even in the case of unstructured grid applications there is usually reasonable spatial locality if the numbering of the mesh is well-behaved - my research into the effects of mesh reordering are discussed in Section 7.3.

Unstructured grid algorithms tend to be bandwidth-bound: the amount of computations carried out per bytes of data moved is relatively low, therefore even if there is good spatial locality, the computational units are under-utilised. This is closely related to how PDE discretisations are implemented; computational sweeps over the entire grid, computing and updating state variables which are read in the next sweep to carry out further updates. This structure implies that entire datasets are streamed in from memory, computed upon and then streamed out, only to be streamed back in again in the next sweep. Clearly, if both sweeps were for example on vertices, only accessing data directly, then the two loops could be fused to carry out both operations right after one another on the same vertex - this would enable temporal locality as well. However, most computational loops are not trivial like that, for example if the first sweep is over cells, indirectly updating vertices, and the next sweep were on edges, accessing values on the vertices previously updated, then there is a data dependency; before an edge of the second sweep could execute, all cells that include either of the vertices at the two ends of the edge would have had to have been updated already. Reorganising computations in a way that satisfies these data dependencies but still provides temporal locality is called *tiling*.

Tiling has been demonstrated for structured grids [34, 22], but it is quite obvious, even for such trivial problems, that writing scientific code in a way that would support tiling would be a very error-prone process and would result in practically unreadable and unmaintainable code. Thus it is easy to see why tiling has rarely been applied to anything larger than a proof of concept benchmark. In fact, this is the first known practical application of tiling to unstructured grid applications, based on our paper [9].

The key observation to be able to reorganise execution and respect data dependencies is that OP2 makes no assumptions about the order in which set elements in a parallel loop are executed. Therefore it is possible for OP2 to determine the order in which elements are executed in a single loop to improve spatial locality, and, as I demonstrate here, to execute certain parts of subsequent computational loops so that temporal locality is achieved. The most important component in enabling this is *lazy or delayed execution* [84]; when an `op_par_loop` is encountered computations are not carried out immediately, rather they are placed in a queue. Having gathered a number of loops, it is possible to reason about inter-loop data dependencies. By employing this technique I show that it is possible to support tiling in a way that is completely opaque to the application programmer; only the implementation of `op_par_loop` changes; the only requirement is that if an `op_arg_gbl` is encountered, the output of which may be used for control, then computations have to be triggered to ensure correct output and control flow.

At a very high level, the tiling algorithm can be described as follows:

1. Collect a number of kernels to be tiled over into $kernels[1...N]$.
2. Perform multi-set partitioning of the mesh into the desired number of tiles.
3. For each tile, compute the dependency tree by iterating over kernels backwards $N...1$:
   (a) Based on previous data dependencies, determine which set elements have to be executed in the current loop so that the data dependency is satisfied.
   (b) Determine data dependencies required to execute set elements from the previous step.
4. Execute tiles one by one:
   (a) Copy data the tile is dependent upon into scratch memory.
   (b) Iterate over kernels $1...N$, only executing set elements determined in previous steps.
   (c) Copy final data back from scratch.

This algorithm describes redundant compute tiling for a sequence of general unstructured grid operations; redundant because the execution of a particular set element may update data on different set elements that may belong to different tiles, therefore both tiles have to carry out the computation to satisfy their data dependencies. Figure 6.2 illustrates one iteration of dependency analysis; given an initial data dependency for a given tile, highlighted in red, it is necessary to determine which set elements need to be executed in the current loop so that data is updated, in this example the loop iterates over cells, updating edges, therefore the highlighted cells need to be executed. However, in order to execute these cells, it is necessary to read data on the vertices of these cells, thus these are added to the list of dependencies. This process is repeated for all loops that are being tiled over, resulting in a list of elements to be executed for each one to satisfy the data dependencies of the next loop.

The implementation is available at [83], since it consists of several thousand lines of C++ code, the specifics are omitted here, only a high-level overview is presented. As usual, the high-level user application does not change, the code generated for each user kernel enqueues the kernel instead of immediately executing it, the rest of the tiling logic is implemented within the OP2 framework, as described by Algorithm 9.

Using this structure, it is possible to separate the construction and management of tiles from the actual execution of individual kernels within the tile, which provides the

Algorithm 9 Redundant compute tiling in OP2

```
Input: data structure holding description of kernels, along with op_args
if tiling plan does not exist for this sequence of kernels then
    create a new tile plan by first partitioning the set the last loop iterates over into the desired number of tiles
    inherit partitioning to all of the other sets, through mapping connections to the partitioned set
    for each tile do
        from each set, add elements that belong to the current tile to the list of dependencies
        for each kernel, in reverse order do
            iterate over the non read-only arguments of the kernel and determine for each element in the kernel's execution set whether it accesses any set element in the list of dependencies, if so, add the set element to the list of elements to be executed for this kernel
            iterate over the execution set elements and determine what data it reads, read-writes or increments on which set elements, and if those set elements are not yet in the list of dependencies, add them
        end for
        based on existing global mapping tables, create tile-specific mapping tables that only include entries from set elements that are in the dependency list of the tile
        for each dataset, determine the required scratch memory size based on the dependency list and the size of the entries in the dataset
        perform colouring on the entire tile with respect to the mappings to support shared-memory parallelism
        sort tile elements, datasets and mappings by colour and record index ranges for each colour
        save tiling plan
    end for
end if
load tiling plan
for each tile do
    iterate over kernel descriptors in the current tile and substitute op_dats and op_maps specific to the tile in all op_args
    copy data dependencies from global op_dats to scratch memory
    for each kernel do
        call kernel
        for each colour do
            execute set elements of the same colour with all available OpenMP threads
        end for
    end for
    write back data on set elements that belonged to the initial partitioning to a copy of the global op_dats
end for
swap pointers of the original and the copies of global op_dats
```

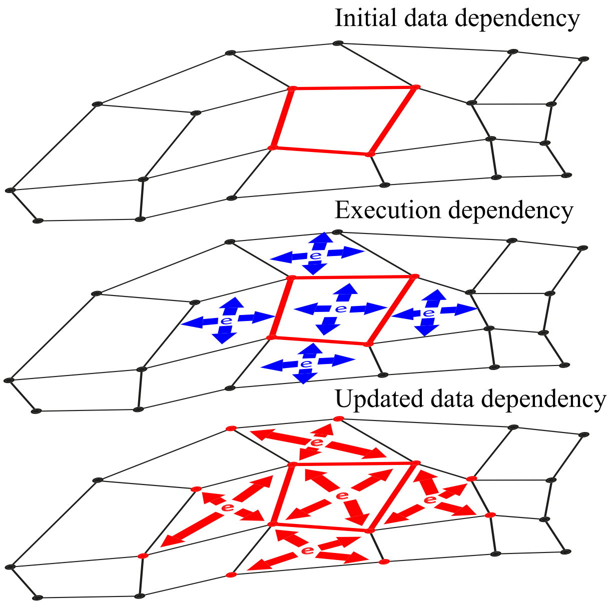


Figure 6.2. Dependency analysis required for tiling for a given loop that iterates over cells, reading vertices and incrementing edges indirectly

modularity that makes it possible to apply this tiling technique to any "front-end" or target architecture. Since this approach implements redundant compute tiling, the size of the "halo", set elements that have to be executed due to data dependencies, is important; if the tile size is too small, then the ratio of the core part of the tile to the halo is low, possibly resulting in too many redundant computations. Exact parameters depend heavily on the target hardware, on factors like the size of the low-latency memory that would hold the tile, the ratio of computational throughput the bandwidth to low-latency memory and off-chip memory, and the parallelisation approach used to process the tile.

Currently, only CPU architectures support meaningful tile sizes; GPUs and the Xeon Phi rely on a high level of parallelism to hide the latency of memory transactions, thus they have relatively small cache sizes compared to the amount of data parallelism needed to fully utilise the computational cores. Despite this, even on CPUs the ratio of computational capacity to memory bandwidth is not yet high enough for this tiling approach to offset the cost of redundant computations and additional bookkeeping; I haven't measured better performance yet, but current trends show memory bandwidth lagging behind computational performance, therefore tiling approaches are expected to become more important.

DOI:10.15774/PPKE.ITK.2014.003

6.3 Heterogeneous execution

Today's systems are increasingly heterogeneous; multi-core CPUs are commonly paired with accelerators such as GPUs or the Xeon Phi, and there is a convergence in architectures; already AMD's Fusion chips incorporate a general purpose programmable GPU next to the CPU cores, resulting in what AMD call the Accelerated Programming Unit (APU) - indeed in some cases more than half the chip is dedicated to graphics processing. NVIDIA's mobile chips also have GPU and ARM CPU cores. Future architectures are expected to follow the same trends, but some degree of separation will remain because it is the differences in fundamental architecture that make accelerators so energy efficient at certain tasks. It is a fact that systems are becoming heterogeneous, but it is still the exception rather than the rule that software use all the available resources; this leads to underutilisation of the system. Most related work published on many-core acceleration, and GPU acceleration in particular, focuses on migrating the entire code base to the GPU and then comparing performance with the CPU. However, modern GPU supercomputers, such as Titan at Oak Ridge NL National Laboratory, consist of roughly the same number of GPUs and CPU sockets, and often pricing is only calculated on a per-node basis. Thus, if an application only exploits the computational resources of the GPUs, then the CPUs are idling, even though they might have considerable computational power themselves.

It is precisely because of these fundamental architectural differences that different hardware have different performance characteristics when executing various tasks, and this is perhaps the reason why heterogeneous execution has rarely been utilised. Several papers address this issue by employing different techniques, where the CPU and the GPU either have the same role, such as in the case of shared task-queues [85], or where the GPU computes on the bulk of the workload and the CPU handles the parts where the GPU would be under-utilised, such as the boundary in domain decomposition systems [86, 87, 88], but these tend to discuss very specific problems.

In the case of unstructured grid computations this is compounded by having computations with various degrees of irregularity; the more regular the more efficient architectures with long vector units are. The goal of my research was to enable heterogeneous execution, that is the utilisation of different computational resources, and to understand and model the performance characteristics when carrying out unstructured mesh computations. The first step in facilitating heterogeneous execution in OP2 is to carry out the domain decomposition of the unstructured mesh and utilise distributed memory parallelism to assign resulting partitions to different hardware. In the MPI setting, this means that some ranks

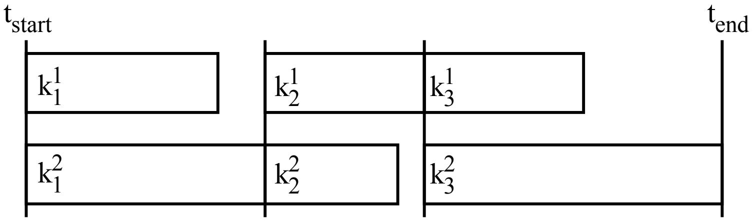


Figure 6.3. Hybrid scheme with fully synchronous execution

are assigned to one type of hardware, other ranks to a second type of hardware, and so on. This implies a static load balancing scheme; while it is possible to create partitions of different sizes, it will not be possible to change it afterwards - this is acceptable because moving large amounts of data associated with re-balancing would be extremely expensive. The key challenge then is to come up with a partition size balance between different hardware that reflects their performance differences.

To determine the right load balance, it is important to understand the performance differences of individual loops on different hardware, and how synchronisation - either implicit or explicit - is carried out between them. The most simplistic model would assume that there is synchronisation between each computational loop, thus for a sequence of N loops with different execution times on K different processes (each may be mapped to different hardware) $k_1^1, k_2^1, ...k_N^1, ..., k_1^K, k_2^K, ...k_N^K$ the total runtime would be:

$$t = \sum_{i=1}^{N} max(k_i^1, k_i^2, ..., k_i^K) \tag{6.1}$$

This essentially describes the scenario where the execution is fully synchronised and runtime is determined by the slowest hardware in each loop, as illustrated in Figure 6.3.

To satisfy data dependencies in distributed memory parallelism, OP2 uses halo exchanges, in combination with a technique called latency hiding; the computational domain on each MPI process is split into two parts: elements that need data from neighbouring processes (*boundary*) and elements that do not (*interior*). By beginning with the transmission of data, then carrying out computations over the interior, and only then waiting for communications to finish, followed by the computations on the boundary, it is possible to hide the latency of communications, assuming the computations on the interior take longer than communications. This approach results in implicit pairwise synchronisation; computations over the boundary cannot begin until data from neighbouring partitions are received, and it does allow for slight performance differences to be hidden through asynchronicity. Some loops however do not require halo exchanges (for example direct loops) and other loops have global reductions which results in blocking synchronisation with no

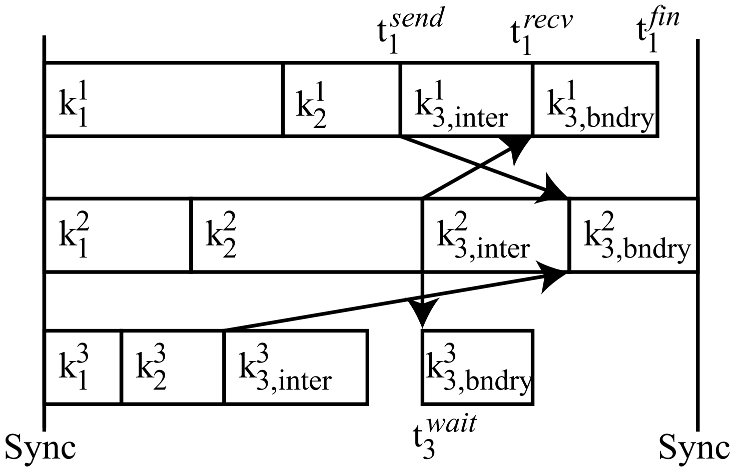


Figure 6.4. Hybrid scheme with latency hiding

possibility of overlap.

To model heterogeneous execution, I study regions between synchronisation points (such as start, global reductions and finish), since total runtime is simply a sum of these. Each region is then broken up into groups of loops between ones that need halo exchanges: take a sequence of N loops, out of which $L < N$ need halo exchanges, at positions $1 \leq p_1, p_2, ..., p_L \leq N$, and for these loops, let $k_{p_l}^{i,inter}$ denote the time it takes to execute the interior of loop p_l by process i, and $k_{p_l}^{i,bndry}$ for the boundary. Thus, to determine $t_i^{fin}(l)$, the time elapsed until the full execution of loop l (a loop that requires halo exchanges) on process i since the last synchronisation point, we have:

$$t_i^{send}(l) = t_i^{fin}(l-1) + \sum_{j=p_{l-1}+1}^{p_l-1} k_j^i \tag{6.2}$$

util the time that process i sends its halo to its neighbours, after which executes the interior of loop p_l and starts receiving data:

$$t_i^{recv}(l) = t_i^{send}(l) + k_{p_l}^{i,inter} \tag{6.3}$$

and waits until it has received all halo data from its neighbours $N(i)$:

$$t_i^{wait}(l) = t_i^{recv}(l) + f\left(\max_{k \in N(i)}\left(t_k^{send}(l)\right) - t_i^{recv}\right) \tag{6.4}$$

where f returns 0 for negative values and the value itself for positive values. This is followed by the execution of the boundary to yield the total time elapsed:

$$t_i^{fin}(l) = t_i^{wait}(l) + k_{p_l}^{i,bndry} \tag{6.5}$$

which simply gives the total time for the current region by taking the maximum execution time until the completion of the last loop over all K processes:

DOI:10.15774/PPKE.ITK.2014.003

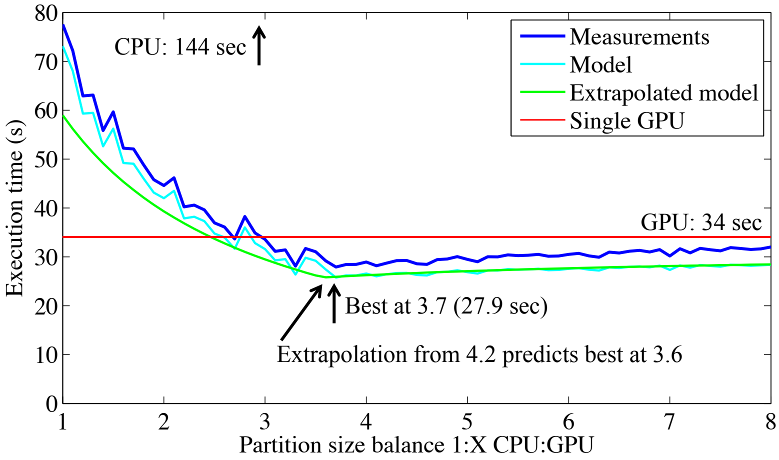


Figure 6.5. Hybrid execution of the Airfoil benchmark, using an NVIDIA Tesla K20c card and an Intel Xeon E5-2640 CPU with 6 cores. The extrapolation is from a balance of 4.2, based on the stand-alone runtime of the CPU (144 sec) and the GPU (34 sec): 144/34 = 4.2

$$t_{region} = \max_{i \in K} \left(t_i^{fin}(L) + \sum_{j=p_L+1}^{N} k_j^i \right). \tag{6.6}$$

Figure 6.4 shows a simplified example of a timing region where there is only one loop that requires a halo exchange, the last one. It highlights the communications between processes and shows that in case of severe load imbalance, process 3 may spend a lot of its time waiting for halo exchanges to arrive as well as at the end of timing regions, due to explicit synchronisation.

6.3.1 Performance evaluation

The hybrid heterogeneous execution scheme was applied to the Airfoil benchmark, described in 5.2, to utilise the GPU (an NVIDIA Tesla K20c) and the CPU (an Intel Xeon E5-2640 with 6 cores) in a development system. The former uses CUDA, the latter OpenMP and the two communicate through MPI - the specifics of how execution is organised on different hardware is discussed in the Chapter 7.

Figure 6.5 describes the results; running the benchmark on a single GPU gives an execution time of 34 seconds and running it on the CPU alone gives 144 seconds. When utilising both hardware, I use standard domain decomposition, facilitated by the ParMetis [89] graph partitioner, specifying a partition size balance as follows; the CPU partitions have a weight of 1, and the weight of GPU partitions can be set w_{gpu}. Thus the fraction of the total problem size that is assigned to the CPU is $1/(1 + w_{gpu})$ and to the GPU: $w_{gpu}/(1 + w_{gpu})$. Then, I carry out a set of measurements, exhaustively searching for the best value of w_{gpu}, between 1 and 8; the resulting execution times are shown with the blue

DOI:10.15774/PPKE.ITK.2014.003

line on Figure 6.5. The reason for the irregularity of the results is due to the non-linearity of the partitioner as well as the performance scaling of the CPU and the GPU as the problem size is changed, not system noise (the results are averaged from 3 runs). The best performance is achieved at a balance of 3.7, giving an execution time of 27.9 seconds, an 18% reduction over the execution time of the GPU alone (and a 5.2 speedup over the CPU alone).

Extracting information on the execution times of individual loops (k^i_j) that do not need halo exchanges and on ones that do ($k^{i,inter}_{pl}, k^{i,bndry}_{pl}$), I substitute these values back into the model, giving predictions displayed by the cyan line in Figure 6.5. While these consistently underestimate the actual execution time, due to the fact that MPI messaging latency is not accounted for, qualitatively it gives the correct results, following the measurements curve accurately, taking its minimum at the same partition balance, 3.7.

The model of course can be used to predict the correct partition balance without having to exhaustively search the parameter space. Thus, a reasonable assumption would be to first try a partition balance of $144/34 = 4.2$, which already gives a good performance improvement (28.7 sec), but feeding measurements into the model gives the prediction shown with the green line in Figure 6.5. This prediction shows the region where it is the CPU that is waiting for the GPU (less than 3.6) and where the GPU is waiting for the CPU (above 3.6), and gives the point where the least amount of waiting is involved at 3.6, which is less then 3% from the measured optimum of 3.7. Note that this extrapolation, since it is based on a single datapoint, assumes linear partitioning and performance behaviour, hence it gives a smooth prediction.

