DOI:10.15774/PPKE.ITK.2016.002

Pázmany Péter Catholic University

Doctoral Thesis

Parallelization of Numerical Methods on Parallel Processor Architectures

Author:
Endre László

Thesis Advisor:
Dr. Péter Szolgay, D.Sc.

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Roska Tamás Doctoral School of Sciences and Technology
Faculty of Information Technology and Bionics

February 12, 2016

DOI:10.15774/PPKE.ITK.2016.002

DOI:10.15774/PPKE.ITK.2016.002

Abstract

Common parallel processor microarchitectures offer a wide variety of solutions to implement numerical algorithms. The efficiency of different algorithms applied to the same problem vary with the underlying architecture which can be a multi-core CPU (Central Processing Unit), many-core GPU (Graphics Processing Unit), Intel's MIC (Many Integrated Core) or FPGA (Field Programmable Gate Array) architecture. Significant differences between these architectures exist in the ISA (Instruction Set Architecture) and the way the compute flow is executed. The way parallelism is expressed changes with the ISA, thread management and customization available on the device. These differences pose restrictions for the efficiency of the algorithm to be implemented. The aim of the doctoral work is to analyze the efficiency of the algorithms through the architectural differences and find efficient ways and new efficient algorithms to map problems to the selected parallel processor architectures. The problems selected for the study are numerical algorithms from three problem classes of the 13 Berkeley "dwarves" [1].

Engineering, scientific and financial applications often require the simultaneous solution of a large number of independent tridiagonal systems of equations with varying coefficients. The dissertation investigates the optimal choice of tridiagonal algorithm for CPU, Intel MIC and NVIDIA GPU with a focus on minimizing the amount of data transfer to and from the main memory using novel algorithms and register blocking mechanism, and maximizing the achieved bandwidth. It also considers block tridiagonal solutions which are sometimes required in CFD (Computational Fluid Dynamic) applications. A novel work-sharing and register blocking based Thomas solver is also presented.

Structured grid problems, like the ADI (Alternating Direction Implicit) method which boils down the solution of PDEs (Partial Differential Equation) into a number of solutions of tridiagonal system of equations is shown to improve performance by utilizing new, efficient tridiagonal solvers. Also, solving the one-factor Black-Scholes option pricing PDE with explicit and implicit time-marching algorithms on FPGA solution with the Xilinx Vivado HLS (High Level Synthesis) is presented. Performance of the FPGA solver is analyzed and efficiency is discussed. A GPU based implementation of a CNN (Cellular Neural Network) simulator using NVIDIA's Fermi architecture is presented.

The OP2 project at the University of Oxford aims to help CFD domain scientists to decrease their effort to write efficient, parallel CFD code. In order to increase the efficiency of parallel incrementation on unstructured grids OP2 utilizes a mini-partitioning and a two level coloring scheme to identify parallelism during run-time. The use of the GPS (Gibbs-Poole-Stockmeyer) matrix bandwidth minimization algorithm is proposed to help create better mini-partitioning and block coloring by improving the locality of neighboring blocks (also known as mini-paritions) of the original mesh. Exploiting the capabilities of OpenCL for achieving better SIMD (Single Instruction Multiple Data) vectorization on CPU and MIC architectures throught the use of the SIMT (Single Instruction Multiple Thread) programming approach in OP2 is also discussed.

DOI:10.15774/PPKE.ITK.2016.002

Acknowledgements

I would like to thank my supervisor Prof. Dr. Péter Szolgay from the Pázmány Péter Catholic University for his guidance and valuable support.

I am also grateful to my colleagues István, Zoltán, Norbi, Bence, András, Balázs, Antal, Csaba, Vamsi, Dániel and many others whose discussion helped in the hard problems.

I spent almost two years at the University of Oxford, UK where I met dozens of great minds. First of all, I would like to thank for the supervision of Prof. Dr. Mike Giles who taught me a lot about parallel algorithms, architectures and numerical methods. I am also grateful to Rahim Lakhoo for the disucssions on technical matters, Eamonn Maguire for his inspiring and motivating thoughts.

All of these would not have been possible without the support of Hajni and my family throughout my PhD studies.

I would like to acknowledge the support of TÁMOP-4.2.1.B-11/2/KMR-2011-0002 and TÁMOP-4.2.2/B-10/1-2010-0014 and Campus Hungary. The research at the Oxford e-Research Centre has been partially supported by the ASEArch project on Algorithms and Software for Emerging Architectures, funded by the UK Engineering and Physical Sciences Research Council. Use of the facilities of the University of Oxford Advanced Research Computing (ARC) is also acknowledged.

DOI:10.15774/PPKE.ITK.2016.002

Contents

DOI:10.15774/PPKE.ITK.2016.002

DOI:10.15774/PPKE.ITK.2016.002

DOI:10.15774/PPKE.ITK.2016.002

List of Figures

DOI:10.15774/PPKE.ITK.2016.002

DOI:10.15774/PPKE.ITK.2016.002

DOI:10.15774/PPKE.ITK.2016.002

DOI:10.15774/PPKE.ITK.2016.002

List of Tables

DOI:10.15774/PPKE.ITK.2016.002

Abbreviations

ADI	**A**lternating **D**irections **I**mplicit method
ALU	**A**rithmetic **L**ogic **U**nit
AoS	**A**rray **o**f **S**tructures
ARM	**A**dvanced **R**ISC **M**achine Holdings Plc.
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
ASM	**As**se**m**bly
AVX	**A**dvanced **V**ector E**x**tensions
BRAM	**B**lock **RAM**
BS	**B**lack-**S**choles
CFD	**C**omputational **F**luid **D**ynamics
CLB	**C**onfigurable **L**ogic **B**lock
CMOS	**C**omplementary **M**etal-**O**xide-**S**emiconductor
CNN	**C**ellular **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
CR	**C**yclic **R**eduction
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DDR	**D**ouble **D**ata **R**ate
DP	**D**ouble **P**recision
DSL	**D**omain **S**pecific **L**anguage
DSP	**D**igital **S**ignal **P**rocessing
FLOP	**F**loating **P**oint **OP**eration
FLOPS	**F**loating **P**oint **O**erations **P**er **S**econd
FMA	**F**used **M**ultiply and **A**dd
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GPS	**G**ibbs-**P**oole-**S**tockmeyer

DOI:10.15774/PPKE.ITK.2016.002

GPGPU	**G**eneral **P**urpose **GPU**
GPU	**G**raphics **P**rocessing **U**nit
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**igh **P**erformance **C**omputing
HT	**H**yper **T**hreading
HTT	**H**yper **T**hreading **T**echnology
IC	**I**ntegrated **C**ircuit
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
IMCI	**I**nitial **M**any **C**ore **I**nstructions
IPP	(Intel) **I**ntegrated **P**erformance **P**rimitives
ISA	**I**nstruction **S**et **A**rchitecture
KNC	**KN**ights **L**anding
L1/2/3	**L**evel 1/2/3
LAPACK	**L**inear **A**lgebra **Pack**age
LHS	**L**eft **H**and **S**ide
LLC	**L**ast **L**evel **C**ache
MAD	**M**ultiply **A**nd **A**dd
MIC	**M**any **I**ntegrated **C**ore
MKL	**M**ath **K**ernel **L**ibrary
MLP	**M**ulti **L**ayer **P**erceptron
MOS	**M**etal-**O**xide-**S**emiconductor
MPI	**M**essage **P**assing **I**nterface
MSE	**M**ean **S**quare **E**rror
NNZ	**N**umber of **N**on-**Z**ero elements
NUMA	**N**on-**U**niform **M**emory **A**ccess
NVVP	**NV**IDIA **V**isual **P**rofiler
NZ	**N**on-**Z**ero element
ODE	**O**rdinary **D**ifferential **E**quations
OpenACC	**Open Acc**elerators
OpenCL	**Open C**omputing **L**anguage
OpenMP	**Open M**ulti-**P**rocessing
PCR	**P**arallel **C**yclic **R**eduction

DOI:10.15774/PPKE.ITK.2016.002

PDE	**P**artial **D**ifferential **E**quations
PTX	**P**arallel **T**hread E**x**execution
RAM	**R**andom **A**ccess **M**emory
RHS	**R**ight **H**and **S**ide
ScaLAPACK	**Sca**lable **L**inear **A**lgebra **Pack**age
SDRAM	**S**ynchronous **D**ynamic **RAM**
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SIMT	**S**ingle **I**nstruction **M**ultiple **T**hread
SMP	**S**imultaneous **M**ulti **P**rocessing
SMT	**S**imultaneous **M**ulti **T**hreading
SMX	**S**treaming **M**ultiprocessor - e**X**tended
SoA	**S**tructures **o**f **A**rray
SP	**S**ingle **P**recision
SPARC	**S**calable **P**rocessor **Arc**hitecture International, Inc.
TLB	**T**ranslation **L**ookaside **B**uffer
TDP	**T**hermal **D**esign **P**ower
UMA	**U**niform **M**emory **A**ccess
VHDL	**VHSIC** **H**ardware **D**escription **L**anguage
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
VLIW	**V**ery **L**ong **I**nstruction **W**ord
VLSI	**V**ery **L**arge **S**cale **I**ntegration
WAR	**W**rite **A**fter **R**ead

DOI:10.15774/PPKE.ITK.2016.002

Notations and Symbols

$\mathbb{R}$	Set of real numbers
$\mathbb{R}_+$	Set of non-negative real numbers
V	Volt
W	Watt
GB	Giga bytes, 2^{10} bytes; used instead of IEC GiB standard
GB/s	Giga bytes per second, 2^{10} bytes per second; used instead of IEC GiB/s standard

DOI:10.15774/PPKE.ITK.2016.002

Chapter 1

Introduction

Limits of physics Today the development of every scientific, engineering and financial field that rely on computational methods is severely limited by the stagnation of computational performance caused by the physical limits in the VLSI (Very Large Scale Integration) technology. This is caused by the single processor performance of the CPU (Central Processing Unit) due to vast heat dissipation that can not be increased. Around 2004 this physical limit made it necessary to apply more processors onto a silicon die and so the problem of efficient parallelization of numerical methods and algorithms on a processor with multiple cores was born.

The primary aim of my thesis work is to take advantage of the computational power of various modern parallel processor architectures to accelerate the computational speed of some of the scientific, engineering and financial problems by giving new algorithms and solutions.

After the introduction of the problems and limits behind this paradigm shift in the current section, a more detailed explanation of the classification and complexity of parallelization is presented in Section 1.1. Section 1.2 introduces the numerical problems on which the new scientific results of Section 8.1 rely.

Gordon Moore in the 1960's studied the development of the CMOS (Complementary Metal-Oxide-Semiconductor) manufacturing process of integrated circuits (IC). In 1965 he concluded from five manufactured ICs, that the number of transistors on a given silicon area will double every year. This finding is still valid after half a century with

DOI:10.15774/PPKE.ITK.2016.002

a major modification: the number of transistors on a chip doubles every two years (as opposed to the original year).

This law is especially interesting since around 2004 the manufacturing process reached a technological limit, which prevents us from increasing the clock rate of the digital circuitry. This limit is due to the heat dissipation. The amount of heat dissipated on the surface of a silicon die of size of a few square centimetres is in the order 100s of Watts. This is the absolute upper limit of the TDP (Thermal Design Power) that a processor can have. The ever shrinking VLSI feature sizes cause: 1) the resistance (and impedance) of conductors to increase; 2) the parasitic capacitance to increase; 3) leakage current on insulators like the MOS transistor gate oxide to increase. These increasing resistance and capacitance related parameters limit the clock rate of the circuitry. In order to increase the clock rate with these parameters the current needs to be increased and that leads to increased power dissipation. Besides limiting the clock rate the data transfer rate is also limited by the same physical facts.

The TDP of a digital circuit is composed in the following way: $P_{TDP} = P_{DYN} + P_{SC} + P_{LEAK}$, where P_{DYN} is the dynamic switching, P_{SC} is the instantaneous short circuit (due to non-zero rise or fall times) and P_{LEAK} is the leakage current induced power dissipation. The most dominant power dissipation is due to the dynamic switching $P_{DYN} \propto CV^2f$, where $\propto$ indicates proportionality, f is the clock rate, C is the capacitance arising in the circuitry and V is the voltage applied to the capacitances.

Temporary solution to avoid the limits of physics – Parallelisation Earlier, the continuous development of computer engineering meant increasing the clock rate and the number of transistors on a silicon die, as it directly increased the computational power. Today, the strict focus is on increasing the computational capacity with new solutions rather than increasing the clock rate on the chip. One solution is to parallelise on all levels of a processor architecture with the cost of cramming more processors and transistors on a single die. The parallelisation and specialization of hardware necessarily increases the hardware, algorithmic and software complexity. Therefore, since 2004 new processor architectures appeared with multiple processor cores on a single silicon die. Also, more and more emphasise is put to increase the parallelism on the lowest levels of the architecture. As there are many levels of parallelism built into the systems today,

DOI:10.15774/PPKE.ITK.2016.002

the classification of these levels is important to understand what algorithmic features can be exploited during the development of an algorithm.

Amdahl's law Gene Amdahl at a conference in 1967 [2] gave a talk on the achievable scaling of single processor performance to multiple processors assuming a fixed amount of work to be performed. Later, this has been formulated as Eq. (1.1) and now it is know as Amdahl's law. Amdahl's law states that the speedup due to putting the P part of the workload of a single processor onto N identical processors results in $S(N)$ speedup compared to single processor.

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{1.1}$$

One may think of the proportion P as the workload that has been parallelized by an algorithm (and implementation). The larger this proportion, the better the scaling of the implementation will be. This setup is also known as strong scaling. This is a highly important concept that is implicitly used in the arguments when parallelization is discussed.

Gustafson's law John Gustafson in 1988 reevaluated Amdahl's law [3] and stated that in the framework of weak scaling the speedup is given by Eq. (1.2). Weak scaling measures the execution time when the problem size is increased N fold when the problem is scheduled onto N parallel processors. Eq. (1.2) states that the execution time ($S_{latency}(s)$) decreases when the latency due to the P proportion - which benefits from the parallelization - speeds up by a factor of s.

$$S_{latency}(s) = 1 - P + sP \tag{1.2}$$

1.1 Classification of Parallelism

The graph presented on Figure 1.1. shows the logical classification of parallelism on all levels. Leaves of the graph show the processor features and software components that implement the parallelism. Processor and software features highlighted as GPU

DOI:10.15774/PPKE.ITK.2016.002

(Graphics Processing Unit) and CPU/MIC (MIC - Many Integrated Core) features are key components used in the work.

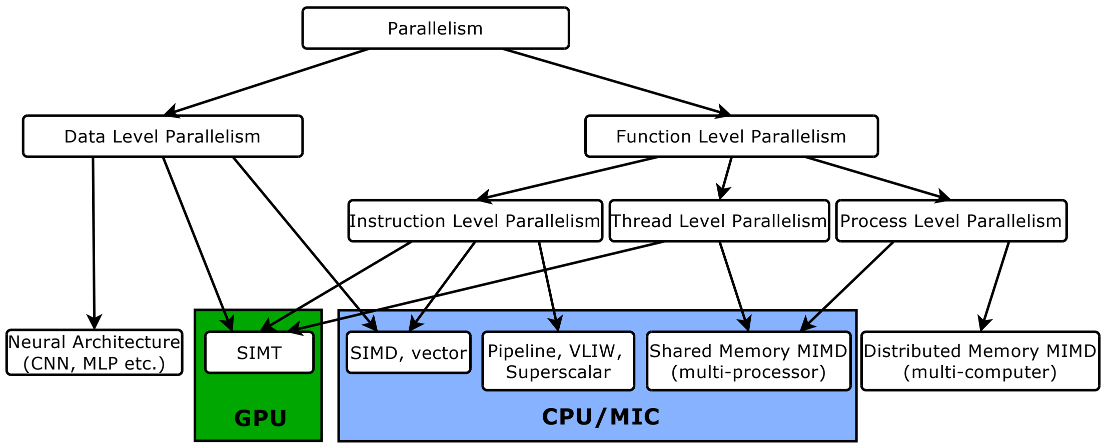

Parallelism
Data Level Parallelism
Function Level Parallelism
Instruction Level Parallelism
Thread Level Parallelism
Process Level Parallelism
Neural Architecture (CNN, MLP etc.)
SIMT
GPU
SIMD, vector
Pipeline, VLIW, Superscalar
Shared Memory MIMD (multi-processor)
CPU/MIC
Distributed Memory MIMD (multi-computer)

FIGURE 1.1: Classification of parallelism along with parallel computer and processor architectures that implement them.

Architecture dependent performance bounds Depending on the design aims of a parallel processor architecture the computational performance of these architectures vary with the problem. The performance can be bound by the lack of some resources needed for that particular problem, eg. more floating point units, greater memory bandwidth etc. Therefore it is common to refer to an implementation being: 1) *compute bound* if further performance gain would only be possible with more compute capability or 2) *memory bandwidth bound* if the problem would gain performance from higher memory bandwidth. The roofline model [4] helps identifying whether the computational performance of the processor on a given algorithm or implementation is bounded by the available computational or memory controller resources and also helps approximating the extent of the utilization of a certain architecture. See Figure 1.2 for comparing parallel processor architectures used in the dissertation. Processor architectures that were recently introduced to the marker and which are to be announced are also noted on the figure.

On Figure 1.2 the graphs were constructed based on the maximum theoretical computational capacity (GFLOP/s) and data bandwidth (GB/s) metrics. In the case of FPGAs the number of implementable multipliers and the number of implementable memory interfaces working at the maximum clock rate gives the base for the calculation of the graphs. The exact processor architectures behind the labels are the following: 1) GPU-K40: NVIDIA Tesla K40m card with Kepler GK110B microarchitecture working with

DOI:10.15774/PPKE.ITK.2016.002

"Boost" clock rate; 2) GPU-K80: NVIDIA Tesla K80 card with Kepler GK210 microarchitecture working with "Boost" clock rate; 3) CPU-SB: Intel Xeon E5-2680 CPU with Sandy Bridge microarchitecture; 4) CPU-HW: Intel Xeon E5-2699v3 CPU with Haswell microarchitecture; 5) MIC-KNC: Intel Xeon Phi 5110P co-processor card with Knights Corner microarchitecture; 6) MIC-KNL: Intel Xeon Phi co-processor with Knights Landing microarchitecture - the exact product signature is yet to be announced; 7) FPGA-V7: Xilinx Virtex-7 XC7VX690T; 8) FPGA-VUSP: Xilinx Virtex UltraScale+ VU13P.

As the aim of my work was achieving the fastest execution time by parallelization the roofline model is used to study the computational performance of the selected problems where applicable.

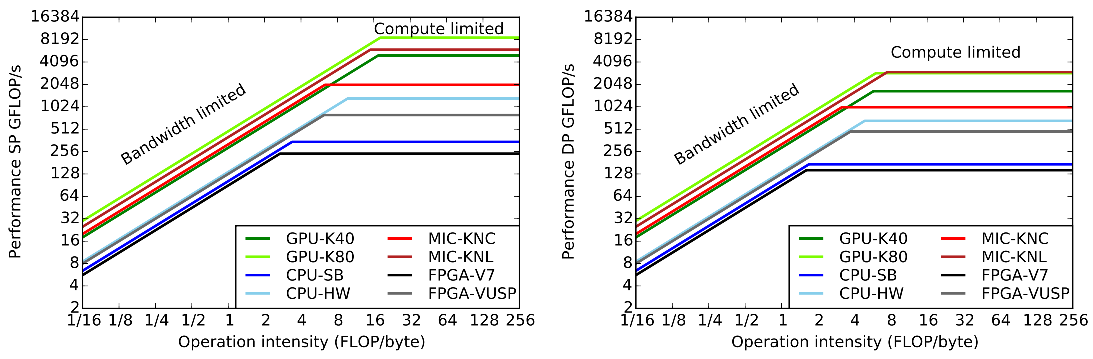


FIGURE 1.2: Roofline model for comparing parallel processor architectures. Note: SP stands for Single Precision and DP stands for Double Precision.

Parallel Processor Architectures and Languages A vast variety of parallel architectures are used and experimented in HPC (High Performance Computing) to compute scientific problems. Multi-core Xeon class server CPU by Intel is the leading architecture used nowadays in HPC. GPUs (Graphics Processing Unit) originally used for graphics-only computing has become a widely used architecture to solve certain problems. In recent years Intel introduced the MIC (Many Integrated Core) architecture in the Xeon Phi coprocessor family. IBM with the POWER and Fujitsu with the SPARC processor families are focusing on HPC. FPGAs (Field Programmable Gate Array) by Xilinx and Altera are also used for the solution of some special problems. ARM as an IP (Intellectual Property) provider is also making new designs for x86 CPU processors which find application in certain areas of HPC. Also, research is conducted to create new heterogenous computing architectures to solve the power efficiency and programmability issues of CPUs and accelerators.

DOI:10.15774/PPKE.ITK.2016.002

Programming languages like FORTRAN, C/C++ or Python are no longer enough to exploit the parallelism of these multi- and many-core architectures in a productive way. Therefore, new languages, language extensions, libraries, frameworks and DSLs (Domain Specific Language) appeared in recent years, see Figure 1.3. Without completeness the most important of these are: 1) CUDA (Compute Unified Device Architecture) C for programming NVIDIA GPUs; 2) OpenMP (Open Multi Processing) directive based languages extension for programming multi-core CPU or many-core MIC architectures; 3) OpenCL (Open Compute Language) for code portable, highly parallel abstraction; 4) AVX (Advanced Vector eXtension) and IMCI (Initial Many Core Instruction) vectorized, SIMD (Single Instruction Multiple Data) ISA (Instruction Set Architecture) and intrinsic instructions for increased ILP (Instruction Level Parallelism) in CPU and MIC; 5) OpenACC (Open Accelerators) directive based language extension for accelerator architectures; 6) HLS (High Level Synthesis) by Xilinx for improved code productivity on FPGAs.

All these new architectural and programming features raise new ways to solve existing parallelisation problems, but non of them provide high development productivity, code- and performance portability as one solution. Therefore, these problems are the topic of many ongoing research in the HPC community.

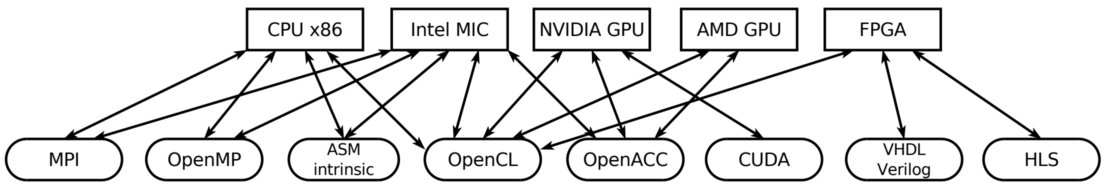


FIGURE 1.3: Relations between processor architectures, languages and language extensions.

Classification of problem classes according to parallelization Classification of the selected problems is based on the 13 "dwarves" of "A view of the parallel computing landscape" [1]. My results are related to the dwarves highlighted with bold fonts:

1. Dense Linear Algebra
2. **Sparse Linear Algebra**
3. Spectral Methods
4. N-Body Methods
5. **Structured Grids**
6. **Unstructured Grids**
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming

DOI:10.15774/PPKE.ITK.2016.002

11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

1.2 Selected and Parallelized Numerical Problems

The selected numerical algorithms cover the fields of numerical mathematics that aim for the solution of PDEs (Partial Differential Equation) in different engineering application areas, such as CFD (Computational Fluid Dynamics), financial engineering, electromagnetic simulation or image processing.

1.2.1 Tridiagonal System of Equations

Engineering, scientific and financial applications often require the simultaneous solution of a large number of independent tridiagonal systems of equations with varying coefficients [5, 6, 7, 8, 9, 10]. The solution of tridiagonal systems also arises when using line-implicit smoothers as part of a multi-grid solver [11], and when using high-order compact differencing [12, 13]. Since the number of systems is large enough to offer considerable parallelism on many-core systems, the choice between different tridiagonal solution algorithms, such as Thomas, CR (Cyclic Reduction) or PCR (Parallel Cyclic Reduction) needs to be re-examined. In my work I developed and implemented near optimal scalar and block tridiagonal algorithms for CPU, Intel MIC and NVIDIA GPU with a focus on minimizing the amount of data transfer to and from the main memory using novel algorithms and register blocking mechanism, and maximizing the achieved bandwidth. The latter means that the achieved computational performance is also maximized (see Figure 1.4) as the solution of tridiagonal system of equations is bandwidth limited due to the operation intensity.

On Figure 1.4 the computational upper limits on the graphs were constructed based on the maximum theoretical computational capacity (GFLOP/s) and data bandwidth (GB/s) metrics of the three processors. The operation intensity is calculated from the amount of data that is moved through the memory bus and the amount of floating point operations performed on this data. The total amount of floating point operations is calculated and divided by the execution time to get the performance in GFLOP/s. The

DOI:10.15774/PPKE.ITK.2016.002

operation intensity and GFLOP/s performance metrics are calculated for each solver and they are represented with stars on the figures.

In most cases the tridiagonal systems are scalar, with one unknown per grid point, but this is not always the case. For example, computational fluid dynamics applications often have systems with block-tridiagonal structure up to 8 unknowns per grid point [7]. The solution of block tridiagonal system of equations are also considered which are sometimes required in CFD (Computational Fluid Dynamic) applications. A novel work-sharing and register blocking based Thomas solver for GPUs is also created.