6.4 Summary and associated theses

In this chapter, I have built upon the OP2 abstraction to carry out my research addressing resiliency, locality and the utilisation of heterogeneous systems. My research involves transformations to execution patterns at a high level; without assuming specifics about the hardware the algorithm is going to execute on. First, I have presented a novel checkpointing algorithm, that based on the information provided through the OP2 API, can automatically create backups of the state space during execution, and in the event of a failure it can fast-forward and continue execution - all this without the user altering the application. Furthermore, I have shown that it is possible to find sub-optimal or optimal points during execution where the state space saved is minimal, using one of three ways; (1) near-optimal local decision, looking for a loop in which data is written to within a timeout, (2) by recording the execution history of loops, finding a repeating pattern and locating the minimal point in that, and (3) by reporting the minimum points to the user, after the fact, so that an OP2 API call may be inserted that triggers the checkpoint. Thus, as Thesis II.1 states: *I have designed and implemented a checkpointing method in the context of OP2 that can automatically locate points during execution where the state space is minimal, save data and recover in the event of a failure.*

I have designed and implemented a novel tiling algorithm for general unstructured grid computations defined through the OP2 API that concatenates the execution of subsequent computational loops, carrying out the data dependency analysis and forming an execution plan over a subset of the computational domain that fits in on-chip cache [9]. Thus, as Thesis II.2 states: *I gave an algorithm for redundant compute tiling in order to provide cache-blocking for modern architectures executing general unstructured grid algorithms, and implemented it in OP2, relying on run-time dependency analysis and delayed execution techniques.*

I have also investigated the issue of utilisation in heterogeneous systems, and designed a model to estimate the performance behaviour when a domain decomposition technique is used to share work between different architectures. I have implemented this execution strategy in OP2 and evaluated it on the Airfoil benchmark, showing the accuracy and the predicting capability of the model [4, 13] . As Thesis II.3 states: *I gave a new performance model for the collaborative, heterogeneous execution of unstructured grid algorithms where multiple hardware with different performance characteristics are used, and introduced support in OP2 to address the issues of hardware utilisation and energy efficiency.*

DOI:10.15774/PPKE.ITK.2014.003

Chapter 7

Mapping to Hardware with OP2

The key challenge addressed by the OP2 domain specific active library is the execution of unstructured grid algorithms based on high-level specifications on a range of heterogeneous architectures. Previous chapters discussed the abstraction defined by OP2 as well as the high-level transformation, in this chapter I discuss how OP2 facilitates the use of multiple levels of parallelism in a generic way, and subsequently my own research on how it maps to different architectures, programming languages and abstractions through a combination of back-end logic and code generation. By studying performance on both benchmark and real applications, I show that it is indeed possible to have an application code written once that is then easily maintainable and performance portable to a range of contrasting hardware, thereby delivering future-proofing to these applications - arguably the most important promise of domain specific languages. In Section 7.1 I briefly discuss how OP2 facilitates the use of multiple levels of parallelism, based on [16] and my own research. Section 7.2 presents my research into how execution is mapped to the GPU hardware, its SIMT programming abstraction, and the CUDA C and CUDA Fortran languages. Section 7.3 discusses mapping execution to modern CPUs and CPU-like architectures such as the Xeon Phi, by utilising the SMT and the SIMD programming abstraction with OpenMP and AVX. The discussion of results on distributed-memory supercomputers is postponed to the Appendix, it will analyse performance to demonstrate the scalability of OP2 when solving large, industrial problems.

7.1 Constructing multi-level parallelism

The basic assumption that OP2 makes is that the order in which elements in any given loop are iterated over does not affect the result, within machine precision. This makes it possible to organise parallelism in any way as long as data dependencies and race conditions are handled. To enable execution on today's hardware that support multiple levels of parallelism, we first create a highly parametrisable execution plan, that is still independent of the hardware being used. There are three levels of parallelism created and supported;

1. Distributed memory parallelism based on the Communicating Sequential Processes (CSP) abstraction.
2. Coarse-grained shared memory parallelism for levels where communication and synchronisation are expensive.
3. Fine-grained shared memory parallelism for levels where communication and synchronisation are cheap.

The first level, distributed memory parallelism, is built entirely into the backend, it is based on MPI and uses standard graph partitioning techniques (similar to ideas developed previously in OPlus [72, 80]) in which the domain is partitioned among the compute nodes of a cluster, and import/export halos are constructed for message-passing. OP2 utilises either of two well established parallel mesh partitioning libraries, ParMETIS [89] and PT-Scotch [90] to obtain high quality partitions.

The two-level shared memory design is motivated by several key factors. Firstly a single node may have different kinds of parallelism depending on the target hardware: on multi-core CPUs shared memory multi-threading is available with the possibility of each thread using vectorisation to exploit the capabilities of SSE/AVX vector units. On GPUs, multiple thread blocks are available with each block having multiple threads. Secondly, memory bandwidth is a major limitation on both existing and emerging processors. In the case of CPUs this is the bandwidth between main-memory and the CPU cores, while on the GPUs this is the bandwidth between the main graphics (global) memory and the GPU cores. Thus this design is motivated to reduce the data movement between memory and cores. Based on ideas from FFTW [91], OP2 constructs for each parallel loop an execution "plan" (`op_plan`) which breaks up the distributed-memory partition into blocks (or mini-partitions) for coarse-grained shared memory parallelism, and then works out possible data races within each of these blocks to enable fine-grained shared memory parallelism.

DOI:10.15774/PPKE.ITK.2014.003

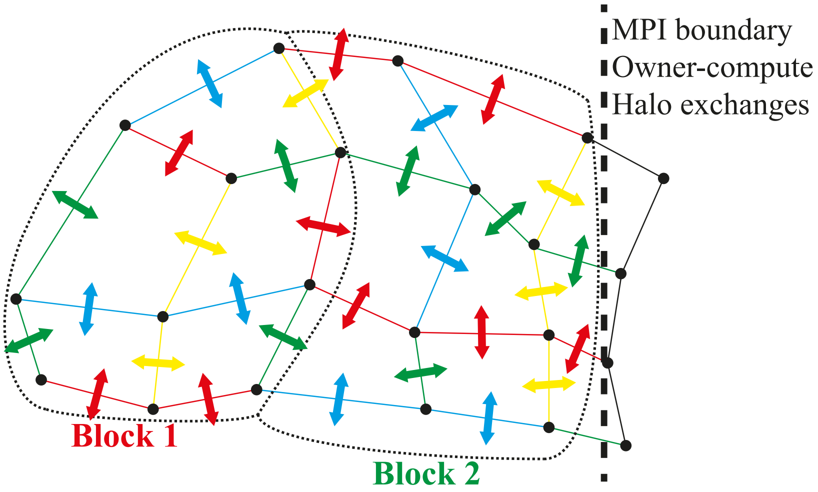


Figure 7.1. Handling data dependencies in the multi-level parallelism setting of OP2

7.1.1 Data Dependencies

One key design issue in parallelising unstructured mesh computations is managing data dependencies encountered when using indirectly referenced arrays [16]. For example, in a mesh with cells and edges, with a loop over edges updating cells a potential problem arises when multiple edges update the same cell.

At the higher distributed-memory level, we follow the OPlus approach [80, 72] in using an "owner compute" model in which the partition which "owns" a cell is responsible for performing the edge computations which will update it. If the computations for a particular edge will update cells in different partitions, then each of those partitions will need to carry out the edge computation. This redundant computation is the cost of this approach. However, we assume that the distributed-memory partitions are very large such that the proportion of redundant computation becomes very small. It is possible that one partition may need to access data which belongs to another partition; in that case a copy of the required data is provided by the other partition. This follows the standard "halo" exchange mechanism used in distributed memory message passing parallel implementations. These data dependencies classify set elements as either "interior", ones that do not depend on data owned by different processes, or "boundary", ones that do. OP2 implements latency hiding; it overlaps the computation over interior set elements with the communication of boundary data.

Within a shared-memory setting the size of the blocks can be very small, and so the proportion of redundant computations would be unacceptably large if we used the owner-compute approach. Instead we use the approach described in [16], in which we adopt the "colouring" idea used in vector computing [50], illustrated in Figure 7.1. The domain is

DOI:10.15774/PPKE.ITK.2014.003

broken up into blocks of elements, the blocks are coloured so that no two blocks of the same colour will update the same cell. This allows for parallel execution of blocks, with synchronisation between different colours. Key to the success of this approach is the fact that the number of blocks is very large and so even if 10-20 colours are required, there are still enough blocks of each colour to mitigate the cost of synchronisation and to ensure good load-balancing

On the level of fine-grained shared memory parallelism it is important to have data reuse during indirect accesses, but there may be conflict when trying to update the same cell. Here, we again use the colouring approach, assigning colours to individual edges so that no two edges of the same colour update the same cell, as illustrated in Figure 7.1. When incrementing data on cells, it is possible to first compute the increments for each edge, and then loop over the different edge colours applying the increments by colour, with synchronisation between each colour. This results in a slight loss of parallelism during the incrementing process, but permits data reuse.

Finally, in order to address the issue of serialisation on the fine level, I have introduced two additional colouring strategies that mitigate serialisation at the cost of increased irregularity. These strategies also enable the use of higher-level automatic parallelisation approaches, such as compiler auto-vectorisation, because execution can be formulated as a simple loop nest with the innermost loop being over elements of the same colour, which requires no synchronisation constructs. I present two new colouring approaches: the first permutes the execution of elements within blocks by colour (referred to as "block permute" and the second that creates a single level of colouring for all elements, and provides a permutation to execute them by colour (referred to as "full permute").

7.2 Mapping to Graphical Processing Units

In order to map execution to GPUs, I use the SIMT programming abstraction, which to some degree reflects the architectural specifics; execution is grouped into coarse-grained thread blocks which have practically no way of communicating and synchronising, these are assigned to different SMX units, and fine-grained threads within each thread block, they can communicate via the on-chip shared memory and synchronise relatively cheaply, these are assigned to the execution units within the SMX units. This aligns well with the multi-level parallelism scheme constructed in `op_plan`, blocks of the same colour are assigned to different thread blocks during a kernel launch, and a separate kernel is launched for each colour. These blocks are sized so that a set element is assigned to each thread,

DOI:10.15774/PPKE.ITK.2014.003

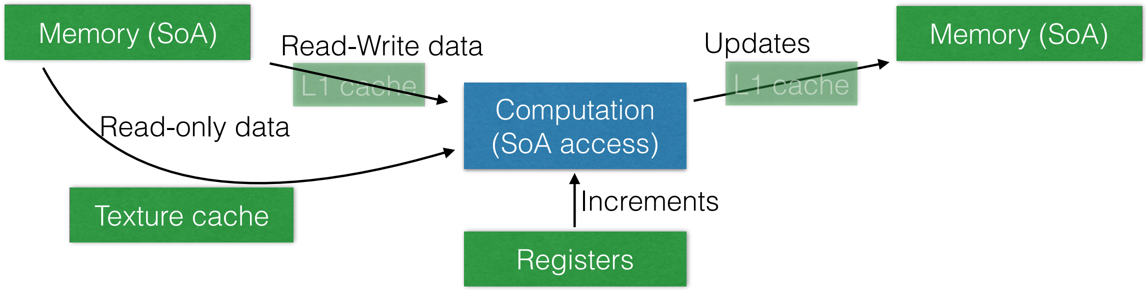


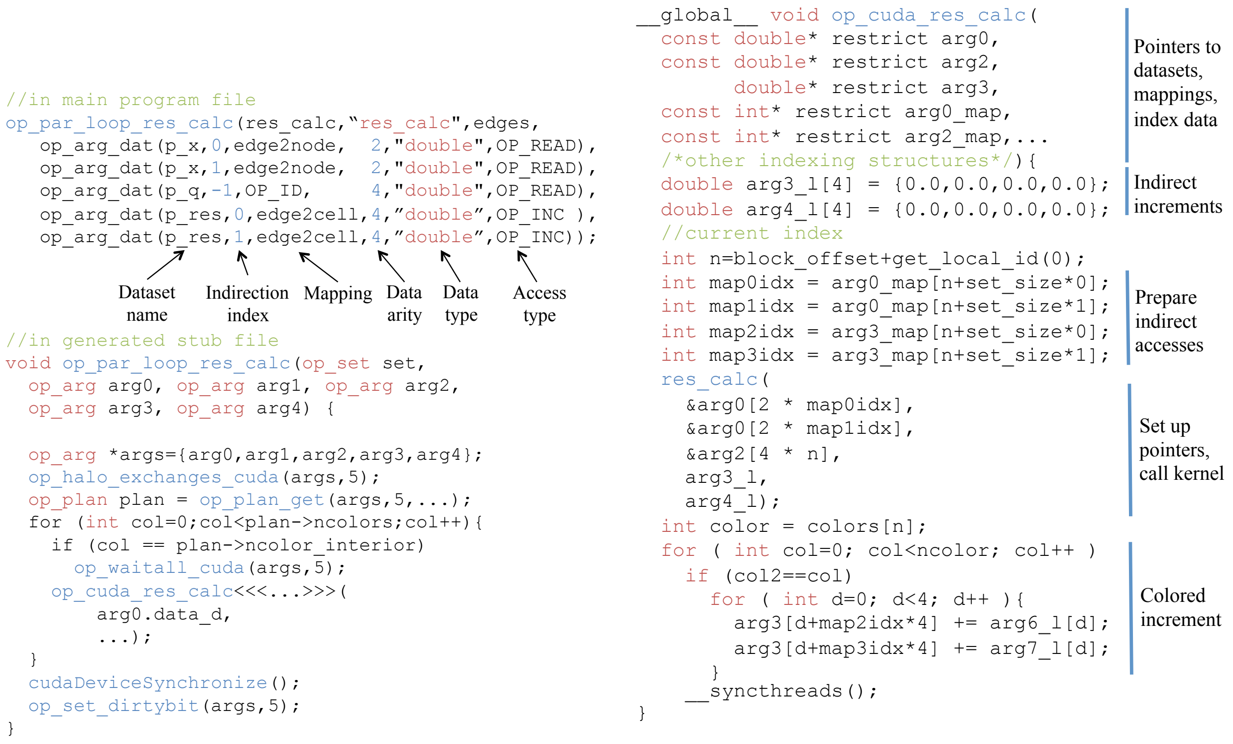
Figure 7.2. Organising data movement for GPU execution

increments are applied colour by colour with a block-level synchronisation call in-between.

To program the GPU, I use the CUDA programming language (and its C and Fortran variants), which defines computations and data movement from the perspective of a single thread using the SIMT abstraction. Due to the way the OP2 API is constructed, there is a separation between the iteration over the execution set and the execution of computations - leaving the responsibility of moving data and handing it to the user-defined kernel function to the library. Given the deep memory hierarchy and the assumption that the `op_plan` is created to give good data reuse, I construct the execution scheme described by Figure 7.2; multi-dimensional datasets may be transposed to a Structure of Arrays (SoA) layout, read-only data is transferred through the texture cache, data that is written or read-and-written is accessed directly from global memory, optionally cached in L1, and for datasets that are incremented, a register array is provided to store the increments in. These pointers are then handed to the user-defined kernel, once done, increments are applied to global memory colour by colour.

This mapping integrates seamlessly with OP2's MPI distributed memory execution; concurrently to the execution on the interior, I run a kernel that collects data on boundary vertices and then I asynchronously copy them to the host, handing it to the MPI backend to be sent off to neighbouring GPUs. The execution of blocks that contain boundary elements cannot begin until updated halo data from neighbours is received, therefore having queued the GPU kernels, the CPU does a blocking wait for data, uploads it and launches a GPU kernel which moves the data to its place, before launching the boundary GPU kernels.

The challenge of course is to fully automate a code generation process which implements this strategy, based on the high-level specification provided through the OP2 API. The example from the previous chapter (Figure 5.2b) is used to describe to illustrate the code generated for execution on the GPU, using CUDA; in Figure 7.3 the original call to the abstraction is transferred to a loop-specific implementation which constructs an `op_plan` and launches GPU kernels, colour by colour. These GPU kernels then arrange



```
//in main program file
op_par_loop_res_calc(res_calc,“res_calc",edges,
   op_arg_dat(p_x,0,edge2node,  2,"double",OP_READ),
   op_arg_dat(p_x,1,edge2node,  2,"double",OP_READ),
   op_arg_dat(p_q,-1,OP_ID,     4,"double",OP_READ),
   op_arg_dat(p_res,0,edge2cell,4,”double”,OP_INC ),
   op_arg_dat(p_res,1,edge2cell,4,”double”,OP_INC));
```

```
//in generated stub file
void op_par_loop_res_calc(op_set set,
  op_arg arg0, op_arg arg1, op_arg arg2,
  op_arg arg3, op_arg arg4) {

  op_arg *args={arg0,arg1,arg2,arg3,arg4};
  op_halo_exchanges_cuda(args,5);
  op_plan plan = op_plan_get(args,5,...);
  for (int col=0;col<plan->ncolors;col++){
    if (col == plan->ncolor_interior)
      op_waitall_cuda(args,5);
    op_cuda_res_calc<<<...>>>(
        arg0.data_d,
        ...);
  }
  cudaDeviceSynchronize();
  op_set_dirtybit(args,5);
}
```

(a) Host stub

```
__global__ void op_cuda_res_calc(
  const double* restrict arg0,
  const double* restrict arg2,
        double* restrict arg3,
  const int* restrict arg0_map,
  const int* restrict arg2_map,...
  /*other indexing structures*/){
  double arg3_l[4] = {0.0,0.0,0.0,0.0};
  double arg4_l[4] = {0.0,0.0,0.0,0.0};
  //current index
  int n=block_offset+get_local_id(0);
  int map0idx = arg0_map[n+set_size*0];
  int map1idx = arg0_map[n+set_size*1];
  int map2idx = arg3_map[n+set_size*0];
  int map3idx = arg3_map[n+set_size*1];
  res_calc(
    &arg0[2 * map0idx],
    &arg0[2 * map1idx],
    &arg2[4 * n],
    arg3_l,
    arg4_l);
  int color = colors[n];
  for ( int col=0; col<ncolor; col++ )
    if (col2==col)
      for ( int d=0; d<4; d++ ){
        arg3[d+map2idx*4] += arg6_l[d];
        arg3[d+map3idx*4] += arg7_l[d];
      }
    __syncthreads();
}
```

Pointers to datasets, mappings, index data
Indirect increments
Prepare indirect accesses
Set up pointers, call kernel
Colored increment

(b) CUDA kernel

Figure 7.3. Code generated for execution on the GPU with CUDA

the data movement and call the user-defined function.

The construction of this code relies on first parsing the call to the OP2 API (in Figure 7.3a); recording information regarding the number of arguments, and for each argument the name of the dataset accessed, the type of access (direct or indirect), the dimensionality of the dataset, its datatype and the access descriptor (read, write, increment). This is stored in a data structure that I then use to generate code similar to what is described in Figure 7.3 using a combination of string manipulation and pre-defined patterns for different cases. Figure 7.4 illustrates this parsing process and the data it generates. Once parsing is completed, the backend-specific code generator is invoked to produce code similar to the one shown in Figure 7.3, Figure 7.5 describes this process step by step. More details are available in [83].

7.2.1 Performance and Optimisations

Here, I carry out the performance analysis of the Airfoil benchmark, the Volna tsunami simulation code and the Hydra application on GPUs, applying optimisations through changes to the code generator or by using auto-tuning techniques, and showing that near-optimal performance is indeed achieved. Since the two applications are using different programming languages (Airfoil and Volna are written in C, Hydra in Fortran), I demon-

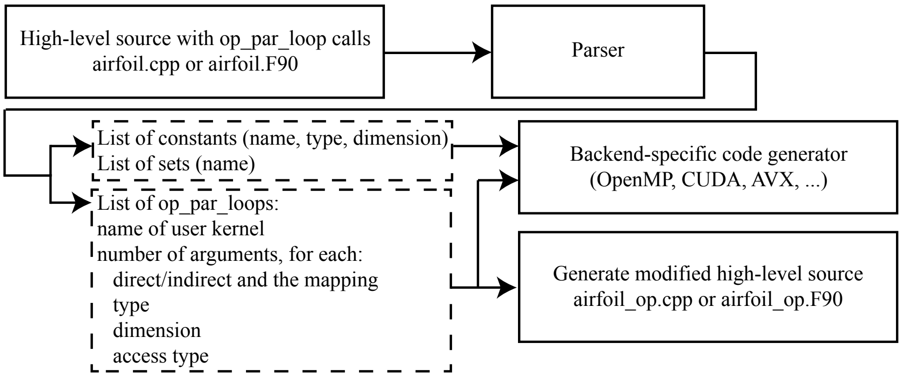

High-level source with op_par_loop calls
airfoil.cpp or airfoil.F90
Parser
List of constants (name, type, dimension)
List of sets (name)
List of op_par_loops:
name of user kernel
number of arguments, for each:
direct/indirect and the mapping
type
dimension
access type
Backend-specific code generator
(OpenMP, CUDA, AVX, ...)
Generate modified high-level source
airfoil_op.cpp or airfoil_op.F90

Figure 7.4. Parsing the high-level file containing `op_par_loop` calls

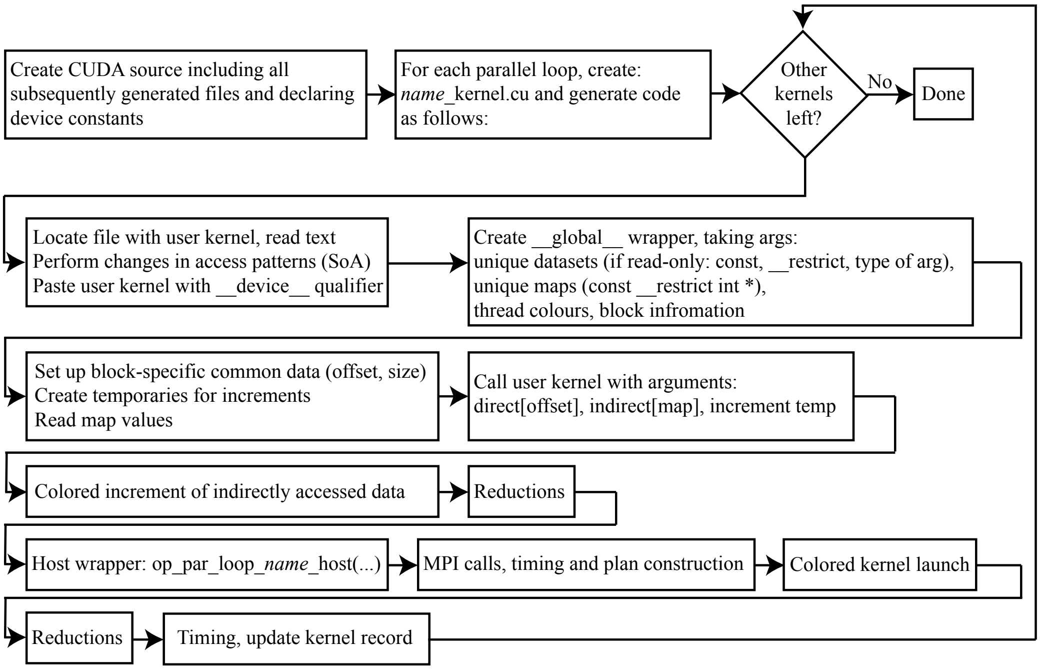

Create CUDA source including all subsequently generated files and declaring device constants
For each parallel loop, create: name_kernel.cu and generate code as follows:
Other kernels left?
No
Done
Locate file with user kernel, read text
Perform changes in access patterns (SoA)
Paste user kernel with __device__ qualifier
Create __global__ wrapper, taking args:
unique datasets (if read-only: const, __restrict, type of arg),
unique maps (const __restrict int *),
thread colours, block infromation
Set up block-specific common data (offset, size)
Create temporaries for increments
Read map values
Call user kernel with arguments:
direct[offset], indirect[map], increment temp
Colored increment of indirectly accessed data
Reductions
Host wrapper: op_par_loop_name_host(...)
MPI calls, timing and plan construction
Colored kernel launch
Reductions
Timing, update kernel record

Figure 7.5. Generating CUDA code based on the parsed data for execution on the GPU

strate the applicability of the library and the code generation to both languages. Tests are carried out on the Ruby development machine which contains two Tesla K20 GPUs and an SKA test node equipped with a Tesla K40 GPU - hardware specifications are presented in 7.1.

First, I investigate the performance of the Airfoil benchmark; it was written to be representative of larger applications, such as Hydra, but it is much smaller so that tests are easier to carry out. It consists of five parallel loops, two direct, three indirect as detailed in Table 5.1. GPU execution is compared to OP2's basic MPI backend on the Ruby development machine, detailed in Table 7.2. The GPU baseline uses the code generated for Fermi-generation GPUs, published in [2, 16], which used shared memory to stage indirectly accessed data, thereby explicitly improving data reuse. Figure 7.6 shows a moderate improvement over the old Fermi C2070 card when running the same code on the Kepler

DOI:10.15774/PPKE.ITK.2014.003

Table 7.1. GPU benchmark systems specifications

System	K20	K40
Architecture	Tesla K20c	Tesla K40
Clock frequency	0.71 GHz	0.87 GHz
Core count	2496	2880
Last level cache	1,.5MB	1.5MB
Peak bandwidth	208 GB/s	288 GB/s
Peak compute DP(SP)	1.17(3.52) TFLOPS	1.43 (4.29) TFLOPS
Stream bandwidth	172 GB/s	229 GB/s
D(S)GEMM throughput	1100(2610) GFLOPS	1420(3730) GFLOPS
FLOP/byte achieved DP(SP)	6.4(15.1)	6.2(16.3)

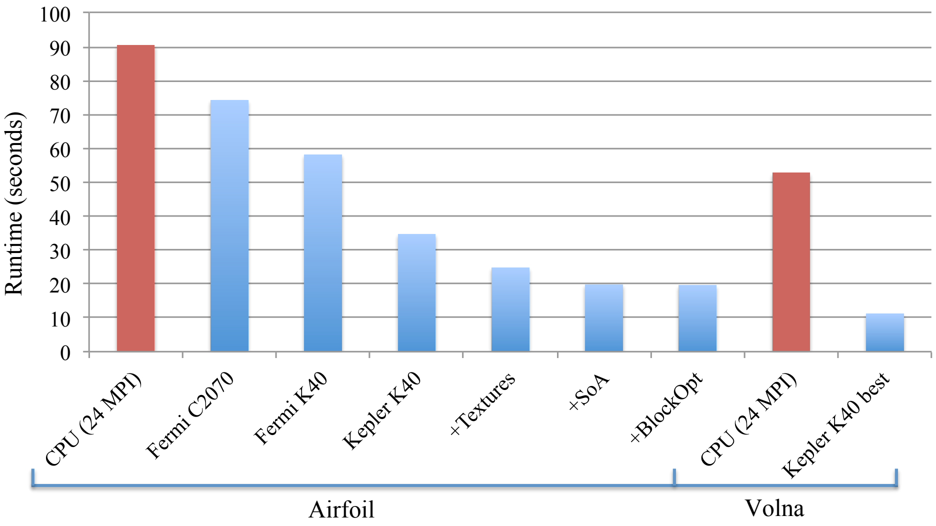


Figure 7.6. Performance of the Airfoil benchmark and the Volna simulation with optimisations on a Tesla K40 card

K40 card, which in part is because the amount of shared memory available didn't change between the two GPU generations, but the required amount of parallelism increased - therefore we eliminated the staging in shared memory by directly loading the data from global memory into SMXs, and rely on the increased L2 and texture cache size to speed up memory accesses that have spatial and temporal locality. The next bar in Figure 7.6 shows the performance of the Kepler-specific code without any further optimisations. By decorating pointers with `const __restrict` in C or `intent(in)` in Fortran, it is possible to give hints to the compiler that data accessed using those pointers should be brought through the read-only texture cache; the performance benefits are obvious, as shown by the fourth bar in Figure 7.6.

Running the generated CUDA code through the NVIDIA Visual Profiler reveals more opportunities for optimisations. It appears that the switch to directly loading data from global memory (without staging in shared memory) has made most parallel loops limited by bandwidth utilisation. Further investigation showed that a high amount of cache

DOI:10.15774/PPKE.ITK.2014.003

Table 7.2. Specifications of the Ruby development machine

System	Ruby
Node Architecture	2×Tesla K20c GPUs+ 2×6-core Intel Xeon E5-2640 2.50GHz
Memory/Node	5GB/GPU + 64GB
O/S	Red Hat Linux 6.3
Compilers	PGI 13.3, ICC 13.0.1, OpenMPI 1.6.4, CUDA 5.0 (Hydra), 5.5 (Airfoil, VOLNA)
Compiler flags	-O2 -xAVX -Mcuda=5.0,cc35

contention is caused by the default data layout of `op_dats`. This layout, called Array-of-Structs (AoS), was found to be the best layout for indirectly accessed data in my previous work [5] on Fermi GPUs. However without staging in shared memory it is damaging performance as many of the `op_dats` have a large number of components (dimensions) for each set element (typically related to the number of PDEs, in this case 4). Thus, adjacent threads are accessing memory locations that are far apart, resulting in high numbers of cache line loads (and evictions, since the cache size is limited). Therefore I have introduced the ability in OP2 to effectively transpose these datasets and use a Struct-of-Arrays (SoA) layout, so when adjacent threads are accessing the same data components, they have a high probability of being accessed from the same cache line. While the OP2 back-end can easily transpose existing data, it is also important to alter access patterns during the execution of the user kernel to also use the SoA layout; this is facilitated by parsing its function signature to determine the names of variables that use SoA, and then subsequently swapping all array accesses involving those variables with SoA accesses. This optimisation resulted in an increase of about 20% to the single GPU performance as shown in Figure 7.6.

To improve performance further, two other aspects of GPU performance are investigated. The goal is to allow as many threads to be active simultaneously so as to hide the latency of memory operations. The first consideration is to limit the number of registers used per thread. A GPU's SM can hold at most 2048 threads at the same time, but it has only a fixed number of registers available (65k 32-bit registers on Kepler K20 GPUs). Therefore kernels using excessive amounts of registers per thread decrease the number of threads resident on an SM, thereby reducing parallelism and performance. Limiting register count (through compiler flags) can be beneficial to occupancy and performance if the spilled registers can be contained in the L1 cache. Thus for the most time-consuming loops, I have manually adjusted register count limitations so as to improve occupancy.

The second consideration is to adjust the number of threads per block. Thread block

Table 7.3. Bandwidth (BW - GB/s) and computational (Comp - GFLOP/s) throughput of the Airfoil benchmark in double precision on the 2.8M cell mesh and on the Volna simulation, running on the K40 GPU

Kernel	CUDA		
	Time	BW	Comp
save_soln	0.80	230	14
adt_calc	2.6	116	133
res_calc	10.8	62	75
bres_calc	0.09	32	5
update	3.22	235	29
RK_1	0.87	198	15
RK_2	0.72	242	24
compute_flux	3.21	101	309
numerical_flux	1.14	120	17
space_disc	1.92	73	31

size not only affects occupancy, but also has non-trivial effects on performance due to synchronisation overheads, cache locality, etc. that are difficult to predict. I have used auto-tuning techniques described in a previous work [5] to find the best thread block size for different parallel loops, and stored them in a look-up table. This optimisation can be carried out once for different hardware, and performance is unlikely to vary significantly when solving different problems. Together with the all the other optimisation, switching from the Fermi architecture to the Kepler architecture and the new K40 GPU gave 3.75× performance improvement, and it is 4.6× faster than the baseline CPU implementation, running on Ruby.

Performance breakdowns for each loop in Airfoil and Volna are shown in Table 7.3; given the theoretical computational performance of the K40 card, it is clear that none of them are bound by the throughput of computations, but the direct loops show bandwidth utilisation very close to the theoretical maximum. The loops `adt_calc` and `numerical_flux` also show relatively high bandwidth utilisation for a kernel with a large number of indirect accesses, but the loops `res_calc` and `space_disc` are hit by the indirect updating of data, which causes serialisation and a loss in parallelism.

Since one of the major limiting factors is this kind of serialisation when indirectly updating data, I have evaluated the performance of the two additional execution and colouring schemes that I have introduced in Section 7.1.1, and applied them to `res_calc` in the Airfoil benchmark. In case of both new approaches, any regularity of execution is lost, because set elements are listed in a permutation table, therefore any data that was accessed directly with the "original" execution scheme is now accessed indirectly (although in case of Airfoil and `res_calc` this is only the mapping from edges to cells

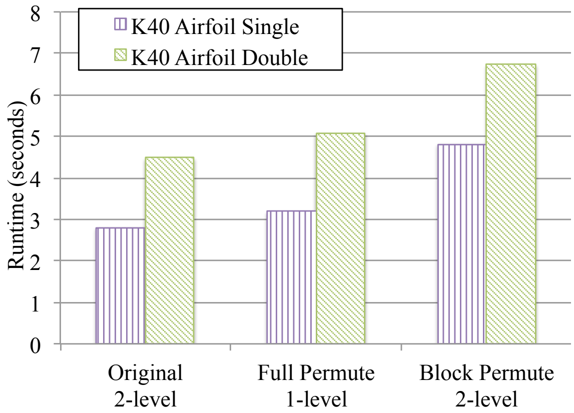


Figure 7.7. Performance of different colouring approaches on the GPU, on the Airfoil benchmark

or nodes, all other data was already indirectly accessed). Another important implication of these colouring schemes is that no data is shared between set elements of the same colour (otherwise race conditions would occur) - in case of the “full permute” approach it is unlikely that any data will remain in cache between the execution of elements of different colour, therefore the same data will be moved repeatedly, while with the “block permute” approach it is possible to have data reuse because the number of elements in a block is much lower. Figure 7.7 presents the results - it is clear that the original two-level colouring approach is superior, and that neither of the two other approaches achieves good data reuse. The “block permute” approach is further hit by the increased complexity and irregularity, despite the fact that race conditions were avoided.

As it can be seen, a range of low-level features had to be taken into account and a significant re-evaluation of the GPU optimisations had to be done to gain optimal performance even when going from one generation of GPUs by the same vendor/designer to the next. In this case the previous optimisations implemented for the Fermi GPUs had to be considerably modified to achieve good performance on the next generation Kepler GPUs. However, as the code was developed under the OP2 framework, radical changes to the parallel code could be easily implemented. In contrast, a directly hand-ported application would cause the application programmer significant difficulties to maintain performance for each new generation of GPUs, not to mention new processor architectures. Given these results, I am confident that given the irregular nature of these computations and memory accesses, near-optimal performance is achieved for this benchmark.

The benefit of using a library such as OP2, is that these optimisations can be introduced and tested on benchmark applications and then immediately transferred to large-scale production applications, such as Rolls-Royce Hydra, although in this case due to the

DOI:10.15774/PPKE.ITK.2014.003

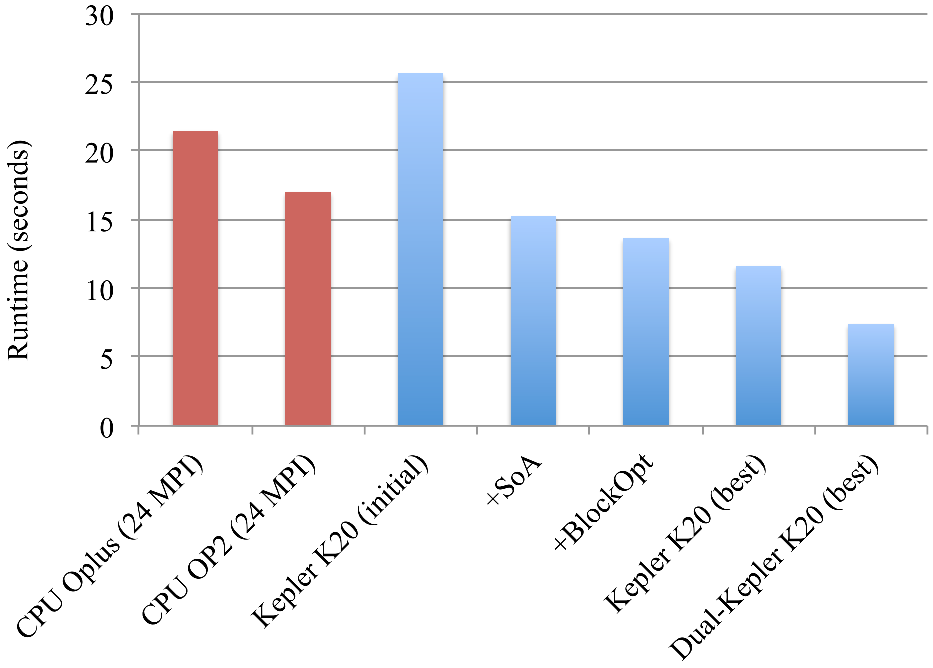


Figure 7.8. OP2 Hydra's GPU performance (NASA Rotor 37, 2.5M edges, 20 iterations)

change in language (C to Fortran), some keywords had to be altered and some new text had to be added to the code generator. Since Hydra is such a massive application, in-depth analysis would be out of scope for this work, I only give a broad overview. For the rest of this work whenever I reference to the performance of the *OPlus* version, I refer to the original implementation of Hydra that uses OPlus and distributed memory parallelisation [72].

In Hydra, I use Fortran to C bindings to call functions in the OP2 back-end and to connect C pointers to Fortran pointers. Following the same sequence of optimisations, the initial code that only uses the texture cache optimisations runs on a single GPU about 45% slower than the best CPU performance on Ruby (2 CPUs, 24 MPI processes) as shown in Figure 7.8. Since for Hydra, the dimensionality of datasets is even larger than for Airfoil (up to 25), the importance of transposing the data layout to a Structure of Arrays format is even greater. Subsequently, by limiting the number of registers that some of the most time-consuming loops use and by auto-tuning the thread block size, additional performance can be gained.

In the case of several Hydra loops that are bandwidth-intensive and do little computation, the thread colouring technique (that avoids write conflicts) introduces a significant branching overhead, due to adjacent edges (mapped to adjacent threads) having different colours, and execution over colours getting serialised. The negative effect on performance

Table 7.4. Hydra single GPU performance:
NASA Rotor 37, 2.5M edges, 20 iterations

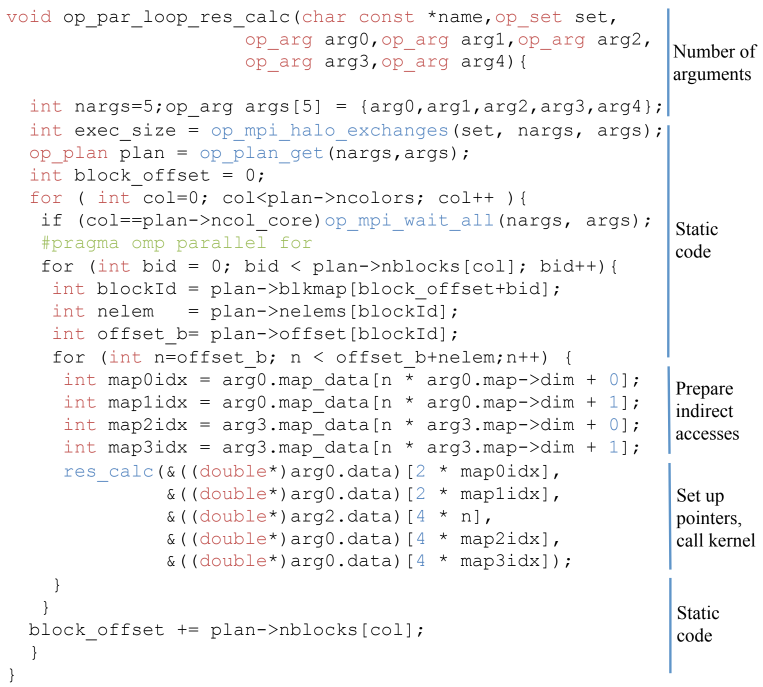
Loop	Time (sec)	GB/sec	% runtime
`accumedges`	2.07	13.87	19.30
`edgecon`	2.46	33.55	22.87
`ifluxedge`	1.03	71.90	9.59
`invjacs`	0.41	25.94	3.85
`srck`	0.34	108.09	3.20
`srcsa`	0.34	121.72	3.15
`updatek`	0.54	115.91	5.01
`vfluxedge`	2.32	53.82	21.62
`volap`	0.23	142.85	2.14
`wffluxedge`	0.11	31.55	1.00
`wvfluxwedge`	0.08	25.69	0.77

can be mitigated by sorting edges in blocks by colour, ensuring that threads in a warp have the same colour (or at least fewer colours), which in turn reduces branching within the warps. However, this solution degrades the access patterns to directly accessed data in memory, as adjacent threads are no longer processing adjacent set elements. Thus, I only apply it to the Hydra loops that are limited by branching overhead. The final best runtime on a single K20 GPU and on both the K20 GPUs on Ruby is presented in the final two bars of Figure 7.8. A single K20 GPU achieves about 1.8× speedup over the original OPlus version of Hydra on Ruby and about 1.5× over the MPI version of Hydra with OP2. However, considering the full capabilities of a node, the best performance on Ruby with GPUs (2 GPUs with MPI) gives a 2.34× speedup over the best OP2 runtime with CPUs (2 CPUs, 24 MPI processes) and about 2.89× speedup over the best OPlus runtime with CPUs (2 CPUs, 24 MPI processes). Table 7.4 notes the achieved memory bandwidth utilisation of some of the top kernels on a K20 GPU. A majority of the most time consuming loops achieve 20 - 50 % of the advertised peak bandwidth of 208 GB/s on the GPU [38]. Bandwidth utilisation is particularly significant during the direct loops `srck`, `updatek` and `volapf`.

These results show that the mapping to GPU architectures I introduced in this section indeed yields near-optimal performance on both the Airfoil benchmark and the Hydra application, and that it is possible to deploy optimisations through the code generation technique across hundreds of computational loops - without changes to the user application.

7.3 Mapping to CPU-like architectures

The most commonly used parallel programming abstractions are the communicating sequential processes (CSP) and the simultaneous multithreading (SMT); the former com-



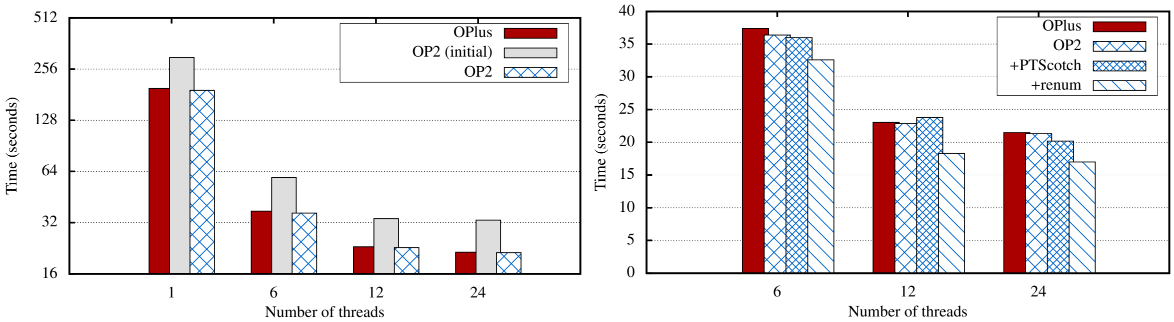
```
void op_par_loop_res_calc(char const *name,op_set set,
                     op_arg arg0,op_arg arg1,op_arg arg2,
                     op_arg arg3,op_arg arg4){

  int nargs=5;op_arg args[5] = {arg0,arg1,arg2,arg3,arg4};
  int exec_size = op_mpi_halo_exchanges(set, nargs, args);
  op_plan plan = op_plan_get(nargs,args);
  int block_offset = 0;
  for ( int col=0; col<plan->ncolors; col++ ){
   if (col==plan->ncol_core)op_mpi_wait_all(nargs, args);
   #pragma omp parallel for
   for (int bid = 0; bid < plan->nblocks[col]; bid++){
    int blockId = plan->blkmap[block_offset+bid];
    int nelem   = plan->nelems[blockId];
    int offset_b= plan->offset[blockId];
    for (int n=offset_b; n < offset_b+nelem;n++) {
     int map0idx = arg0.map_data[n * arg0.map->dim + 0];
     int map1idx = arg0.map_data[n * arg0.map->dim + 1];
     int map2idx = arg3.map_data[n * arg3.map->dim + 0];
     int map3idx = arg3.map_data[n * arg3.map->dim + 1];
     res_calc(&((double*)arg0.data)[2 * map0idx],
              &((double*)arg0.data)[2 * map1idx],
              &((double*)arg2.data)[4 * n],
              &((double*)arg0.data)[4 * map2idx],
              &((double*)arg0.data)[4 * map3idx]);
    }
   }
  block_offset += plan->nblocks[col];
  }
}
```

Figure 7.9. Example of code generated for MPI+OpenMP execution

pletely isolates threads effectively simulating a distributed memory environment but the latter relies on the fact that they share caches and the system memory. In the case of both abstractions, communication and synchronisation are regarded expensive although for SMT only the synchronisation is explicit. Thus, I will use MPI, discussed in Section 7.1, for distributed memory parallelism and OpenMP for coarse-grained shared memory parallelism. For OpenMP, an `op_plan` is constructed, and blocks of the same colour are assigned to the available threads in the system, these iterate through the set elements in their respective blocks sequentially, therefore no fine-grained parallelism is used. Since the construction of the `op_plan` and the MPI halo exchanges are implemented in the back-end, the generated code is fairly simple, Figure 7.9 shows code for MPI+OpenMP, but when only MPI is used, the code is slightly simplified and no `op_plan` is required. The process of code parsing and code generation is very similar to what I described in Section 7.2, therefore for the OpenMP example I do not go into further details.