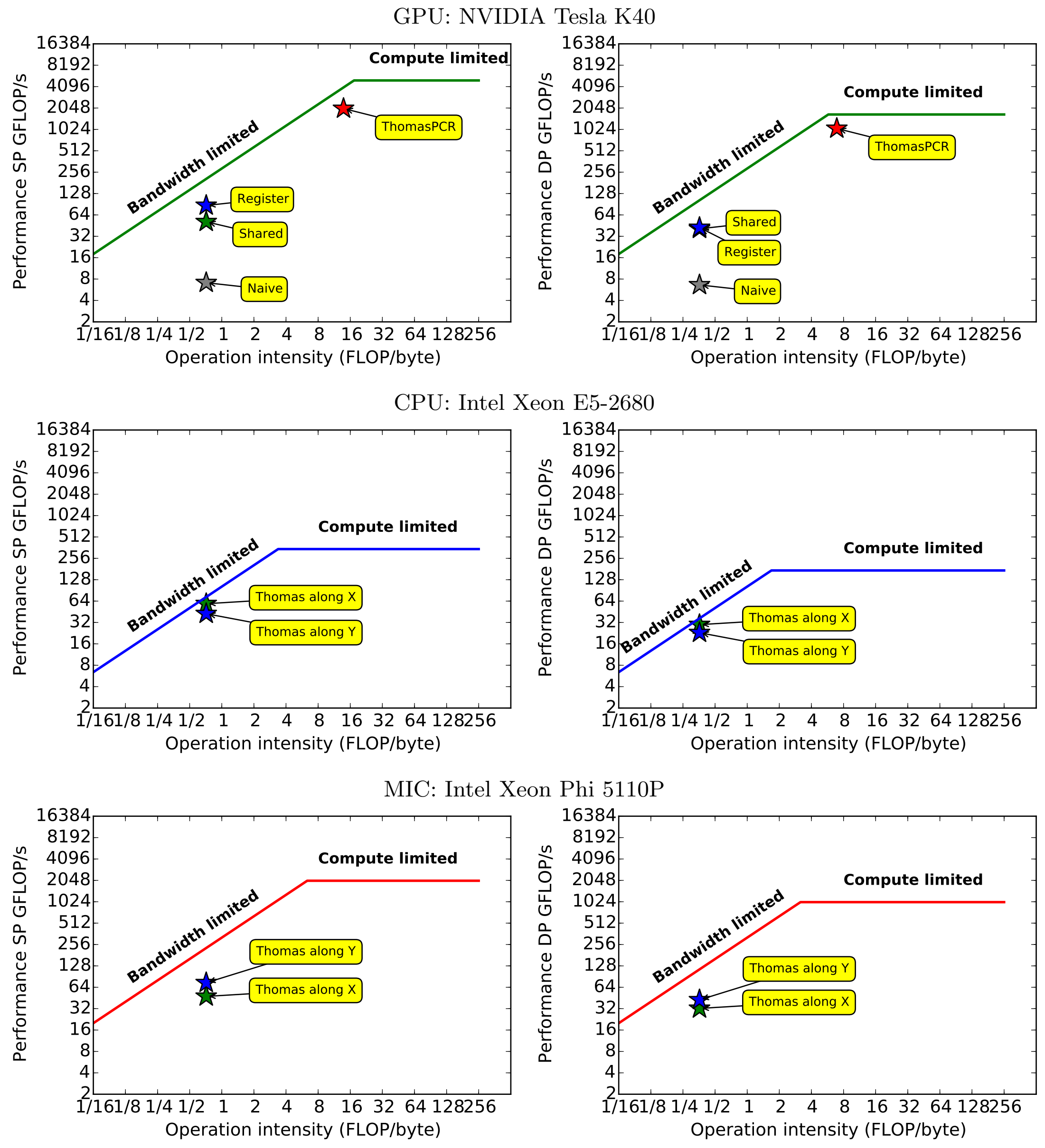


FIGURE 1.4: Roofline model applied to the implemented scalar tridiagonal solvers on GPU, CPU and MIC processor architectures. The proximity of stars to the upper computational limits shows the optimality of the implementation on the architecture.

DOI:10.15774/PPKE.ITK.2016.002

1.2.2 Alternating Directions Implicit Method

The numerical approximation of multi-dimensional PDE problems on regular grids often requires the solution of multiple tridiagonal systems of equations. In engineering applications and computational finance such problems arise frequently as part of the ADI (Alternating Direction Implicit) time discretization favored by many in the community, see [10]. The ADI method requires the solution of multiple tridiagonal systems of equations in each dimension of a multi-dimensional problem, see [14, 9, 15, 16].

1.2.3 Cellular Neural Network

The CNN (Cellular Neural Network) [17] is a powerful image processing architecture whose hardware implementation is extremely fast [18, 19]. The lack of such hardware device in a development process can be substituted by using an efficient simulator implementation. A GPU based implementation of a CNN simulator using NVIDIA's Fermi architecture provides a good alternative. Different implementation approaches are considered and compared to a multi-core, multi-threaded CPU and some earlier GPU implementations. A detailed analysis of the introduced GPU implementation is presented.

1.2.4 Computational Fluid Dynamics

Achieving optimal performance on the latest multi-core and many- core architectures depends more and more on making efficient use of the hardware's vector processing capabilities. While auto-vectorizing compilers do not require the use of vector processing constructs, they are only effective on a few classes of applications with regular memory access and computational patterns. Other application classes require the use of parallel programming models, and while CUDA and OpenCL are well established for programming GPUs, it is not obvious what model to use to exploit vector units on architectures such as CPUs or the MIC. Therefore it is of growing interest to understand how well the Single Instruction Multiple Threads (SIMT) programming model performs on a an architecture primarily designed with Single Instruction Multiple Data (SIMD) ILP parallelism in head. In many applications the OpenCL SIMT model does not map efficiently to CPU vector units. In my dissertation I give solutions to achieve vectorization and avoid synchronization - where possible - using OpenCL on real-world CFD simulations

DOI:10.15774/PPKE.ITK.2016.002

and improve the block coloring in higher level parallelization using matrix bandwidth minization reordering.

1.3 Parallel Processor Architectures

A variety of parallel processor architectures appeared since the power wall became unavoidable in 2004 and older processor architectures were also redesigned to fit general computing aims. CPUs operate with fast, heavy-weight cores, with large cache, out-of-order execution and branch prediction. GPUs and MICs on the other hand use light weight, in-order, hardware threads with moderate, but programmable caching capabilities and without branch prediction. FPGAs provide full digital circuit customization capabilities with low power consumption. A detailed technical comparison or benchmarking is not topic of the thesis, but the in-depth knowledge of these architectures is a key to understanding the theses. The reader is suggested to study the cited manuals and whitepapers for more details on the specific architecture. The list of architectures tackled in the thesis is the following:

1. CPU/MIC architectures: conventional, x86 architectures – with CISC and RISC instruction sets augmented with SIMD vector instructions – like the high-end, dual socket Intel Xeon server CPU and the Intel Xeon Phi MIC coprocessors, more specifically:
   (a) Intel **Sandy-Bridge** (E5-2680) CPU architecture
   (b) Intel **Knights Corner** (5110P) MIC architecture
2. GPU architectures: generalized for general purpose scientific computing, specifically from vendor NVIDIA:
   (a) NVIDIA **Fermi** (GF114) GPU architecture
   (b) NVIDIA **Kepler** (GK110b) GPU architecture
3. FPGA architecture: fully customizable, specifically form vendor Xilinx:
   (a) Xilinx **Virtex-7** (VX690T) FPGA architecture

All processor specifications are listed in the Appendix A and Section 5.

DOI:10.15774/PPKE.ITK.2016.002

1.3.1 GPU: NVIDIA GPU architectures

The use of GPU platforms is getting more and more general in the field of HPC and in real-time systems as well. The dominancy of these architectures versus the conventional CPUs are shown on Figure 1.5 and 1.6. In the beginning only the game and CAD industry was driving the chip manufacturers to develop more sophisticated devices for their needs. Recently, as these platforms have gotten more and more general to satisfy the need of different type of users, the GPGPU (General Purpose GPU) computing has evolved. Now the scientific community is benefiting from these result as well.

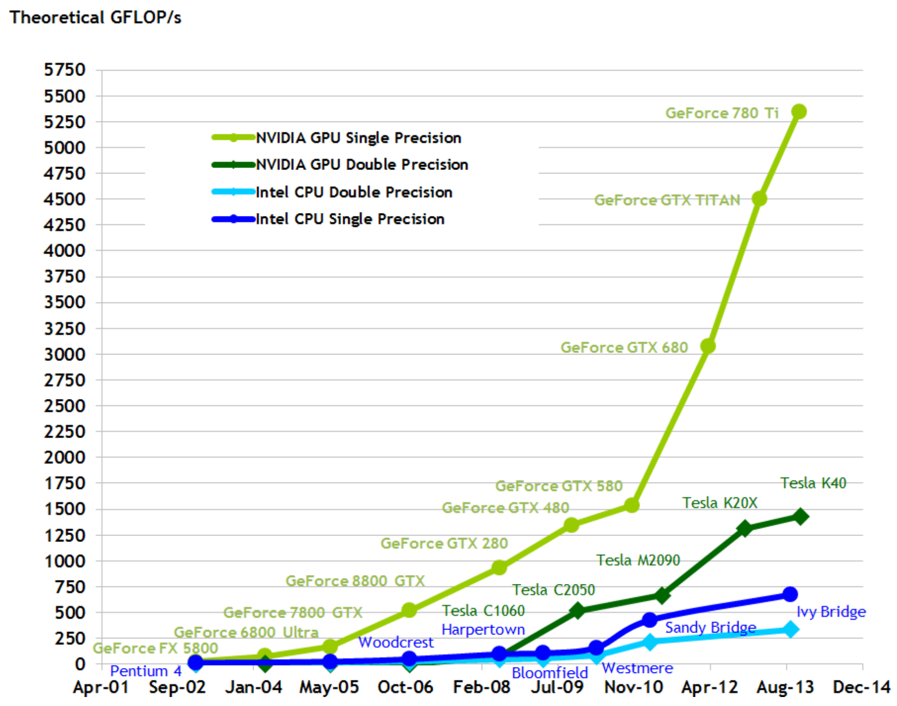


FIGURE 1.5: The increase in computation power. A comparison of NVIDIA GPUs and Intel CPUs. [20]

1.3.1.1 The NVIDIA CUDA architecture

CUDA is a combination of hardware and software technology [20] to provide programmers the capabilities to create well optimized code for a specific GPU architecture that is general within all CUDA enabled graphics cards. The CUDA architecture is designed to provide a platform for massively parallel, data parallel, compute intensive applications. The hardware side of the architecture is supported by NVIDIA GPUs, whereas the software side is supported by drivers for GPUs, C/C++ and Fortran compilers and the

DOI:10.15774/PPKE.ITK.2016.002

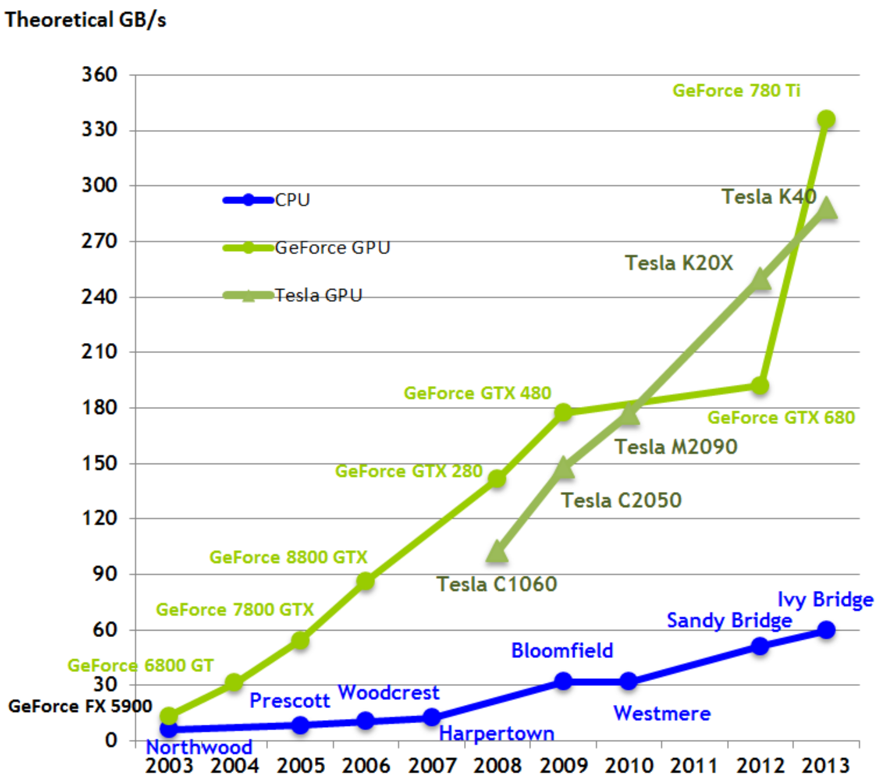


FIGURE 1.6: The increase in memory bandwidth. A comparison of NVIDIA GPUs and Intel CPUs. [20]

necessary API libraries for controlling these devices. As CUDA handles specialised keywords within the C/C++ source code, this modified programming language is referred to as the CUDA language.

The programming model, along with the thread organisation and hierarchy and memory hierarchy is described in [20]. A thread in the CUDA architecture is the smallest computing unit. Threads are grouped into thread blocks. Currently, in the Fermi and Kepler architectures maximum 1024 threads can be allocated within a thread block. Thousands of thread blocks are organized into a block grid and many block grids form a CUDA application. This thread hierarchy with thread blocks and block grids make the thread organisation simpler. Thread blocks in a grid are executed sequentially. At a time all the Streaming Multiprocessors (SM or SMX) are executing a certain number of thread blocks (a portion of a block grid). The execution and scheduling unit within an SM (or SMX) is a warp. A warp consists of 32 threads. Thus the scheduler issues an instruction for a warp and that same instruction is executed on each thread. This procedure is similar to SIMD (Single Instruction Multiple Data) instruction execution and is called SIMT (Single Instruction Multiple Threads).

DOI:10.15774/PPKE.ITK.2016.002

The memory hierarchy of the CUDA programming model can be seen in Fig. 1.7. Within a block each thread has its own register, can access the shared memory that is common to every thread within a block, has access to the global (graphics card RAM) memory and has a local memory that is allocated in the global memory but private for each thread. Usually the latter memory is used to handle register spill. Threads can also read data through the Read-Only Cache (on Kepler architecture) or Texture Cache (on Fermi architecture).

1.3.1.2 The Fermi hardware architecture

The CUDA programming architecture is used to implement algorithms above NVIDIA's hardware architectures such as the Fermi. The Fermi architecture is realized in chips of the GF100 series. Capabilities of the Fermi architecture are summarized in [21]. The base Fermi architecture is implemented using 3 billion transistors, features up to 512 CUDA cores which are organized in 16 SMs each encapsulating 32 CUDA cores. Compared to the earlier G80 architecture along with others a 10 times faster context switching circuitry, a configurable L1 cache (16 KB or 48 KB) and unified L2 cache (768 KB) is implemented. In earlier architectures the shared memory could be used as programmer handled cache memory. The L1 cache and the shared memory is allocated within the same 64 KB memory segment. The ratio of these two memories can be chosen as 16/48 KB or as 48/16 KB.

Just like the previous NVIDIA architectures, a read-only texture cache is implemented in the Fermi architecture. This cache is embedded between the L2 cache and ALU, see Fig. 1.7. Size of the texture cache is device dependent and varies between 6 KB and 8 KB per multiprocessor. This type of cache has many advantages in image processing. During a cache fetch, based on the given coordinates, the cache circuitry calculates the memory location of the data in the global memory, if a miss is detected the geometric neighbourhood of that data point is fetched. Using normalized floating point coordinates (instead of integer coordinates) the intermediate points can be interpolated. All these arithmetic operations are for free.

Beside the texture cache the Fermi architecture has a constant memory cache (prior architectures had it as well). This memory is globally accessible and its size is 64 KB. Constant data are stored in the global memory and cached through a dedicated circuitry.

DOI:10.15774/PPKE.ITK.2016.002

It is designed for use when the same data is accessed by many threads in the GPU. If many threads want to access the same data but a cache miss happens, only one global read operation is issued. This type of cache is useful if the same, small amount of data is accessed repeatedly by many threads. If one would like to store this data in the shared memory and every thread in the same block wanted to access the same memory location, that would lead to a bank conflict. This conflict is resolved by serializing the memory access, which leads to a serious performance fall.

In Fermi NVIDIA introduced the FMA (Fused Multiply and Add) [21] operation using subnormal floating point numbers. Earlier a MAD (Multiply and Add) operation is performed by executing a multiplication. The intermediate result is truncated finally the addition is performed. FMA performs the multiplication, and all the extra digits of the intermediate result are retained. Then the addition is performed on the denormalised floating number. Truncation is performed in the last step. This highly improves the precision of iterative algorithms.

1.3.1.3 The Kepler hardware architecture

The Kepler generation [22] of NVIDIA GPUs has essentially the same functionality as the previous Fermi architectures with some minor differences. Kepler GPUs have a number of SMX (Streaming Multiprocessor – eXtended) functional units. Each SMX has 192 relatively simple in-order execution cores. These operate effectively in warps of 32 threads, so they can be thought of as vector processors with a vector length of 32. To avoid poor performance due to the delays associated with accessing the graphics memory, the GPU design is heavily multi-threaded, with up to 2048 threads (or 64 warps) running on each SMX simultaneously; while one thread in a warp waits for the completion of an instruction, execution can switch to a different thread in the warp which has the data it requires for its computation. Each thread has its own share (up to 255 32 bit registers) of the SMX's registers (65536 32 bit registers) to avoid the context-switching overhead usually associated with multi-threading on a single core. Each SMX has a local L1 cache, and also a shared memory which enables different threads to exchange data. The combined size of these is 64kB, with the relative split controlled by the programmer. There is also a relatively small 1.5MB global L2 cache which is shared by all SMX units.

DOI:10.15774/PPKE.ITK.2016.002

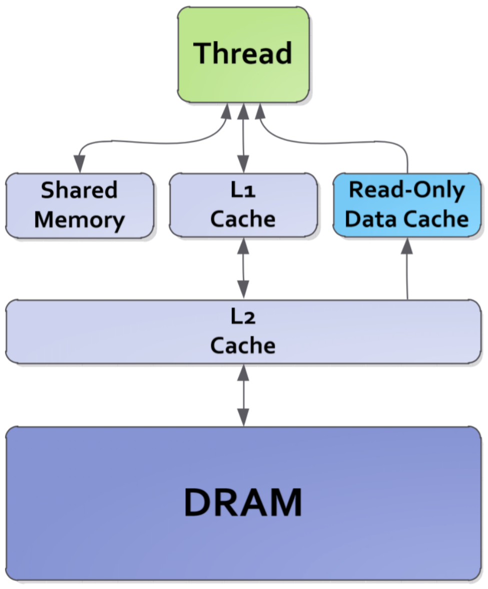


FIGURE 1.7: Kepler and Fermi memory hierarchy according to [22]. The general purpose read-only data cache is known on Fermi architectures as texture cache, as historically it was only used to read textures from global memory.

1.3.2 CPU: The Intel Sandy Bridge architecture

The Intel Sandy Bridge [23, 24] CPU cores are very complex out-of-order, shallow-pipelined, low latency execution cores, in which operations can be performed in a different order to that specified by the executable code. This on-the-fly re-ordering is performed by the hardware to avoid delays due to waiting for data from the main memory, but is done in a way that guarantees the correct results are computed. Each CPU core also has an AVX [25] vector unit. This 256-bit unit can process 8 single precision or 4 double precision operations at the same time, using vector registers for the inputs and output. For example, it can add or multiply the corresponding elements of two vectors of 8 single precision variables. To achieve the highest performance from the CPU it is important to exploit the capabilities of this vector unit, but there is a cost associated with gathering data into the vector registers to be worked on. Each cores owns an L1 (32KB) and an L2 (256KB) level cache and they also own a portion of the distributed L3 cache. These distributed L3 caches are connected with a special ring bus to form a large, usually 15-35 MB L3 or LLC (Last Level Cache). Cores can fetch data from the cache of another core.

DOI:10.15774/PPKE.ITK.2016.002

1.3.3 MIC: The Intel Knights Corner architecture

The Intel Knights Corner [26] MIC architecture on the Intel Xeon Phi card is Intel's HPC coprocessor aimed to accelerate compute intensive problems. The architecture is composed of 60 individual simple, in-order, deep-pipelined, high latency, high throughput CPU cores equipped with 512-bit wide floating point vector units which can process 16 single precision or 8 double precision operations at the same time, using vector registers for inputs and output. Low level programming is done through the KNC (or IMCI) instruction set in assembly. This architecture faces mostly the same programmability issues as the Xeon CPU, although due to the in-order execution the consequences of inefficiencies in the code can result in more server performance deficit. The MIC coprocessor uses a similar caching architecture as the Sandy Bridge Xeon CPU but has only two levels: L1 with 32KB and the distributed L2 with 512KB/core.

1.3.4 FPGA: The Xilinx Virtex-7 architecture

FPGA (Field Programmable Gate Array) devices have a long history stemming from the need to customize the functionality of the IC design on demand according to the needs of a customer, as opposed to ASIC (Application Specific Integrated Circuit) design where the functionality is fixed in the IC. An FPGA holds an array of CLB (Configurable Logic Block), block memories, customized digital circuitry for arithmetic operations (eg. DSP slices) and reconfigurable interconnect, which connects all these units. The functionality (ie. the digital circuit design) of the FPGA is defined using the HDL (Hardware Description Language) languages like VHDL or Verilog. After the synthetization of the circuit from the HDL description the FPGA is configured using an external smart source. FPGAs come in different flavours according to the application area, such as network routers, consumer devices, automotive applications etc. The Xilinx Virtex-7 FPGA family [27] is optimized for high performance, therefore it has a large number of DSP slices and a significant amount of block RAM which makes it suitable for HPC applications.

DOI:10.15774/PPKE.ITK.2016.002

Chapter 2

Alternating Directions Implicit solver

2.1 Introduction

Let us start the discussion of the ADI solvers with the numerical solution of the heat equation on a regular 3D grid of size 256^3 – the detailed solution is discussed in Section 2.2. An ADI time-marching algorithm will require the solution of a tridiagonal system of equations along each line in the first coordinate direction, then along each line in the second direction, and then along each line in the third direction. There are two important observations to make here. The first is that there are 256^2 separate tridiagonal solutions in each phase, and these can be performed independently and in parallel, ie. there is plenty of natural parallelism to exploit on many core processors. The second is that a data layout which is optimal for the solution in one particular direction, might be far from optimal for another direction. This will lead us to consider using different algorithms and implementations for different directions. Due to this natural parallelism in batch problems, the improved parallelism of CR (Cyclic Reduction) and PCR (Parallel Cyclic Reduction) are not necessarily advantageous and the increased work-load of these methods might not necessarily pay off. If we take the example of one of the most modern accelerator cards, the NVIDIA Tesla K40 GPU, we may conclude that the 12GB device memory is suitable to accommodate a N^d multidimensional problem domain with $N^d = 12\,\text{GB}/4\,\text{arrays} = 0.75 \times 10^9$ single precision grid points – assuming 4 arrays ($\mathbf{a}$,$\mathbf{b}$,$\mathbf{c}$

DOI:10.15774/PPKE.ITK.2016.002

and **d**) are necessary to store and perform in-place computation of tridiagonal system of equations as in Algorithm 1. This means that the length N along each dimension is $N = \sqrt[d]{0.75 \times 10^9}$ for dimensions $d = 2 - 4$ as shown on Table 2.1.

d	N	# parallel systems
2	27386	27386
3	908	824464
4	165	4492125

Table 2.1: Number of parallel systems increases rapidly as dimension d is increased. N is chosen to accommodate an N^d single precision problem domain on the available 12 GB of an NVIDIA Tesla K40 GPU.

The ADI (Alternating Direction Implicit) method is a finite difference scheme and has long been used to solve PDEs in higher dimension. Originally it was introduced by Peaceman and Rachford [14], but many variant have been invented throughout the years [15, 16, 9]. The method relies on factorization of the spatial operators: approximate factorization is applied to the Crank-Nicolson scheme. The error introduced by the factorization is second order accurate in both space ($O(\Delta x^2)$ and time (Δt^2)). Since the Crank-Nicolson is also second order accurate, the factorization conserves the accuracy. The benefit of using ADI instead of other (implicit) methods is the better performance. Splitting the problem this way changes the sparse matrix memory access pattern into a well defined, structured access pattern. In case of the Crank-Nicolson method a sparse matrix with high (matrix) bandwidth has to be handled.

Solving the tridiagonal systems can be performed either with the Thomas (aka. TDMA - Tridiagonal Matrix Algorithm) 3.2.1, Cyclic Reduction (CR) 3.2.2, Parallel Cyclic Reduction (PCR) 3.2.3, Recursive Doubling (RD) [28] etc. In engineering and financial applications the problem size is usually confined in a N^d hypercubic domain, where N is typically in the order of hundreds.

2.2 Solving the 3D linear diffusion PDE with ADI

The exact formulation of the ADI method used in the dissertation is discussed through the solution of a three dimensional linear diffusion PDE, ie. the heat diffusion equation

DOI:10.15774/PPKE.ITK.2016.002

with $\alpha = 1$ thermal diffusivity, see Eq.(2.1).

$$\frac{\partial u}{\partial t} = \nabla^2 u \tag{2.1}$$

or

$$\frac{\partial u}{\partial t} = \left(\frac{\partial^2}{\partial_x^2} + \frac{\partial^2}{\partial_y^2} + \frac{\partial^2}{\partial_z^2} \right) u \tag{2.2}$$

where $u = u(x, y, z, t)$ and u is a scalar function $u : \mathbb{R}^4 \mapsto \mathbb{R}$, $x, y, z \in \mathbb{R}$ are the free spatial domain variables and $t \in \mathbb{R}$ free time domain variable.

In brief: splitting the spatial, Laplace operator is feasible with approximate factorization. The result is a chain of three systems of tridiagonal equations along the three spatial dimensions X,Y and Z. The partial solution of the PDE along the dimension X is used to compute the next partial solution of the PDE along the Y dimension and then the solution of Y is used to solve the partial solution in Z dimension. The formulation of the solution presented here computes the contribution Δu which in the last step is added to the dependent varaible u.

The formulation starts with the Crank-Nicolson discretization, where a backward and forward Euler discretization is combined with the trapezoidal formula with equal 1/2 weights, see Eq.(2.3). Due to the backward Euler discretization ADI is an implicit method central difference spatial discretizations (Crank-Nicolson discretization):

$$\frac{u_{ijk}^{n+1} - u_{ijk}^{n}}{\Delta t} = \frac{1}{2} \left(\frac{1}{h} \left(\delta_x^2 + \delta_y^2 + \delta_z^2 \right) u_{ijk}^{n+1} + \frac{1}{h} \left(\delta_x^2 + \delta_y^2 + \delta_z^2 \right) u_{ijk}^{n} \right) \tag{2.3}$$

here $(x, y, z) \in \Omega = \{(i * \Delta x, j * \Delta y, k * \Delta z) \,|\, i, j, k = 0, .., N-1; h = \Delta x = \Delta y = \Delta z\}$, $t = n * \Delta t$ and $\delta_x^2 u_{ijk}^n = u_{i-1jk}^n - 2u_{ijk}^n + u_{i+1jk}^n$ is the central difference operator.

From $u_{ijk}^{n+1} = u_{ijk}^n + \Delta u_{ijk}$ and Eq.(2.3) it follows that

$$\Delta u_{ijk} = \lambda \left(\delta_x^2 + \delta_y^2 + \delta_z^2 \right) \left(2u_{ijk}^n + \Delta u_{ijk} \right) \tag{2.4}$$

where $\lambda = \frac{\Delta t}{h}$.

$$\Delta u_{ijk} - \lambda \left(\delta_x^2 + \delta_y^2 + \delta_z^2 \right) \Delta u_{ijk} = 2\lambda \left(\delta_x^2 + \delta_y^2 + \delta_z^2 \right) u_{ijk}^n \tag{2.5}$$

DOI:10.15774/PPKE.ITK.2016.002

$$\left(1-\lambda\delta_x^2-\lambda\delta_y^2-\lambda\delta_z^2\right)\Delta u_{ijk}=2\lambda\left(\delta_x^2+\delta_y^2+\delta_z^2\right)u_{ijk}^n \tag{2.6}$$

Eq.(2.6) is transformed into Eq.(2.7) by approximate factorization. One may verify the second order accuracy of Eq.(2.7) by performing the multiplication on the LHS.

$$\left(1-\lambda\delta_x^2\right)\left(1-\lambda\delta_y^2\right)\left(1-\lambda\delta_z^2\right)\Delta u_{ijk}=2\lambda\left(\delta_x^2+\delta_y^2+\delta_z^2\right)u_{ijk}^n \tag{2.7}$$

The solution of Eq.(2.7) above boils down to performing a preprocessing stencil operation Eq.(2.8), solving three tridiagonal system of equations Eqs. (2.9),(2.10) and (2.11) and finally adding the contribution Δu to the dependent variable in Eq.(2.12).

$$\begin{aligned}
u^{(0)} &= 2\lambda\left(\delta_x^2+\delta_y^2+\delta_z^2\right)u_{ijk}^n && (2.8)\\
\left(1-\lambda\delta_x^2\right)u^{(1)} &= u^{(0)} && (2.9)\\
\left(1-\lambda\delta_y^2\right)u^{(2)} &= u^{(1)} && (2.10)\\
\left(1-\lambda\delta_z^2\right)\Delta u_{ijk} &= u^{(2)} && (2.11)\\
u_{ijk}^{n+1} &= u_{ijk}^n+\Delta u_{ijk} && (2.12)
\end{aligned}$$

where $u^{(0)}$,$u^{(1)}$,$u^{(2)}$ denote the intermediate values and not time instance. Eq.(2.13) represents the solution with tridiagonal system of equations.

$$\begin{aligned}
u_{ijk}^{(0)} =& 2\lambda\Big(u_{i-1jk}^n+u_{i+1jk}^n+\\
&u_{ij-1k}^n+u_{ij+1k}^n+\\
&u_{ijk-1}^n+u_{ijk+1}^n\\
&-6u_{ijk}^n\Big)\\
a_{ijk}^x u_{i-1jk}^{(1)}+b_{ijk}^x u_{ijk}^{(1)}+c_{ijk}^x u_{i+1jk}^{(1)} =& u_{ijk}^{(0)}\\
a_{ijk}^y u_{ij-1k}^{(2)}+b_{ijk}^y u_{ijk}^{(2)}+c_{ijk}^y u_{ij+1k}^{(2)} =& u_{ijk}^{(1)}\\
a_{ijk}^z \Delta u_{ijk-1}+b_{ijk}^z \Delta u_{ijk}+c_{ijk}^z \Delta u_{ijk+1} =& u_{ijk}^{(2)}\\
u_{ijk}^{n+1} =& u_{ijk}^n+\Delta u_{ijk}
\end{aligned} \tag{2.13}$$

DOI:10.15774/PPKE.ITK.2016.002

Note that the super and subscript indices of coefficients a, b and c mean that the coefficients are stored in a cubic datastructure.

DOI:10.15774/PPKE.ITK.2016.002

Chapter 3

Scalar Tridiagonal Solvers on Multi and Many Core Architectures

3.1 Introduction

Scalar tridiagonal solvers are used in many applications in science, engineering and finance, usually as part of more complex solvers like implicit line smoothers or ADI solvers etc. Application cases are detailed in Section 1.2.1. Before discussing the specific implementations on different architectures, we review a number of different algorithms for solving tridiagonal systems, and discuss their properties in terms of the number of floating point operations and the amount of memory traffic generated. Research has been conducted by [29, 30] to solve tridiagonal system of equations on GPUs. The Recursive Doubling algorithm has been developed by [31]. Earlier work by [32, 33, 34, 35, 36] has decomposed a larger tridiagonal system into smaller ones that can be solved independently, in parallel. The algorithms described in the following subsections are the key building blocks of a high performance implementation of a tridiagonal solver. We introduce the tridiagonal system of equations as shown on Eq. (3.1) or in its matrix form shown on Eq. (3.2) and Eq. (3.3).

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d_i, \quad i = 0, 1, \ldots, N-1 \tag{3.1}$$

DOI:10.15774/PPKE.ITK.2016.002

$$\begin{pmatrix} b_0 & c_0 & 0 & 0 & \cdots & 0 \\ a_1 & b_1 & c_1 & 0 & \cdots & 0 \\ 0 & a_2 & b_2 & c_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{N-1} & b_{N-1} \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N-1} \end{pmatrix} \tag{3.2}$$

$$\mathbf{Au} = \mathbf{d} \tag{3.3}$$

where $a_0 = c_{N-1} = 0$, $\mathbf{A}$ is called the tridiagonal matrix, $\mathbf{u}$ is the unknown and $\mathbf{d}$ is the RHS (Right Hand Side).

3.2 Tridiagonal algorithms

In this section the standard algorithms like Thomas, PCR (Parallel Cyclic Reduction) and CR (Cyclic Reduction) are presented along with a new hybrid Thomas-PCR algorithm which is the combination of the Thomas and the PCR algorithms.

3.2.1 Thomas algorithm

The Thomas algorithm [37] is a sequential algorithm which is described in [38]. It is a specialized version of Gaussian elimination to the case in which the matrix is tridiagonal. The algorithm has a forward pass in which the lower diagonal elements a_i are eliminated by adding a multiple of the row above. This is then followed by a reverse pass to compute the final solution using the modified c_i values. In both passes the d_i is also modified according to the row operations.

The derivation of the Thomas algorithm is the following. Suppose Gaussian elimination of a_i-s is already done and the equations are normalized with b_i, then we have Eq.(3.4).

$$u_i + c_i^* u_{i+1} = d_i^* \qquad i = 0, ..., N-1 \tag{3.4}$$

where c_i^* and d_i^* are assumed to be unknowns for now. Substituting Eqs. (3.5) into the original equation Eq. (3.1) we get Eq. (3.6).

DOI:10.15774/PPKE.ITK.2016.002

$$\begin{aligned} u_i =& -c_i^* u_{i+1} + d_i^* \\ u_{i-1} =& -c_{i-1}^* u_i + d_{i-1}^* \\ i =& 0, ..., N-1 \end{aligned} \tag{3.5}$$

$$\begin{aligned} a_i(-c_{i-1}^* u_i + d_{i-1}^*) + b_i u_i + c_i u_{i+1} =& d_i \\ (b_i - a_i c_{i-1}^*) u_i + c_i u_{i+1} =& d_i - a_i d_{i-1}^* \\ u_i + \frac{c_i}{b_i - a_i c_{i-1}^*} u_{i+1} =& \frac{d_i - a_i d_{i-1}^*}{b_i - a_i c_{i-1}^*} u \end{aligned} \tag{3.6}$$

ALGORITHM 1: Thomas algorithm

```
Require: $thomas(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
 1: $d_0^* := d_0 / b_0$
 2: $c_0^* := c_0 / b_0$
 3: for $i = 1, \ldots, N-1$ do
 4:     $r := 1 \,/\, (b_i - a_i \, c_{i-1})$
 5:     $d_i^* := r \, (d_i - a_i \, d_{i-1})$
 6:     $c_i^* := r \, c_i$
 7: end for
 8: $d_{K-1} := d_{K-1}^*$
 9: for $i = N-2, \ldots, 0$ do
10:     $d_i := d_i^* - c_i^* \, d_{i+1}$
11: end for
12: return $\mathbf{d}$
```

The full Thomas algorithm with in-place solution (RHS vector is overwritten) is given in Algorithm 1. Note that this does not perform any pivoting; it is assumed the matrix is diagonally dominant, or at least sufficiently close to diagonal dominance so that the solution is well-conditioned. Floating point multiplication and addition, as well as FMA (Fused Multiply and Add) have the same instruction throughput on almost every hardware. Also, the use of FMA is considered to be a compiler optimization feature and therefore optimistic calculations based on this instruction give a good lower estimate on the number of actual instruction uses. In the following discussion the FMA instruction

DOI:10.15774/PPKE.ITK.2016.002

is used as the basis of estimating the work complexity of the algorithm. The computational cost per row is approximately three FMA operations, one reciprocal and two multiplications. If we treat the cost of the reciprocal as being equivalent to five FMAs (which is the approximate cost on a GPU for a double precision reciprocal), then the total cost is equivalent to 10 FMAs per grid point.

On the other hand, the algorithm requires at the minimum the loading of a_i, b_i, c_i, d_i, and the storing of the final answer d_i. Worse still, it may be necessary to store the c_i^*, d_i^* computed in the forward pass, and then read them back during the reverse pass. This shows that the implementation of the algorithm is likely to be memory-bandwidth limited, because there are very few operations per memory reference. Thus, when using the Thomas algorithm it will be critical to ensure the maximum possible memory bandwidth.

3.2.2 Cyclic Reduction

Buneman descibed first the the cyclic even-odd algorithm [39] (also known as cyclic reduction) for the solution of tridiagonal system of equations. A comparison on the arithmetic complexity of the cyclic reduction and other algorithms is studied in [40]. Consider the algebraic formulation of the system of tridiagonal system of equations as Eq. (3.7), which assumes that the equation is normalized by b_n and coefficients a_n and c_n are negated.

$$-a_n u_{n-1} + u_n - c_n u_{n+1} = d_n \qquad n = 0, ..., N-1 \tag{3.7}$$

CR has two phases, a forward and a backward pass. Each pass takes $O(\log(N))$ steps to perform. Alltogether the CR algorithm takes $2 \times O(\log(N))$ steps to finish. The forward pass is performed the following way: for each even n row (equation) add a_n times row $n-1$ and c_n times row $n+1$, see Eq.(3.8).

$$\begin{aligned} &-a_n a_{n-1} u_{n-2} + (1 - a_n c_{n-1} - c_n a_{n+1}) u_n - c_n c_{n+1} u_{n+2} = d_n + a_n d_{n-1} + c_n d_{n+1} \\ &n = 0, 2, 4, ... \end{aligned} \tag{3.8}$$

DOI:10.15774/PPKE.ITK.2016.002

Once the constant terms of the expression in Eq.(3.8) is normalized, we get a new tridiagonal system of equation Eq.(3.9) with new constant coefficients.

$$\begin{aligned}
-a_n^* u_{n-2} + u_n - c_n^* u_{n+2} &= d_n^* \\
a_n^* &= \frac{a_n a_{n-1}}{(1 - a_n c_{n-1} - c_n a_{n+1})} \\
c_n^* &= \frac{a_n a_{n+1}}{(1 - a_n c_{n-1} - c_n a_{n+1})} \\
n &= 2, 4, ...
\end{aligned} \tag{3.9}$$

The steps of the forward pass are repeated until the resulting system size reduces to a single expression. In the backward pass the known values are back-substituted to the equations and the results are acquired. Due to the lack of pivoting diagonal dominance is a strict criteria of the algorithm for stability.

3.2.3 Parallel Cyclic Reduction

Historically the PCR algorithm stems from the CR algorithm, but it has a higher parallel efficiency. In each of the $O(\log(N))$ steps the CR algorithm creates a single new tridiagonal system that is two times smaller than the previous one. The PCR on the other hand creates two new tridiagonal systems (both for odd and even indices of unknowns) in each of the $O(\log(N))$ steps that is two times smaller than the previous one. The main difference is that by the end of the forward sweep the PCR is done with the computation, whereas the CR still has to use the computed values to solve the system in the backward sweep. The overall advantage of the PCR algorithm is that all the processors computing the solution are active in each of the $O(\log(N))$ steps as opposed to the CR algorithm.

PCR [41] is an inherently parallel algorithm which is ideal when using multiple execution threads to solve each tridiagonal system.

DOI:10.15774/PPKE.ITK.2016.002

ALGORITHM 2: PCR algorithm

```
Require: $pcr(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
 1: for $p = 1, \dots, P$ do
 2:     $s := 2^{p-1}$
 3:     for $i = 0, \dots, N-1$ do
 4:         $r \;:= 1 \,/\, (1 - a_i^{(p-1)}\, c_{i-s}^{(p-1)} - c_i^{(p-1)} a_{i+s}^{(p-1)})$
 5:         $a_i^{(p)} := -r\, a_i^{(p-1)}\, a_{i-s}^{(p-1)}$
 6:         $c_i^{(p)} := -r\, c_i^{(p-1)}\, c_{i+s}^{(p-1)}$
 7:         $d_i^{(p)} := r\, (d_i^{(p-1)} - a_i^{(p-1)}\, d_{i-s}^{(p-1)} - c_i^{(p-1)}\, d_{i+s}^{(p-1)})$
 8:     end for
 9: end for
10: return $\mathbf{d}^{(P)}$
```

If we start with the same tridiagonal system of equations, but normalized so that $b_i = 1$,

$$a_i u_{i-1} + u_i + c_i u_{i+1} = d_i, \quad i = 0, 1, \dots, N-1,$$

with $a_0 = c_{N-1} = 0$, then subtracting the appropriate multiples of rows $i \pm 1$, and re-normalising, gives

$$a'_i u_{i-2} + u_i + c'_i u_{i+2} = d'_i, \quad i = 0, 1, \dots, N-1,$$

with $a'_0 = a'_1 = c'_{N-2} = c'_{N-1} = 0$. Repeating this by subtracting the appropriate multiples of rows $i \pm 2$ gives

$$a''_i u_{i-4} + u_i + c''_i u_{i+4} = d''_i, \quad i = 1, 2, \dots, N,$$

with $a''_j = c''_{N-1-j} = 0$ for $0 \le j < 4$.

After P such steps, where P is the smallest integer such that $2^P \ge N$, then all of the modified a and c coefficients are zero (since otherwise the value of u_i would be coupled to the non-existent values of $u_{i \pm 2^P}$) and so we have the value of u_i.

The PCR algorithm is given in Algorithm 2. Note that any reference to a value with index j outside the range $0 \le j < N$ can be taken to be zero; it is multiplied by a zero coefficient so as long as it is not an IEEE exception value (such as a NaN) then any valid floating point value can be used. If the computations within each step are performed simultaneously for all i, then it is possible to reuse the storage so that $a^{(p+1)}$

DOI:10.15774/PPKE.ITK.2016.002

and $c^{(p+1)}$ are held in the same storage (e.g. the same registers) as $a^{(p)}$ and $c^{(p)}$. The computational cost per row is approximately equivalent to 14 FMAs in each step, so the total cost per grid point is $14 \log_2 N$. This is clearly much greater than the cost of the Thomas algorithm with 10 FMA, but the data transfer to/from the main memory may be lower if there is no need to store and read back the various intermediate values of the a and c coefficients.

3.2.4 Hybrid Thomas-PCR Algorithm

As noted earlier CR is a slightly different algorithm from PCR in which the p^{th} pass of the Algorithm 2 is performed only for those i for which $i \bmod 2^p = 0$. This gives the forward pass of the PCR algorithm; there is then a corresponding reverse pass which performs the back solve. This involves fewer floating point operations overall, but there is less parallelism on average and it requires twice as many steps so it is slower than PCR when there are many active threads. Hence, the PCR algorithm is used in the hybrid method.

The hybrid algorithm – developed specifically for GPU architectures – is a combination of a modified Thomas and PCR algorithms. Some hybrid algorithms have been developed over the years with different application aims, like [32] who decomposed tridiagonal systems in an upper tridiagonal form to be solved multiple RHS. The slides of [42] show that the PCR algorithm can be used to divide a larger system into smaller ones which can be solved using the Thomas algorithm. It is shown that a larger system can be divided into smaller ones using a modified Thomas algorithm and the warp level PCR can be used to solve the remaining system. The complete computation with the presented algorithms can be done in registers using the *__shfl()* instructions introduced in the NVIDIA Kepler GPU architecture. In the case of the Thomas solver intermediate values c_i^* and d_i^* had to be stored and loaded in main memory. With the hybrid algorithm the input data is read once and output is written once without the need to move intermediate values in global memory. Therefore this algorithm allows an optimal implementation.

DOI:10.15774/PPKE.ITK.2016.002

ALGORITHM 3: First phase of hybrid algorithm

```
Require: $hybridFW(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
 1: $d_0 := d_0 / b_0$
 2: $a_0 := a_0 / b_0$
 3: $c_0 := c_0 / b_0$
 4: for $i = 1, \ldots, M-2$ do
 5:     $r \;:= 1 \,/\, (b_i - a_i\, c_{i-1})$
 6:     $d_i := r\,(d_i - a_i\, d_{i-1})$
 7:     $a_i := -r\, a_i\, a_{i-1}$
 8:     $c_i := r\, c_i$
 9: end for
10: for $i = M-3, \ldots, 1$ do
11:     $d_i := d_i - c_i\, u_{i+1}$
12:     $a_i := a_i - c_i\, a_{i+1}$
13:     $c_i := -c_i\, c_{i+1}$
14: end for
```

ALGORITHM 4: Middle phase: modified PCR algorithm.

```
Require: $pcr2(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
 1: for $i = 1 : 2 : N-1$ do
 2:     $r = 1/(1 - a_i^{(0)}\, c_{i-1}^{(0)} - c_i^{(0)}\, a_{i+1}^{(0)})$
 3:     $d_i^{(1)} = r\,(d_i^{(0)} - a_i^{(0)}\, d_{i-1}^{(0)} - c_i^{(0)}\, d_{i+1}^{(0)})$
 4:     $a_i^{(1)} = -r\, a_i^{(0)}\, a_{i-1}^{(0)}$
 5:     $c_i^{(1)} = -r\, c_i^{(0)}\, c_{i+1}^{(0)}$
 6: end for
 7: for $p = 1, \ldots, P$ do
 8:     $s := 2^p$
 9:     for $i = 1 : 2 : N-1$ do
10:         $r = 1/(1 - a_i^{(p-1)}\, c_{i-s}^{(p-1)} - c_i^{(p-1)}\, a_{i+s}^{(p-1)})$
11:         $d_i^{(p)} = r\,(d_i^{(p-1)} - a_i^{(p-1)}\, d_{i-s}^{(p-1)} - c_i^{(p-1)}\, d_{i+s}^{(p-1)})$
12:         $a_i^{(p)} = -r\, a_i^{(p-1)}\, a_{i-s}^{(p-1)}$
13:         $c_i^{(p)} = -r\, c_i^{(p-1)}\, c_{i+s}^{(p-1)}$
14:     end for
15: end for
16: for $i = 1 : 1 : N-1$ do
17:     $d_i^{(P)} = d_i^{(P-1)} - a_i^{(P-1)}\, d_i^{(P-1)} - c_i^{(P-1)}\, d_{i+1}^{(P-1)}$
18: end for
19: return $\mathbf{d}^{(P)}$
```

DOI:10.15774/PPKE.ITK.2016.002

ALGORITHM 5: Last phase: solve for the remaining unknowns.

```
Require: $hybrid1b(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
1: for $i = 1, \ldots, M-1$ do
2:     $d_i := d_i - a_i\, r\, d_0 - c_i\, d_{M-1}$
3: end for
```

Suppose the tridiagonal system is broken into a number of sections of size M (show in Figure 3.1), each of which will be handled by a separate thread. Within each of these pieces, using local indices ranging from 0 to $M-1$, a slight modification to the Thomas algorithm (shown in Algorithm 3) operating on rows 1 to $M-2$ enables one to obtain an equation of the form

$$a_i\, u_0 + u_i + c_i\, u_{M-1} = d_i, \quad i = 1, 2, \ldots, M-2 \tag{3.10}$$

expressing the central values as a linear function of the two end values, u_0 and u_{M-1}. The forward and backward phases of the modified Thomas algorithm are shown on Figure 3.2 and 3.3. Using equation 3.10 for $i = 1, M-2$ to eliminate the u_1 and u_{M-2} entries in the equations for rows 0 and $M-1$, leads to a reduced tridiagonal system (shown in Figure 3.3) of equations involving the first and last variables within each section. This reduced tridiagonal system can be solved using PCR using Algorithm 4, and then Algorithm 5 gives the interior values.

In the last phase the solution of the interior unknowns using the outer $i = 0$ and $i = M-1$ values needs to be completed with Algorithm 5.

When there are 32 CUDA threads and 8 unknowns per thread the cost is approximately 14 FMAs per point, about 40% more than the cost of the Thomas algorithm. The data transfer cost depends on whether these intermediate coefficients a_i, c_i, d_i need to be stored in the main memory. If they do, then it is again more costly than the Thomas algorithm, but if not then it is less costly. We need to add this to the cost of the PCR solution. The relative size of this depends on the ratio $(\log_2 N)/M$. We will discuss this in more detail later.

The hybrid Thomas-PCR solver is validated against the Thomas solver by computing the MSE (Mean Square Error) between the two solutions. The results show MSE error in the order of 10^{-9} in single precision and 10^{-18} in double precision.

DOI:10.15774/PPKE.ITK.2016.002

$$\left(\begin{array}{cccc|cccc|cccc}
b_0 & c_0 & & & & & & & & & & \\
a_1 & b_1 & c_1 & & & & & & & & & \\
 & a_2 & b_2 & c_2 & & & & & & & & \\
 & & a_3 & b_3 & c_3 & & & & & & & \\
\hline
 & & & a_4 & b_4 & c_4 & & & & & & \\
 & & & & a_5 & b_5 & c_5 & & & & & \\
 & & & & & a_6 & b_6 & c_6 & & & & \\
 & & & & & & a_7 & b_7 & c_7 & & & \\
\hline
 & & & & & & & a_8 & b_8 & c_8 & & \\
 & & & & & & & & a_9 & b_9 & c_9 & \\
 & & & & & & & & & a_{10} & b_{10} & c_{10} \\
 & & & & & & & & & & a_{11} & b_{11}
\end{array}\right)
\left(\begin{array}{c}
u_0 \\ u_1 \\ u_2 \\ u_3 \\ \hline u_4 \\ u_5 \\ u_6 \\ u_7 \\ \hline u_8 \\ u_9 \\ u_{10} \\ u_{11}
\end{array}\right)
=
\left(\begin{array}{c}
d_0 \\ d_1 \\ d_2 \\ d_3 \\ \hline d_4 \\ d_5 \\ d_6 \\ d_7 \\ \hline d_8 \\ d_9 \\ d_{10} \\ d_{11}
\end{array}\right)$$

FIGURE 3.1: Starting state of the equation system. 0 values outside the bar are part of the equation system and they are needed for the algorithm and implementation.

$$\left(\begin{array}{cccc|cccc|cccc}
1 & c_0 & & & & & & & & & & \\
a_1^* & 1 & c_1 & & & & & & & & & \\
a_2^* & & 1 & c_2 & & & & & & & & \\
a_3^* & & & 1 & c_3 & & & & & & & \\
\hline
 & & & a_4^* & 1 & c_4 & & & & & & \\
 & & & & a_5^* & 1 & c_5 & & & & & \\
 & & & & a_6^* & & 1 & c_6 & & & & \\
 & & & & a_7^* & & & 1 & c_7 & & & \\
\hline
 & & & & & & & a_8^* & 1 & c_8 & & \\
 & & & & & & & & a_9^* & 1 & c_9 & \\
 & & & & & & & & a_{10}^* & & 1 & c_{10} \\
 & & & & & & & & a_{11}^* & & & 1
\end{array}\right)
\left(\begin{array}{c}
u_0 \\ u_1 \\ u_2 \\ u_3 \\ \hline u_4 \\ u_5 \\ u_6 \\ u_7 \\ \hline u_8 \\ u_9 \\ u_{10} \\ u_{11}
\end{array}\right)
=
\left(\begin{array}{c}
d_0^* \\ d_1^* \\ d_2^* \\ d_3^* \\ \hline d_4^* \\ d_5^* \\ d_6^* \\ d_7^* \\ \hline d_8^* \\ d_9^* \\ d_{10}^* \\ d_{11}^*
\end{array}\right)$$

FIGURE 3.2: State of the equation system after the forward sweep of the modified Thomas algorithm.

$$\left(\begin{array}{cccc|cccc|cccc}
\mathbf{1} & & & \mathbf{c_0^*} & & & & & & & & \\
a_1^* & 1 & & c_1^* & & & & & & & & \\
a_2^* & & 1 & c_2^* & & & & & & & & \\
\mathbf{a_3^*} & & & \mathbf{1} & \mathbf{c_3^*} & & & & & & & \\
\hline
 & & & \mathbf{a_4^*} & \mathbf{1} & & & \mathbf{c_4^*} & & & & \\
 & & & & a_5^* & 1 & & c_5^* & & & & \\
 & & & & a_6^* & & 1 & c_6^* & & & & \\
 & & & & \mathbf{a_7^*} & & & \mathbf{1} & \mathbf{c_7^*} & & & \\
\hline
 & & & & & & & \mathbf{a_8^*} & \mathbf{1} & & & \mathbf{c_8^*} \\
 & & & & & & & & a_9^* & 1 & & c_9^* \\
 & & & & & & & & a_{10}^* & & 1 & c_{10}^* \\
 & & & & & & & & \mathbf{a_{11}^*} & & & \mathbf{1}
\end{array}\right)
\left(\begin{array}{c}
\mathbf{u_0} \\ u_1 \\ u_2 \\ \mathbf{u_3} \\ \hline \mathbf{u_4} \\ u_5 \\ u_6 \\ \mathbf{u_7} \\ \hline \mathbf{u_8} \\ u_9 \\ u_{10} \\ \mathbf{u_{11}}
\end{array}\right)
=
\left(\begin{array}{c}
\mathbf{d_0^*} \\ d_1^* \\ d_2^* \\ \mathbf{d_3^*} \\ \hline \mathbf{d_4^*} \\ d_5^* \\ d_6^* \\ \mathbf{d_7^*} \\ \hline \mathbf{d_8^*} \\ d_9^* \\ d_{10}^* \\ \mathbf{d_{11}^*}
\end{array}\right)$$

FIGURE 3.3: State of the equation system after the backward sweep of the modified Thomas algorithm. Bold variables show the elements of the reduced system which is to be solved with the PCR algorithm.