Since a detailed study of Airfoil's MPI and OpenMP performance was already presented in [16], I immediately proceed to investigating Hydra. Figure 7.10a presents the performance of Hydra with both OPlus and OP2 on up to 12 cores (and 24 SMT threads) on the Ruby single node system using the message passing (MPI) parallelisation. This is a like-for-like comparison where the same mesh is used by both versions. The partitioning routine used in both cases is a recursive coordinate bisection (RCB) mesh partitioning [92].



(a) OPlus vs OP2 (MPI only)

(b) OPlus vs OP2 (with PTSotch and renumbering)

Figure 7.10. Single node performance on Ruby (NASA Rotor 37, 2.5M edges, 20 iterations)

Initial performance was poor, up to 50% slower than the original OPlus version. This proved to be due to the compiler's inability to effectively inline the user function as well as a Fortran-specific issue having to do with how arrays are represented and arguments are passed internally [93]; this was resolved by first calling a wrapper function that uses a cheaper and more efficient indexing scheme, which in turn iterates over the execution set, calling the user kernel. Clearly, doing this by hand would have been infeasible for an application the size of Hydra, but since I have developed a code generation technique to produce the code shown in Figure 7.9, applying this workaround was only a matter of changing the code generator. Results show that the performance of the original hand-coded Hydra application can be matched; this is hugely important because it shows that a domain specific language that uses a high-level abstraction does not sacrifice performance in exchange for generality.

A number of new features implemented in OP2 allow further improvements for the MPI parallelisation. Firstly, the OP2 design allows the underlying mesh partitioner for distributing the mesh across MPI processes to be changed to state-of-the-art unstructured mesh partitioners such as ParMetis [89] or PTScotch [90]. Secondly, I have added the ability to OP2 to optimise the ordering or numbering of mesh elements in an unstructured mesh. The renumbering of the execution set and related sets that are accessed through indirections has an important effect on performance [94]: the efficiency of pre-fetch engines and cache locality can be improved by making sure that data accessed by elements which are executed consecutively are close, so that data and cache lines are reused. I have added a multi-set reordering algorithms to OP2 that can be called to convert the input data meshes based on the Gibbs-Poole-Stockmeyer algorithm in Scotch [90]. The results in Figure 7.10b show the effect of the above two features. The use of PTScotch resulted in about 8% improvement over the recursive coordinate bisection partitioning, and

DOI:10.15774/PPKE.ITK.2014.003

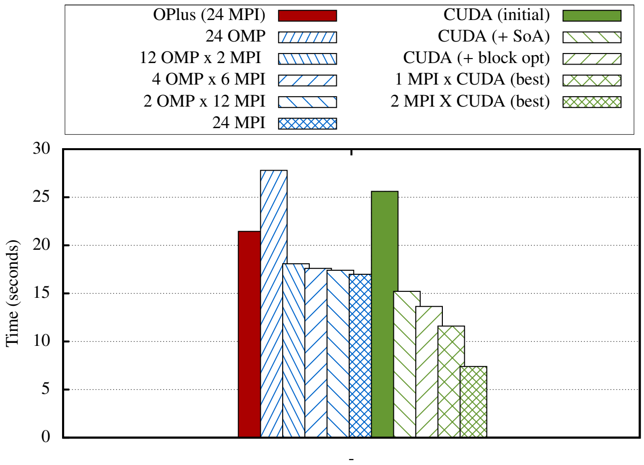


Figure 7.11. OP2 Hydra Multi-core/Many-core performance on Ruby (NASA Rotor 37, 2.5M edges, 20 iterations) with PTScotch partitioning

renumbering resulted in about 17% gains for Hydra solving the NASA Rotor 37 mesh. Overall, partitioning with PTScotch gave marginally better performance than ParMetis (not shown here). In the results presented in the rest of this work I make use of OP2's mesh renumbering capability, unless stated otherwise.

Figure 7.11 presents the runtime performance of the OpenMP parallel back-end. The experiments varied from running a fully multi-threaded version of the application (OpenMP only), to a heterogeneous version using both MPI and OpenMP. We see that executing Hydra with 24 OpenMP threads (i.e. OpenMP only) resulted in significantly poorer performance than when using only the MPI parallelisation on this two socket CPU node. We have observed similar performance with the Airfoil CFD benchmark code [2]. The causes for MPI outperforming OpenMP on SMPs/CMPs have been widely discussed in literature. These mainly consist of the non-uniform memory access issues (NUMA [95, 36]), and thread creation and tear-down overheads [96]. In my case, an additional cause may also be the reduced parallelism due to colouring for blocks that avoid data races. The hybrid MPI+OpenMP parallelisation provided better performance but was still about 10% slower than the pure MPI version; again the overhead of shared-memory multi-threading techniques may be causing performance bottlenecks. To explore the causes further, in Figure 7.12 and Table 7.5 I present the variation in the runtime and the number of block colours for different block sizes for a number of combinations of MPI processes and OpenMP threads. The number of colours and blocks per colour are automatically output by OP2 for each `op_par_loop` and I have presented here a range of numbers taken from the most time consuming loops in Hydra.

With the shared memory parallelism execution scheme employed by OP2, I reason that the size of the block determines the amount of work that a given thread carries

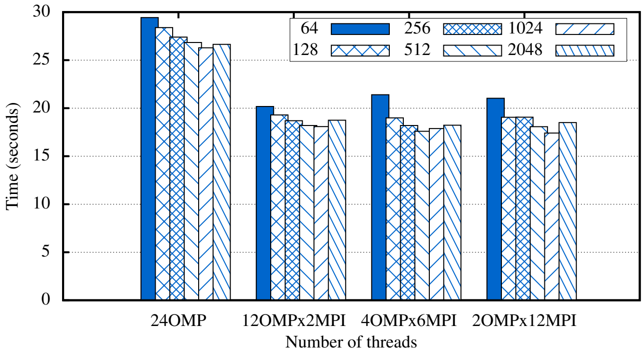


Figure 7.12. Hydra OpenMP and MPI+OpenMP performance for different block sizes on Ruby (NASA Rotor 37, 2.5M edges, 20 iterations) with PTScotch partitioning

Table 7.5. Hydra single node performance, Number of blocks (nb) and number of colours (nc) (K = 1000): 2.5M edges, 20 iterations

	OMP × MPI							
Block	24		12×2		4×6		2×12	
size	nb	nc	nb	nc	nb	nc	nb	nc
	(K)		(K)		(K)		(K)	
64	39	14	20	16	6.8	16	3	16
128	20	15	10	17	3.4	17	1.7	15
256	10	14	5	14	1.7	15	0.8	14
512	5	12	2.5	14	0.8	14	0.4	12
1024	2.5	10	1	11	0.4	11	0.2	11
2048	1	9	0.6	11	0.2	11	0.1	10

out uninterrupted. The bigger it is, the higher data reuse within the block with better cache and prefetch engine utilisation. At the same time, some parallelism is lost due to the coloured execution. In other words, only those blocks that have the same colour can be executed at the same time by different threads, with an implicit synchronisation step between colours. This makes the execution scheme prone to load imbalances, especially when the number of blocks with a given colour is comparable to the number of threads that are available to execute them.

The above two causes have given rise to the runtime behaviour seen with each MPI and OpenMP combination in Figure 7.12. The average number of colours (nc) and the average number of blocks (nb) in Table 7.5 supports this reasoning where, overall I see a reduction in runtime as the block size is increased but only until there are enough blocks per colour (nb/nc) to achieve good load balancing in parallel. I also experimented by using OpenMP's static and dynamic load balancing functionality but did not obtain any significant benefits as all the blocks executed (except for the very last one) have the same size.

Table 7.6 details the achieved bandwidth for the most time consuming loops of Hydra with OpenMP (6 MPI × 4 OMP). These loops together take up about 90% of the

DOI:10.15774/PPKE.ITK.2014.003

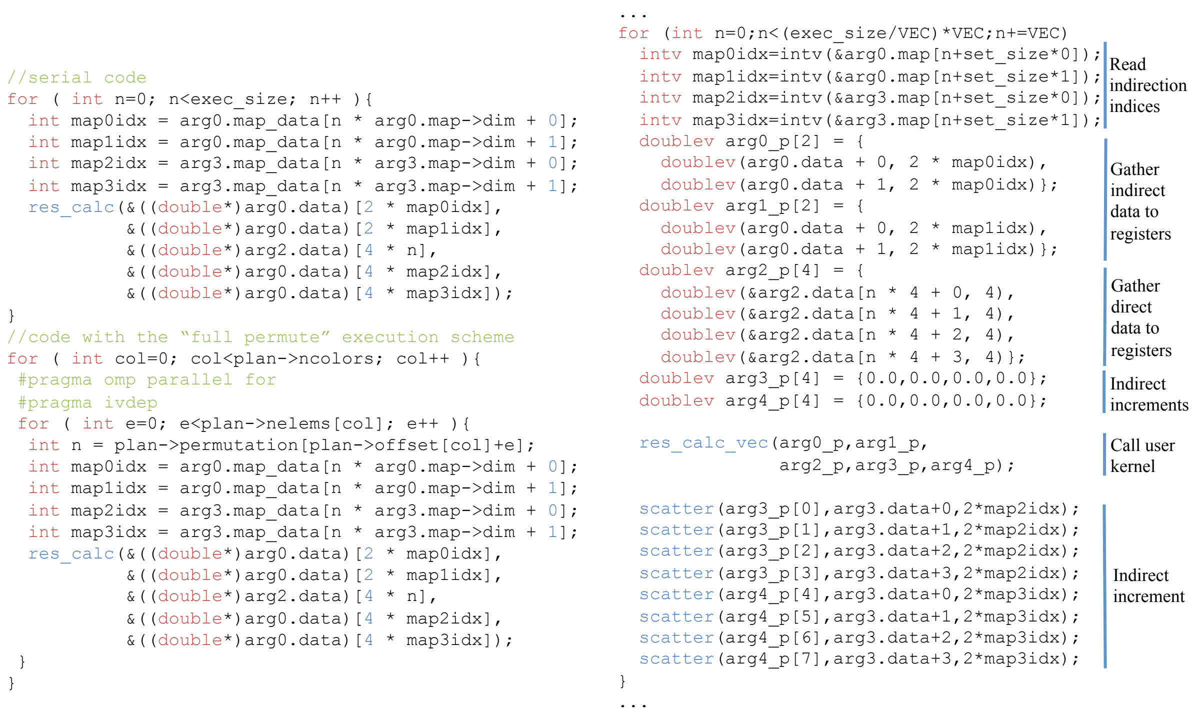
Table 7.6. Hydra single node performance, 6 MPI x 4 OMP with PTScotch on Ruby: 2.5M edges, 20 iterations

Loop	Time (sec)	GB/sec	% runtime
`accumedges`	1.46	28.57	7.99
`edgecon`	2.00	58.40	10.95
`ifluxedge`	1.88	48.88	10.32
`invjacs`	0.16	67.37	0.90
`srck`	0.47	81.72	2.57
`srcsanode`	1.34	31.62	7.38
`updatek`	0.94	68.67	5.14
`vfluxedge`	7.05	16.11	38.70
`volapf`	0.47	72.19	2.57
`wffluxedge`	0.26	25.24	1.41
`wvfluxedge`	0.26	16.81	1.41

total runtime. The achieved bandwidths are reported through code generated by OP2 for recording performance statistics. The bandwidth figure was computed by counting the useful amount of data bytes transferred from/to global memory during the execution of a parallel loop and dividing it by the runtime of the loop. The achieved bandwidth by a majority of the key loops on Ruby is over or close to 60% of the advertised peak bandwidth of the Sandy Bridge processors (2×42.6 GB/s). The best performing loops, such as `srck, updatek` and `volapf`, are direct loops that only access datasets defined on the set being iterated over, hence they achieve a very high fraction of the peak bandwidth, due to trivial access patterns to memory. Indirect loops use mapping tables to access memory, and they often require coloured execution in order to avoid data races; both of these factors contribute to lower bandwidth achieved by loops such as `accumedges, ifluxedge` and `edgecon`. Additionally some loops are also compute and control-intensive, such as `vfluxedge`. Finally, loops over boundary sets, such as `wvfluxedge`, have highly irregular access patterns to datasets and generally much smaller execution sizes, leading to a lower utilisation of resources. My experiments also showed that the trends on achieved bandwidth utilisation remain very similar to the observed results from previous work [16, 10] for the Airfoil benchmark application.

7.3.1 Vectorising unstructured grid computations

Intel provides several programming approaches to achieve vectorisation - the use of the SIMD execution units - the most trivial being the auto-vectorisation features implemented in the Intel Compilers; this takes loops that iterate over arrays executing a “kernel function" and vectorises them. However, it has some limitations, for example a kernel function with a non-trivial execution path, a loop-carried dependency or indirect memory accesses



```
//serial code
for ( int n=0; n<exec_size; n++ ){
  int map0idx = arg0.map_data[n * arg0.map->dim + 0];
  int map1idx = arg0.map_data[n * arg0.map->dim + 1];
  int map2idx = arg3.map_data[n * arg3.map->dim + 0];
  int map3idx = arg3.map_data[n * arg3.map->dim + 1];
  res_calc(&((double*)arg0.data)[2 * map0idx],
           &((double*)arg0.data)[2 * map1idx],
           &((double*)arg2.data)[4 * n],
           &((double*)arg0.data)[4 * map2idx],
           &((double*)arg0.data)[4 * map3idx]);
}
//code with the “full permute” execution scheme
for ( int col=0; col<plan->ncolors; col++ ){
 #pragma omp parallel for
 #pragma ivdep
 for ( int e=0; e<plan->nelems[col]; e++ ){
  int n = plan->permutation[plan->offset[col]+e];
  int map0idx = arg0.map_data[n * arg0.map->dim + 0];
  int map1idx = arg0.map_data[n * arg0.map->dim + 1];
  int map2idx = arg3.map_data[n * arg3.map->dim + 0];
  int map3idx = arg3.map_data[n * arg3.map->dim + 1];
  res_calc(&((double*)arg0.data)[2 * map0idx],
           &((double*)arg0.data)[2 * map1idx],
           &((double*)arg2.data)[4 * n],
           &((double*)arg0.data)[4 * map2idx],
           &((double*)arg0.data)[4 * map3idx]);
 }
}
```

(a) Code for serial execution and auto-vectorisation

```
...
for (int n=0;n<(exec_size/VEC)*VEC;n+=VEC)
  intv map0idx=intv(&arg0.map[n+set_size*0]);
  intv map1idx=intv(&arg0.map[n+set_size*1]);
  intv map2idx=intv(&arg3.map[n+set_size*0]);
  intv map3idx=intv(&arg3.map[n+set_size*1]);
  doublev arg0_p[2] = {
    doublev(arg0.data + 0, 2 * map0idx),
    doublev(arg0.data + 1, 2 * map0idx)};
  doublev arg1_p[2] = {
    doublev(arg0.data + 0, 2 * map1idx),
    doublev(arg0.data + 1, 2 * map1idx)};
  doublev arg2_p[4] = {
    doublev(&arg2.data[n * 4 + 0, 4),
    doublev(&arg2.data[n * 4 + 1, 4),
    doublev(&arg2.data[n * 4 + 2, 4),
    doublev(&arg2.data[n * 4 + 3, 4)};
  doublev arg3_p[4] = {0.0,0.0,0.0,0.0};
  doublev arg4_p[4] = {0.0,0.0,0.0,0.0};

  res_calc_vec(arg0_p,arg1_p,
               arg2_p,arg3_p,arg4_p);

  scatter(arg3_p[0],arg3.data+0,2*map2idx);
  scatter(arg3_p[1],arg3.data+1,2*map2idx);
  scatter(arg3_p[2],arg3.data+2,2*map2idx);
  scatter(arg3_p[3],arg3.data+3,2*map2idx);
  scatter(arg4_p[4],arg3.data+0,2*map3idx);
  scatter(arg4_p[5],arg3.data+1,2*map3idx);
  scatter(arg4_p[6],arg3.data+2,2*map3idx);
  scatter(arg4_p[7],arg3.data+3,2*map3idx);
}
...
```

(b) Code with vector intrinsics

Figure 7.13. Code generated for serial and parallel CPU execution

usually does not get auto-vectorised, because the compiler first and foremost has to produce correct code. Since access patterns in unstructured grid computations are irregular, this class of applications does not lend itself well to auto-vectorisation over computations carried out on different set elements; data races cannot be determined statically. Since previously I haven't discussed the use of vector units, in this section I present research into ways of achieving vectorised execution, giving a special attention to programmability considerations.

My experiments have shown that the primary obstacle to getting auto-vectorisation is the scatter-gather nature of memory accesses; for vectorised execution, the values of adjacent set elements have to be packed next to each other. Figure 7.13a shows a code snippet with a typical gather-type memory access that occurs when iterating over edges, accessing data on different sets. Clearly the simple serial code shown in Figure 7.13a cannot be parallelised due to potential race conflicts. First I evaluated the auto-vectorisation capabilities of the compiler, by using execution schemes that guarantee the independence of different loop iterations; the "full permute" and the "block permute" approaches. This way, it is possible to add a compiler pragma (`#pragma ivdep`) in front of the innermost `for` loop over independent set elements, thereby telling the compiler that it *valid* to vectorise the loop. In practise however, this rarely led to vectorisation when compiling for the CPU, but when compiling for the Xeon Phi, simpler kernels with no internal control flow did vectorise (in case of Airfoil, `adt_calc` and `res_calc`).

By using vector types, it is possible to explicitly vectorise the user kernel, and this approach only requires changing scalar types to vector types in the user kernel, which in most cases is a matter of a simple find-and-replace during code generation. While operators are not defined for these types, the Intel Compiler includes a header file that defines container classes for these types, overloading some operators. By extending these classes it is possible to maintain the original simple arithmetic expressions in the user kernels, but instead of scalars they will now operate on vectors. Furthermore, using preprocessor macros, I have shown that it possible to have a single source code generated for different vector lengths, and select one at compile-time. An important obstacle is the lack of support for branching in kernels; without a compiler technology I cannot support conditionals through operator overloading. Therefore the user has to alter conditional code to use $select()$ instructions instead, which can be supported through function overloading. While this was easy to apply to the Airfoil benchmark, this is by no means a generic and flexible solution to this problem.

A range of load and store instructions are implemented that support aligned addresses, strided gather/scatter or mapping-based gather/scatter operations, which enables us to utilise different instructions on different hardware - for example the IMCI (Initial Many Core Instructions - Xeon Phi) has a gather intrinsic. In the case of mapping-based scatter operations it is necessary to avoid data races; while in case of the GPU I use coloured updates, this proved to be inefficient on the Xeon Phi (which supports masked scatter operations), therefore I simply sequentially scatter data from the vector container.

Code resulting from the use of vector containers is rather verbose, and results in a much higher number of source lines due to explicit packing and unpacking of data. Furthermore, the iteration range for any given thread where vectorisation can take place must be divisible by the vector length, in order to support aligned load instructions and fully packed vectors, which is not always the case (especially in an MPI+OpenMP setting), therefore there are actually three loops generated; a scalar pre-sweep to get aligned to the vector length, the main vectorised loop, and a scalar post-sweep to compute set elements left over. Based on the parsed information from the high-level source, the steps of code generation are shown in Figure 7.14.