DOI:10.15774/PPKE.ITK.2016.002

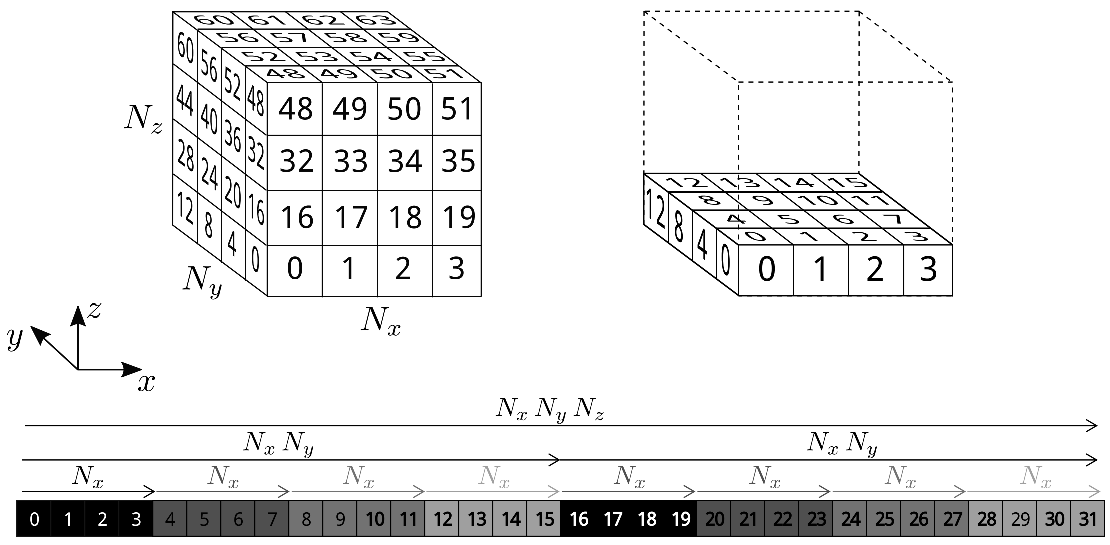


FIGURE 3.4: Data layout of 3D cube data-structure.

3.3 Data layout in memory

Knowledge of the data layout and the access patterns of the executed code with various algorithms is essential to understand the achieved performance. Taking the example of a 3D application shown on Figure 3.4, each of the coefficients arrays is a 3D indexed array which is mathematically of the form $u_{i,j,k}$. Ie. each arrays **a**, **b**, **c**, **d** and **u** are stored in a separate cubic data array.

However this is stored in memory as a linear array and so we require a mapping from the index set (i, j, k) to a location in memory. If the array has dimensions (N_x, N_y, N_z), we choose to use the mapping:

$$\text{index} = i + j\,N_x + k\,N_x\,N_y.$$

This means that if we perform a Thomas algorithm solution in the first coordinate direction, then consecutive elements of the same tridiagonal system are contiguous in memory. On the other hand, in the second coordinate direction they are accessed with a stride of N_x, and in the third direction the stride is $N_x\,N_y$. This data layout extends naturally to applications with higher number of dimensions.

To understand the consequences of this layout and access pattern, it is also necessary to understand the operation of the cache memory hierarchy. The memory bus which transfers data from the main memory (or graphics memory) to the CPU (or GPU) does

DOI:10.15774/PPKE.ITK.2016.002

so in cache lines which have a size usually in the range of 32-128 bytes. These cache lines are held in the cache hierarchy until they are displaced to make room for a new cache line. One difference between CPUs and GPUs is that CPUs have a lot more cache memory per thread, and so cache lines are usually resident for longer.

The effectiveness of caches depends on two kinds of locality: 1) *temporal locality*, which means that a data item which has been loaded into the cache is likely to be used again in the near future; 2) *spatial locality*, which means that when a cache line is loaded to provide one particular piece of data, then it is likely that other data items in the same cache line will be used in the near future. An extreme example of spatial locality can occur in a GPU when a set of 32 threads (known as a *warp*) each load one data item. If these 32 items form a contiguous block of data consisting of one or more cache lines then there is perfect spatial locality, with the entire line of data being used. This is referred to as a *coalesced* read or write. See [20] for further details.

Aligned memory access for efficient data transfer [43], [26], [20], TLB (Translation Lookaside Buffer) [43], [44] hit rate reduction for lower cache latency are among the techniques used to reduce execution time. Avoiding cache and TLB cache thrashing [43] can be done by proper padding which is the responsibility of the user of the library.

3.4 Notable hardware characteristics for tridiagonal solvers

Processors used in the present study include a high-end, dual socket Intel Xeon server CPU, Intel Xeon Phi coprocessor and NVIDIA K40m GPU. Processor specifications are listed in the Appendix A.1. CPUs operate with fast, heavy-weight cores, with large cache, out-of-order execution and branch prediction. GPUs on the other hand use light weight, in-order, hardware threads with moderate, but programmable caching capabilities and without branch prediction.

The CPU cores are very complex out-of-order, shallow-pipelined, low latency execution cores, in which operations can be performed in a different order to that specified by the executable code. This on-the-fly re-ordering is performed by the hardware to avoid delays due to waiting for data from the main memory, but is done in a way that guarantees the correct results are computed. Each CPU core also has an AVX vector unit. This 256-bit unit can process 8 single precision or 4 double precision operations at the same time,

DOI:10.15774/PPKE.ITK.2016.002

using vector registers for the inputs and output. For example, it can add or multiply the corresponding elements of two vectors of 8 single precision variables. To achieve the highest performance from the CPU it is important to exploit the capabilities of this vector unit, but there is a cost associated with gathering data into the vector registers to be worked on. Each cores owns an L1 (32KB) and an L2 (256KB) level cache and they also own a portion of the distributed L3 cache. These distributed L3 caches are connected with a special ring bus to form a large, usually 15-35 MB L3 or LLC (Last Level Cache). Cores can fetch data from the cache of another core.

The Xeon Phi or MIC (Many Integrated Core) is Intel's HPC coprocessor aimed to accelerate compute intensive problems. The architecture is composed of 60 individual simple, in-order, deep-pipelined, high latency, high throughput CPU cores equipped with 512-bit wide floating point vector units which can process 16 single precision or 8 double precision operations at the same time, using vector registers for inputs and output. This architecture faces mostly the same programmability issues as the Xeon CPU, although due to the in-order execution the consequences of inefficiencies in the code can result in more server performance deficit. The MIC coprocessor uses a similar caching architecture as the Xeon CPU but has only two levels: L1 with 32KB and the distributed L2 with 512KB/core. The IMCI ISA used on the MIC is equipped with FMA instruction which is beneficial for tridiagonal solvers as well as many other applications.

The Kepler generation of NVIDIA GPUs have a number of SMX (Streaming Multiprocessor – eXtended) functional units. Each SMX has 192 relatively simple in-order execution cores. These operate effectively in warps of 32 threads, so they can be thought of as vector processors with a vector length of 32. To avoid poor performance due to the delays associated with accessing the graphics memory, the GPU design is heavily multi-threaded, with up to 2048 threads (or 64 warps) running on each SMX simultaneously; while one thread in a warp waits for data, execution can switch to a different thread in the warp which has the data it requires for its computation. Each thread has its own share (up to 255 32 bit registers) of the SMX's registers (65536 32 bit registers) to avoid the context-switching overhead usually associated with multi-threading on a single core. Each SMX has a local L1 cache, and also a shared memory which enables different threads to exchange data. The combined size of these is 64kB, with the relative split controlled by the programmer. There is also a relatively small 1.5MB global L2 cache which is shared by all SMX units.

DOI:10.15774/PPKE.ITK.2016.002

3.5 CPU and MIC solution

In the present section the CPU and MIC solutions are described. The MIC implementation is essentially the same as the CPU implementation with some minor differences in terms of the available ISA (Instruction Set Architecture). The ZMM (512 bits wide Multi Media register) vector registers are 512 bit wide therefore they allow handling 8x8 double precision or 16x16 single precision data transposition.

To have an efficient implementation of the Thomas algorithm running on a CPU, compiler auto-vectorization and single thread, sequential execution is not sufficient. To exploit the power of a multi-core CPU the vector instruction set together with the multi-threaded execution needs to be fully utilized. Unfortunately, compilers today are not always capable of making full use of the vector units of a CPU even though the instruction set would make it feasible. Often, algorithmic data dependencies and execution flow can prevent the vectorization of a loop. Usually these conditions are related to the data alignment, layout and data dependencies in the control flow graph. Such conditions are live-out variables, inter-loop dependency, non-aligned array, non-contiguous data-access pattern, array aliasing, etc. If the alignment of arrays cannot be proven at compile time, then special vector instructions with unaligned access will be used in the executable code, and this leads to a significant performance deficit. In general, the largest difference between the vector level parallelism of the CPU and GPU are the differences how the actual parallelism is executed. In the case of the CPU and MIC code the compiler decides how a code segment can be vectorized with the available instruction set – this is called SIMD (Single Instruction Multiple Data) parallelism. The capabilities of the ISA influences the efficiency of the compiler to accomplish this task. GPUs on the other hand leave the vector level parallelization for the hardware – this is SIMT (Single Instruction Multiple Threads). It is decided in run-time by the scheduling unit whether an instruction can be executed in parallel or not. If not, than the threads working in a group (called a warp in CUDA terms) are divided into a set of idle and active threads. When the active threads complete the task, another set of idle threads are selected for execution. This sequential scheduling goes on while there are unaccomplished threads left. A more detailed study on this topic in the case of unstructured grids can be found in [45].

DOI:10.15774/PPKE.ITK.2016.002

Since the Thomas algorithm needs huge data traffic due to the varying coefficients, one would expect the implementation of the algorithm to be limited by memory bandwidth. If an implementation is bandwidth limited and can be implemented with a non-branching computation stream the GPU is supposed to be more efficient than the CPU due to the 2-4 times larger memory bandwidth. But, this is not true for CPUs with out-of-order execution, large cache and sophisticated caching capabilities. A single socket CPU in Appendix A.1 has 20 MB of LLC/L3 (Last Level Cache) per socket, 8 x 256 KB L2 and 8 x 32KB L1 Cache per socket, which is in total 40 MB L3, 4 MB L2 and 512 KB L1 cache total. This caching/execution mechanism makes the CPU efficient when solving a tridiagonal algorithm with the Thomas algorithm. The two temporary arrays ($\mathbf{c}^*$ and $\mathbf{d}^*$) can be held in this cache hierarchy and reused in the backward sweep. Therefore input data ($\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ and $\mathbf{d}$) only passes through the DDR3 memory interface once when it is loaded and result array ($\mathbf{u}$) passes once when it is stored.

This means that a system in single precision with 3 coefficient arrays (a RHS array and 2 temporary arrays) can stay in L1 cache up to system length 1365 if no hardware hyper-threading (HTT) is used or half the size, namely 682 if HTT is used. The efficiency of the solver still remains very good above this level and a gradual decrease can be observed as L2 and L3 get more significant role in caching.

Thomas solver in X dimension The data layout described in Section 3.3 doesn't allow for natural parallelism available with AVX vector instructions. The loading of an 8-wide SP (Single Precision) or 4-wide DP (Double Precision) chunk of array into a register can be accomplished with the use of *_mm256_load_p*{*s,d*} intrinsic or the *vmovap*{*s,d*} assembly instruction if the first element of the register is on an aligned address, otherwise a *_mm256_loadu_p*{*s,d*} intrinsic or the *vmovup*{*s,d*} assembly instruction needs to be used.

Unfortunately, since the data in the vector register belongs to one tridagonal system, it needs to be processed sequentially. When the algorithm acquires a value of a coefficient array, the consecutive 7 SP (or 3 DP) values will also be loaded into the L1 cache. The L1 cache line size is the same as the vector register width and therefore no saving on the efficiency of the main memory traffic can be achieved. According to Appendix A.1 this memory can be accessed with 4 clock cycle latency. On a Xeon server processor this latency is hidden by the out-of-order execution unit if data dependencies allow it. When

DOI:10.15774/PPKE.ITK.2016.002

the instructions following the memory load instruction are independent from the loaded value – eg. the instructions to calculate the index of the next value to be read – the processor can skip waiting for the data to arrive and continue on with the independent computations. When the data arrives or the execution flow reaches an instruction that depends on the loaded value, the processor enters in idle state until the data arrives. L1 caching and out-of-order execution on the CPU therefore enables the Thomas algorithm in the X dimension to run with high efficiency, although some performance is still lost due to non-vectorized code.

Vectorization and efficient data reuse can be achieved by applying local data transposition. This local transposition combines the register or cache blocking optimization with butterfly transposition. Butterfly or XOR matrix transposition used in the dissertation is similar to that detailed in Section 4. "The Butterfly Algorithm" of [46]. Historically the butterfly transposition was first implemented on the Connection Machine and today it is used on distributed memory systems implemented with the MPI (Message Passing Interface) All-to-all communication pattern. If the code is written using intrinsic operations, the compiler decides where the temporary data is stored, either in registers or cache. The advantage of this approach is that it allows for the use of vector load/store and arithmetic instructions and vector registers (or L1 cache) which leads to highly efficient execution.

The process in the case in double precision coefficient arrays is depicted on Figure 3.6. As the AVX YMM (256 bits wide Multi Media register) registers are capable holding 4 DP values, 4 tridiagonal systems can be solved in parallel. The first 4 values of the 4 systems are loaded and stored in 4 separate vector registers for each coefficient array $\mathbf{a}, \mathbf{b}$ etc. This procedure allows for perfect cache line utilization and it is depicted on Figure 3.6 for the first 4x4 array case. The same procedure is applied for the 8x8 single precision case with one additional transposition of 4x4 blocks. The process in the case of 4x4 of array $\mathbf{a}$ is the following. Load the first 4 values of the first tridiagonal system into YMM0 register, load the first 4 values of the second tridiagonal system into YMM1 register etc. Once data is in the register perform transposition according to Figure 3.6 and perform 4 steps of the Thomas algorithm. Do the same for the next 4 values of the system. Repeat this procedure until the end of the systems are reached.

DOI:10.15774/PPKE.ITK.2016.002

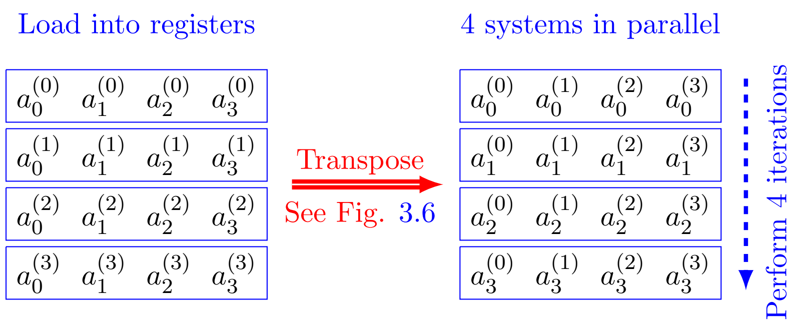


FIGURE 3.5: Vectorized Thomas solver on CPU and MIC architectures. Data is loaded into M wide vector registers in M steps. On AVX $M = 4$ for double precision and $M = 8$ for single precision. On IMCI $M = 8$ for double precision and $M = 16$ for single precision. Data is then transposed within registers as described in Figure 3.6 and after that M number of iterations of the Thomas algorithm is performed. When this is done the procedure repeats until the and of the systems is reached.

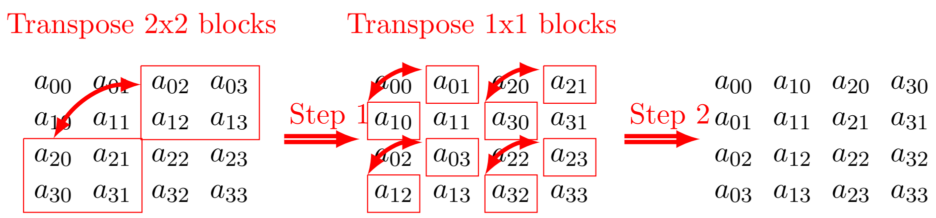


FIGURE 3.6: Local data transposition of an $M \times M$ two dimensional array of M consecutive values of M systems, where a_{ti} is the ith coefficient of tridiagonal system t. Transposition can be performed with $M \times \log_2 M$ AVX or IMCI shuffle type instructions such as swizzle, permute and align or their masked counter part. On AVX $M = 4$ for double precision and $M = 8$ for single precision. On IMCI $M = 8$ for double precision and $M = 16$ for single precision.

Thomas solver in Y and Z dimension The data layout described in Section 3.3 suggests natural parallel execution with *_mm256_load{u}_p{s,d}* loads. Unfortunately, the compiler is not able to determine how to vectorize along these dimensions, since it can not prove that the neighboring solves will access neighboring elements in the nested *for* loops of the forward and backward phases. Not even Intel's elemental function and array notation is capable of handling this case correctly.

Since solving tridiagonal problems is inherently data parallel and data reads fit the AVX load instructions, manual vectorization with AVX intrinsic functions are used to vectorize the method. Compared to a scalar, multi-threaded implementation, using the explicit vectorization gives a 2.5 times speedup in dimension Y in single precision.

The vectorized code reads 8/4 consecutive SP/DP scalar elements into YMM registers (*__m256* or *__m256d*) from the multidimensional coefficient array. The implementation is straightforward with *_mm256_load_p{s,d}* intrinsic and the *vmovap{s,d}* assembly instruction if the first element of the register is on aligned address or with

DOI:10.15774/PPKE.ITK.2016.002

_mm256_loadu_p{*s,d*} intrinsic and the *vmovup*{*s,d*} assembly instruction when data is not aligned. In case the length along the X dimension is not a multiple of 8/4 padding is used to achieve maximum performance. Arithmetic operations are carried out using *_mm256_*{*add,sub,mul,div*}*_p*{*s,d*} intrinsics.

When stepping in the Z dimension to solve a system, one may hit the NUMA (Non-Uniform Memory Access) penalty. The NUMA penalty is the consequence of threads accessing data in a memory domain which is not attached to their CPU [47, 48]. The consequences of ccNUMA (cache coherent NUMA) are two-fold. First, the LLC miss rate increases since the values of **d** are still in the cache where they have been used the last time. The coherent cache is implemented through the QPI bus which introduces an extra access latency when the cache line needs to be fetched from a different socket. LLC cache miss rate can go up to a level of 84.7% of all LLC-cache access in the Z dimension. The second consequence is the TLB (Translation Lookaside Buffer) miss penalty which occurs when elements in an array are accessed with large stride. A TLB page only covers a 4KB range of the dynamically allocated memory. Once the data acquired is outside this range, a new page frame pointer needs to be loaded into the TLB cache. The latency of doing this is the time taken to access the main memory and time taken by the Linux kernel to give the pointer and permission to access that particular page.

3.6 GPU solution

The GPU implementation of the tridiagonal solver is not as straightforward as the CPU solver with the Thomas algorithm. Regardless of the underlying algorithms (Thomas or Thomas-PCR hybrid), solving tridiagonal systems is usually considered to be memory bandwidth limited, since the ratio of the minimal amount of data that needs to be loaded ($4N$) and stored (N) and the minimal amount of FLOPs need to be carried out is $\frac{9NFLOP}{(4+1)\,N\,values}$. $9N$ FLOP is the total amount of addition, multiplication and division within the two for loops of the Thomas algorithm. This figure is called the compute ratio and implies that for every loaded value this amount of compute needs to be performed. This figure also depends on the SP/DP floating point data and processing unit throughput. Thus the required compute ratio for single and double precision is $0.45\frac{FLOP}{byte}$ and $0.225\frac{FLOP}{byte}$ respectively. An algorithm is theoretically expected to be memory bandwidth

DOI:10.15774/PPKE.ITK.2016.002

limited if the compute ratio capability of the specific hardware exceeds the compute ratio of the algorithm. For the K40 GPU these figures are $16.92\frac{FLOP}{byte}$ and $14.89\frac{FLOP}{byte}$ respectively which suggest that solving the problem with the most compute-efficient algorithm – the Thomas algorithm – is memory bound. In the forthcoming discussion the aim is to improve this bottleneck and achieve high memory utilization with suitable access patterns and reduced memory traffic.

Global memory load on the GPU poses strict constraints on the access patterns used to solve systems of equations on a multidimensional domain. The difference with regards to CPUs comes from the way SIMT thread level parallelism provides access to data in the memory. Solvers using unit-stride access pattern (along the X dimension) and the long-strided access pattern (along Y and higher dimensions) need to be distinguished. The former is explained in Section 3.6.1 while the latter is explained in Section 3.6.2.

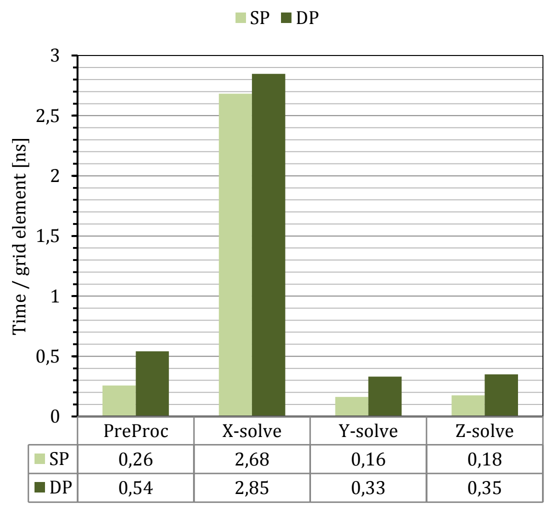
The memory load instructions are issued by threads within the same warp. In order to utilize the whole 32 (or 128) byte cache line threads need to use all the data loaded by that line. At this point we need to distinguish between the two efficient algorithms discussed in the dissertation for solving tridiagonal problems along the X dimension, because the two needs different optimization strategies. These algorithms are

1. Thomas algorithm with optimized memory access pattern, detailed in Sections 3.6.1
2. Thomas-PCR hybrid algorithm, detailed in Section 3.6.3

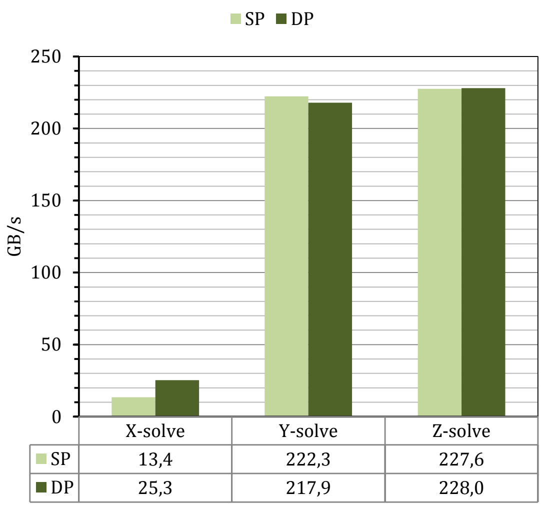
3.6.1 Thomas algorithm with optimized memory access pattern

The optimization is performed on the Thomas algorithm, which is detailed in Section 3.2.1. The problem with the presented solver is two-fold: 1) the access pattern along dimension X is different from the pattern along dimensions Y and higher ; 2) in X dimension the actual data transfer is more than the theoretically required lower limit. In this section it is shown how to overcome problem 1) and how to optimize the effect of 2) on a GPU. In order to make the discussion unambiguous, the focus of the following discussion is put on the single precision algorithm. The same argument applies for the access pattern of double precision.

DOI:10.15774/PPKE.ITK.2016.002



	PreProc	X-solve	Y-solve	Z-solve
SP	0,26	2,68	0,16	0,18
DP	0,54	2,85	0,33	0,35



	X-solve	Y-solve	Z-solve
SP	13,4	222,3	227,6
DP	25,3	217,9	228,0

FIGURE 3.7: ADI kernel execution times on K40m GPU, and corresponding bandwidth figures for X, Y and Z dimensional solves based on the Thomas algorithm. Single precision X dimensional solve is slower $\times 16.75$ than the Y dimensional solve. Bandwidth is calculated from the amount of data that supposed to loaded and stored and the elapsed time. High bandwidth in the case of Y and Z dimensional solve is reached due to local memory caching and read-only (texture) caching.

The first problem becomes obvious when one considers execution time along the three dimensions of and ADI solution step, see Figure 3.7. The X dimensional execution time is more than one order of magnitude lower than the solution along the Y and Z dimension. This is due to two factors: 1) high cache under-utilization and 2) high TLB thrashing.

The high cache line underutilization can be explained with the access pattern of coefficient a_i in Algorithm 1. A whole 32 byte cache line with 8 SP values is loaded into the L2/L1 or non-coherent cache, but temporarily only one value is used by the algorithm before the cache line is subsequently evicted – this results in poor cache line utilization. The other values within the cache line are used in a different time instance of the solution process, but at that point the cache line will already be flushed from the cache. The same applies to arrays $\mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{u}$ as well. To overcome this issue one needs to do local data transposition with cache or register blocking similar to that discussed in Section 3.5. Two solutions to perform this optimization are introduced in Section 3.6.1.1 and Section 3.6.1.2. The idea of these optimizations is to increase cache line utilization by reading consecutive memory locations with consecutive threads into a scratch memory (shared memory or registers) and then redistributing them as needed.

The issue with TLB thrashing in the X dimensional solution relates to the mapping of threads to the problem. One might map a 2 dimensional thread block on the Y-Z plane

DOI:10.15774/PPKE.ITK.2016.002

of the 3 dimensional ADI problem, and process the systems along the X dimension. In this case the start index (ind in $a[ind]$) calculation is easy, according to the following formula:

ALGORITHM 6: The common way of mapping CUDA threads on a 3D problem.

```
1: y = threadIdx.x
2: z = threadIdx.y
3: ind = z*NX*NY + y*NX
```

The problem with this mapping is that it introduces significant TLB (Translation Lookaside Buffer) thrashing. TLB is an important part of the cache hierarchy in every processor with virtual memory. A virtual memory page covers a certain section of the memory, usually 4 kB in the case of systems with relatively low memory and 2 MB "huge memory page" on systems with large memory. Since NVIDIA does not reveal the details of its cache and TLB memory subsystem, further details on the supposed TLB structure for the older GT200 architecture can be found in [44]. Note that significant architecture changes have been made since GT200, but the latency is expected to be dominated by DRAM access latency, which did not change significantly in the last few years. The problem with TLB thrashing arises as z changes and when the different coefficient arrays are accessed. This access pattern induces reads from $NX \times NY \times 4$ bytes distance and even larger when reading different arrays, which are $NX \times NY \times NZ$ apart. For sufficiently large domain this requires the load of a new TLB page table pointer for every z. The TLB cache can only handle a handful of page pointers, thus in such a situation thrashing is more significant. Depending on the level of TLB page misses the introduced latency further increases. Explaining the in-depth effects of TLB is out of the scope of the dissertation and the reader is suggested to read [44] for more details.

The solution for this problem is simple. One needs to preserve data locality when accessing the coefficients of the systems, by mapping the threads to always solve the closest neighboring system. One may map the threads with a 1 dimensional thread block according to:

ALGORITHM 7: Mapping threads to neighboring systems.

```
1: tid = threadIdx.x + blockIdx.x*blockDim.x
2: ind = tid*NX
```

DOI:10.15774/PPKE.ITK.2016.002

The inefficiency due to non-coalesced memory access has also been shown on the slides of [49] and a global transposition solution has been given. This essentially means that the data has been transposed before performing the execution of the tridiagonal solve and therefore data is read and written unnecessarily during the transposition. In the dissertation two local data transposition solutions are given, both which keep the data in registers for the time of the tridiagonal solution. These solutions work in a register-blocking manner and therefore avoid the need of the load and store of the global transposition resulting in much higher efficiency.

3.6.1.1 Thomas algorithm with local transpose in shared memory

Local data transpose can be performed in shared memory available on most GPU devices. The optimization is based on warp-synchronous execution, therefore there is no need for explicit synchronization. Although a warp size of 32 is assumed in the following, the implementation uses macros to define warp size and thus it allows for changes in the future. Threads within a warp cooperate to read data into shared memory. The data is then transposed via shared memory and stored back in registers. The threads perform 8 steps of the Thomas algorithm with the data in registers and then read the next 8 values, and so on. Meanwhile the updated coefficients and intermediate values $\mathbf{c}^*, \mathbf{d}^*$ are stored in the local memory, which automatically creates coalesced and cached load/store memory transactions. The algorithm is shown in Algorithm 8 and further detailed in Figure 3.8.

ALGORITHM 8: Thomas algorithm with shared memory transpose

```
Forward pass:
  1: Wrap a warp (32 threads) into $4 \times 8$ blocks to perform non-caching (32byte) loads
  2: Load $32 \times 8$ size tiles into shared memory: in 8 steps of $4 \times 8$ block loads
  3: Transpose data by putting values into *float a[8]* register arrays;
  4: Perform Thomas calculation with the 8 values along the $X$ dimension
  5: Repeat from step 2 until end of $X$ dimension is reached
Backward pass:
  1: Compute backward stage of Thomas in chunks of 8
  2: Transpose results and store in global memory
```

The shared memory on an NVIDIA GPU can be accessed through 32 banks of width 32 or 64 bits – latter only applicable from NVIDIA Compute Capability 3.0 and higher.

DOI:10.15774/PPKE.ITK.2016.002

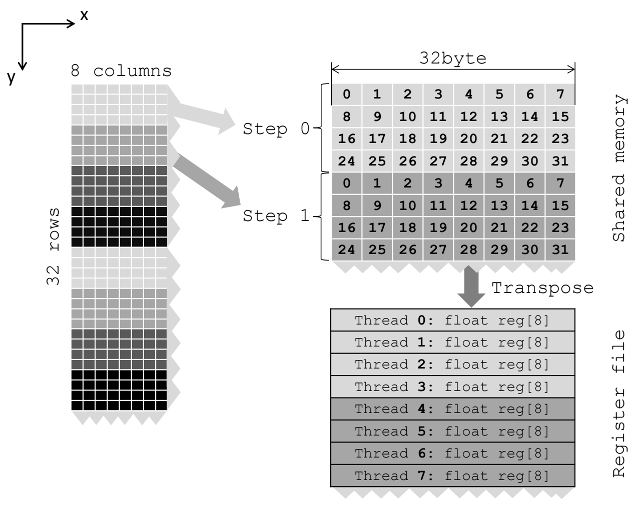


FIGURE 3.8: Local transpose with shared memory.

Since the width of the block that is stored in shared memory is 8, this leads to shared memory bank conflicts in the 3rd step of Algorithm 8. In the first iteration the first 4 threads will access banks 0,8,16,24, the next 4 threads will again access 0,8,16,24 and so on. To overcome this problem the leading dimension (stride) in shared memory block is padded to 9 instead of 8. This common trick helps to get rid of most of the bank conflicts. The first 4 threads will access banks 0,9,18,27, the next 4 threads will access banks 1,10,19,28 and so on. The amount of shared memory utilized for the 4 register arrays is $9 * 32 * 4 * 4\, bytes/warp = 4608\, bytes/warp$.

3.6.1.2 Thomas algorithm with local transpose using register shuffle

Local data transposition can also be performed in registers exclusively. This solution fits recent NVIDIA GPU architectures, starting with the Kepler architecture with Compute Capability 3.0. The optimization is again based on warp-synchronous execution with the use of *__shfl()* register shuffle intrinsic instructions. Although a warp size of 32 is assumed in the following, the implementation uses C macros definitions to set warp size and thus it allows for future architecture change. Threads within a warp cooperate to read a 32×16 SP block or 32×8 DP block of the x-y plane, ie. threads cooperate to read 16 SP or 8 DP values of 32 neighboring systems. Every 16 or 8 threads within the warp initiate a read of two cache-lines (2×32 bytes). They store the data into their register arrays of size 16 for SP and 8 for DP. These local arrays will be kept in registers for the same reasons that are discussed in Section 3.6.1.1. Once data is read, threads cooperatively distribute values with the XOR (Butterfly) transpose algorithm using the

DOI:10.15774/PPKE.ITK.2016.002

__shfl_xor() intrinsic in two steps. Once every thread has the data they perform 16 SP or 8 DP steps of the Thomas algorithm with the data in registers and then read the next 16 SP or 8 DP long array, and so on. Meanwhile the updated coefficients and intermediate values $\mathbf{c}^*, \mathbf{d}^*$ are stored in the local memory, which automatically creates coalesced, cached load/store memory transactions. The algorithm is shown in Algorithm 9 and further detailed in Figure 3.9.

ALGORITHM 9: Thomas algorithm with register shuffle transpose

```
Forward pass:
1: Wrap 32 threads into 8 × 4 blocks to perform 4× float4 vector loads
2: Load 32 × 16 size tiles into registers:
3:     4 threads read 4 consecutive float4 vectors = 64 bytes
4:     Do this 4 times for rows under each other
5: Transpose data within 4 threads:
6:     4 threads exchange data on a 4 × 4 2D array with __shfl(float4)
7:     Each element in the 2D array is a float4 vector
8: Perform Thomas calculation with the 16 values along X dimension
9: Repeat from 2 until end of X dimension is reached
Backward pass:
1: Compute backward stage of Thomas in chunks of 8
2: Transpose results and store in global memory
```

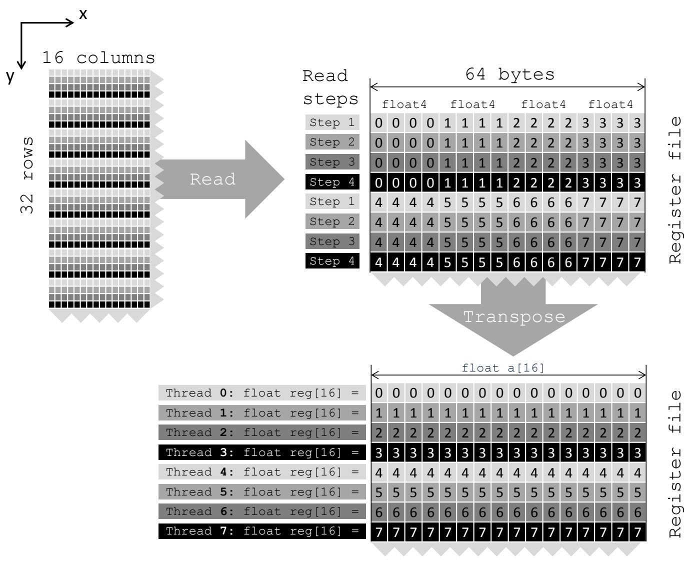

x
y
16 columns
32 rows
Read
Read steps
64 bytes
float4
Step 1
Step 2
Step 3
Step 4
Register file
Transpose
float a[16]
Thread 0: float reg[16] =
Thread 1: float reg[16] =
Thread 2: float reg[16] =
Thread 3: float reg[16] =
Thread 4: float reg[16] =
Thread 5: float reg[16] =
Thread 6: float reg[16] =
Thread 7: float reg[16] =
Register file

FIGURE 3.9: Local transpose with registers.

The use of the register shuffle instruction increases the register pressure compared to the shared memory version, but since the data is kept in registers, all the operations within

DOI:10.15774/PPKE.ITK.2016.002

a 16 SP or 8 DP solve are performed in the fastest possible way, thus the overall gain is significant.

3.6.2 Thomas algorithm in higher dimensions

The Thomas algorithm gives itself to natural parallelization and efficient access pattern in the dimensions Y and above. The memory access patterns fit the coalesced way of reading and storing data. Therefore, the Thomas algorithm is efficient in these directions and the only limiting factor is the bandwidth throughput requirement posed by reading 3 coefficients and 1 RHS arrays, writing and later reading the 2 temporary arrays and finally writing out the solution array.

3.6.3 Thomas-PCR hybrid algorithm

The Thomas-PCR hybrid algorithm introduced in Section 3.2.4 is implemented using either shared memory or register shuffle instructions. The advantage of the hybrid algorithm is that it allows for storing the entire system in the registers of a warp. Every thread in the warp is responsible for part of the system of size M, eg. if the system size is 256, then $M = 8$ and each thread stores an 8 long chunk of the 3 coefficients, the RHS and the solution vector in its registers. This storage method puts a high pressure on register allocation, and it stays efficient only up to a system size which allows the sub-arrays to be stored in registers. Every thread individually performs the first phase of the modified Thomas algorithm and then switches to PCR to cooperatively solve the second phase of the hybrid algorithm. Meanwhile the intermediate, modified coefficients and RHS values are kept in registers. Once the PCR finished, its solution can be used to solve the subsystems with the recently computed boundary values. Then follows the third phase where the back-substitution of the modified Thomas algorithm is performed.

In the case of the X dimensional solve, threads of a warp need contiguous chunks of size M of the whole array. This poses the same need for reading and transposing data in the X dimension, just like in Section 3.6.1.1 and 3.6.1.2.

Data transposition can be performed both using *shared memory* or register shuffles. To achieve good efficiency warp-synchronous implementation is used in the solver. Threads of a warp read consecutive values in a coalesced way, in multiple steps of 32 values and

DOI:10.15774/PPKE.ITK.2016.002

store data in shared memory. Then each thread reads its own M values from shared memory.

ALGORITHM 10: Thomas-PCR hybrid in X dimension

```
1: Read $N$ values with 32 threads in a coalesced way
2: Transpose array data so that every thread has its own $M$ interior values
3: For each thread, compute interior $M$ elements in terms of outer 2
4: The new tridiagonal system of size 2*32 is solved by PCR, using shuffles or data exchange
   through shared memory
5: Perform back-substitution for interior elements in terms of outer 2 values
```

In Y and higher dimensions the access pattern fits the algorithm better, just like in the case of the standard Thomas algorithm. In this case there is no need for data transposition, but there is need for thread block synchronization. The solution along these dimensions is the most efficient with few restrictions. Algorithm 11 is similar to Algorithm 10 in terms of the underlying hybrid algorithm, with the significant difference that warps cooperate to solve multiple systems. Each thread within a warp solves an M size chunk. The chunks within a warp don't necessarily contribute to the same system. At a certain phase warps share the necessary information to connect the matching systems. Sharing data is done through shared memory with thread block synchronization, but the solution of the reduced system is done with PCR the same way as in Algorithm 10.

ALGORITHM 11: Thomas-PCR hybrid in dimension Y and above

```
1: Threads within a warp are grouped into $(W/G) \times G$ blocks, where $G$ is the number of
   systems solved by a warp.
2: The first $G$ threads read the first $M$ values of $G$ systems, second $G$ threads read the second
   $M$ values and so on.
3: Data of $G$ systems are now loaded into registers
4: Every thread computes an independent $M$ size system: thread 0 computes the first $M$
   values of problem 0, thread $G$ computes the second $M$ values, thread $2G$ computes the
   second $M$ values etc.
5: Threads cooperate through shared memory to each reduced system into one warp.
6: Every warp computes an independent reduced system independently.
7: Reduced solution is propagated back to threads to solve interior systems.
8: Data is stored the same way as it was read.
```

3.7 Performance comparison

The performance of the solvers presented are discussed in terms of scalability on different system sizes and along different dimensions on all three architectures – namely on an NVIDIA GPU, Intel Xeon CPU and Xeon Phi coprocessor. Further in the disseration the

DOI:10.15774/PPKE.ITK.2016.002

GPU will also be referred as SIMT (Single Instruction Multiple Thread) architecture and the CPU and Xeon Phi collectively will also be referred to as SIMD (Single Instruction Multiple Data) architecture.

SIMD (CPU and Xeon Phi) measurement results are heavily contaminated with system noise. Therefore significantly longer integration time had to be used for averaging execution time. There is however a significant recurring spike in the execution time of the Xeon Phi results, that is worth discussing. In single precision almost every system size with a multiple of 128 and almost every system size in double precision with a multiple of 64 results in a significant, more than $\times 2$ slowdown. This is the consequence of cache-thrashing. Cache-thrashing happens when cache-lines on 2^n bytes boundaries with same n are accessed frequently. These memory addresses have different tag ID-s, but the same address within a tag. This means that threads accessing the two cache-lines of such boundaries contaminate the shared L3 (or LLC) level cache, ie. they thrash the shared cache. This problem can be overcome if the cache architecture uses set-associative cache with high associativity – at least as many threads are sharing the cache. This is true in the case of a modern processor. The size of L2 cache is so low in the case of the Xeon Phi coprocessor, that only $512KB/4threads = 128KB/thread$ is available and that might not allow for an efficient set-associativity.

The presented scaling measurements in the following subsections are run with fixed, $256^2 = 65536$ number of systems. The large number of systems ensures enough parallelism even for the Thomas solver – which by nature contains the most sequential computations – in the case of the GPU. Since GPUs need a tremendous amount of work to be done in parallel to hide memory latency, the 65536 parallel work units are enough to saturate the CUDA cores and more importantly the memory controllers. The length of the systems along the different dimensions are changed by extruding the dimension under investigation from 64 up to 1024 with steps of size 4. The resulting execution times are averaged over 100-200 iterations to integrate out system noise. The execution time reported is the proportion of the total execution time to one element of the three dimensional cube and does not include the data transfer to the accelerator card. This gives good reference to compare the different solvers, since it is independent of the number and the length of the systems to be solved. It is expected that the execution time per element be constant for a solver, since the execution time is limited by the memory transfer bottleneck. The execution time differences of the algorithms presented in this

DOI:10.15774/PPKE.ITK.2016.002

section only relate to the memory transfer and memory access pattern of the particular algorithm. All implementations presented here utilize a whole cache line except the SIMD X solvers and the naïve GPU solver. The dependence on system size relates to running out of scratch memory (registers caching, L1/L2/L3 cache, TLB cache) for large systems and having enough workload for efficiency in the case of small systems. These dependences are discussed in the following in the discussion of the corresponding solver. The results are also compared against the *dtsvb()* function of the Intel Math Kernel Library 11.2 Update 2 for Linux library [50] and the *gtsv()* function of the NVIDIA's cuSPARSE library [51]. The *dtsvb()* solver is a diagonally dominant solver which according to the Intel documentation is two times faster than the partial pivoting based *gtsv()* solver. The execution of multiple *dtsvb()* solvers is done using OpenMP. Details of the hardware used are discussed in Appendix A.1. Since the implementations for X and higher dimensions differ we also need separate discussion for these cases.

In a realistic scenario in high dimensions the data transfer between the NUMA nodes is unavoidable since an NUMA-efficient layout for the X dimensional solve is not efficient for the Y dimensional solve and vice versa. Therefore no special attention was made to handle any NUMA related issues in the 2 socket CPU implementation for any dimensions.

3.7.1 Scaling in the X dimension

Figures 3.10 and 3.11 show the performance scaling in the X dimension for single and double precision for all the architectures studied in the dissertation. Figure 3.13 compares the solvers for a specific setup.

3.7.1.1 SIMD solvers

The SIMD solvers rely on the regular Thomas algorithm with or without local data transposition. The detailed description of these solvers is in Section 3.5. The MKL *dtsvb()* solver for diagonally dominant tridiagonal systems was chosen as the baseline solver. In the following comparison all the presented SIMD implementations take advantage of multithreading with OpenMP. Threads using *KMP_AFFINITY=compact* were pinned to the CPU and MIC cores to avoid unnecessary cache reload when threads are

DOI:10.15774/PPKE.ITK.2016.002

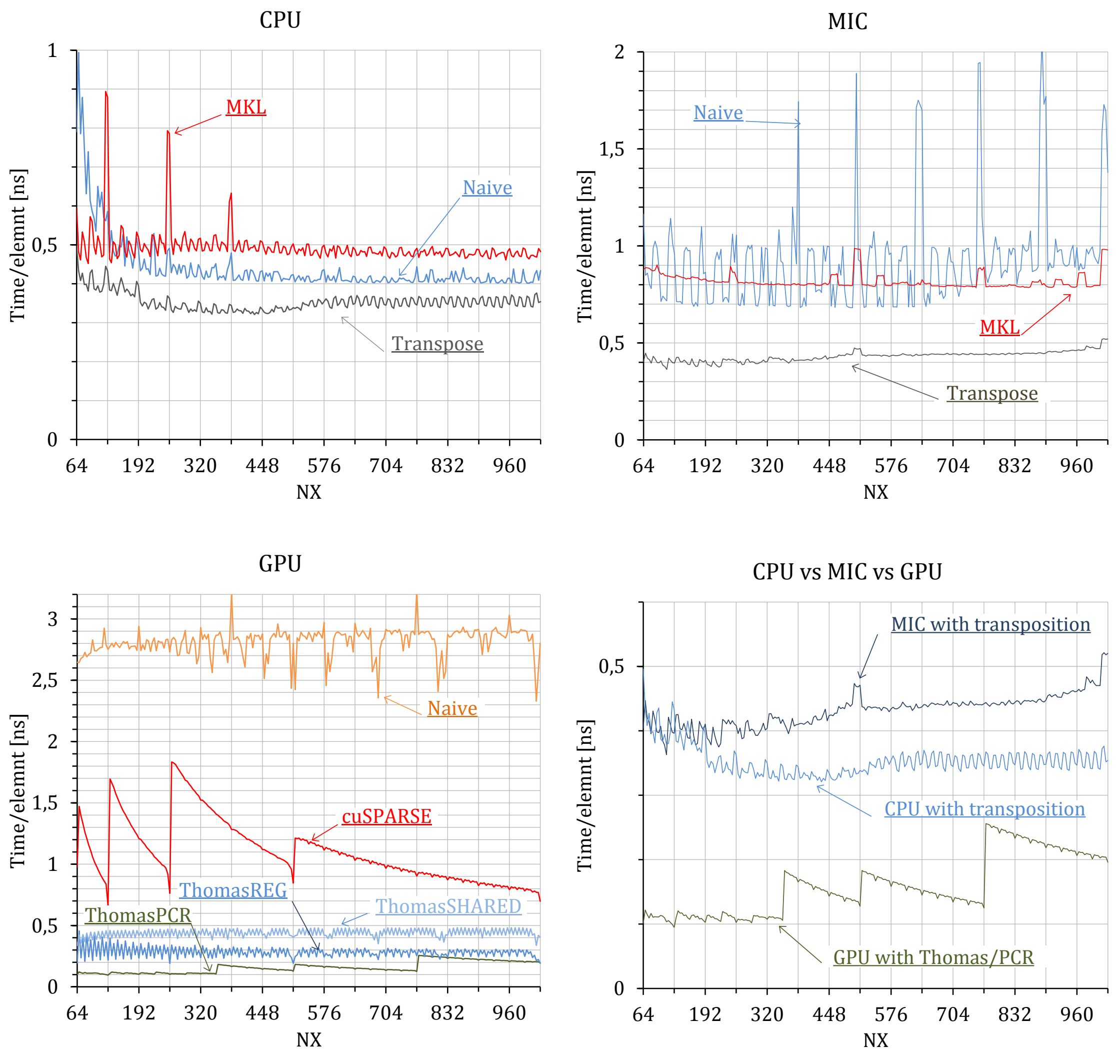


FIGURE 3.10: Single precision execution time per grid element. 65536 tridiagonal systems with varying NX length along the X dimension are solved. CPU, MIC and GPU execution times along with the comparison of the best solver on all three architectures are show.

scheduled to another core. Other run-time optimization approaches using *numactl* and *membind* were also considered, but they did not provide any significant speedup.

As seen in Figure 3.10 and 3.11 the naïve Thomas algorithm with OpenMP outperforms the MKL *dtsvb()* solver and the transposition based Thomas solver further increases the speedup on a 2 socket Xeon processor. The naïve Thomas solver in the X dimension is not capable of utilizing any AVX instruction due to the inherent nature of vector units on CPUs, however the transposition based solvers are capable of taking advantage of the vector units as it is described in Section 3.5. The execution time declines and saturates in both single and double precision as the system size increases, due to the workload necessary to keep threads busy on the CPU. Forking new threads with OpenMP has a constant overhead which becomes less significant when threads have more work. Above

DOI:10.15774/PPKE.ITK.2016.002

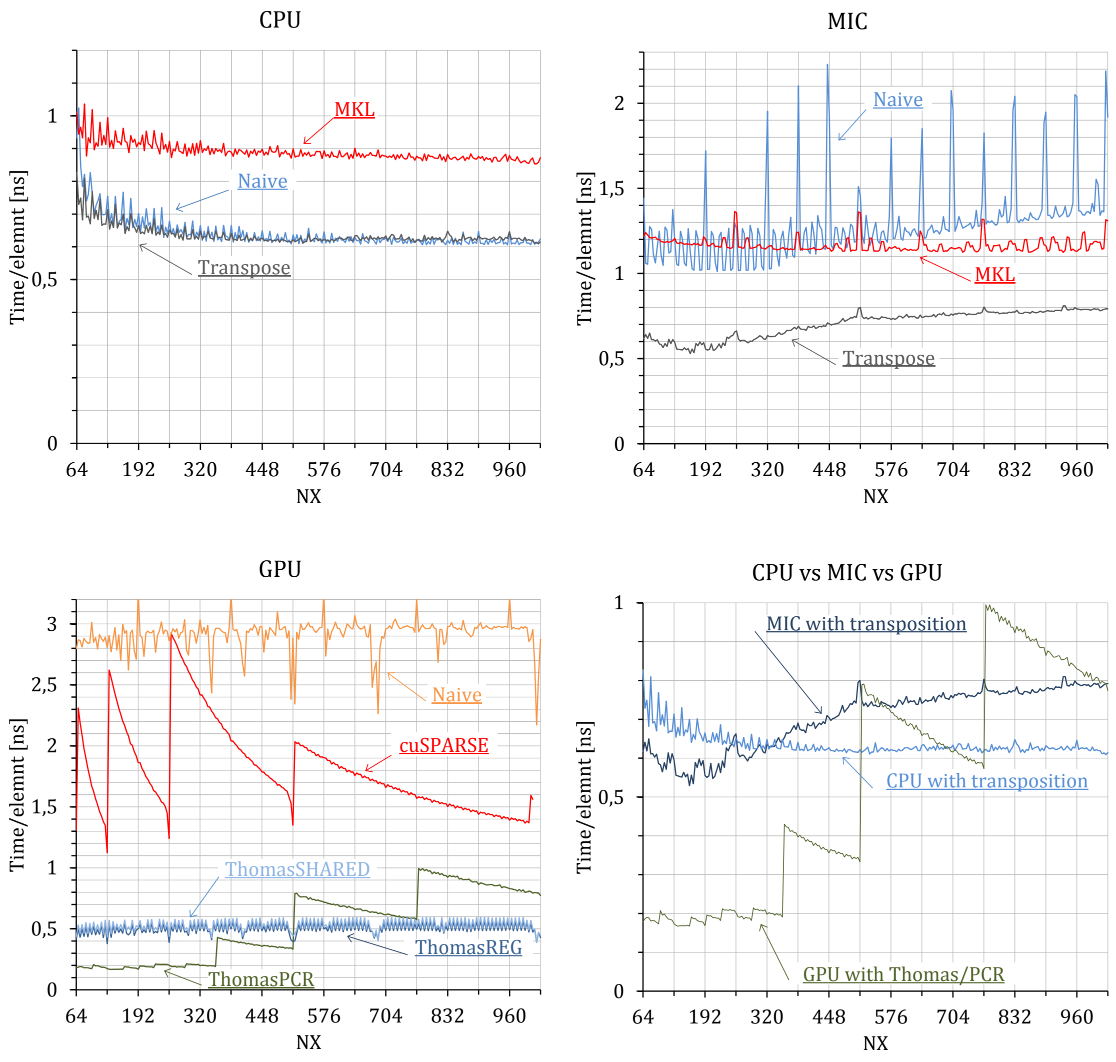


Figure 3.11: Double precision execution time per grid element. 65536 tridiagonal systems with varying NX length along the X dimension are solved. CPU, MIC and GPU execution times along with the comparison of the best solver on all three architectures are show.

a certain system size the CPU provides stable execution time. The efficiency of the CPU relies on the cache and out-of-order execution performance of the architecture. The size of the temporary arrays during the solution is small enough to fit into the L1 cache. The out-of-order execution is capable of hiding some of the 4 clock cycle latency of accessing these temporary, cached values which in total results in high efficiency.