To evaluate performance, I use three different platforms; a mid-range Xeon server CPU pair (CPU 1, Ruby), a high-end Xeon server CPU pair (CPU 2) and the Intel Xeon Phi;

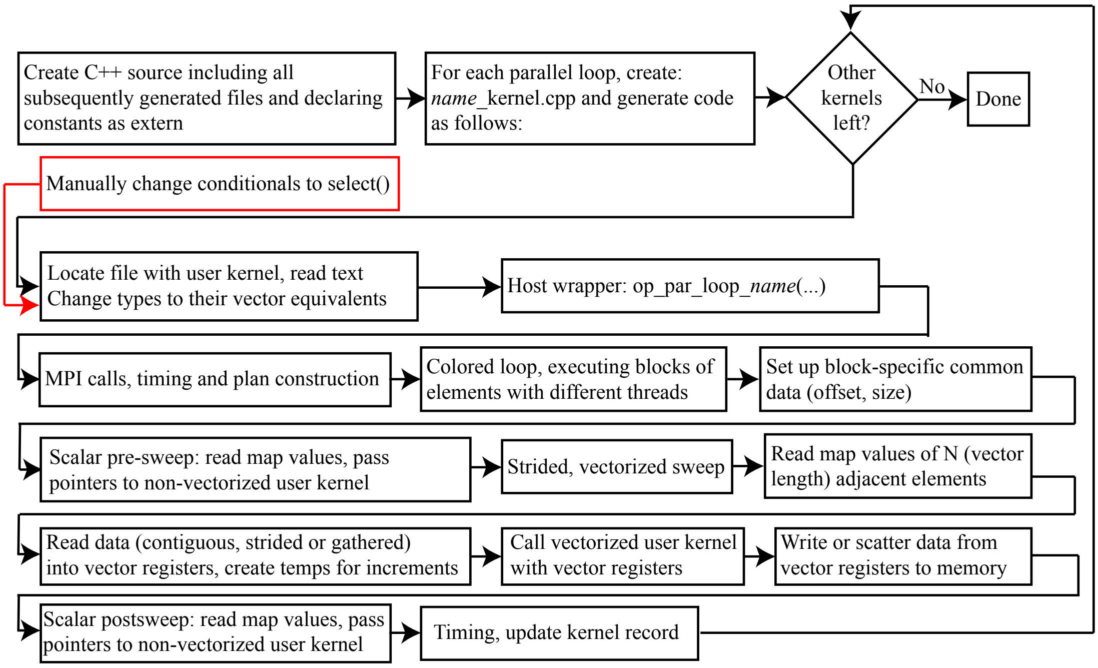

Create C++ source including all subsequently generated files and declaring constants as extern
For each parallel loop, create: *name*_kernel.cpp and generate code as follows:
Other kernels left?
No
Done
Manually change conditionals to select()
Locate file with user kernel, read text
Change types to their vector equivalents
Host wrapper: op_par_loop_*name*(...)
MPI calls, timing and plan construction
Colored loop, executing blocks of elements with different threads
Set up block-specific common data (offset, size)
Scalar pre-sweep: read map values, pass pointers to non-vectorized user kernel
Strided, vectorized sweep
Read map values of N (vector length) adjacent elements
Read data (contiguous, strided or gathered) into vector registers, create temps for increments
Call vectorized user kernel with vector registers
Write or scatter data from vector registers to memory
Scalar postsweep: read map values, pass pointers to non-vectorized user kernel
Timing, update kernel record

Figure 7.14. Generating code for vectorised execution on CPUs or the Xeon Phi

Table 7.7. CPU benchmark systems specifications

System	CPU 1 (Ruby)	CPU 2	Xeon Phi
Architecture	2×Xeon E5-2696-v2	2×Xeon E5-2680	Xeon Phi 5110P
Clock frequency	2.4 GHz	2.7 GHz	1.053 GHz
Core count	2×6	2×12	61 (60 used)
Last level cache	2×15MB	2×30MB	30MB
Peak bandwidth	2×42.6 GB/s	2×59.7 GB/s	320 GB/s
Peak compute DP(SP)	2×120(240) GFLOPS	2×259(518) GFLOPS	1.01 (2.02) TFLOPS
Stream bandwidth	65 GB/s	98.7 GB/s	171 GB/s
D/SGEMM throughput	188(414) GFLOPS	510(944) GFLOPS	833(1729) GFLOPS
FLOP/byte achieved DP(SP)	2.9(6.37)	5.43(9.34)	4.87(10.1)
FLOP/byte w/o vectorisation	0.73(0.79)	1.35(1.16)	0.6(0.63)
Intel Compiler	13.1 (Hydra), 14.0	14.0	14.0

these all have different balances in computational performance, memory bandwidth and the amount of parallelism supported - specifications are described in Table 7.7. Tests are carried out on the Airfoil benchmark using two meshes (720k cells, 2.8M cells), the second being for times bigger as well as Volna. Baseline performance breakdowns are provided in Table 7.8 for the non-vectorised execution with MPI in double precision on the 2.8M cell mesh for Airfoil and single precision for Volna.

7.3.2 Vector Intrinsics on CPUs

With many parallel programming abstractions and methods such as MPI, OpenMP and OpenCL, the programmer does not have direct control over what gets vectorised,

DOI:10.15774/PPKE.ITK.2014.003

Table 7.8. Useful bandwidth (BW - GB/s) and computational (Comp - GFLOP/s) throughput baseline implementations on Airfoil (double precision) and Volna (single precision) on CPU 1 and CPU 2

Kernel	MPI CPU 1			MPI CPU 2		
	Time	BW	Comp	Time	BW	Comp
save_soln	4	46	3.2	2.9	63	4
adt_calc	24.6	13	14.6	7.6	43	48
res_calc	25.2	27	32	13.6	50	61
bres_calc	0.09	29	12	0.05	52	16
update	14.05	56	8	9.7	81	10
RK_1	3.24	53	4	2.16	79	6
RK_2	2.88	59	5	1.92	89	9
compute_flux	23.34	14	42	12.1	27	82
numerical_flux	4.68	29	4	2.88	57	6
space_disc	16.86	21	9	4.54	79	33

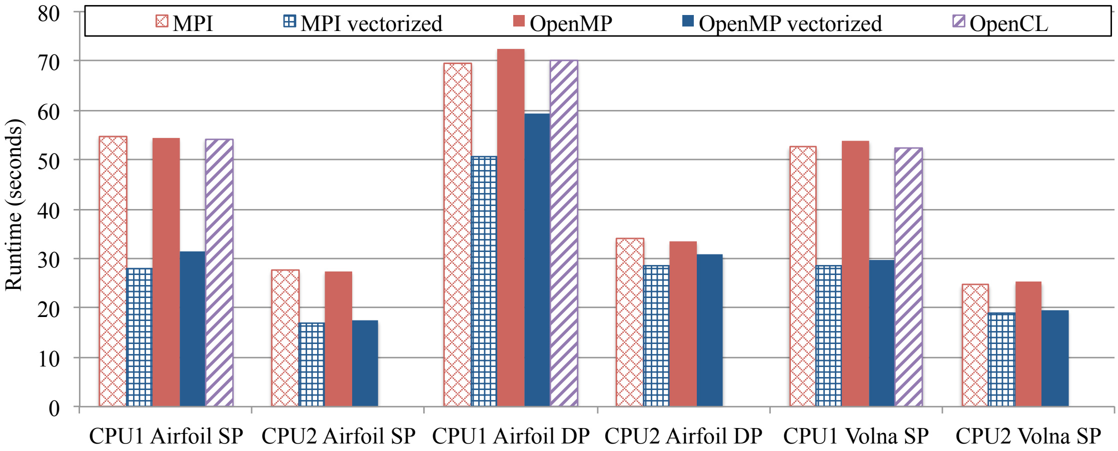


Figure 7.15. Vectorisation with explicit vector intrinsics and OpenCL, performance results from Airfoil in single (SP) and double (DP) precision on the 2.8M cell mesh and from Volna in single precision on CPU 1 and 2

but when using vector intrinsics, vectorisation is explicit. Figure 7.15 shows performance results comparing the vectorised and the non-vectorised code on CPU 1 and CPU 2 in both single and double precision on the mesh with 2.8M cells. The pure MPI versions are consistently faster than the MPI+OpenMP versions, because at this scale the overhead of MPI communications is almost completely hidden, while OpenMP has some overhead due to threading [96] and the coloured execution.

Observe that the improvement on CPU 1 in single precision is 67-97%, but in double precision it is only 15-37%; this is due to the fact that twice as much data has to be transferred in double precision, but the length of the vector registers is the same, therefore only half the number of floating point operations can be carried out at the same time. On the other hand, on CPU 2 the computational resources are doubled a the clock frequency is higher, but the available bandwidth only increased by 50%, therefore the performance difference between the vectorised and the non-vectorised versions is much smaller. Another important observation is that comparing the runtimes in single and double precision,

Table 7.9. Timing and bandwidth breakdowns for the Airfoil benchmarks in double(single) precision on the 2.8M cell mesh and Volna using the vectorised pure MPI backend on CPU 1 and CPU 2

Kernel	CPU 1		CPU 2	
	Time	BW	Time	BW
save_soln	4.08(2.04)	45(45)	2.9(1.5)	62(61)
adt_calc	12.7(5.2)	25(31)	5.6(3.5)	57(46)
res_calc	19.5(13.5)	35(26)	9.9(7.1)	69(48)
update	14.6(7.1)	53(56)	9.8(4.7)	79(83)
RK_1	3.27	52	2.19	78
RK_2	2.88	59	1.86	92
compute_flux	8.82	37	6	54
numerical_flux	4.59	30	3.18	43
space_disc	7.47	48	4.56	79

there is only a 30-40% difference in the baseline (recall that without vectorisation the computational throughput is the approximately the same in single and double precision), while the vectorised version shows an almost perfect 80-110% speedup when going from double to single precision.

Table 7.9 shows per-loop breakdowns of the vectorised Airfoil and Volna on CPU 1 and 2. Comparing double precision bandwidth values with that of Table 7.8, it is clear that the direct kernels (`save_soln`, `update`, `RK_1` and `RK_2`) were already bandwidth-limited, therefore their performance remains the same, however, `adt_calc` and `compute_flux` almost doubled in performance, and `res_calc` and `space_disc` also sped up by 30% on CPU 1 providing further evidence that these kernels were, to some extent, bound by compute throughput on the slower CPU. Switching to single precision should effectively half the runtime of different loops, because half the amount of data is moved, which matches the timings of direct loops on the large mesh as shown in Table 7.9. The kernel `adt_calc` gains further speedups (2.4 times) by moving to single precision, due to the higher computational throughput, which hints at this kernel still being compute-limited in double precision on CPU 1, due to the expensive square root operations. The kernels `res_calc` and `space_disc` is affected by a high number of gather and serialised scatter operations, moving to single precision only improves runtime by 30%, which hints at this kernel being limited by control and caching behaviour.

7.3.3 Vector Intrinsics on the Xeon Phi

One of Intel's main arguments in favour of the Xeon Phi is that applications running on the CPU are trivially ported to the Phi. While this is true to some extent as far as compilation and execution goes, performance portability is a much more important issue.

DOI:10.15774/PPKE.ITK.2014.003

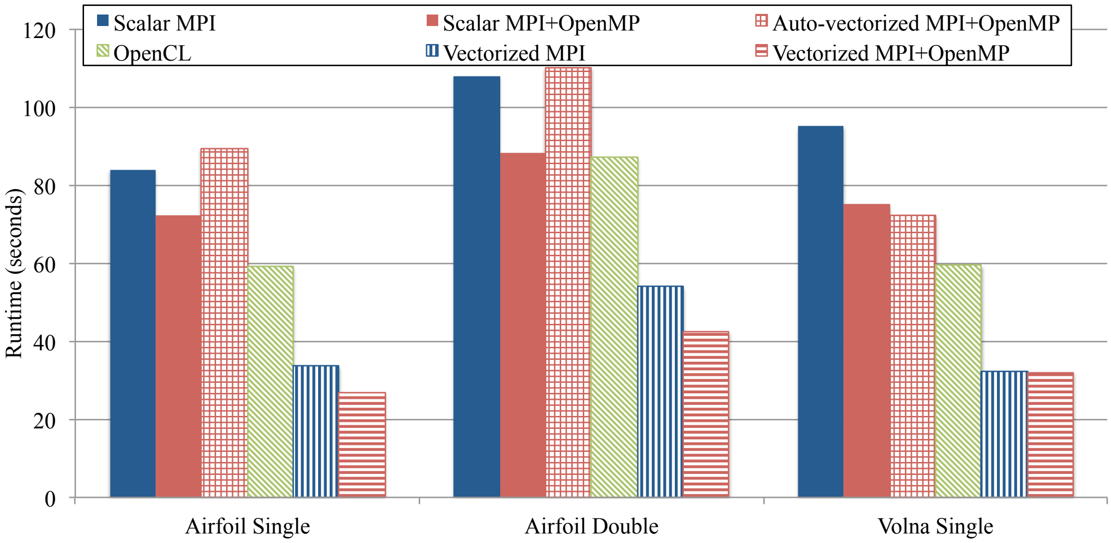


Figure 7.16. Performance of the Xeon Phi on Airfoil (2.8M cell mesh) and on Volna with non-vectorised, auto-vectorised and vector intrinsic versions

On the higher level, the balance of MPI and/or OpenMP is usually slightly different, and on a lower level vectorisation is needed more than ever to achieve high performance. As long as one depends on compiler auto-vectorisation this is automatic to some extent, but as I will show the performance of auto-vectorised code is poor. Therefore, the use of intrinsics is necessary, and since the instruction set is not backwards-compatible, they have to be changed. My approach of wrapping vectors in C++ classes, using constructors and operator overloading to hide vector intrinsic instructions permits us to generate the same code for both CPU and Phi vectorisation, and then through compiler preprocessor macros select classes that are appropriate for the hardware being used. Through this, I can exploit new features in the Phi, such as gather instructions, vector reduction, etc., and integrate it seamlessly into the OP2 toolchain. I had to override the `malloc, free` and `realloc` functions to ensure allocations are aligned to a 512 bit boundary (which does not happen automatically), so the aligned load instructions could be used when appropriate.

To compile for the Phi, I used the `-O3 -mmic -fno-alias -mcmodel=medium -inline-forceinline` flags, and executed all applications natively on the device, offload mode is currently not supported, because it requires extensive changes to the backend in order to manage halo exchanges. Tests included pure MPI execution and different combinations of MPI processes and OpenMP threads, setting `I_MPI_PIN_DOMAIN=auto`. Due to the nature of the OP2 OpenMP parallelisation approach, there is no data reuse between threads, which is likely why I observed very little (<3%) performance differences between the settings of `KMP_AFFINITY`, therefore I only report results with the `compact` setting. In all MPI+OpenMP hybrid tests, the total number of OpenMP threads is 240 (60 cores,

DOI:10.15774/PPKE.ITK.2014.003

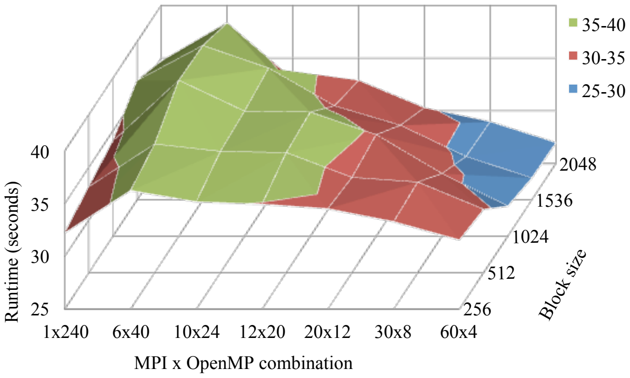


Figure 7.17. Performance on the Xeon Phi when varying OpenMP block size and the MPI+OpenMP combination on the 2.8M cell mesh in double precision

4 threads each), as the number of MPI processes vary, so does the number of OpenMP threads per process to give a total of 240.

Figure 7.16 shows performance figures on the Xeon Phi. Similar to the CPU, vectorisation gives a significant performance boost, but the difference is even higher, up to 3 times in single and 2.2 times in double precision. Unlike in the case of the CPU, the hybrid MPI+OpenMP gives better performance than the pure MPI version - which is due to the MPI messaging overhead becoming significant when the number of processes goes beyond 120. Comparing the runtimes of the small and the large mesh however reveals that the Xeon Phi is actually quite sensitive to being fully utilised; while the problem size is four times bigger, it only takes 2.97 times more time in single and 3.25 times more time in double precision to finish the execution for the vectorised MPI+OpenMP version. As a comparison, these figures are almost exactly 4 times (or slightly higher due to caching effects) on the CPU, and 3.85 times on the GPU.

An important factor affecting the performance is the hybrid MPI+OpenMP setup - how many MPI processes and how many OpenMP threads each. This has non-trivial effects on the cost of communications, shared cache space, NUMA effects and others. In addition to this, the size of mini-partitions or blocks formed and assigned to threads by OpenMP can be modified, trading off the number of blocks (load balancing) with block size (cache locality). Figure 7.17 shows the effects of varying these parameters on the performance, observe that as the number of MPI processes increases, a larger block size is preferred, up to a point where the load imbalance is significant.

Per-loop breakdowns for the Xeon Phi are shown in Table 7.10 for the best combination of MPI+OpenMP in each case. Performance is generally slightly better compared to CPU 1 (Table 7.9). `adt_calc` does not show signs of being compute-limited, but the performance penalty from gather operations and serialised scatter operations in `res_calc` and

Table 7.10. Timing and bandwidth breakdowns for Airfoil (2.8M mesh) in double(single) precision and Volna using different MPI+OpenMP backends on the Xeon Phi

Kernel	Scalar		Auto-vectorized		Intrinsics	
	Time	BW	Time	BW	Time	BW
save_soln	1.95(1.01)	94(92)	1.94(1.02)	95(90)	2.17(1.11)	84(83)
adt_calc	27.7(25.5)	12(6.3)	14.35(13.1)	23(12)	6.86(3.54)	47(45)
res_calc	48.8(36.8)	14(9.4)	84.03(66.5)	8(5)	27.22(18.4)	25(18)
update	11.8(9.59)	66(41)	8.33(8.45)	94(46)	8.77(4.6)	89(85)
RK_1	2.16	79	2.19	78	1.35	128
RK_2	2.37	70	3.24	53	1.32	130
compute_flux	32.1	10	29.3	11	10.95	30
numerical_flux	12.9	11	11.3	12	7.29	19
space_disc	23.6	15	24.5	15	9.93	36

`space_disc` is more severe, making these loops significantly slower than on the CPU. An important factor influencing runtime is the time spent in MPI communications, either sending/receiving halo data or performing a global reductions (in `update` or `numerical_fluxes` to compute the residual or the minimum timestep. Both of these operations result in implicit synchronisation; the time spent actually transferring data is negligible compared to the time spent waiting for data to arrive due to different processes getting slightly out of sync and then waiting for each other to catch up at a synchronisation point. On the smaller problem this takes up to 30% of total runtime, but is reduced to 13% on the larger problem, whereas on the CPU this is only 7% and 4% respectively. This points to load balancing issues, explaining some of the performance differences between the small and the large mesh.

7.4 Summary and associated theses

In this chapter I started by giving a short overview of how OP2 architects the multiple levels of parallelism that can be used when mapping to hardware, based on previous work of our research group [2, 5]. This is followed by my own research on the mapping of unstructured grid algorithms defined through the OP2 API to the SIMT parallel programming model and GPU architectures in combination with distributed-memory operations using MPI. I showed that through OP2 it is possible to automatically generate CUDA code and apply optimisations through the code generator. I demonstrated that near optimal performance is achieved and that these techniques apply immediately to the large-scale industrial application Hydra [4, 6, 11, 13]. Thus, as Thesis III.1. states: *I have designed and developed an automated mapping process to GPU hardware that employs a number of data and execution transformations in order to make the best use of limited hardware resources, the multiple levels of parallelism and memory hierarchy, which I proved experimentally.*

DOI:10.15774/PPKE.ITK.2014.003

By designing and facilitating the automatic mapping of execution to CPU cores and vector units, I have shown that there is no performance loss involved in using a high-level abstraction such as OP2, comparing Hydra's original implementation with the OP2 implementation, and I demonstrated that by applying techniques I integrated into OP2, such as mesh reordering, the original can be significantly outperformed. By demonstrating a fully automatic mapping to the SIMD parallel programming model and to AVX vector units specifically, I have demonstrated that near-optimal performance can be achieved on modern CPUs and the Xeon Phi architecture [4, 6, 11, 13]. As Thesis III.2 states: *I have designed and implemented an automated mapping process to multi- and many-core CPUs, such as Intel Xeon CPUs and the Intel Many Integrated Cores (MIC) platform, to make efficient use of multiple cores and large vector units in the highly irregular setting of unstructured grids, which I proved experimentally.*

This chapter discussed how my results integrate into a distributed memory environment, but I do not claim OP2's MPI backend as my own research, therefore the investigation of performance scaling on large-scale distributed-memory supercomputers is postponed to the Appendix, which will serve to show that the results discussed in this chapter are indeed scalable to 16 GPUs and 4096 CPU cores, completing the claims of Thesis III.1 and III.2.

DOI:10.15774/PPKE.ITK.2014.003

Chapter 8

Conclusions

I have set out to address the some of today's challenges to programmability; parallelism, locality, load balancing and resilience, in the field of unstructured grid computations.

By first narrowing my focus to the Finite Element Method, I have shown that it is possible to apply transformations to the high-level algorithm, exposing parallelism and handling data dependencies in a way that produces different formulations that trade off computations for communications. By pairing these with established and novel data structures, I have given descriptions of FE assembly and solution algorithms with very different characteristics in terms of the inherent spatial and temporal locality in them, how data dependencies and data races are handled, and others.

Subsequently, I have demonstrated how these algorithms map to GPU hardware, and what considerations had to be made to maximise performance. By presenting a detailed analysis of both the assembly and the solution phase I have shown the performance implications of different algorithmic formulations and data structures, and demonstrated the viability and the benefits of the Local Matrix Approach that I introduced. I have given further attention to the sparse matrix-vector operation - a basic building block in many iterative solvers - and addressed the related field of sparse linear algebra as well. I have shown how the operation can be parametrised for the GPU architecture and how performance is affected by the sparsity patterns, and I introduced novel heuristics and a run-time tuning algorithm that aims to find the optimal parameters. The resulting implementation significantly outperform the state of the art on a wide range of general sparse matrices, and my research into the distributed-memory execution of the operation helps make these results applicable to a large domain of applications.

As a natural generalisation of the first part of my research, I have shifted focus to general unstructured grid algorithms, defined through the OP2 domain specific library. OP2

DOI:10.15774/PPKE.ITK.2014.003

defines an abstraction for unstructured grid computations, its goal is to enable an application written once at a high level to be easily maintainable and performance portable, thereby making it future-proof. Thus, my research addressed these challenges to show that this is indeed achieved; first I studied high-level transformations to execution patterns, relying on domain-specific information in order to (1) provide resiliency through checkpointing and recovery, (2) improve spatial and temporal locality by introducing a novel redundant-compute tiling algorithm and (3) enable execution on heterogeneous systems with very different architectures and performance characteristics.

I have introduced mappings to different multi- and many-core architectures, such as CPUs, GPUs and the Xeon Phi that utilise the multi-level parallelism and deep memory hierarchies present in these hardware, and demonstrated automatic code generation techniques that are able to produce code in different programming languages and parallelisation approaches from a single high-level specification. Extending this, I have discussed how execution maps to heterogeneous distributed memory clusters. By analysing performance on a range of applications that I adopted to use OP2, including the Hydra, used by Rolls-Royce for the design of turbomachinery, I have shown that it is indeed possible to have an abstraction general enough to support a wide range of unstructured grid applications and achieve near-optimal performance on a wide range of contrasting hardware; from a single GPU to a 4000 core supercomputer.

While the hardware used and the measurements in this work only represent today's state of the art, I believe that the essence of my results will continue to be relevant as hardware evolves: since trends show hardware architectures becoming more complex and a further shift towards a bandwidth-scarcity, it is going to be increasingly important to re-assess numerical algorithms and to apply aggressively hardware-specific optimisations - something my research addressed by investigating the balance of computations and communications, and by demonstrating that through high-level domain-specific abstractions, it is possible to achieve near-optimal performance on a wide range of hardware, but still maintain productivity for the domain scientist, thereby future-proofing the application.

DOI:10.15774/PPKE.ITK.2014.003

Chapter 9

Theses of the Dissertation

This chapter gives a concise summary of the main scientific contributions of this dissertation as well as the methods and tools used, and briefly discusses the applicability of the results.

9.1 Methods and Tools

During the course of my research a range of numerical methods and analytical methods were used in conjunction with different programming languages, programming and execution models, and hardware. Indeed, one of my goals is to study the interaction of these in today's complex systems. The first part of my research (Thesis group I.) is based on a popular discretisation method for Partial Differential Equations (PDEs); the Finite Element Method - I used a Poisson problem, implemented loosely based on [45]. During the study of the solution of sparse linear systems, I used the Conjugate Gradient iterative method preconditioned by the Jacobi and the Symmetric Successive Over-Relaxation (SSOR) method [44]. The sparse matrix-vector multiplication, as the principal building block for sparse linear algebra algorithms, is studied in further detail. This initial part of my research served as an introduction to unstructured grid algorithms, gaining invaluable experience that would later be applied to the rest of my research.

The second and third parts of my research are based on the OP2 Domain Specific Language (or "active library"), introduced by Prof. Mike Giles at the University of Oxford [16], its abstraction carried over from OPlus [72]. There is a suite of finite volume applications that were written using the OP2 abstraction and are used to evaluate the algorithms presented in this dissertation; a benchmark simulating airflow around the wing of an aircraft (Airfoil), a tsunami simulation software called Volna [97], and a large-scale

DOI:10.15774/PPKE.ITK.2014.003

production application, Hydra [74], used by Rolls-Royce plc. for the design and simulation of turbomachinery. While I do not claim authorship of the original codes, I did do most of the work transforming the latter two to the OP2 abstraction. OP2 and the Airfoil benchmark are available at [83]

Computer code was implemented using either the C or the Fortran language, using the CUDA's language extensions when programming for GPUs. Python was used to facilitate text manipulation and code generation. A number of parallel programming models were employed to support the hierarchical parallelism present in modern computer systems; at the highest level, message passing for distributed memory parallelism, using MPI libraries. For coarse-grained shared memory parallelism I used simultaneous multithreading (SMT), using OpenMP and CUDA thread blocks. Finally for fine-grained shared memory parallelism I used either Single Instruction Multiple Threads (SIMT) using CUDA, or Single Instruction Multiple Data (SIMD) using Intel vector intrinsics.

A range of contrasting hardware platforms were used to evaluate the performance of algorithms and software. When benchmarking at a small scale, single workstations were used, consisting of dual-socket Intel Xeon server processors (Westmere X5650, Sandy-Bridge E2640 and E2670). The accelerators used were: an Intel Xeon Phi 5110P, and NVIDIA Tesla cards (C2070, M2090, K20, K20X, K40). For large-scale tests the following supercomputers were used: HECToR (the UK's national supercomputing machine, a Cray XE6, with 90112 AMD Opteron cores), Emerald (the UK's largest GPU supercomputer, with 372 NVIDIA M2090 GPUs, 512 cores each) and Jade (Oxford University's GPU cluster, with 16 NVIDIA K20 GPUs, 2496 cores each). Timings were collected using standard UNIX system calls, usually ignoring the initial set-up cost (due to e.g. file I/O) because production runs of the benchmarked applications have an execution time of hours or days, compared to which, set-up costs are negligible. In most cases, results are collected from 3-5 repeated runs and averaged. Wherever possible, I provide both absolute and relative performance numbers, such as achieved bandwidth (in GB/s), computational throughput (10^9 Floating Operations Per Second - GFLOPS), and speedup over either a reference implementation on the GPU or a fully utilised CPU, not just a single core.

9.2 New scientific results

Thesis group I. *(area: Finite Element Method)* - I have introduced algorithmic transformations, data structures and new implementations of the Finite Element Method (FEM) and corresponding sparse linear algebra methods to GPUs in order

DOI:10.15774/PPKE.ITK.2014.003

to address different aspects of the concurrency, locality, and memory challenges and quantified the trade-offs.

Related publications: [3, 8, 12, 14].

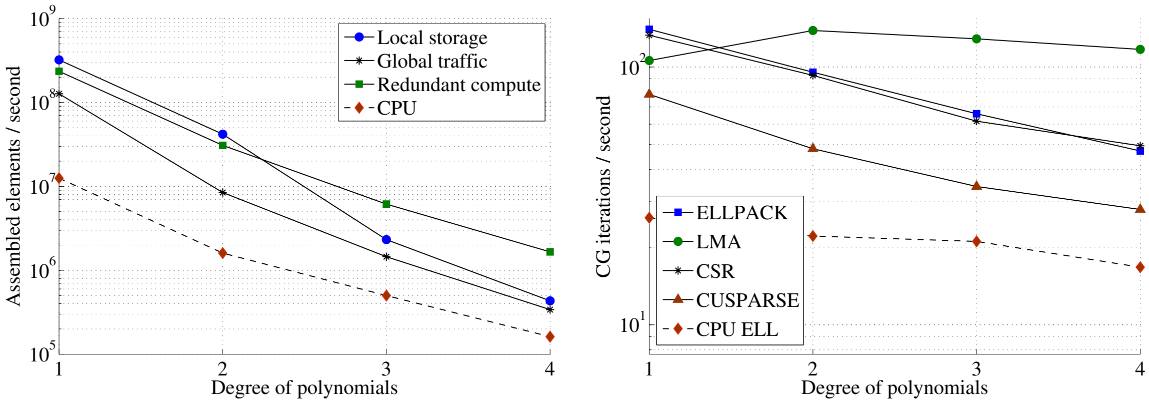
Thesis I.1. - By applying transformations to the FE integration that trade off computations for communications and local storage, I have designed and implemented new mappings to the GPU, and shown that the redundant compute approach delivers high performance, comparable to classical formulations for first order elements, furthermore, it scales better to higher order elements without loss in computational throughput.

Through algorithmic transformations to the Finite Element integration, I gave per-element formulations that have different characteristics in terms of the amount of computations, temporary memory usage, and spatial and temporal locality in memory accesses. The three variants are: (1 - redundant compute), where the outer loop is over pairs of degrees of freedom and the inner loop over quadrature points recomputing the Jacobian for each one, (2 - local storage) structured as (1) but Jacobians are pre-computed and re-used in the innermost loop, effectively halving the number of computations, and (3 - global memory traffic), that is commonly used in Finite Element codes, where the outermost loop is over quadrature points, computing the Jacobian once, and then the inner loop is over pairs of degrees of freedom, adding the contribution from the given quadrature point to the stiffness values. As illustrated in Figure 9.1a, I have demonstrated that approach (1) is scalable to high degrees of polynomials because only the number of computations changes, whereas with (2) the amount of temporary storage, and with (3) the number of memory transactions also increase. Implementations of these variants in CUDA applied to a Poisson problem show that for low degree polynomials (1) and (2) perform almost the same, but at higher degrees (1) is up to 8× faster than (2), and generally 3× faster than (3). Overall, an NVIDIA C2070 GPU is demonstrated to deliver up to 400 GFLOPS (66% of the ideal[1]), it is up to 10× faster than a two-socket Intel Xeon X5650 processor, and up to 120× faster than a single CPU core.

Thesis I.2. - I introduced a data structure for the FEM on the GPU, derived storage and communications requirements, shown its applicability to both the integration and the sparse iterative solution, and demonstrated superior performance due to improved locality.

One of the key challenges to performance in the FEM is the irregularity of the

[1] The same card delivers 606 Giga Floating Operations per Second (GFLOPS) on a dense matrix-matrix multiplication benchmark



(a) FE integration transformations (b) Iterative solution and data structures

Figure 9.1. Performance of Finite Element Method computations mapped to the GPU

problem and therefore of the memory accesses, which is most apparent during the matrix assembly and the sparse iterative solution phases. By storing stiffness values on a per-element basis laid out for optimal access on massively parallel architectures, I have shown that it is possible to regularise memory accesses during integration by postponing the handling of race conditions until the iterative solution phase, where it can be addressed more efficiently. This approach, called the Local Matrix Approach (LMA), consists of a storage format and changes to the FE algorithms in both the assembly and the solution phases, and is compared to traditional storage formats, such as CSR and ELLPACK on GPUs. I show that it can be up to 2× faster during both phases of computations, due to reduced storage costs, as shown in Figure 9.1b, and regularised memory access patterns. A conjugate gradient iterative solver is implemented, supporting all three storage formats, using a Jacobi and a Symmetric Successive Over-Relaxation (SSOR) preconditioner, performance characteristics are analysed, and LMA is shown to deliver superior performance in most cases.

Thesis I.3. - I have parametrised the mapping of sparse matrix-vector products (spMV) for GPUs, designed a new heuristic and a machine learning algorithm in order to improve locality, concurrency and load balancing. Furthermore, I have introduced a communication-avoiding algorithm for the distributed execution of the spMV on a cluster of GPUs. My results improve upon the state of the art, as demonstrated on a wide range of sparse matrices from mathematics, computational physics and chemistry.

The sparse matrix-vector multiplication operation is a key part of sparse linear algebra; virtually every algorithm uses it in one form or another. The most commonly used storage format for sparse matrices is the compressed sparse row (CSR) format; it

DOI:10.15774/PPKE.ITK.2014.003

Table 9.1. Performance metrics on the test set of 44 matrices.

	CUSPARSE	Fixed rule	Tuned
Throughput single GFLOPS/s	7.0	14.5	15.6
Throughput double GFLOPS/s	6.3	8.8	9.2
Min Bandwidth single GB/s	28.4	58.9	63.7
Min Bandwidth double GB/s	38.7	54.0	56.8
Speedup single over CUSPARSE	1.0	2.14	2.33
Speedup double over CUSPARSE	1.0	1.42	1.50

is supported by a wide range of academic and industrial software, thus I chose it for the basis for my study. By appropriately parametrising the multiplication operation for GPUs, using a dynamic number of cooperating threads to carry out the dot product between a row of the matrix and the multiplicand vector, in addition to adjusting the thread block size and the granularity of work assigned to thread blocks, it is possible to outperform the state of the art CUSPARSE library. I have introduced an O(1) heuristic that gives near-optimal values for these parameters that immediately results in 1.4-2.1× performance increase. Based on the observation that in iterative solvers, the spMV is evaluated repeatedly with the same matrix, I have designed and implemented a machine learning that tunes these parameters and increases performance by another 10-15% in at most 10 iterations, achieving 98% of the optimum, found by exhaustive search. Results are detailed in Table 9.1. I have also introduced a communication avoiding algorithm for the distributed memory execution of the spMV, that uses overlapping graph partitions to perform redundant computations and decrease the frequency of communications, thereby mitigating the impact of latency, resulting in up to 2× performance increase.

Thesis group II. *(area: High-Level Transformations with OP2)* - I address the challenges of resilience, the expression and exploitation of data locality, and the utilisation of heterogeneous hardware, by investigating intermediate steps between the abstract specification of an unstructured grid application with OP2 and its parallel execution on hardware; I design and implement new algorithms that apply data transformations and alter execution patterns.

Related publications [2, 5, 4, 9]

Thesis II.1. - I have designed and implemented a checkpointing method in the context of OP2 that can automatically locate points during execution where the state space is minimal, save data and recover in the event of a failure.

As the number of components in high performance computing systems increases, the mean time between hardware or software failures may become less than the execution time of a large-scale simulation. I have introduced a checkpointing method in order to provide means to recover after a failure, that relies on the information provided through the OP2 API to reason about the state space of the application at any point during the execution and thereby to (1) find a point where the size of the state space is minimal and save it to disk and (2) in case of a failure, recover by fast-forwarding to the point where the last backup happened. This is facilitated by the OP2 library, in a way that is completely opaque to the user, requiring no intervention except for the re-launch of the application after the failure. This ensures the resiliency of large-scale simulations.

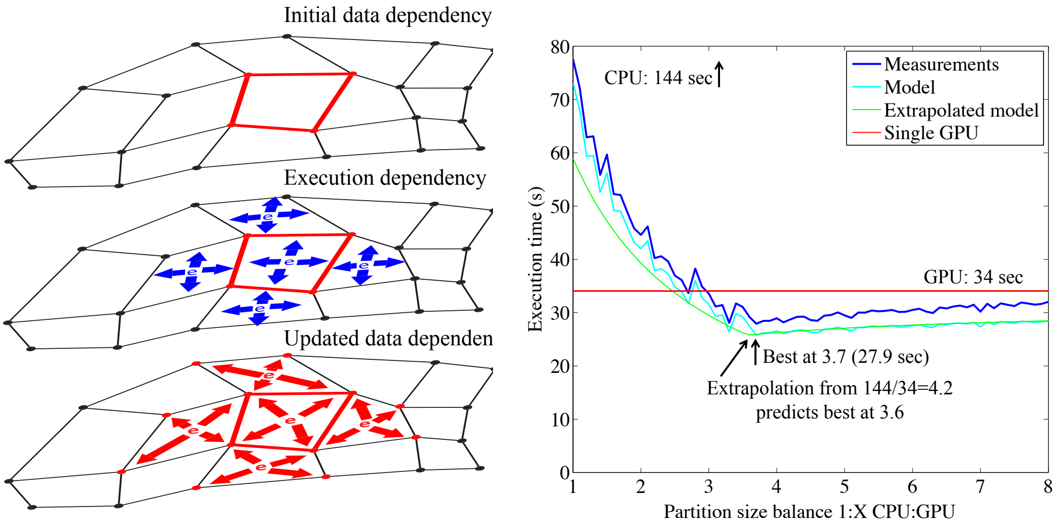
Thesis II.2. - I gave an algorithm for redundant compute tiling in order to provide cache-blocking for modern architectures executing general unstructured grid algorithms, and implemented it in OP2, relying on run-time dependency analysis and delayed execution techniques.

Expressing and achieving memory locality is one of the key challenges of high performance programming; but the vast majority of scientific codes are still being designed and implemented in a way that only supports very limited locality; it is common practice to carry out one operation on an entire dataset and then another - as long as the dataset is larger than the on-chip cache, this will result in repeated data movement. However, doing one operation after the other on just a part of the dataset is often non-trivial due to data dependencies. I have devised and implemented a tiling algorithm for general unstructured grids defined through the OP2 abstraction, that can map out these data dependencies, as illustrated in Figure 9.2a, and enable the concatenation of operations over a smaller piece of the dataset, ideally resident in cache, thereby improving locality. The tiling algorithm can be applied to any OP2 application without the intervention of the user.

Thesis II.3. - I gave a new performance model for the collaborative, heterogeneous execution of unstructured grid algorithms where multiple hardware with different performance characteristics are used, and introduced support in OP2 to address the issues of hardware utilisation and energy efficiency.

Modern supercomputers are increasingly designed with many-core accelerators such as GPUs or the Xeon Phi. Most applications running on these systems tend to only utilise the accelerators, leaving the CPUs without useful work. In order to make the best use of these systems, all available resources have to be kept busy, in a way that takes

DOI:10.15774/PPKE.ITK.2014.003



(a) Resolving data and execution dependencies for unstructured mesh tiling

(b) Heterogeneous execution and modelling

Figure 9.2. High-level transformations and models based on the OP2 abstraction

their different performance characteristics into account. I have developed a model for the hybrid execution of unstructured grid algorithms, giving a lower bound for expected performance increase and added support for utilising heterogeneous hardware in OP2, validating the model and evaluating performance, as shown in Figure 9.2b.

Thesis group III. *(area: Mapping to Hardware with OP2)* - One of the main obstacles in the way of the widespread adoption of domain specific languages is the lack of evidence that they can indeed deliver performance and future proofing to real-world codes. Through the Airfoil benchmark, the tsunami-simulation code Volna and the industrial application Hydra, used by Rolls-Royce plc. for the design of turbomachinery, I provided conclusive evidence that an unstructured grid application, written once using OP2, can be automatically mapped to a range of heterogeneous and distributed hardware architectures at near-optimal performance, thereby providing maintainability and longevity to these codes.

Related publications: [2, 5, 4, 6, 10, 11, 15, 13]

Thesis III.1. - I have designed and developed an automated mapping process to GPU hardware that employs a number of data and execution transformations in order to make the best use of limited hardware resources, the multiple levels of parallelism and memory hierarchy, which I proved experimentally.

Mapping execution to GPUs involves the use of the Single Instruction Multiple

DOI:10.15774/PPKE.ITK.2014.003

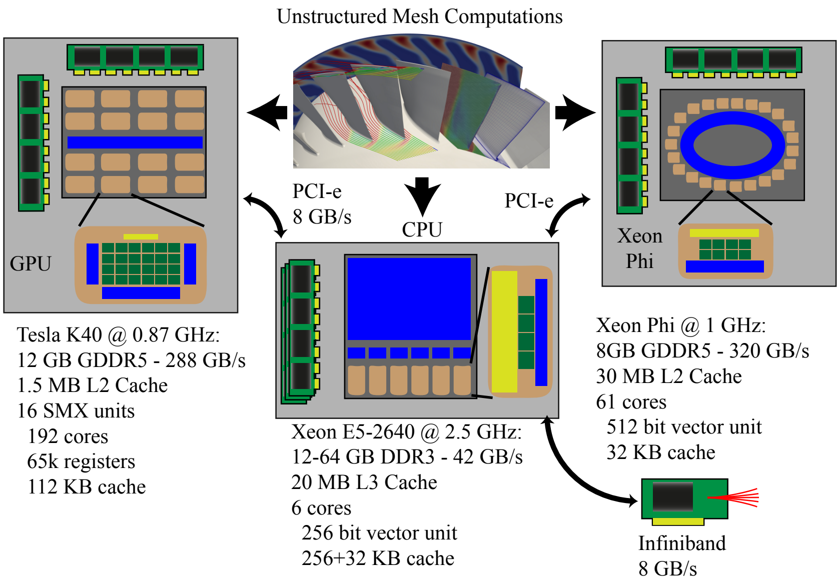


Figure 9.3. The challenge of mapping unstructured grid computations to various hardware architectures and supercomputers

Threads (SIMT) model, and the CUDA language. I have created an automatic code generation technique that, in combination with run-time data transformation, facilitates near-optimal execution on NVIDIA Kepler-generation GPUs. I show how state-of-the-art optimisations can be applied through the code generator, such as the use of the read-only cache, or data structure transformation from Array-of-Structures (AoS) to Structure-of-Arrays (SoA), in order to make better use of the execution mechanisms and the memory hierarchy. These are then deployed to a number of applications and tested on different hardware, giving 2-5× performance improvement over fully utilised Intel Xeon CPUs. Performance characteristics are analysed, including compute and bandwidth utilisation, to gain a deeper understanding of the interaction of software and hardware, and to verify that near-optimal performance is indeed achieved. I discuss how OP2 is able to utilise supercomputers with many GPUs, by automatically handling data dependencies and data movement using MPI, and I demonstrate strong and weak scalability on Hydra.

Thesis III.2. - I have designed and implemented an automated mapping process to multi- and many-core CPUs, such as Intel Xeon CPUs and the Intel Many Integrated Cores (MIC) platform, to make efficient use of multiple cores and large vector units in the highly irregular setting of unstructured grids, which I proved experimentally.