The performance of the naïve Thomas algorithm on the Xeon Phi coprocessor is far from the expected. Due to the difficulty in vectorizing in the X dimension described in Section 3.5, the coprocessor needs to process the data with a scalar code. Since the scalar code is more compute limited than a vectorized code, and since the clock rate of the Xeon Phi is almost a third of the Xeon processor, the efficiency of the code drops to about the third of the vectorized code. However significant x2 increase in speedup is reached on

DOI:10.15774/PPKE.ITK.2016.002

the MIC architecture both in single and double precision when using the transposition based Thomas solver. The performance increase due to vectorization would imply a 8 or 16 times speedup in single or double precision respectively, but the underlying execution architecture, compiler and caching mechanism is not capable of providing this speedup. The overall performance of the Xeon Phi with the transposition based Thomas algorithm is comparable to the CPU for small system size and becomes slower than the CPU as the system size increases above 200-300. Significant spikes in execution time of the naïve solver can be seen in both single and double precision on almost every 512 byte steps, either in steps of 128 in single precision or in steps of 64 in double precision. Thread pining with *KMP_AFFINITY=compact* option prevents thread migration and improves the performance on the coprocessor significantly. To understand the difference between the CPU and the MIC architecture the reader is suggested to study the ISA manuals [43] and [26] of the two architectures. The two architectures are radically different. The CPU works at high clock rates with complex control logic, out-of-order execution, branch prediction, low latency instructions and large, low latency distributed cache per thread. The MIC (Knights Corner) architecture is the opposite in many of these properties. It works at low clock rate, has simple control logic, in-order execution, no branch prediction, higher latency instructions and small, higher latency cache per thread. These fact suggest that when the input data is fetched into the cache, the heavy weight CPU cores utilize these data much efficiently than the light weight MIC cores. One may note that the upcoming Intel Knights Landing architecture with its out-of-order execution unit may significantly improve the MIC performance.

3.7.1.2 SIMT solvers

One may notice that the worst performance is achieved by the naïve GPU solver – even worse than any SIMD implementation. The inefficiency comes from the poor cache line utilizations discussed in Section 3.6. The measurements are contaminated with deterministic, non-stochastic noise that may come from cache efficiency and cache thrashing etc. This noise can not be attenuated by longer integration time. Due to the low cache line utilization the implementation is fundamentally latency limited. This is supported by the fact that the execution time is almost the same for the single and double precision cases. Every step of the solver requires the load of a new cache line and since cache lines

DOI:10.15774/PPKE.ITK.2016.002

on GPU architectures are not prefetched, both single and double precision versions move the same amount of data (32 byte cache line) through the memory bus.

The cuSPARSE v7.0 solver provides better performance than the naïve solver, but it also has a significant recurring dependence on system size. The *cusparse{S,D}gtsvStridedBatch()* solver performs the best when the system size is a power of two in both single and double precision cases. The performance can vary approximately by two times speed difference in both single and double precision. This dependence is due to the internal implementation of the hybrid CR/PCR algorithm that is used inside the solver [51]. The performance of this solver is even worse than the MKL *dtsvb()*library function on the CPU in both single and double precision. The slowdown of cuSPARSE versus the MKL solver is only valid for the regime of short systems in the order of thousands of length. For larger systems this tendency changes for the advantage of the cuSPRASE library. Since the practical applications detailed in the introductory Section 1.2.1 involves the solution of systems below the thousand size regime, systems above this limit are not the scope of the dissertation.

The transposition based Thomas solvers perform better than cuSPARSE by a factor of $2.5-3.1$ in the case of transposition in shared memory and by $4.3-3.3$ in the case of register shuffle depending on floating point precision and system size. The advantage of making extra effort for improving the cache line utilization is obvious. The achieved speedup compared to the naïve Thomas solver is $6.1-7.2$ in the case of single and double precision respectively. The efficiency of the Thomas solver with transposition remains constant as the system size increases, ie. there is no significant fluctuation in performance.

One may notice that the padding is important for the register transposition so that aligned *float4* or *double2* loads may be done, which can not always be ensured. There is a factor 2 speed difference between the shared memory transposition and the register shuffle transposition in single precision. This difference is negligible in double precision since 64bit wide shared memory bank access is used to access the scratch memory instead of 32bit in the case of single precision. The wider bank access has been introduced in the Kepler architectures and it effectively doubles the bandwidth of the shared memory when 64bit access can be ensured. In case of shared memory a read throughput of 1211 GB/s and store throughput of 570 GB/s is measured with NVVP (NVIDIA Visual

DOI:10.15774/PPKE.ITK.2016.002

Profiler) on a 256^3 single precision cube domain. Also, the transposition using shared memory happens by reading 32bytes at a time, whereas in the case of the register shuffle based transposition 64 bytes are read from the global memory at a time.

The hybrid Thomas-PCR algorithm outperforms every solver in the X dimension in single precision. In double precision however the performance drops significantly beyond system size 512. The efficiency is due to the register blocking nature of the algorithm. Each system that is solved is completely resident in registers and therefore only the input data is read once and the output is stored once. This results in the minimum amount of data transfers and leads to the best possible performance.

3.7.2 Scaling in higher dimensions

Figure 3.12 shows the performance scaling in the Y dimension for single and double precision for all the architectures studied. Since the solution of tridiagonal systems with non-unit stride is not implemented in any library up to date, the higher dimensional execution time benchmarks do not contain any standard library execution times. Transposing the data structure would allow for the use of the standard solvers, but it has not been done for two reasons: 1) the efficiency of transposing the whole data structure would involve further optimization of implementations attached to standard library code and the overall performance would be influenced by the extra programming effort; 2) even with a highly optimized transposition the overall execution time would be higher then in the X dimension.

3.7.2.1 SIMD solvers

The SIMD solvers rely on the regular Thomas algorithm. The detailed description is in Section 3.5. Both the CPU and the MIC SIMD architectures are able to utilize the 256 bits and 512 bits wide AVX and IMCI vector units to solve a tridiagonal system. This means that 8(4) and 16(8) systems can be solved by a single thread in parallel in single(double) precision on CPU and MIC respectively. The efficiency of the CPU relies on the cache and out-of-order execution performance of the architecture. The size of the temporary arrays for the systems solved in parallel is small enough to fit into the L1 cache and the L2 cache. The out-of-order execution is capable of hiding some of

DOI:10.15774/PPKE.ITK.2016.002

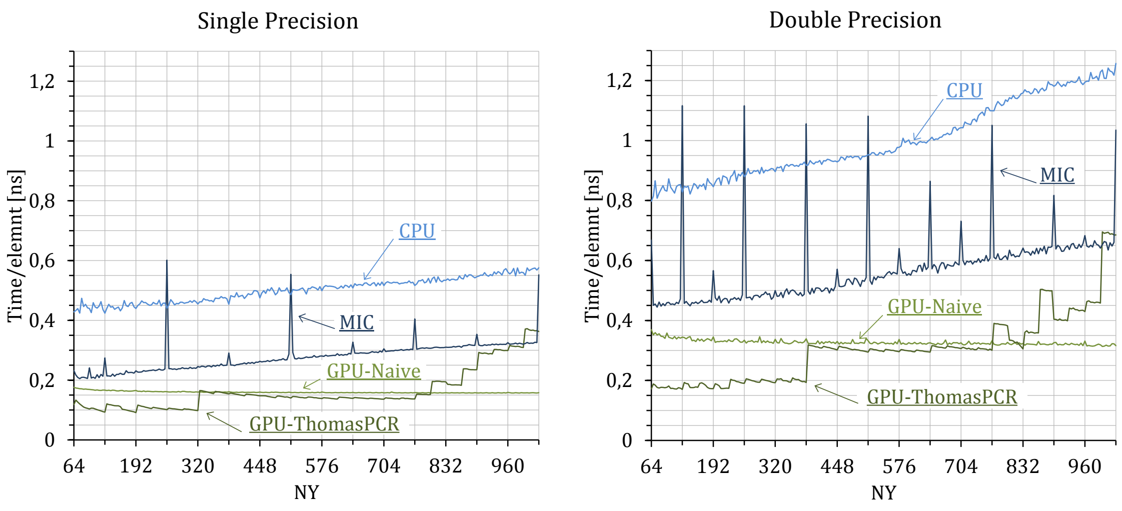


FIGURE 3.12: Single and double precision execution time per grid element. 65536 tridiagonal systems with varying NY length along the Y dimension are solved. CPU, MIC and GPU execution times along with the comparison of the best solver on all three architectures are show.

the 4 clock cycle latency of accessing these temporary in the L1 cache. On Figure 3.12 it can be seen that the CPU and MIC performance changes in parallel. Although the scaling should be constant, it is changing almost linearly in the case of single precision and super linearly in double precision. The reason for the latter is that the cores are running out of L1 cache and the L2 cache performance starts to dominate. Two arrays (c^* and d^*) need to be cached for good performance. For instance, system size $N = 1024$ needs $1024element * 8bytes/element * 2arrays * 4parallelsystems = 64KB$ to be cached, which is twice the size of the L1 cache. The single precision solver remains linear in this regime, because even 1024 long system fits into the L1 cache. The order of the two for loops (X and Z) iterating on the 65526 systems have set so that better data locality is achieved and thus better TLB hit rate is reached. The Thomas solver on the dual socket Xeon CPU is the slowest among all the solvers. The MIC architecture is by a factor $1.8 - 2$ faster then the CPU. The SIMD architectures require aligned loads for best performance which can be ensured with padding otherwise the performance is hit by unaligned data loads and stores. The vector operations can still be performed, the non-alignment only hits data transfer.

3.7.2.2 SIMT solvers

The naïve GPU solver provides stable execution time and it is up to 3.6(3.8) times faster than the dual socket Xeon CPU and 2.1(2.5) times faster then the Xeon Phi in

DOI:10.15774/PPKE.ITK.2016.002

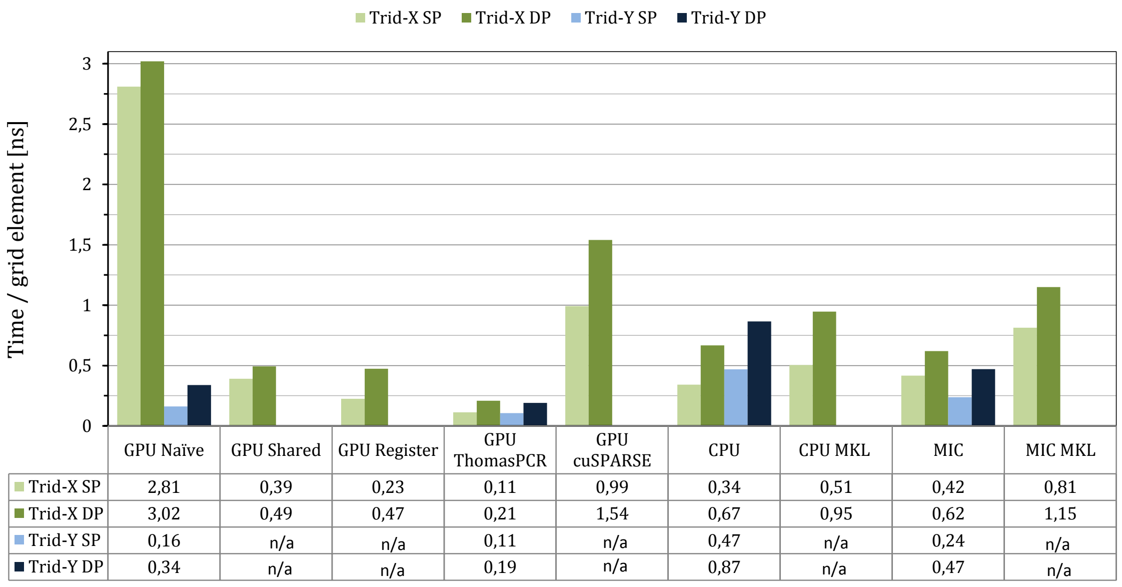
single(double) precision. The GPU implementation is capable of solving 32 systems within a warp using coalesced memory access. The performance is therefore predictable and very efficient. The only drawback of the solver is that it is moving more data than the Thomas-PCR hybrid solver (detailed in Section 3.2.4) which caches the data in registers. Therefore the Thomas-PCR hybrid algorithm is up to 1.5(1.8) faster than the naïve GPU solver in case of single(double) precision. Compared to the highly optimized two socket CPU solver the Thomas-PCR solver is 4.3(4.6) faster in single(double) precision. Compared to the highly optimized MIC the speedup is $2.2 - (2.5)$. These are significant differences which can be maintained until there is enough register to hold the values of the processed systems. Once the compiled code runs out of register, the effect of register spill becomes obvious, since the execution time jumps by more the 50% – this happens with system size 320 in single precision and 384 in double precision. In case of register spill the advantage of the Thomas-PCR over the naïve solver is negligible and above certain system size it is even worse.

3.8 Conclusion

In the past many algorithms have been introduced to solve a system of tridiagonal equations, but only a few of them took advantage of the fact that in certain cases (eg. an ADI solver) the problem to be solved contains a large number of small systems to solve and the access pattern of the system elements might change in data structures with 2 dimensions and above. It has been shown that in the X dimension the standard Thomas algorithm with modified access pattern using local data transposition on both CPU, MIC and GPU can outperform library quality, highly optimized code, such as: 1) *dtsvb()* MKL diagonally dominant solver running on multiple threads on a dual socket Xeon processor and 2) the PCR-CR hybrid algorithm implemented in the cuSPARSE library provided by NVIDIA. If the system size allows caching in registers, then a new proposed Thomas-PCR hybrid algorithm on the GPU can be used to solve the problem even more efficiently with a speedup of about 2 compared to the Thomas algorithm with local data transposition. It has been shown that in higher dimensions (Y, Z etc.) the naïve solver allows for coalesced (or almost coalesced) access pattern and therefore there is no need for transposition and the performance is high. The Thomas-PCR for X dimension is modified to handle systems in higher dimensions by using more warps to

DOI:10.15774/PPKE.ITK.2016.002

load and store the data required in the computation. The register and shared memory pressure is higher in these cases and register spills occur above system size 320 in single and 384 in double precision. The Thomas algorithm performs better for above these system sizes. Figure 3.13 summarizes the exection times for the X and Y dimensions in the case of a $240 \times 256 \times 256$ domain.



	GPU Naïve	GPU Shared	GPU Register	GPU ThomasPCR	GPU cuSPARSE	CPU	CPU MKL	MIC	MIC MKL
Trid-X SP	2,81	0,39	0,23	0,11	0,99	0,34	0,51	0,42	0,81
Trid-X DP	3,02	0,49	0,47	0,21	1,54	0,67	0,95	0,62	1,15
Trid-Y SP	0,16	n/a	n/a	0,11	n/a	0,47	n/a	0,24	n/a
Trid-Y DP	0,34	n/a	n/a	0,19	n/a	0,87	n/a	0,47	n/a

FIGURE 3.13: Grid element execution times per grid element on a $240 \times 256 \times 256$ cube domain. X dimension is chosen to be 240 to alleviate the effect of caching and algorithmic inefficiencies which occurs on sizes of power of 2 as it can be seen on Figures 3.10,3.11,3.12. Missing bars and n/a values are due to non-feasible or non-advantagous implementations in the Y dimension.

The conclusion is, that the Thomas algorithm with modified access pattern is advantageous up to relatively large system sizes in the order of thousands. The Thomas-PCR hybrid gives better performance in the X dimension in this regime. In the Y dimensions and above the Thomas-PCR is the best performing up to system size 320(384) in single(double) precision, but above this size the Thomas regains its advantage due to its simplicity.

DOI:10.15774/PPKE.ITK.2016.002

Chapter 4

Block Tridiagonal Solvers on Multi and Many Core Architectures

4.1 Introduction

In many real-world CFD and financial applications the multidimensional PDEs have interdependent state variables. The state variable dependence creates a block structure in the matrix used in the implicit solution of the PDE. In certain cases these matrices are formed to be tridiagonal with block matrices in the diagonal and off-diagonals. The M^2 block sizes are usually in the range of $M = 2, .., 8$ and therefore the tridiagonal matrix takes the forms shown in Equation 4.1 and 4.2.

$$\mathbf{A_i u_{i-1}} + \mathbf{B_i u_i} + \mathbf{C_i u_{i+1}} = \mathbf{d_i} \tag{4.1}$$

$$\begin{pmatrix} \mathbf{B_0} & \mathbf{C_0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{A_1} & \mathbf{B_1} & \mathbf{C_1} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{A_2} & \mathbf{B_2} & \mathbf{C_2} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{A_{N-1}} & \mathbf{B_{N-1}} \end{pmatrix} \begin{pmatrix} \mathbf{u_0} \\ \mathbf{u_1} \\ \mathbf{u_2} \\ \vdots \\ \mathbf{u_{N-1}} \end{pmatrix} = \begin{pmatrix} \mathbf{d_0} \\ \mathbf{d_1} \\ \mathbf{d_2} \\ \vdots \\ \mathbf{d_{N-1}} \end{pmatrix} \tag{4.2}$$

DOI:10.15774/PPKE.ITK.2016.002

where $\mathbf{A_i}, \mathbf{B_i}, \mathbf{C_i} \in \mathbb{R}^{MxM}$, $\mathbf{u}_i, \mathbf{d}_i \in \mathbb{R}^M$ and $M \in [2, 8]$.

4.2 Block-Thomas algorithm

ALGORITHM 12: Block Thomas algorithm

```
Require: $block\_thomas(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{d})$
    Forward pass
 1: $\mathbf{C_0}^* = \mathbf{B_0^{-1}}\mathbf{C_0}$
 2: $\mathbf{d_0}^* = \mathbf{B_0^{-1}}\mathbf{d_0}$
 3: for $i = 1, \ldots, N-1$ do
 4:     $\mathbf{C_i}^* = (\mathbf{B_i} - \mathbf{A_i}\mathbf{C^*_{i-1}})^{-1}\mathbf{C_i}$
 5:     $\mathbf{d_i}^* = (\mathbf{B_i} - \mathbf{A_i}\mathbf{C^*_{i-1}})^{-1}(\mathbf{d_i} - \mathbf{A_i}\mathbf{d^*_{i-1}})$
 6: end for
    Backward pass
 7: $\mathbf{u_{N-1}} = \mathbf{d^*_i}$
 8: for $i = N-2, \ldots, 0$ do
 9:     $\mathbf{u_i} = \mathbf{d^*_i} - \mathbf{C^*_i}\mathbf{u_{i+1}}$
10: end for
11: return u
```

The block Thomas algorithm is essentially the same as the scalar algorithm, assuming that scalar operations are exchanged with matrix operations. The lack of commutative property of the matrix multiplication, the order of these matrix operations have to be maintained throughout the implementation. See Algorithm 12 for details. The block structure introduces matrix operations such as 1) block matrix inversion with $O(M^3)$; 2) block matrix-matrix multiplication with $O(M^3)$ and addition with $O(M^2)$; 3) block matrix-vector multiplication with $O(M^2)$. On the other hand the data transfer is only $O(M^2)$ per block matrix. As the matrix operations are dominated by $O(M^3)$ versus the $O(M^2)$ data transfer, the solution of the block tridiagonal system becomes compute limited as the block size increases. Once the data is read and stored in scratch memory, the cost of making the matrix operations is the bottleneck, both because arithmetic complexity and control flow complexity are significant. Let us define the O complexity in terms of block matrix operations with the arithmetic complexity states above, and define the total work as the complexity of solving a single system times the number of systems

DOI:10.15774/PPKE.ITK.2016.002

to be solved on a given number of parallel processors. When solving N long systems on N processors the Thomas algorithm has $N\,O(N)$ total work complexity versus the $N\,O(N\log N)$ total work complexity of the PCR algorithm. This significant difference establishes the use of the Thomas algorithm over the PCR in the block tridiagonal case.

Algorithms for solving block-tridiagonal system of equations have been previously developed in [52, 53, 33]. [54] gave a GPU based solution using the block PCR algorithm motivating their choice by the inherent parallelism given by the algorithm and the demand for high parallelism by the GPU. The overall arithmetic complexity of the PCR is known to be higher than that of the Thomas algorithm as it is detailed in Section 4.2. As many CFD applications consider block sizes of $M = 2, .., 8$ the motivation is to make use of the computationally less expensive algorithm, namely the Thomas algorithm and exploit parallelism in the block-matrix operations. In other words, the overall arithmetic complexity is kept low by the Thomas algorithm and the parallelism is increased by the work-sharing threads which solve the block-matrix operations.

[54] used a highly interleaved SOA (Structure of Arrays) storage format to store the coefficients of the systems in order to achieve coalesced memory access pattern suitable for the GPU. In that approach the data on the host is stored in AOS (Array of Structures) format which had to be converted into SOA format to suit the needs of the GPU. In our approach the work-sharing and register blocking solution alleviates the need for AOS-SOA conversion, ie. the data access efficiency remains high.

4.3 Data layout for SIMT architecture

The data layout is a critical point of the solver, since this influences the effectiveness of the memory access pattern. Coalesced memory access is needed to achieve the best theoretical performance, therefore SOA data structures are used in many GPU applications. In this section an AOS data storage is presented in which blocks of distinct tridiagonal system are interleaved. Block coefficients $\mathbf{A^p_n}, \mathbf{B^p_n}, \mathbf{C^p_n}, \mathbf{d^p_n}$ are stored in separate arrays. The data layout of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{C}^*$ coefficient block arrays are the same. Within the array of blocks the leading dimension is the row of a block, ie. blocks are stored in row major format. The block of system p is followed by the block of system block $p+1$ in

DOI:10.15774/PPKE.ITK.2016.002

the array, ie. the blocks in array $\mathbf{A}$ are stored by interleaving the nth blocks of problems $p = 0, .., P-1$ in the way shown on Figure 4.1.

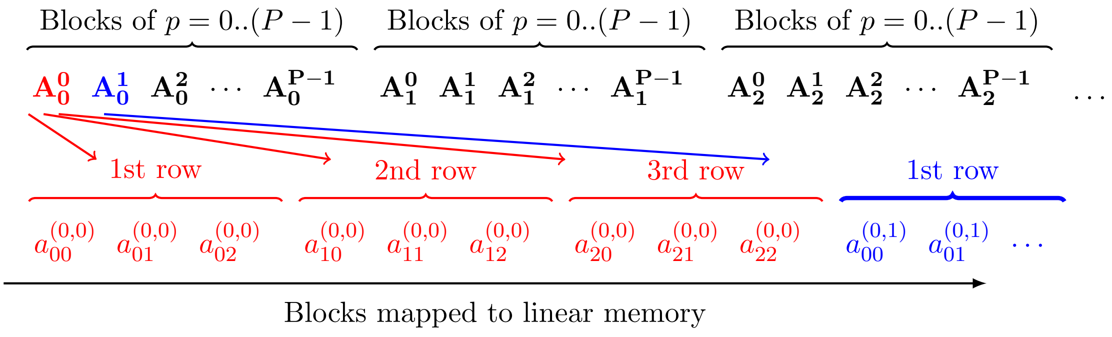

Blocks of $p = 0..(P-1)$
Blocks of $p = 0..(P-1)$
Blocks of $p = 0..(P-1)$
$\mathbf{A_0^0}$ $\mathbf{A_0^1}$ $\mathbf{A_0^2}$ $\cdots$ $\mathbf{A_0^{P-1}}$
$\mathbf{A_1^0}$ $\mathbf{A_1^1}$ $\mathbf{A_1^2}$ $\cdots$ $\mathbf{A_1^{P-1}}$
$\mathbf{A_2^0}$ $\mathbf{A_2^1}$ $\mathbf{A_2^2}$ $\cdots$ $\mathbf{A_2^{P-1}}$ $\ldots$
1st row
2nd row
3rd row
1st row
$a_{00}^{(0,0)}$ $a_{01}^{(0,0)}$ $a_{02}^{(0,0)}$ $a_{10}^{(0,0)}$ $a_{11}^{(0,0)}$ $a_{12}^{(0,0)}$ $a_{20}^{(0,0)}$ $a_{21}^{(0,0)}$ $a_{22}^{(0,0)}$ $a_{00}^{(0,1)}$ $a_{01}^{(0,1)}$ $\ldots$
Blocks mapped to linear memory

FIGURE 4.1: Data layout within coefficient array $\mathbf{A}$. This layout allows nearly coalesced access pattern and high cache hit rate when coalescence criteria is not met. Here $\mathbf{A_n^p}$ is the nth block (ie. nth block-row in the coefficient matrix) of problem p and $p \in [0, P-1]$, $n \in [0, N-1]$. Notation for the scalar elements in the nth block of problem p are shown on Eq. (4.3). Bottom part of the figure shows the mapping of scalar values to the linear main memory.

$$\mathbf{A_n^p} = \begin{pmatrix} a_{00}^{(n,p)} & a_{01}^{(n,p)} & a_{02}^{(n,p)} \\ a_{10}^{(n,p)} & a_{11}^{(n,p)} & a_{12}^{(n,p)} \\ a_{20}^{(n,p)} & a_{21}^{(n,p)} & a_{22}^{(n,p)} \end{pmatrix} \tag{4.3}$$

Vectors $\mathbf{d}$, $\mathbf{d}^*$ and $\mathbf{u}$ are stored in a similar, interleaved fashion as depicted on Figure 4.1. Subvectors in array $\mathbf{d}$ are stored by interleaving the nth subvector of problems $p = 0, .., P-1$ in the way shown on Figure 4.2. The notation of the scalar values of $\mathbf{d_n^p}$ can be seen on Eq. (4.4).

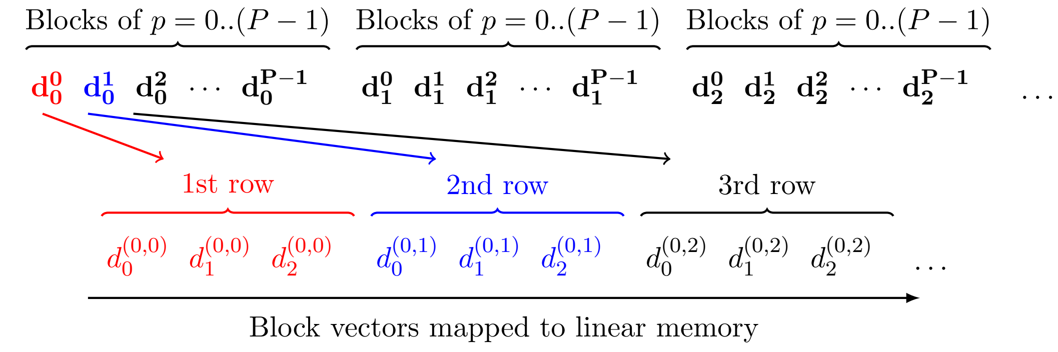

Blocks of $p = 0..(P-1)$
Blocks of $p = 0..(P-1)$
Blocks of $p = 0..(P-1)$
$\mathbf{d_0^0}$ $\mathbf{d_0^1}$ $\mathbf{d_0^2}$ $\cdots$ $\mathbf{d_0^{P-1}}$
$\mathbf{d_1^0}$ $\mathbf{d_1^1}$ $\mathbf{d_1^2}$ $\cdots$ $\mathbf{d_1^{P-1}}$
$\mathbf{d_2^0}$ $\mathbf{d_2^1}$ $\mathbf{d_2^2}$ $\cdots$ $\mathbf{d_2^{P-1}}$ $\ldots$
1st row
2nd row
3rd row
$d_0^{(0,0)}$ $d_1^{(0,0)}$ $d_2^{(0,0)}$ $d_0^{(0,1)}$ $d_1^{(0,1)}$ $d_2^{(0,1)}$ $d_0^{(0,2)}$ $d_1^{(0,2)}$ $d_2^{(0,2)}$ $\ldots$
Block vectors mapped to linear memory

FIGURE 4.2: Data layout of $\mathbf{d}$ as block vectors. This layout allows nearly coalesced access pattern and high cache hit rate when coalescence criteria is not met. Here $\mathbf{d_n^p}$ is the nth block (ie. nth block-row in the coefficient matrix) of problem p and $p \in [0, P-1]$, $n \in [0, N-1]$. Notation for the scalar elements in the nth block of problem p are shown on Eq. (4.4). Bottom part of the figure shows the mapping of scalar values to the linear main memory.

$$\mathbf{d_n^p} = \begin{pmatrix} d_0^{(n,p)} & d_1^{(n,p)} & d_2^{(n,p)} \end{pmatrix}^T \tag{4.4}$$

DOI:10.15774/PPKE.ITK.2016.002

This data layout allows the threads to load data through texture cache from global memory with efficient access pattern and cache line utilization. A block is read in M steps. In each step a row of the block is read. The scalar values are stored in the registers as shown of Figure 4.3. In the second step threads read the second row in the same manner, and so on.

This storage format allows for high texture cache hit rate by assuring that most of the data that is read in within a 32 bytes cache line. Let us take the case of $M = 3$ in single precision when reading A_0^0. In the first step only the first row of the block is read, ie. 12 bytes of the 32 bytes cache line is utilized. Once a texture cache line is loaded it will be reused immediately in the following few instructions or by the neighboring thread. In the next step, when the second row of the block is read, the cache line is already in the texture cache and this time 12 bytes are directly read from the texture cache. In the third step, when the third row is read, the 12 bytes are again in the cache, since 8 bytes out of the 12 bytes are in the same cache line as the first two rows, and the remaining 4 bytes are in the cache line read by the next group of 3 threads which read the block A_0^1. All this is done in parallel by $\lfloor 32/3 \rfloor = 30$ threads. The total amount of data that is read by the warp is $3^2 \frac{\text{scalars}}{\text{block}} \times 4 \frac{\text{blocks}}{\text{system}} \times 10 \text{ systems} = 360\text{bytes}$, which fits into 12 cache lines. Only $3 \times 10 = 30$ threads are active in the warp (which has 32 threads). The probability of a cache line being evicted is low, since the cache lines are reused by the threads in the same warp. Since the sequence of instructions of loading a block doesn't contain data dependency, there is no reason for the scheduler to idle the active threads which started loading the data.

4.4 Cooperating threads for increased parallelism on SIMT architecture

SIMT architectures are inherently sensitive to parallelism, ie. they need enough running threads to hide the latency of accessing the memory and filling up the arithmetic pipelines efficiently. Also, the implemented solver has to consider the constraints of the processor architecture being used. The Kepler GK110b architecture has 48KiB shared

DOI:10.15774/PPKE.ITK.2016.002

memory and 64K 32bit registers available for use within a thread block with up to 255 registers/thread. Computing block matrix operations with a single thread is not efficient due to the following reasons: 1) the limited number of registers and shared memory doesn't allow for temporary storage of block matrices and reloading data from global memory is costly; 2) in order to utilize coalesced memory access a high level of input data interleaving would be required, which is not useful in real application environment. As a consequence the problem would become memory bandwidth limited rather than compute limited. The workload of computing a single systems therefore needs to be shared among threads, so that block matrix data is distributively stored in the registers (and shared memory) of threads. This means that both workload and data storage is distributed among the cooperating threads.

We propose to use M number of threads to solve the M dimensional block matrix operations, so that every thread stores one column of a block and one scalar value of a vector that is being processed, see Figure 4.3 for details. Subsequent M threads are computing a system and a warp computes $\lfloor 32/M \rfloor$ number of systems. This means that $\lfloor 32/M \rfloor * M$ threads are active during the computation. The rest are idle. With this work distribution in the $M = 2, .., 8$ range the worst case is $M = 7$ when 4 out of 32 threads are inactive and thus 87.5% is the actual computation performance that can be reached.

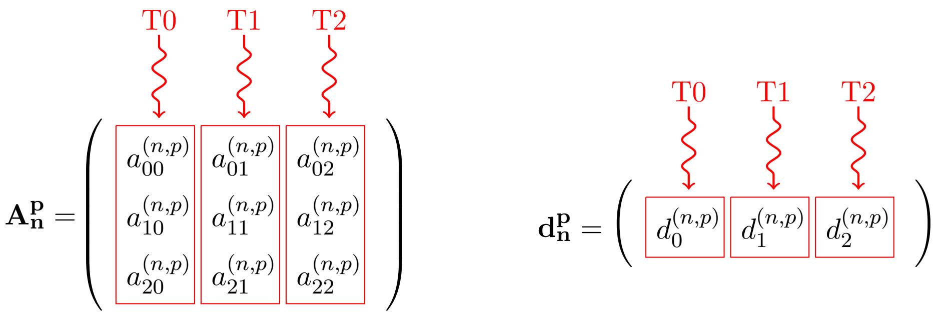


FIGURE 4.3: Shared storage of blocks in registers. Arrows show load order. Thread $T0$,$T1$ and $T2$ stores the first, second and third columns of block $\mathbf{A_n^p}$ and first, second and third values of $\mathbf{d_n^p}$.

The communication to perform matrix operations is carried out using either shared memory or register shuffles (*__shfl()* intrinsic). Just like in the case of the scalar solver, register and shared memory is not enough to store the intermediate $\mathbf{C}^*, \mathbf{d}^*$ block arrays

DOI:10.15774/PPKE.ITK.2016.002

and for this purpose local memory is used – this is an elegant way of getting coalesced memory access.

In the following the essential block matrix operations are discussed using Figure 4.3. Important to notice that all the implementation with register shuffle is written in a way that the local arrays storing block columns are held in registers. Two criteria need to be satisfied to achieve this, 1) local array indexing needs to be known at compile time and 2) local array size can not exceed a certain size defined internally in the compiler.

Matrix-vector multiplication

When performing matrix-vector multiplication, threads have the column of a block and the corresponding scalar value to perform the first step of matrix-vector multiplication, namely scalar-vector multiplication – this is done in parallel, independently by the threads. The second step is to add up the result vectors of the previous step. In this case threads need to share data, which can either be done through shared memory or using register shuffle instructions. In the case of *shared memory* the result is stored in shared memory. It is initialized to 0. In the mth step (where $m = 0, .., M-1$) thread m adds its vector to the shared vector and thread block synchronizes. In the case of *register shuffle* the multiplication is also done in M steps. In the mth step the M threads compute a scalar product of the mth row and shuffle the computed values around (in round-robin) to accumulate these values. The actual addition of the accumulation is done by the mth thread.

Matrix-matrix multiplication

Matrix-matrix multiplication needs to communicate the $M \times M$ values of one of the blocks. This can either be done through shared memory or register shuffle. In the case of shared memory this approach uses M^2 number of *__syncthreads()* which would suggest heavy impact on performance, but the results are still satisfying. The register shuffle approach doesn't require synchronization, thus it is expected to work faster then the shared memory approach.

DOI:10.15774/PPKE.ITK.2016.002

Gauss-Jordan elimination

The solution of block systems in line 4 and 5 in Algorithm 12 is done by immediately solving these systems when performing a Gauss-Jordan elimination, ie. the systems are solved without composing the explicit inverse of matrix $\mathbf{B_i} - \mathbf{A_i}\mathbf{C^*_{i-1}}$. The Gauss-Jordan elimination is computed cooperatively by using either shared memory of register shuffle instructions. Both versions have high compute throughput.

4.5 Optimization on SIMD architecture

The approach for efficient solution in the case of SIMD architectures is different in terms of data storage and execution flow. The data layout needs to be changed to get better data locality (for better cache performance) and second, each individual HT (Hyper Thread) thread computes an independent system without sharing the workload of the solution of one system. The latter consideration is the natural way of doing computation on multi-core systems. As each thread solves an independent system, best cache locality is reached when the blocks of array A are reordered to store the blocks A^p_0, A^p_1, ..., A^p_N next to each other. Figure 4.4 depicts this in more details.

$$\overbrace{\mathbf{A^0_0}\ \ \mathbf{A^0_1}\ \ \mathbf{A^0_2}\ \cdots\ \mathbf{A^0_{N-1}}}^{\text{Blocks of problem } p=0}\quad \overbrace{\mathbf{A^1_0}\ \ \mathbf{A^1_1}\ \ \mathbf{A^1_2}\ \cdots\ \mathbf{A^1_{N-1}}}^{\text{Blocks of problem } p=1}\quad \overbrace{\mathbf{A^2_0}\ \ \mathbf{A^2_1}\ \ \mathbf{A^2_2}\ \cdots\ \mathbf{A^2_{N-1}}}^{\text{Blocks of problem } p=2}\quad \cdots$$

FIGURE 4.4: Data layout within coefficient array $\mathbf{A}$ suited for better data locality on CPU and MIC. Here $\mathbf{A^p_n}$ is the nth block (ie. nth block-row in the coefficient matrix) of problem p and $p \in [0, P-1]$, $n \in [0, N-1]$. Notation for the scalar elements in the nth block of problem p are shown on Eq. (4.3).

On these systems threads are heavy weight with out-of-order execution and have enough L1 cache to efficiently solve a block tridiagonal system. The C++ code is optimized to take advantage of the vector units. Loops are written so that the requirements for auto-vectorization are satisfied. Where needed *#pragma ivdep* or *#pragma simd* is used to help and force the compiler to vectorize loops. Dimension sizes are passed through template variables to make better use of compile time optimizations such as loop unrolling, vectorization and static indexing. Thread level parallelism is provided by OpenMP and threads solving independent systems are scheduled with guided scheduling to balance work load and give an overall better performance.

DOI:10.15774/PPKE.ITK.2016.002

4.6 Performance comparison and analysis

Performances of the implemented solvers are compared in terms of memory bandwidth, computation throughput in the case of GPUs and speedups of GPU and CPU implementations compared to the banded solver *gbsv()*. The size of the problem is always chosen to be such that it saturates both the CPU and the GPU with enough work, so that these architectures can provide the best performance, ie. as the block size is increased the length of the system to be solved is decreased so that the use of the available memory is kept close to maximum. Table A.2. and A.3. show the selected length of a system (N) and the number of systems to be solved (P) respectively.

The Intel Math Kernel Library 11.2 Update 2 library [50] was chosen with its *LAPACKE_{s,d}gbsv_work()* function to perform the banded solution. The library function was deploy using OpenMP to achieve the best performance MKL can provide. According to the MKL manual [50] this routine solves for X the linear equations $AX = B$, where $A \in \mathbb{R}^{n\times n}$ band matrix with kl subdiagonals and ku superdiagonals. The columns of matrix B are individual right-hand sides (RHS), and the columns of X are the corresponding solutions. This function is based on LU decomposition with partial pivoting and row interchanges. The factored form of A is then used to solve the system of equations $AX = B$. The solution of a system is done in two steps. First, a partial solve is done with the upper-triangular U matrix and then a solve with the lower-triangular L matrix is performed. This is an efficient approach when many right hand side exist. In the present case there is always one RHS, ie. $X \in \mathbb{R}^{n\times 1}$ and $B \in \mathbb{R}^{n\times 1}$. As the systems which are solved are defined with diagonally dominant matrices, pivoting is not performed during execution time. Moreover, the *_work* version of the solver neglects any data validity check and thus provides a fair comparison. The scalar elements of diagonal block matrix arrays A_n^p, B_n^p and C_n^p are mapped to band matrix A and the scalar elements of diagonal block vector arrays of d_n^p and u_n^p are mapped to X and B accordingly. The performance of the routine is expected to be high as the triangular sub-matrices are small enough to reside in L1, L2 or L3 cache. Comparing the solutions of the banded solver with the block tridiagonal solver by taking the differences between the corresponding scalar values shows MSE (Mean Square Error) in the order of 10^{-4}. MSE does varies but stays in the order of 10^{-4} as floating point precision, block size, system size or the number of systems is changed.

DOI:10.15774/PPKE.ITK.2016.002

Figure 4.5 and Figure 4.6 show the effective bandwidth and computation throughput. The term effective is used to emphasize the fact that these performance figures are computed on basis of the actual data needed to be transferred and actual floating point arithmetic needed to be performed. Any caching and FMA influencing these figures are therefore implicitly included. In general the register shuffle based GPU solver outperforms the shared memory version, with one major exception in double precision with $M = 8$ block size. In this case the register pressure is too high and registers get spilled into local memory.

One may notice that, as the block size increases the effective bandwidth decreases on Figure 4.5 and the effective compute throughput increases at the same time on Figure 4.6. Therefore it is implied that the problem is becoming compute limited rather than bandwidth limited as it is discussed in Section 4.2 due to the increasing difference between the $O(M^3)$ compute and $O(M^2)$ memory complexity of a single block.

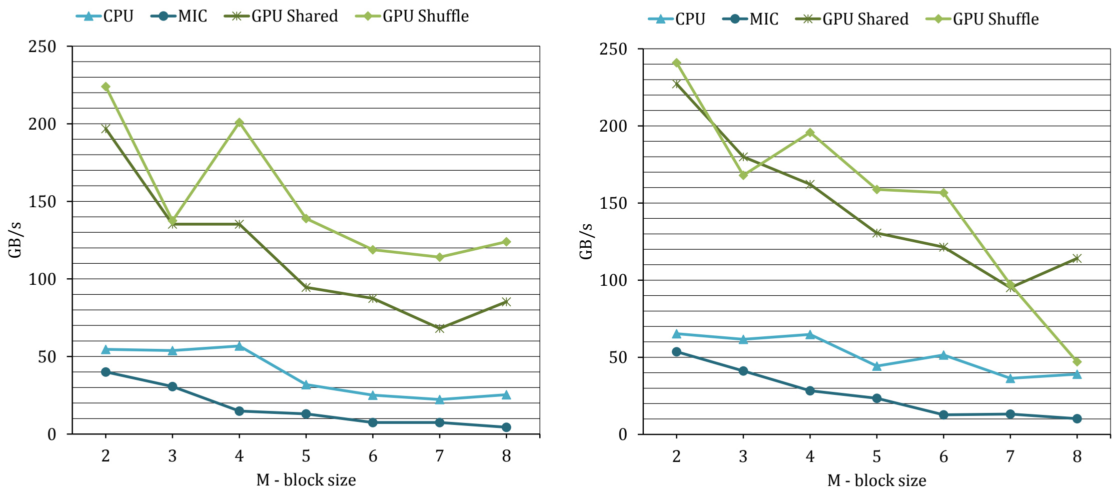


FIGURE 4.5: Single (left) and double (right) precision effective bandwidth when solving block tridiagonal systems with varying M block sizes. Bandwidth is computed based on the amount of data to be transfered. Caching in L1 and registers can make this figure higher then the achievable bandwidth with a bandwidth test.

Execution time per block row are shown of Figure 4.7. The relative execution time measures the efficiency which is independent of problem size. The total execution time of the solver is divided by NP, where N is the length of a system and P is the number system that are solved. Due to alignment effects the performance for $M = 4$ and $M = 8$ is significantly better than for sizes that don't allow perfect alignment. The only exception is the $M = 8$ case where execution time drastically increases for the double precision shuffle version. This is due to register spilling and local memory allocation, ie. data can no longer fit into registers, therefore it is put into local memory, also the small

DOI:10.15774/PPKE.ITK.2016.002

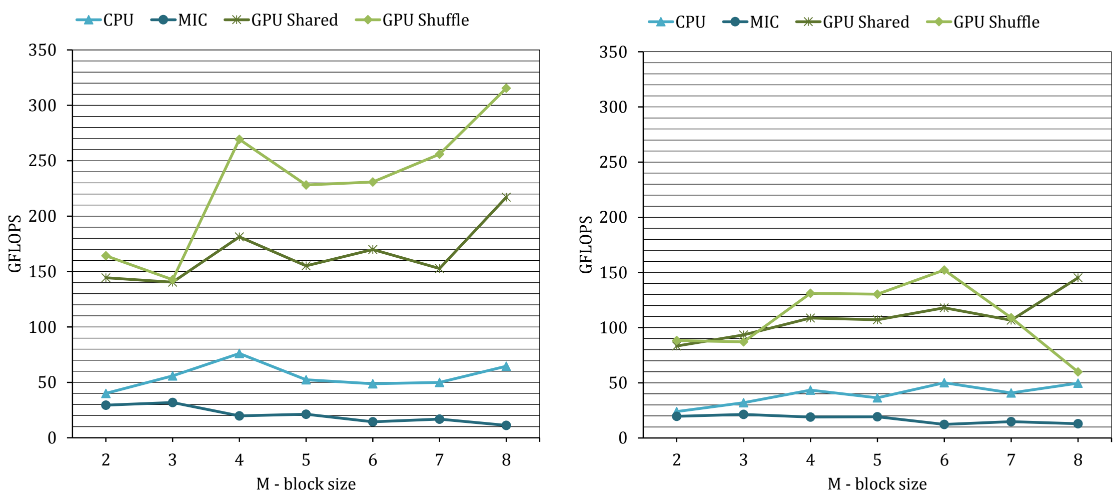


FIGURE 4.6: Single (left) and double (right) precision effective computational throughput when solving block tridiagonal systems with varying M block sizes. GFLOPS is computed based on the amount of floating point arithmetic operations needed to accomplish the task. Addition, multiplication and division is considered as floating point arithmetic operations, ie. FMA (Fused Multiply and Add) is considered as two floating point instructions.

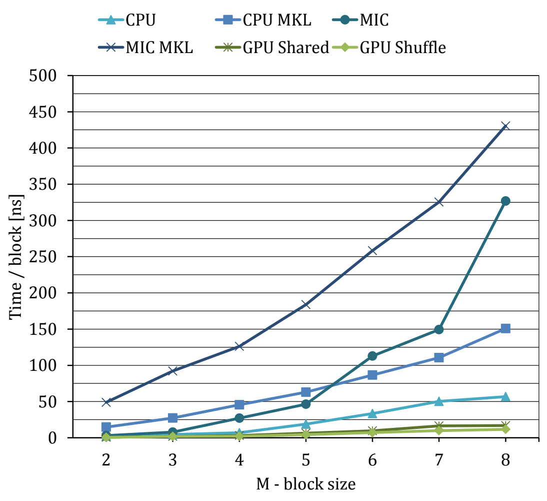
local arrays that supposed to be allocated in registers due to compiler considerations are put into local memory as well. This shows that the presented GPU approaches have very high efficiency for $M = 2, .., 8$ block sizes.

An NVIDIA Tesla C2070 GPU card has been used to compare the results against the PCR solver presented in [54] where an NVIDIA Tesla C2050 was used. The two cards contain identical GPUs with different amount of global memory, see [55]. 3GB is available on the C2050 and 6 GB is available on the C2070. The execution times were measured for the single precision case on block size $M = 4$, on $P = 512$ number of problems each of which is $N = 128$ long. The block PCR based solver completes in 10.5ms and the (shared memory based) block Thomas solver completes in 2.42ms. This is a $\times 4.3$ speed difference for the sake of the Thomas solver. $\times 8.3$ and $\times 9.8$ improvement is achieved if the execution time per block metrics are compared and the number of systems to be solved is increased to $P = 4096$ or $P = 32768$.

The SIMD solution presented performs well on the CPU, but performs poorly on the MIC architecture. On both architectures the compute intensive loops were vectorized as it is reported by the Intel compiler. Both the MKL banded solver and the presented block tridiagonal solver run more efficiently on the CPU.

Figure 4.8 presents the speedup of the block-tridiagonal solvers on GPU over the MKL banded solvers and the CPU and MIC based block tridiagonal solvers. This proves the

DOI:10.15774/PPKE.ITK.2016.002



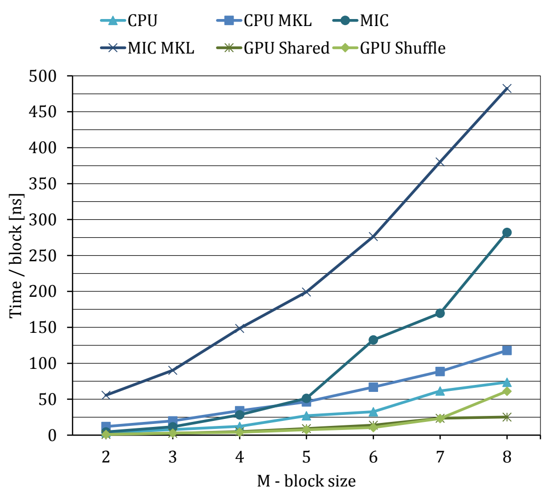
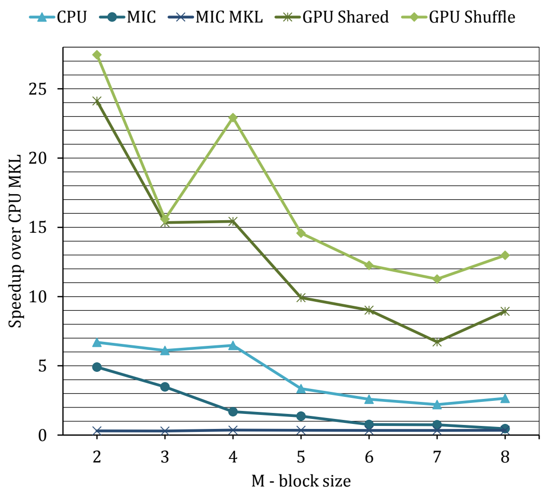


FIGURE 4.7: Single (left) and double (right) precision block-tridiagonal solver execution time per block matrix row for varying block sizes.

benefit of the presented GPU based solutions. Also, the highly efficient CPU and MIC implementations show the benefit of using a block tridiagonal solver over a banded solver for the range of block sizes $M = 2, .., 8$.