Modern CPUs feature increasingly longer vector units, their utilisation is essential to achieving high performance. However, compilers consistently fail at automatically vectorising irregular codes, such as unstructured grid algorithms, therefore low-level vector intrinsics have to be used to ascertain the utilisation of vector processing capabilities. I have introduced a code generation technique that is used in conjunction with C++ classes and operator overloading for wrapping vector intrinsics and show how vectorised execution can be achieved through OP2, by automatically gathering and scattering data. Performance is evaluated on high-end Intel Xeon CPUs and the Xeon Phi, and a 1.5-2.5× improvement is demonstrated over the non-vectorised implementations. In-depth analysis reveals what hardware limitations determine the performance of different stages of computations on different hardware. I demonstrate that these approaches are naturally scalable to hundreds or thousands of cores in modern supercomputers, evaluating strong and weak scalability on Hydra.

9.3 Applicability of the results

The applicability of the results related to the Finite Element Method are manyfold; the practice of designing algorithms that have different characteristics in terms of computations, memory requirements and memory traffic is useful in different contexts as well, but the results can be directly used when designing a general-purpose Finite Element library. There are already some libraries, such as ParaFEM [98], which take a similar matrix-free approach as LMA, therefore my results are directly applicable, should GPU support be introduced, or the need for more advanced sparse linear solvers arise. Results concerning the sparse matrix-vector product are pertinent to a much wider domain of applications: sparse linear algebra. The heuristic published in [8] was subsequently adopted by the NVIDIA CUSPARSE library [59], and the run-time auto-tuning of parameters is a strategy that, though few libraries have adopted, could become standard practice as hardware becomes even more diverse and thus performance predictions become more uncertain. During my internship at NVIDIA, I have developed the distributed memory functionality of the sparse linear solver software package that became AmgX [99], incorporating many of the experiences gained working on the FEM, designing it from the outset in a way so that optimisations such as redundant computations for avoiding communications could be adopted.

DOI:10.15774/PPKE.ITK.2014.003

Results of the research carried out in the context of the OP2 framework are immediately applicable to scientific codes that use OP2; after converting the Volna tsunami simulation code [97] to OP2, it was adopted by Serge Guillas's group at the University College London and subsequently by the Indian Institute of Science in Bangalore, and it is currently being used for the simulation of tsunamis in conjunction with uncertainty quantification, since the exact details of the under-sea earthquakes are often not known. Similarly, the conversion of Rolls-Royce Hydra [74] to OP2 is considered a success, performance bests the original, and support for modern heterogeneous architectures is introduced, thereby future-proofing the application; discussions regarding the use of the OP2 version in production are ongoing. However, many of these results, especially the ones under Thesis II that describe generic algorithms and procedures, are relevant to other domains in scientific computations as well; our subsequent research into structured grid computations is going to be employing many of these techniques, and some are already used in research on molecular dynamics carried out in collaboration with chemical physicists that resulted in [7].

DOI:10.15774/PPKE.ITK.2014.003

Chapter 10

Appendix

In this appendix, I discuss the strong and weak scaling performance of the Hydra application, in order to demonstrate OP2's ability to compose the mapping of execution to individual hardware, discussed in Chapter 7, with distributed memory execution capabilities.

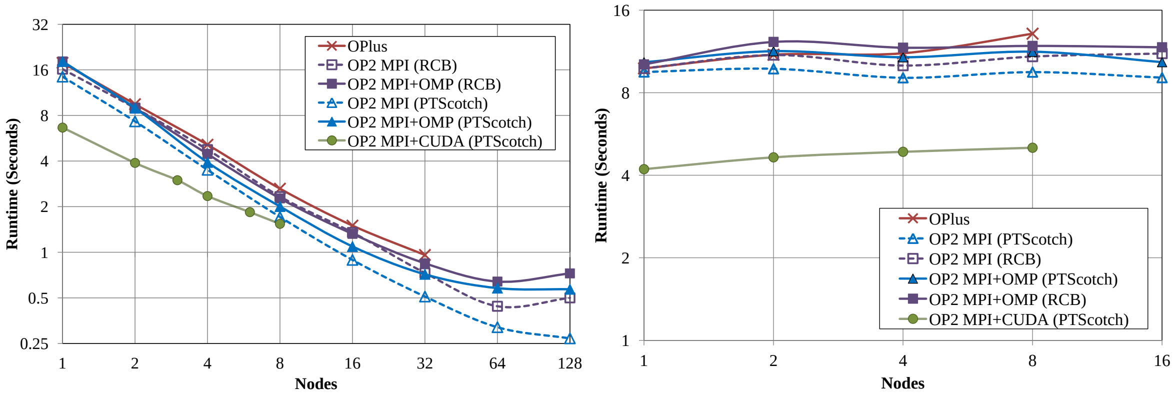
10.1 Mapping to distributed, heterogeneous clusters

The industrial problems simulated by Hydra require significantly larger computational resources than what is available today on single node systems. An example design simulation such as a multi-blade-row unsteady RANS (Reynolds Averaged Navier Stokes) computation, would need to operate over a mesh with about 100 million nodes. Currently with OPlus, Hydra can take more than a week on a small CPU cluster to reach convergence for such a large-scale problem. Future turbomachinery design projects aim to carry out such simulations more frequently, such as on a weekly or on a daily basis, and as such the OP2 based Hydra code needs to scale well on clusters with thousands to tens of thousands of processor cores. In this section I explore the performance on such systems. Table 10.1 lists the key specifications of the two cluster systems I use in my benchmarking. The first system, HECToR, is a large-scale proprietary Cray XE6 system which I use to investigate the scalability of the MPI and MPI+OpenMP parallelisations. The second system, JADE is a small NVIDIA GPU (Tesla K20) cluster that I use to benchmark the MPI+CUDA execution.

DOI:10.15774/PPKE.ITK.2014.003

Table 10.1. Benchmark systems specifications

System	HECToR (Cray XE6)	Jade (NVIDIA GPU Cluster)
Node Architecture	2×16-core AMD Opteron 6276 (Interlagos)2.3GHz	2×Tesla K20m GPUs+ Intel Xeon E5-1650 3.2GHz
Memory/Node	32GB	5GB/GPU
Num of Nodes	128	8
Interconnect	Cray Gemini	FDR InfiniBand
O/S	CLE 3.1.29	Red Hat Linux 6.3
Compilers	Cray MPI 8.1.4	PGI 13.3, ICC 13.0.1, OpenMPI 1.6.4, CUDA 5.0
Compiler flags	-O3 -h fp3 -h ipa5	-O2 -xAVX -Mcuda=5.0,cc35



(a) Strong Scaling (2.5M edges)

(b) Weak Scaling (0.5M edges per node)

Figure 10.1. Scaling performance on HECToR (MPI, MPI+OpenMP) and Jade (MPI+CUDA) on the NASA Rotor 37 mesh (20 iterations)

10.1.1 Strong Scaling

Figure 10.1a reports the run-times of Hydra at scale, solving the NASA Rotor 37 mesh with 2.5M edges in a strong-scaling setting. The x-axis represents the number of nodes on each system tested, where a HECToR node consists of two Interlagos processors, a JADE node consists of two Tesla K20 GPUs. MPI+OpenMP results were obtained by assigning four MPI processes per HECToR node, each MPI process consisting of eight OpenMP threads. This is due to the NUMA architecture of the Interlagos processors which combine two cores to a “module” and packages 4 modules with a dedicated path to part of the DRAM. Here I leverage the job scheduler to exactly place and bind one MPI process per memory region (or die) reducing inter-die communications. Other combinations of processes and threads were also explored, but the above provided the best performance.

On HECToR, I see that the overall scaling of Hydra with OP2 is significantly better than that with OPlus. OP2’s MPI-only parallelisation scales well up to 128 nodes (4096 cores). At 32 nodes (1024 cores) the MPI-only parallelisation partitioned with PTScotch

DOI:10.15774/PPKE.ITK.2014.003

Table 10.2. Halo sizes : Av = average number of MPI neighbours per process, Tot. = average number total elements per process, %H = average % of halo elements per process

nodes	MPI	PTScotch			RCB		
	procs.	Av	Tot	%H	Av	Tot	%H
1	32	7	111907	5.12	9	117723	9.81
2	64	8	56922	6.74	10	61208	13.27
4	128	9	29175	9.02	11	32138	17.41
8	256	10	14997	11.52	11	17074	22.28
16	512	10	7765	14.55	12	9211	27.97
32	1024	10	4061	18.32	12	4949	32.98
64	2048	10	2130	22.16	13	2656	37.58
128	4096	10	1134	26.98	13	1425	41.89

(a) Strong scaling

nodes	MPI	PTScotch			RCB		
	procs.	Av	Tot	%H	Av	Tot	%H
1	32	8	70084	6.00	8	73486	10.35
2	64	9	73527	6.69	9	78934	11.07
4	128	10	79009	6.09	10	73794	12.26
8	256	12	73936	7.33	11	78396	12.98
16	512	13	78671	6.94	12	75224	13.76

(b) Weak scaling

gives about 2x speedup over the runtime achieved with OPlus.

As with all message passing based parallelisations, one of the main problems that limits the scalability is the over-partitioning of the mesh at higher machine scale. This leads to an increase in redundant computations at the halo regions (compared to the non-halo elements per partition) and an increase in time spent during halo exchanges. Evidence for this explanation can be gained by comparing the average number of halo elements and the average number of neighbours per MPI process reported by OP2 after partitioning with PTScotch and RCB (see Table 10.2). I noted these results from runs on HECToR, but the halo sizes and neighbours are only a function of the number of MPI processes (where one MPI process is assigned to one partition) as the selected partitioner gives the same quality partitions for a given mesh for the same number of MPI processes on any cluster.

Columns 4 and 7 of Table 10.2(a) detail the average total number of nodes and edges per MPI process when partitioned with PTScotch and RCB respectively. Columns 5 and 8 (%H) indicate the average proportion of halo nodes and edges out of the total number of elements per MPI process while Columns 3 and 6 (Av) indicate the average number of communication neighbours per MPI process. With PTScotch, the proportion of the halo elements out of the average total number of elements held per MPI process ranged from about 5% (at 32 MPI processes) to about 27% (at 4096 MPI processes). The average number of MPI neighbours per MPI process ranged from 7 to 10. The halo sizes with

DOI:10.15774/PPKE.ITK.2014.003

Table 10.3. Hydra strong scaling performance on HECToR, Number of blocks (nb) and number of colours (nc) for MPI+OpenMP and time spent in communications (comm) and computations (comp) for the hybrid and the pure MPI implementation: 2.5M edges, 20 iterations

Num of	MPI+OMP		MPI+OMP		MPI	
nodes	nb	nc	comm (sec.)	comp (sec.)	comm (sec.)	comp (sec.)
1	9980	17	1.33	16.6	1.2	13.2
2	4950	16	1.04	7.8	0.83	6.5
4	2520	17	0.57	3.3	0.36	3.14
8	1260	15	0.52	1.42	0.23	1.48
16	630	14	0.26	0.81	0.21	0.68
32	325	13	0.28	0.4	0.13	0.38
64	165	10	0.32	0.21	0.12	0.2
128	86	12	0.52	0.11	0.15	0.12

RCB were relatively large, starting at about 10% at 32 MPI processes to about 40% at 4096 processes. Additionally the average number of neighbours per MPI process was also greater with RCB. These causes point to better scaling with PTScotch which agrees with the results in Figure 10.1a.

The above reasoning, however, goes contrary to the relative performance difference I see between OP2's MPI only and MPI+OpenMP parallelisations. I expected MPI+OpenMP to perform better at larger machine scales as observed in previous performance studies using the Airfoil CFD benchmark[2]. The reason was that larger partition sizes per MPI process gained with MPI+OpenMP in turn resulting in smaller proportionate halo sizes. But for Hydra, adding OpenMP multi-threading has caused a reduction in performance, where the gains from better halo sizes at increasing scale have not manifested into an overall performance improvement. Thus, it appears that the performance bottlenecks discussed in Section 7.3 for the single node system are prevalent even at higher machine scales. To investigate whether this is indeed the case, Table 10.3 presents the number of colours and blocks for the MPI+OpenMP runs at increasing scale on HECToR.

The size of a block (i.e. a mini-partition) was tuned from 64 to 1024 for each run, but at higher scale (from upwards of 16 nodes) the best runtimes were obtained by a block size of 64. As can be seen, the number of blocks (nb) reduces by two orders of magnitude when scaling from 1 node up to 128 nodes. However within this scale the number of colours remains between 10 to 20. These numbers provide evidence similar to the ones I observed on the Ruby single node system in Section 7.3 where a reduced number of blocks per colour results in poor load balancing. The time spent computing and communicating during each run at increasing scale shows that although the computation time reduces faster for the MPI+OpenMP version, its communication times increase significantly, compared to the

DOI:10.15774/PPKE.ITK.2014.003

pure MPI implementation. Profiled runs of MPI+OpenMP indicate that the increase in communications time is in fact due to time spent in `MPI_Waitall` statements where due to poor load balancing, MPI processes get limited by their slowest OpenMP thread.

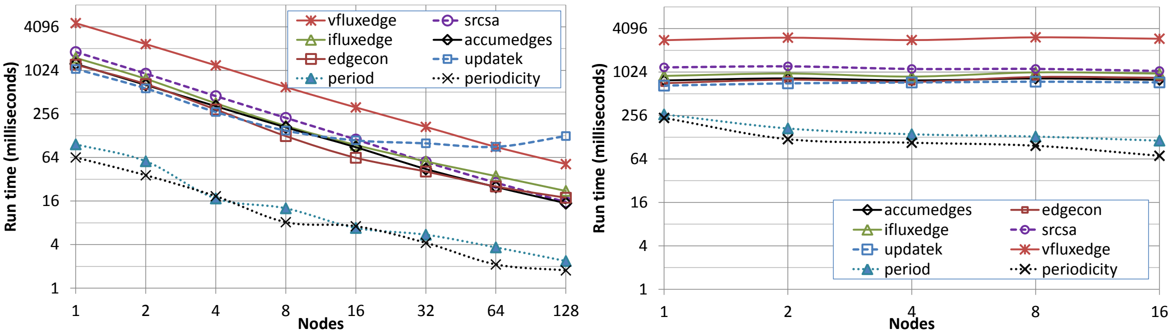
Getting back to Figure 10.1a, comparing the performance on HECToR to that on the GPU cluster JADE, reveals that for the 2.5M edge mesh problem the CPU system gives better scalability than the GPU cluster. This comes down to GPU utilisation issues: the level of parallelism during execution. Since the GPU is more sensitive to these effects than the CPU (where the former relies on increased throughput for speedups and the latter depends on reduced latency), the impact on performance is more significant due to reduced utilisation at increasing scale. Along with the reduction in problem size per partition, the same fragmentation as I observed with the MPI+OpenMP implementation due to colouring is present. Colours with only a few blocks have very low GPU utilisation, leading to a disproportionately large execution time. This is further complemented by the different number of colours on different partitions for the same loop, leading to faster execution on some partitions and then the idling at implicit or explicit synchronisation points waiting for the slower ones to catch up. I further explore these issues and how they affect performance of different types of loops in Hydra later in Section 10.1.3.

10.1.2 Weak Scaling

Weak scaling of a problem investigates the performance of the application at both increasing problem and machine size. For Hydra, I generated a series of NASA rotor 37 meshes such that a near-constant mesh size per node (0.5M vertices) is maintained at increasing machine scale. The results are detailed in Figure 10.1b. The largest mesh size benchmarked across 16 nodes (512 cores) on HECToR consists of about 8 million vertices and 25 million edges in total. Further scaling could not be attempted due to the unavailability of larger meshes at the time of writing.

With OPlus, there is about 8-13% increase in the runtime of Hydra each time the problem size is doubled. With OP2, the pure MPI version with PTScotch partitioning shows virtually no increase in runtime, while the RCB partitioning slows down 3-7% every time the number of processes and problem size is doubled. One reason for this is the near constant halo sizes resulting from PTScotch, but with RCB giving 7-10% larger halos. The second reason is the increasing cost of MPI communications at larger scale, especially for global reductions. Similar to strong scaling, the MPI-only parallelisation performs about 10-15% better than the MPI+OpenMP version. The GPU cluster, JADE, gives the best

DOI:10.15774/PPKE.ITK.2014.003



(a) Strong Scaling (2.5M edges)

(b) Weak Scaling (0.5M edges per node)

Figure 10.2. Scaling performance per loop runtime breakdowns on HECToR (NASA Rotor 37, 20 iterations)

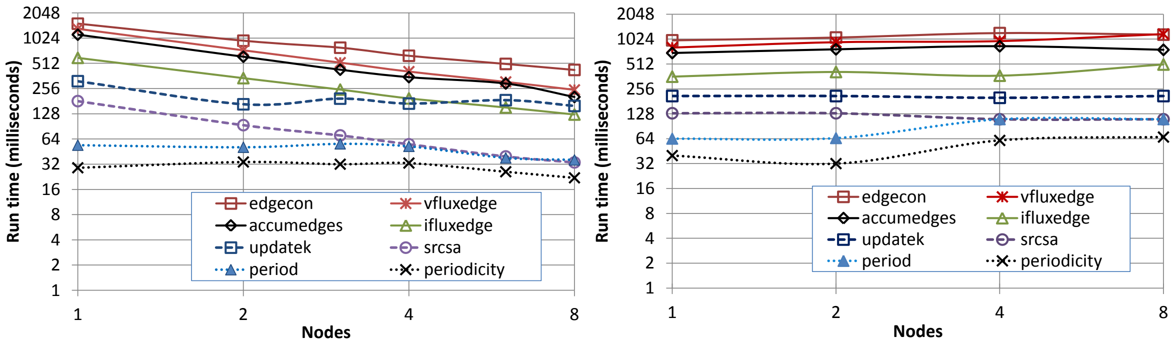
runtimes for weak scaling, with a 4-8% loss of performance when doubling problem size and processes. It roughly maintains a 2× speedup over the CPU implementations at increasing scale. Adjusting the experiment to compare one HECToR node to one GPU (instead of a full JADE node with 2 GPUs) still shows a 10-15% performance advantage for the GPU.

The above scaling results show OP2's ability to give good performance at large machine sizes even for a complex industrial application such as Hydra. We see that the primary factor affecting performance is the quality of the partitions: minimising halo sizes and MPI communication neighbours. These results illustrate that, in conjunction with utilising state-of-the-art partitioners such as PTScotch, the halo sizes resulting from OP2's owner-compute design for distributed memory parallelisation provide excellent scalability. I also see that GPU clusters are much less scalable for small problem sizes and are best utilised in weak-scaling executions.

10.1.3 Performance Breakdown at Scale

In this section I delve further into the performance of Hydra in order to identify limiting factors. The aim is to break down the scaling performance to gain insights into how the most significant loops in the application scale on each of the two cluster systems.

Figure 10.2a shows the timing breakdowns for a number of key loops when, for the MPI only version (partitioned with PTScotch). Note how the loops `vfluxedge, edgecon` and `srcsa` at small scale account for most of total runtime. However as they are loops over either interior vertices or edges, that do not include any global reductions, they have near-optimal scaling. The loop `updatek` contains global reductions, and thus at scale, it is bound by the latency of communications. At 128 nodes (4096 cores) it becomes the single most expensive loop. Loops over boundary sets, such as `period` and `periodicity` scale relatively worse than loops over interior sets, since fewer partitions carry out operations



(a) Strong Scaling (2.5M edges)

(b) Weak Scaling (0.5M edges per node)

Figure 10.3. Scaling performance per loop runtime breakdowns
on JADE (NASA Rotor 37, 20 iterations)

over elements in those sets.

Per-loop breakdowns when strong scaling on the Jade GPU cluster are shown in Figure 10.3a. Observe how the performance of different loops is much more spread out compared to those on the CPU cluster scaling (as shown in Figure 10.2a). Also note how boundary loops such as `period` and `periodicity` are not so much faster than loops over interior sets, which is again due to GPU utilisation. While the loop with reductions (`updatek`) was showing good scaling on the CPU up to about 512 cores, performance stagnates beyond 4 GPUs which is a result of the near-static overhead of on-device reductions and the transferring of data to the host, all of which are primarily latency-limited. Most other loops, such as `vfluxedge, ifluxedge, accumedges` and `srcsa`, scale with 65-80% performance gain when the number of GPUs are doubled, however `edgecon` only shows a 48-60% increase due to the loop being dominated by indirect updates of memory and an increasingly poor colouring at scale.

Figure 10.2b shows timing breakdowns for the same loops when weak scaling on HECToR, with very little increase in time for loops over interior sets and a slight reduction in time for boundary loops, as a result of the boundary (surface) of the problem becoming smaller relative to the interior. Similar results can be observed in Figure 10.3b when weak scaling on the GPU cluster. Here, some of the bigger loops get relatively slower, due to the load imbalance between different GPUs. In this case some partitions need more colours than others for execution, which in turn slows down execution. Boundary loops such as `period` and `periodicity` become slower as more partitions share elements of the boundary set forcing halo exchanges that are limited by latency.

DOI:10.15774/PPKE.ITK.2014.003

Bibliography

Journal publications by the author

[1] M. B. Giles and **I. Z. Reguly**. “Trends in high performance computing for engineering calculations”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* (2014). Accepted with minor revisions.

[2] G. R. Mudalige, M. B. Giles, J. Thiyagalingam, **I. Z. Reguly**, C. Bertolli, P. H. J. Kelly, and A. E. Trefethen. “Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems”. In: *Parallel Computing* 39.11 (2013), pp. 669–692. DOI: `10.1016/j.parco.2013.09.004`.

[3] **I. Z. Reguly** and M. Giles. “Finite Element Algorithms and Data Structures on Graphical Processing Units”. In: *International Journal of Parallel Programming* (2013). ISSN: 0885-7458. DOI: `10.1007/s10766-013-0301-6`.

[4] **I. Z. Reguly**, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. J. Kelly, and D. Radford. “Acceleration of a Full-scale Industrial CFD Application with OP2”. In: *submitted to ACM Transactions on Parallel Computing* (2013). Available at: `http://arxiv.org/abs/1403.7209`.

[5] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and **I. Z. Reguly**. “Designing OP2 for GPU Architectures”. In: *Journal Parallel and Distributed Computing* 73.11 (Nov. 2013), pp. 1451–1460. DOI: `10.1016/j.jpdc.2012.07.008`.

[6] **I. Z. Reguly**, E. László, G. R. Mudalige, and M. B. Giles. “Vectorizing Unstructured Mesh Computations for Many-core Architectures”. In: *submitted to Concurrency and Computation: Practice and Experience special issue on programming models and applications for multicores and manycores* (2014).

[7] L. Rovigatti, P. Šulc, **I. Z. Reguly**, and F. Romano. “A comparison between parallelisation approaches in molecular dynamics simulations on GPUs”. In: *submitted*

DOI:10.15774/PPKE.ITK.2014.003

to The Journal of Chemical Physics (2014). Available at: http://arxiv.org/abs/1401.4350.

International conference publications by the author

[8] **I. Z. Reguly** and M. B. Giles. "Efficient sparse matrix-vector multiplication on cache-based GPUs." In: *Proceedings of Innovative Parallel Computing (InPar '12).* San Jose, CA. US.: IEEE, May 2012. DOI: 10.1109/InPar.2012.6339602.

[9] M. Giles, G. Mudalige, C. Bertolli, P. Kelly, E. László, and **I. Z. Reguly**. "An Analytical Study of Loop Tiling for a Large-Scale Unstructured Mesh Application". In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* 2012, pp. 477–482.

[10] G. R. Mudalige, **I. Z. Reguly**, M. B. Giles, C. Bertolli, and P. H. J. Kelly. "OP2: An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures." In: *Proceedings of Innovative Parallel Computing (InPar '12).* San Jose, CA. US.: IEEE, May 2012. DOI: 10.1109/InPar.2012.6339594.

[11] **I. Z. Reguly**, E. László, G. R. Mudalige, and M. B. Giles. "Vectorizing Unstructured Mesh Computations for Many-core Architectures ". In: *Proceedings of the 2014 International Workshop on Programming Models and Applications for Multicores and Manycores.* PMAM '14. Orlando, Florida, USA: ACM, 2014. DOI: 10.1145/2560683.2560686.

Other publications by the author

[12] **I. Z. Reguly** and M. B. Giles. "Efficient and scalable sparse matrix-vector multiplication on cache-based GPUs." In: *Sparse Linear Algebra Solvers for High Performance Computing Workshop.* July 8-9, Warwick, UK, 2013.

[13] **I. Z. Reguly**, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. J. Kelly., and D. Radford. "Acceleration of a Full-scale Industrial CFD Application with OP2". In: *UK Many-Core Developer Conference 2013 (UKMAC'13).* December 16, Oxford, UK, 2013.

[14] **I. Z. Reguly**, M. B. Giles, G. R. Mudalige, and C. Bertolli. "Finite element methods in OP2 for heterogeneous architectures". In: *European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS 2012)*. September 10-14, Vienna, Austria, 2012.

[15] **I. Z. Reguly**, M. B. Giles, G. R. Mudalige, and C. Bertolli. "OP2: A library for unstructured grid applications on heterogeneous architectures". In: *European Numerical Mathematics and Advanced Applications (ENUMATH 2013)*. August 26-30, Lausanne, Switzerland, 2013.

Related publications

[16] M. B. Giles, G. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly. "Performance Analysis and Optimization of the OP2 Framework on Many-Core Architectures". In: *The Computer Journal* 55.2 (2012), pp. 168–180.

[17] S. Mitra. "Carbon Nanotube Computer: Transforming Scientific Discoveries into Working Systems". In: *Proceedings of the 2014 on International Symposium on Physical Design*. ISPD '14. Petaluma, California, USA: ACM, 2014, pp. 117–118. ISBN: 978-1-4503-2592-9. DOI: 10.1145/2560519.2565872. URL: http://doi.acm.org/10.1145/2560519.2565872.

[18] G. Kestor, R. Gioiosa, D. Kerbyson, and A. Hoisie. "Quantifying the energy cost of data movement in scientific applications". In: *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. 2013, pp. 56–65. DOI: 10.1109/IISWC.2013.6704670.

[19] Top500. *Top 500 Supercomputer Sites*. http://www.top500.org/. 2010.

[20] P. Kogge and J. Shalf. "Exascale Computing Trends: Adjusting to the New Normal for Computer Architecture". In: *Computing in Science Engineering* 15.6 (Nov. 2013), pp. 16–26. ISSN: 1521-9615. DOI: 10.1109/MCSE.2013.95.

[21] M. D. Lam, E. E. Rothberg, and M. E. Wolf. "The cache performance and optimizations of blocked algorithms". In: *ACM SIGARCH Computer Architecture News*. Vol. 19. 2. ACM. 1991, pp. 63–74.

[22] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. "The Pochoir Stencil Compiler". In: *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: ACM, 2011, pp. 117–128. ISBN: 978-1-4503-0743-7. DOI: 10.1145/1989493.1989508.

DOI:10.15774/PPKE.ITK.2014.003

[23] B. Dally. "Power, Programmability, and Granularity: The Challenges of ExaScale Computing". In: *Proceedings of the Parallel Distributed Processing Symposium (IPDPS'11)*. 2011, pp. 878–878. DOI: 10.1109/IPDPS.2011.420.

[24] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. "Avoiding communication in sparse matrix computations". In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE. 2008, pp. 1–12.

[25] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec. 2006. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[26] K. Brown, A. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. "A Heterogeneous Parallel Framework for Domain-Specific Languages". In: *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. Oct. 2011, pp. 89–100. DOI: 10.1109/PACT.2011.15.

[27] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. "Liszt: a domain specific language for building portable mesh-based PDE solvers". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 9:1–9:12. ISBN: 978-1-4503-0771-0.

[28] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. J. Kelly. "Deriving Efficient Data Movement from Decoupled Access/Execute Specifications". In: *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*. HiPEAC '09. Paphos, Cyprus: Springer-Verlag, 2009, pp. 168–182. ISBN: 978-3-540-92989-5.

[29] T. Muranushi. "Paraiso : An Automated Tuning Framework for Explicit Solvers of Partial Differential Equations". In: *Computational Science & Discovery* 5.1 (2012), p. 015003.

[30] D. A. Orchard, M. Bolingbroke, and A. Mycroft. "Ypnos: Declarative, Parallel Structured Grid Programming". In: *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*. DAMP '10. Madrid, Spain: ACM, 2010, pp. 15–24. ISBN: 978-1-60558-859-9.

DOI:10.15774/PPKE.ITK.2014.003

[31] T. Brandvik and G. Pullan. "SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms". In: *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*. CIT '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1181–1188. ISBN: 978-0-7695-4108-2.

[32] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, K. Olukotun, T. Rompf, and M. Odersky. "Implementing Domain-Specific Languages For Heterogeneous Parallel Computing". In: *IEEE Micro* 31 (2011), pp. 42–52.

[33] N. Bell and M. Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation, Dec. 2008.

[34] J. Xue. *Loop tiling for parallelism*. Springer, 2000.

[35] U. Drepper. *What Every Programmer Should Know About Memory*. 2007.

[36] R. P. L. Jr. and C. S. Ellis. "Page placement policies for {NUMA} multiprocessors". In: *Journal of Parallel and Distributed Computing* 11.2 (1991), pp. 112 –129. ISSN: 0743-7315. DOI: `http://dx.doi.org/10.1016/0743-7315(91)90117-R`. URL: `http://www.sciencedirect.com/science/article/pii/074373159190117R`.

[37] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*. 2007. URL: `http://developer.download.nvidia.com/compute/cuda/1\_0/NVIDIA\_CUDA\_Programming\_Guide\_1.0.pdf`.

[38] *Tesla Kepler GPU Accelerators*. `http://www.nvidia.co.uk/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf`. 2012.

[39] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*. 2008. URL: `http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf`.

[40] *OpenACC — Directives for Accelerators*. `http://www.openacc-standard.org`.

[41] *CUDA C Best Practices Guide*. 4.0. NVIDIA Corporation. 2011. URL: `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA\_C\_Best\_Practices\_Guide.pdf`.

[42] NVIDIA. *Kepler GK110 Whitepaper*. 2012. URL: `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[43] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Cambridge University Press, 1987.

[44] Y. Saad. *Iterative Methods for Sparse Linear Systems.* 2nd. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003. ISBN: 0898715342.

[45] M. S. Gockenbach. *Understanding and implementing the finite element method.* SIAM, 2006. ISBN: 978-0-89871-614-6.

[46] F. Vázquez, J. Fernández, and E. Garzón. "Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach". In: *Parallel Computing* (2011). ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.08.003.

[47] G. R. Markall, D. A. Ham, and P. H. Kelly. "Towards generating optimised finite element solvers for GPUs from high-level specifications". In: *Procedia Computer Science* 1.1 (2010), pp. 1815 –1823. ISSN: 1877-0509. DOI: 10.1016/j.procs.2010.04.203. URL: http://www.sciencedirect.com/science/article/pii/S1877050910002048.

[48] C. Cantwell, S. Sherwin, R. Kirby, and P. Kelly. "From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements". In: *Computers & Fluids* 43.1 (2011). Symposium on High Accuracy Flow Simulations. Special Issue Dedicated to Prof. Michel Deville, pp. 23 –28. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2010.08.012. URL: http://www.sciencedirect.com/science/article/pii/S0045793010002057.

[49] G. Alefeld. "On the convergence of the symmetric SOR method for matrices with red-black ordering". In: *Numerische Mathematik* 39.1 (1982), pp. 113–117. ISSN: 0029-599X. DOI: 10.1007/BF01399315.

[50] E. L. Poole and J. M. Ortega. "Multicolor ICCG Methods for Vector Computers". In: *SIAM J. Numer. Anal.* 24.6 (1987), pp. 1394–1418. ISSN: 00361429.

[51] C. Cecka, A. J. Lew, and E. Darve. "Assembly of finite element methods on graphics processors". In: *International Journal for Numerical Methods in Engineering* 85.5 (2011), pp. 640–669. ISSN: 1097-0207. DOI: 10.1002/nme.2989. URL: http://dx.doi.org/10.1002/nme.2989.

[52] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid". In: *ACM Transactions on Graphics* 22 (2003), pp. 917–924.

[53] D. Göddeke, R. Strzodka, and S. Turek. "Accelerating Double Precision FEM Simulations with GPUs". In: *18th Symposium Simulationstechnique (ASIM'05).* Ed. by

F. Hülsemann, M. Kowarschik, and U. Rüde. Frontiers in Simulation. Sept. 2005, pp. 139–144.

[54] D. Komatitsch, D. Michéa, and G. Erlebacher. "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA". In: *Journal of Parallel and Distributed Computing* 69.5 (2009), pp. 451 –460. ISSN: 0743-7315. DOI: `10.1016/j.jpdc.2009.01.006`. URL: `http://www.sciencedirect.com/science/article/pii/S0743731509000069`.

[55] D. Komatitsch, D. Göddeke, G. Erlebacher, and D. Michéa. "Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs". In: *Computer Science Research and Development* 25.1-2 (2010), pp. 75–82. URL: `http://www.springerlink.com/index/10.1007/s00450-010-0109-1`.

[56] C. Flaig and P. Arbenz. "A scalable memory efficient multigrid solver for micro-finite element analyses based on CT images". In: *Parallel Computing* 37.12 (2011). 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10), pp. 846 –854. ISSN: 0167-8191. DOI: `10.1016/j.parco.2011.08.001`. URL: `http://www.sciencedirect.com/science/article/pii/S0167819111001037`.

[57] J. Filipovic, I. Peterlik, and J. Fousek. "GPU Acceleration of Equations Assembly in Finite Elements Method – Preliminary Results". In: *SAAHPC : Symposium on Application Accelerators in HPC* (2009).

[58] P. Plaszewski, P. Maciol, and K. Banas. "Finite element numerical integration on GPUs". In: *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I.* PPAM'09. Wroclaw, Poland: Springer-Verlag, 2010, pp. 411–420. ISBN: 3-642-14389-X, 978-3-642-14389-2. URL: `http://dl.acm.org/citation.cfm?id=1882792.1882842`.

[59] NVIDIA. *cuSPARSE library, last accessed Dec 20th.* `http://developer.nvidia.com/cuSPARSE`. 2012.

[60] NVIDIA. *NVIDIA Tesla C2070 techinical specifications, last accessed Aug 20th.* `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf`. 2012.

[61] NVIDIA. *CUBLAS library, last accessed Sept 12th.* `http://developer.nvidia.com/cublas`. 2013.

[62] K. J. Fidkowski, T. A. Oliver, J. Lu, and D. L. Darmofal. “p-Multigrid solution of high-order discontinuous Galerkin discretizations of the compressible Navier-Stokes equations”. In: *J. Comput. Phys.* 207.1 (July 2005), pp. 92–113. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2005.01.005.

[63] M. M. Baskaran and R. Bordawekar. “Optimising Sparse Matrix-Vector Multiplication on GPUs”. In: *In Ninth SIAM Conference on Parallel Processing for Scientific Computing* (2008).

[64] A. Dziekonski, A. Lamecki, and M. Mrozowski. “A memory efficient and fast sparse matrix vector product on a GPU”. In: *Progress In Electromagnetics Research* 116 (2011), pp. 49–63. DOI: 10.2528/PIER11031607. URL: http://www.jpier.org/pier/pier.php?paper=11031607.

[65] A. H. El Zein and A. P. Rendell. “Generating optimal CUDA sparse matrix-vector product implementations for evolving GPU hardware”. In: *Concurrency and Computation: Practice and Experience* 24.1 (2012), pp. 3–13. ISSN: 1532-0634. DOI: 10.1002/cpe.1732. URL: http://dx.doi.org/10.1002/cpe.1732.

[66] F. Vázquez, G. Ortega, J. Fernández, and E. Garzón. “Improving the Performance of the Sparse Matrix Vector Product with GPUs”. In: *Proceeding of the 10th International Conference on Computer and Information Technology (CIT).* 2010, pp. 1146 –1151. DOI: 10.1109/CIT.2010.208.

[67] T. A. Davis and Y. Hu. “The University of Florida Sparse Matrix Collection”. In: *ACM Transactions on Mathematical Software* 38.1 (2011), pp. 1–25.

[68] R. H. Bisseling. *Parallel scientific computation.* OUP Oxford, 2004.

[69] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface.* Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-57132-3.

[70] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial (2Nd Ed.)* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000. ISBN: 0-89871-462-1.

[71] G. R. Markall, F. Rathgeber, L. Mitchell, N. Loriant, C. Bertolli, D. Ham, and P. Kelly. “Performance-Portable Finite Element Assembly Using PyOP2 and FEniCS”. In: *Supercomputing.* Ed. by J. Kunkel, T. Ludwig, and H. Meuer. Vol. 7905. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 279–289. ISBN:

978-3-642-38749-4. DOI: 10.1007/978-3-642-38750-0_21. URL: http://dx.doi.org/10.1007/978-3-642-38750-0_21.

[72] P. I. Crumpton and M. B. Giles. “Multigrid Aircraft Computations Using the OPlus Parallel Library”. In: *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers* (). A. Ecer, J. Periaux, N. Satofuka, and S. Taylor, (eds.), North-Holland, 1996., pp. 339–346.

[73] M. B. Giles, D. Ghate, and M. C. Duta. “Using Automatic Differentiation for Adjoint CFD Code Development”. In: *Computational Fluid Dynamics Journal* 16.4 (2008), pp. 434–443.

[74] M. B. Giles, M. C. Duta, J.-D. Müller, and N. A. Pierce. “Algorithm Developments for Discrete Adjoint Methods”. In: *AIAA Journal* 42.2 (2003), pp. 198–205.

[75] D. Dutykh, R. Poncet, and F. Dias. “The VOLNA code for the numerical modeling of tsunami waves: Generation, propagation and inundation”. In: *European Journal of Mechanics - B/Fluids* 30.6 (2011). <ce:title>Special Issue: Nearshore Hydrodynamics</ce:title>, pp. 598 –615. ISSN: 0997-7546. DOI: 10.1016/j.euromechflu.2011.05.005. URL: http://www.sciencedirect.com/science/article/pii/S0997754611000574.

[76] P. Moinier, J.-D. Muller, and M. Giles. “Edge-based multigrid and preconditioning for hybrid grids”. In: *AIAA Journal* 40 (10 2002), pp. 1954–1960.

[77] M. Duta, M. Giles, and M. Campobasso. “The harmonic adjoint approach to unsteady turbomachinery design”. In: *International Journal for Numerical Methods in Fluids* 40 (3-4 2002), pp. 323–332.

[78] M. B. Giles, M. C. Duta, J. D. Muller, and N. A. Pierce. “Algorithm Developments for Discrete Adjoint Methods”. In: *AIAA Journal* 42.2 (2003), pp. 198–205.

[79] L. Lapworth. “The Challenges for Aero-Engine CFD”. In: (2008). Invited Lecture, ICFD 25th Anniversary Meeting, Oxford, UK.

[80] D. A. Burgess, P. I. Crumpton, and M. B. Giles. “A Parallel Framework for Unstructured Grid Solvers”. In: *Computational Fluid Dynamics’94:Proceedings of the Second European Computational Fluid Dynamics Conference*. Ed. by S. Wagner, E. Hirschel, J. Periaux, and R. Piva. John Wiley and Sons, 1994, pp. 391–396.

DOI:10.15774/PPKE.ITK.2014.003

[81] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. S. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. *Addressing Failures in Exascale Computing.* Tech. rep. ANL/MCS-TM-332. 2013.

[82] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts.* 5th ed. New York, NY, USA: McGraw-Hill, Inc., 2006. ISBN: 0072958863, 9780072958867.

[83] *OP2 GitHub Repository.* `https://github.com/OP2/OP2-Common`. 2013.

[84] J. Launchbury. "A Natural Semantics for Lazy Evaluation". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '93. Charleston, South Carolina, USA: ACM, 1993, pp. 144–154. ISBN: 0-89791-560-7. DOI: `10.1145/158511.158618`. URL: `http://doi.acm.org/10.1145/158511.158618`.

[85] J. Agulleiro, F. Vázquez, E. Garzón, and J. Fernández. "Hybrid computing: CPU+GPU co-processing and its application to tomographic reconstruction". In: *Ultramicroscopy* 115.0 (2012), pp. 109 –114. ISSN: 0304-3991. DOI: `http://dx.doi.org/10.1016/j.ultramic.2012.02.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0304399112000344`.

[86] C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng. "A peta-scalable CPU-GPU algorithm for global atmospheric simulations". In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming.* PPoPP '13. Shenzhen, China: ACM, 2013, pp. 1–12. ISBN: 978-1-4503-1922-5. DOI: `10.1145/2442516.2442518`. URL: `http://doi.acm.org/10.1145/2442516.2442518`.

[87] B. Bilel, N. Navid, and M. Bouksiaa. "Hybrid CPU-GPU Distributed Framework for Large Scale Mobile Networks Simulation". In: *Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on.* 2012, pp. 44–53.

[88] *Hybrid-CPU/GPU execution mode with GROMACS.* `http://www.scalalife.eu/content/hybrid-cpu-gpu-execution-mode-gromacs`. 2013.

[89] *ParMETIS.* `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

[90] *Scotch and PT-Scotch.* `http://www.labri.fr/perso/pelegrin/scotch/`.

DOI:10.15774/PPKE.ITK.2014.003

[91] M. Frigo and S. Johnson. “The Design and Implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 216 –231. ISSN: 0018-9219.

[92] H. D. Simon. *Partitioning of Unstructured Problems for Parallel Processing.* Tech. rep. RNR-91-008. Ames Research Center Moffett Field, California 94035-1000: National Aeronautics and Space Administration, Feb. 1991. URL: `www.nas.nasa.gov/assets/pdf/techreports/1991/rnr-91-008.pdf`.

[93] ISO/IES. *Further Interoperability of Fortran with C.* ISO SO/IEC TS 29113:2012. Geneva, Switzerland: International Organisation for Standardization and International Electrotechnical Commission, 2012.

[94] D. Burgess and M. Giles. “Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines”. In: *Advances in Engineering Software* 28 (3 1997), pp. 198–201.

[95] C. Lameter. “An overview of non-uniform memory access”. In: *Commun. ACM* 56.9 (Sept. 2013), pp. 59–54. ISSN: 0001-0782.

[96] P. Lindberg. *Basic OpenMP Threading Overhead.* Tech. rep. `http://software.intel.com/en-us/articles/basic-openmp-threading-overhead`. Intel, 2009.

[97] D. Dutykh, R. Poncet, and F. Dias. “The VOLNA code for the numerical modeling of tsunami waves: Generation, propagation and inundation”. In: *European Journal of Mechanics - B/Fluids* 30.6 (2011), pp. 598 –615. ISSN: 0997-7546. DOI: `j.euromechflu.2011.05.005`.

[98] I. Smith, D. Griffiths, and L. Margetts. *Programming the Finite Element Method.* Wiley, 2013. ISBN: 9781118535936. URL: `http://books.google.co.uk/books?id=iUaaAAAAQBAJ`.

[99] *NVIDIA AmgX Library.* `https://developer.nvidia.com/amgx`. 2013.