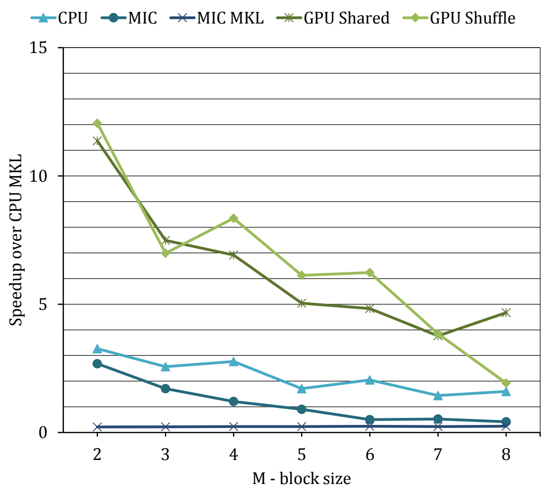


FIGURE 4.8: Single (left) and double (right) precision block tridiagonal solver speedup over the CPU MKL banded solver.

Significant speedup against the CPU MKL banded solver is reached with the GPU-based solver, up to $\times 27$ in single and $\times 12$ in double precision. The multi-threaded CPU code provides $\times 2 - 6$ speedup in single and $\times 1.5 - 3$ speedup in double precision. The multi-threaded MIC performance of the block solver is better than the CPU MKL, but the MKL banded solver perform poorly on the MIC. The efficiency of the CPU relies on the cache and out-of-order execution performance of the architecture. The size of the temporary blocks and arrays of blocks is small enough to fit into the L1 and L2 cache. The out-of-order execution is capable of hiding some of the 4 clock cycle latency

DOI:10.15774/PPKE.ITK.2016.002

of accessing these temporary data structures in the L1 cache. As the MIC lacks the out-of-order execution and the cache size per thread is much smaller the performance is also worse than the CPU. Moreover, the *gbsv()* banded solver of the MKL library does not perform well on the MIC architecture as it shows 3 times lower performance than the CPU MKL version.

The advantage of doing block tridiagonal solve in the range of $M = 2, .., 8$ instead of a banded solve is obvious. It is important to note that, as the block size M increases, the computations involved in performing block-matrix operations make the problem compute limited instead of memory bandwidth limited. The total execution time of computing a system is composed of loading the blocks of data and performing matrix linear algebra on the blocks. The former one scales as $O(NM^2)$ and the latter one as $O(NM^3)$, where N is the system size and M is the block size. As the block size increases the computational part becomes more dominant in the execution time and the memory access time becomes less significant. This can be read from the Figures 4.5 and 4.6 – as M increases, the bandwidth decreases and the GFLOPS increases.

4.7 Conclusion

In the dissertation it has been shown that solving block-tridiagonal systems with $M = 2, .., 8$ block sizes with the Thomas algorithm pays off over the Intel MKL banded solver. It has been shown that the advantage of the block Thomas algorithm is the computational complexity over the CR or PCR algorithm and that parallelism can be increased by exploiting the properties of the block matrix operations. The superior performance of the GPU relies on the low arithmetic complexity of the Thomas algorithm and the efficiency of the parallel block matrix operations allowed by the work sharing and the register blocking capabilities of the GPU threads. Since the work complexity (ie. number of block-matrix operations) of the CR/PCR algorithms are significantly higher than the Thomas algorithm, the CR/PCR has no advantage in block-tridiagonal solvers. Significant speedup is reached with the GPU-based solver with up to $\times 27$ in single and $\times 12$ in double precision. The multi-threaded CPU code provides $\times 2 - 6$ speedup in single and $\times 1.5 - 3$ times speedup in double precision against the MKL banded solver.

DOI:10.15774/PPKE.ITK.2016.002

Chapter 5

Solving the Black-Scholes PDE with FPGA

5.1 Introduction

The aim is to further examine the architectural, programmability and development issues regarding novel CPU, GPU and FPGA architectures in the case of one dimensional finite difference problem like the one-factor Black-Scholes (BS) PDE. The BS PDE is a parabolic type defined with Dirichlet boundary conditions. Therefore the solution of this problem is similar to the solution of the heat equation in one dimension. The problem can be solved using explicit and implicit time marching. Although the explicit solution is programmatically much simpler, it requires significantly more time step computations than the implicit method, due to stability limit of the explicit method. Besides, the implicit method doesn't pose such restrictions on the grid resolution which is defined by the convergence criteria of the explicit method.

The balance of computational speed, programming effort and power efficiency are the key factors that decide where a certain architecture will be used in the engineering and research practice. Therefore, in the current study the following architecture-parallelisation tool combinations were chosen:

- Intel Xeon 2 socket server CPU with Sandy Bridge architecture: algorithms implemented using AVX ISA (Instruction Set Architecture) intrinsics in C/C++.

DOI:10.15774/PPKE.ITK.2016.002

- NVIDIA Tesla K40 GPU with Kepler architecture: algorithms implemented with CUDA C programming language
- Xilinx Virtex-7 FPGA: algorithms implemented with Vivado HLS (High Level Synthesis) C/C++ language.

The way parallelism is executed and implemented on these hardware platforms is usually very diverse. This is especially true in the case of the FPGA where the implementation of parallelism uniquely depends on the problem at hand and the effort of the developer. Although the standard way of FPGA development is through the use of VHDL or Verilog hardware description languages, the development effort with these approaches are not comparable with the C/C++ and CUDA C development efforts, as hardware description languages are more fine-grained. Therefore, Vivado HLS (High Level Synthesis) has been chosen to create a high performing FPGA implementation from C/C++ source code.

The literature on the finite difference solution of the Black Scholes PDE is abundant, but only a few papers provide a thorough comparison of solvers on novel CPU, GPU and FPGA architectures. See [56] for efficient algorithms on CPUs and GPUs, [57] for comparison between GPU and FPGA implementations. FPGA implementations of the explicit solver are studied in [58] and [59], while implicit solution is considered in [60].

5.2 Black-Scholes equation and its numerical solution

The Black-Scholes (BS) PDE for derivative security pricing is a celebrated tool of the Black-Scholes theory [61]. The one-factor BS in the case of European call options is shown on Eq. (5.1).

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} - rV = 0 \tag{5.1}$$

The BS equation is a second order, parabolic, convection-diffusion type PDE sharing common numerical features with the heat equation. Solving these equations with explicit or implicit time-marching is feasible in 1 dimension (one-factor form). In higher dimensions the cost of the implicit solution increases greatly which can be avoided by

DOI:10.15774/PPKE.ITK.2016.002

more sophisticated numerical methods. Such methods are not the scope of the dissertation. Both explicit and implicit solutions iterate in time backwards, since the problem is a final value problem rather than an initial value problem. After explicit (backward time differentiation) discretization of Eq. (5.1) the BS PDE boils down to the algebraic expression shown in Eq. (5.2). Evaluating this expression gives the price curve in the next time-step.

$$V_k^{(n+1)} = a_k V_{k-1}^{(n)} + b_k V_k^{(n)} + c_k V_{k+1}^{(n)} \tag{5.2}$$

with coefficients

$$\begin{aligned} a_k &= \tfrac{1}{2}\sigma^2 k^2 \Delta t - \tfrac{1}{2} r k \Delta t \\ b_k &= 1 - \sigma^2 k^2 \Delta t - r\Delta t \\ c_k &= \tfrac{1}{2}\sigma^2 k^2 \Delta t + \tfrac{1}{2} r k \Delta t \end{aligned}$$

where $(n) = 0, ..., N-1$ superscript is the time coordinate with Δt time step, $k = 0, ..., K-1$ subscript is the price coordinate with ΔS price step (price step is factorized out), σ is the volatility of the underlying (risky) asset, r is the risk-free interest rate.

Eq. (5.3) shows the implicit form of the solution of BS PDE, which requires the solution of a tridiagonal system of equations.

$$a_k V_{k-1}^{(n+1)} + b_k V_k^{(n+1)} + c_k V_{k+1}^{(n+1)} = V_k^{(n)} \tag{5.3}$$

where

$$\begin{aligned} a_k &= -\tfrac{1}{2}\sigma^2 k^2 \Delta t + \tfrac{1}{2} r k \Delta t \\ b_k &= 1 + \sigma^2 k^2 \Delta t + r\Delta t \\ c_k &= -\tfrac{1}{2}\sigma^2 k^2 \Delta t - \tfrac{1}{2} r k \Delta t \end{aligned}$$

Since many options need to be calculated in a real-world scenario, a natural parallelism arises in computing these options in parallel.

DOI:10.15774/PPKE.ITK.2016.002

5.3 Multi and manycore algorithms

The problem of solving the explicit and implicit time-marching is done using stencil operations in the explicit case and using the Thomas algorithm for solving the tridiagonal system of equations arising in the implicit case. The use of stencil operation avoids the explicit construction of a matrix, therefore it is a commonly used matrix-free method for calculating explicit Euler solution of PDEs, see Section 5.4.1 for details. In depth details and comparison of the CPU and GPU implementations can be found in [56].

5.3.1 Stencil operations for explicit time-marching

The one dimensional, stencil-based computations required for the one-factor application can be efficiently implemented on both CPUs and GPUs.

The system specification of the CPU system used in our work consists of a two socket Intel Xeon E5-2690 (Sandy Bridge) 8 core/socket server processors. Each core has 32KB L1 and 256KB L2 cache and all the cores in a socket share a 20MB of L3 shared cache. Each socket has a 51.2GB/s bandwidth to RAM memory and a total 66GB/s bandwidth is measured for the two sockets by the STREAM benchmark [62]. The theoretical maximal floating point computational limit of a single socket is 318 GFLOPS for single and 171 GFLOPS for double precision. The total maximal dissipated power of the two socket is 270 W.

The GPU used in the current comparison is an NVIDIA Tesla K40 card with the GK110b micro architecture. The theoretical maximal bandwidth of the system towards the main memory is 288 GB/s. The maximum bandwidth achieved with the CUDA toolkit's memory bandwidth test program is 229 GB/s. The theoretical maximal floating point computational limit 5.04 TFLOPS for single and 1.68 TFLOPS for double precision. The total maximal dissipated power of the GPU card is 235 W.

CPU implementation

Stencil based computations can be efficiently implemented on CPUs equipped with vector instructions sets such as AVX or AVX2. Due to the efficient and large L1 cache (32KB) these problems on CPUs tend to be compute limited rather than bandwidth

DOI:10.15774/PPKE.ITK.2016.002

limited. On a typical CPU implementation the computations are parallelized over the set of options. Smaller subsets of options are solved using multithreading with OpenMP and options within a subset are solved within the lanes of CPU vectors.

GPU implementation

GPUs, due to the implemented thread level parallelism and compile-time register allocation give more freedom for better vectorization and optimization and therefore the achievable computational efficiency is very high. Among the many possible efficient algorithms discussed in [56] the best performing utilizes the new register shuffle instructions to share the work-load of computing a single timestep. The shuffle instruction (introduced in the Kepler architecture) makes communication possible between the threads inside a warp, ie. shuffle allows data to be passed between lanes (threads) in a vector (warp).

5.3.2 Thomas algorithm for implicit time-marching

The solution of the implicit form of the discretized BS equation requires the solution of a tridiagonal system of equations. The solution of the system is essentially the Gauss-Jordan elimination process detailed in 3.2.1.

CPU implementation

The CPU implementation relies on L1 cache performance and vectorization over options with either auto vectorization or explicit vectorization with instrinsic functions. Due to vectorization, 4 options with double precision or 8 options with single precision are solved in parallel with AVX or AVX2 instructions. The workload of the complete set of options is then parallelised using multithreading with OpenMP.

GPU implementation

Just as in the case of the explicit solver, the GPU solver allows for more optimization. The discussion of these novel, optimized algorithms can be found in [56]. Beside the Thomas and the PCR (Parallel Cyclic Reduction) algorithms, a new hybrid

DOI:10.15774/PPKE.ITK.2016.002

Thomas/PCR algorithm is also introduced in [56]. The efficiency of the latter algorithm relies on the aforementioned register shuffle instruction and compile-time register allocation. The algorithm works on a work-sharing basis in three steps: 1) the system of size N is split into 32 pieces which are partially solved and results in a reduced system of size 64; 2) threads cooperate to solve the 64 size system with PCR; 3) the solution of the reduced system is used to solve the partially computed systems in step 1).

5.4 FPGA implementation with Vivado HLS

For many years there has been no breakthrough for FPGAs in the field of HPC (High Performance Computing). Over the years, research has been carried out to create tools to support development for FPGAs in C language. Recently syntheser tools with OpenCL (Open Computing Language) support appeared for Altera and Xilinx FPGAs. Xilinx in its Vivado HLS (High Level Synthesis) suite started a new software-based syntheser for the C,C++ and System C languages. These tools appeared as a response to the need of faster system realization. The Vivado HLS support for C with its customized support for the Xilinx FPGA is a potentially efficient solution.

5.4.1 Stencil operations for explicit time-marching

Stencil operations are a widely studied area of FPGA algorithms, see [63]. Many signal and image processing algorithms implemented on FPGAs utilize similar solutions to that used in explicit solution of one dimensional PDEs with stencil operations.

One of the key optimizations in FPGA circuit design is the maximization of the utilization of processing elements. This involves careful implementation that allows for pipelining computations to keep the largest possible hardware area busy. In HPC terminology this optimization is similar to cache-blocking, although the problem is not the data movement but rather keeping the system busy.

One way to create such a (systolic) circuitry is to create multiple interleaved processing elements with simple structure to allow low latency and high throughput. Each processing element is capable of handling the computation associated with three neighboring elements $f(u_{k-1}^{(n)}, u_k^{(n)}, u_{k+1}^{(n)})$ within a single timestep n, where f is the stencil operation.

One may stack a number of such processing elements to perform consecutive timestep computations by feeding the result of processor $p = 1$ to $p = 2$ as shown on Figure 5.1. The result $u_{k+1}^{(1)}$ of $f(u_k^{(0)}, u_{k+1}^{(0)}, u_{k+2}^{(0)})$ is fed into the third input of the element processing element $p = 2$. This way the processing elements stay busy until they reach the end of the system. There are a number of warm-up cycles until all the processors get the data on which they can operate. This introduces an insignificant run-up delay if the size of the system to be calculated is larger than the number of the processors. Each processing element needs 2 clock cycles to feed in the necessary 2 elements in their FIFO. For the third clock cycle the third element is also available and the computation can be executed. Since there are P number of processing elements that need to be initialized to perform the lock-step time iteration on a given system, the total delay of the system is $2P$, ie. the Pth processor starts executing a stencil operation after $2 * P$ clock cycles of the start of the simulation.

Larger units, called processors are composed of 80 and 85 processing elements in the case of single and double precision solutions. Resource requirements of these processors are shown in Table 5.1. For single and double precision respectively 3 and 1 such processors can be crammed onto the FPGA used in the study.

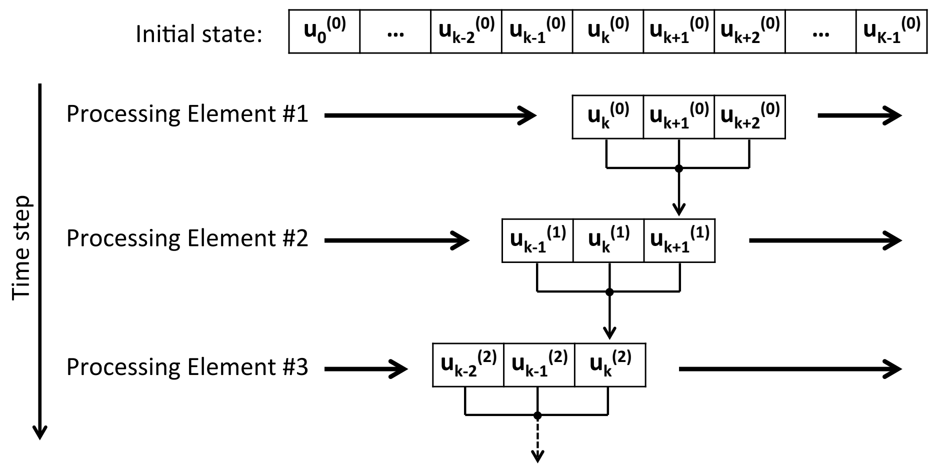


FIGURE 5.1: Stacked FPGA processing elements to pipeline stencil operations in a systolic fashion. The initial system variables $u_0^{(0)}, ..., u_{K-1}^{(0)}$ are swept by the first processing element. The result $u_{k+1}^{(1)}$ is fed into the second processing element.

DOI:10.15774/PPKE.ITK.2016.002

Table 5.1: FPGA Resource statistics for a single processor. The final number of implementable explicit solvers is bounded by the 3600 DSP slices, while the number of implicit solvers is bounded by the 2940 Block RAM Blocks on the Xilinx Virtex-7 XC7V690T processor.

	# BRAM		# DSP slice		Clock [ns]		$\times 10^6$ Ticks	
	SP	DP	SP	DP	SP	DP	SP	DP
Explicit	24	64	1200	3570	4.09	4.01	334	315
Implicit	256	512	37	94	4.26	4.3	14.2	12.6

Note: SP - Single Precision, DP - Double Precision

5.4.2 Thomas algorithm for implicit time-marching

The optimization of the implicit solver relies on the principle of creating independent processors to perform the calculation of independent options. The calculation is based on the Thomas algorithm detailed in Section 3.2.1. Each of these processors is capable of pipelining the computation of more options into the same processor with the associated cost of storing the temporary (c^*, d^*) arrays of each option. c^*, d^* arrays are calculated according to Algorithm 1. The number of options that can be pipelined is defined by the depth of the forward sweep of the Thomas algorithm, which is 67 clock cycles in the present case. The temporary storage is implemented in the available Block RAM memories, but due to the deep pipeline the BRAM memory requirement limits the number of deployable processors.

The current HLS compiler fails to recognize that no data hazard exist between c_i^* and c_{i-1}^* in the two consecutive iterations of the forward pass in the tridiagonal algorithm and therefore refuses to properly pipeline the loop. A secondary temporary array is used to store a copy of the temporary arrays – this changes the data flow and guides the compiler towards pipelining.

A single processor unit for the implicit solver has resource requirements specified on Table 5.1. For single and double precision respectively 11 and 5 of these units can be crammed onto the utilized FPGA.

5.5 Performance comparison

FPGA performance is compared to highly optimized CPU and GPU code. Estimations based on the Xilinx Vivado toolset are made to predict the achievable clock frequency on

DOI:10.15774/PPKE.ITK.2016.002

Table 5.2: Performance - Single Precision

	ps/element			GFLOPS			GFLOPS/W		
	C	G	F	C	G	F	C	G	F
Explicit	15.2	2	17.4	394	3029	344	1.46	12.9	8.6
Implicit	142.7	14.5	766	139	1849	26	0.51	7.9	0.65

Note: C - 2 Xeon CPUs, G - Tesla K40 GPU, F - Virtex-7 FPGA

Table 5.3: Performance - Double Precision

	ps/element			GFLOPS			GFLOPS/W		
	C	G	F	C	G	F	C	G	F
Explicit	29.8	4.1	48.2	201	1463	124	0.74	6.2	3.1
Implicit	358.8	43.5	1748	48	892	9.8	0.18	3.8	0.24

Note: C - 2 Xeon CPUs, G - Tesla K40 GPU, F - Virtex-7 FPGA

a Xilinx Virtex-7 VX690T. Performance metric is calculated on a grid element basis to eliminate the effects of architectural differences, ie. the total execution time is divided by the number of system elements K and total N time steps made. To mimic a real-world scenario the a_k, b_k, c_k coefficients of the explicit and implicit methods are set up in the first phase of the computation and stored in $\mathbf{a}, \mathbf{b}, \mathbf{c}$ arrays for each option independently. Every implementation uses these arrays to perform the computation.

The chosen FPGA is equiped with 108k slices, 3600 DPS slices and 3000 BRAM of size 18Kb. The total maximal power dissipation allowed by the packaging is expected to be less then 40 W.

The results of measurements are present on Table 5.2 and 5.3. Based on the figures it can be stated that the proposed FPGA implementation is slower than the highly optimized CPU implementation, but it is significantly more power efficient. Compared to the GPU version the FPGA is significantly slower and even if the power dissipation of the GPU is higher than the FPGA the overall power efficiency is higher for the sake of the GPU.

5.6 Conclusion

A well studied finite difference problem – the solution of the Black-Scholes PDE – is chosen to compare novel CPU, GPU and FPGA architectures. Efficient FPGA based implementation of the explicit and implicit BS solvers have been created using Vivado

DOI:10.15774/PPKE.ITK.2016.002

HLS. The relatively low programming effort and the achieved efficiency of the resulting circuitry shows a promising step towards the applicability of FPGAs in an HPC environment and more specifically in finite difference calculations. Although the overall performance is not significantly higher than the CPU, the higher power efficiency makes this approach viable in power constrained system and solutions. The results on the other hand clearly show the superior performance of GPUs both in terms of computational efficiency and power efficiency.

DOI:10.15774/PPKE.ITK.2016.002

Chapter 6

GPU based CNN simulator with double buffering

6.1 Introduction

The bare computing power of GPU devices is tremendous compared to the conventional CPUs, see Fig. 1.5. At this point it is worth noting that the memory wall in this computing environment is the most problematic bottleneck. Usually the data is stored in the main memory of the system. A CPU can access this memory through a cache hierarchy (with a maximum 21 GB/s bandwidth), which is quite fast compared to the a GPU that can access this data only through a PCIe (PCI Express) bus (with 4 GB/s aggregated speed through a 16 lane PCIe connector). On the other hand when the data is on the device (GPU) memory it can be accessed by the GPU with 128 GB/s through a 256 bit width GDDR5 memory interface (according to the specification of nVidia GeForce GTX 560 - Asus DCII Top graphics card).

The computing power of these devices is given by the large amount of computing units, the cores. Different GPU manufacturers construct their cores differently. But it is common that each core contains pipelined ALU (Arithmetic Logic Unit) that implements the most essential arithmetic functions. These units have smaller instruction set implemented than CPUs have. The GeForce GTX 560 GPU has 336 CUDA (Compute Unified Device Architecture) cores in it, for which 1.95 billion transistors have been used. An Intel i5 660 CPU has two computing units that are implemented using only 380 million

DOI:10.15774/PPKE.ITK.2016.002

transistors (without the integrated GPU unit). These two units are used as 4 cores by the help of a hardware implemented Hyper-Threading Technology developed by Intel.

6.2 The CNN model

During the analisys the conventional CNN model introduced in [64, 17] is used. Briefly describing the Cellular Neural Network is locally connected recurrent neural network, as shown on Figure 6.1.

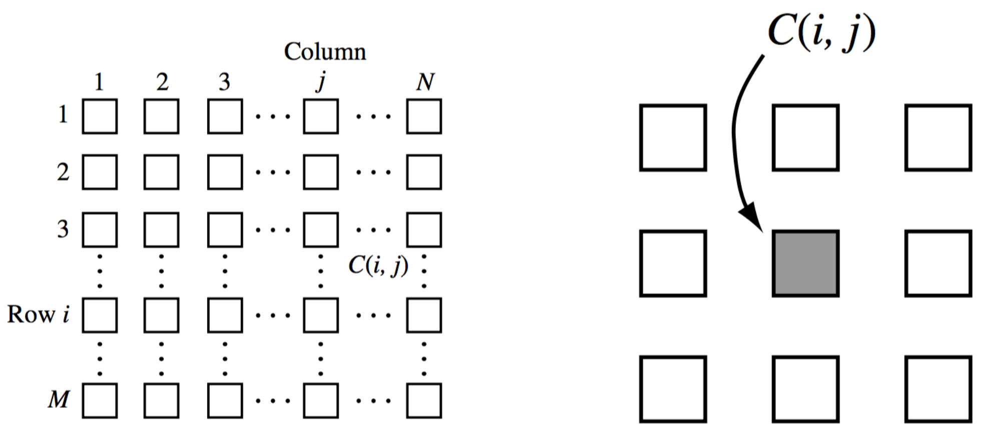


FIGURE 6.1: Left: standard CNN (Cellular Neural Network) architecture. Right: CNN local connectivity to neighbor cells. Source: [64]

The local connectivity is defined by a radius of neighbourhood. The original model is an analogue one defined by the differential equation Eq. (6.1).

$$\dot{x}_{i,j}(t) = -x_{i,j}(t) + \sum_{C(k,l)\in S_r(i,j)} A(i,j,k,l)y_{k,l}(t) + \sum_{C(k,l)\in S_r(i,j)} B(i,j,k,l)u_{k,l}(t) + z_{i,j} \tag{6.1}$$

where $x(t)$ is the state variable, $C(k,l)$ is an element in the $S_r(i,j)$ neighbourhood, $A(i,j,k,l)$ and $B(i,j,k,l)$ are the feed-back and feed-forward templates, $u_{k,l}(t)$ is the input, $z_{i,j}$ is the offset and $y_{k,l}(t)$ is the output that is calculated according to the formula on Eq. (6.2) and shown on Figure 6.2.

$$y_{k,l}(t) = f(x_{k,l}) = 0.5(|x_{k,l}+1| - |x_{k,l}-1|) \tag{6.2}$$

DOI:10.15774/PPKE.ITK.2016.002

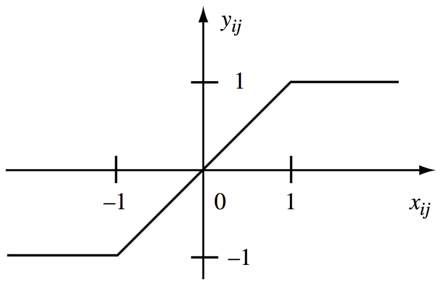


FIGURE 6.2: Piecewise linear output function defined in Eq. (6.2). Source: [64]

The solution of this state equation is approximated using the forward-Euler method shown in Eq. (6.3).

$$x_{i,j}(k+1) = x_{i,j}(k) + h\Bigg(-x_{i,j}(k) + \sum_{C(k,l)\in S_r(i,j)} A(i,j,k,l)y_{k,l}(k) + \sum_{C(k,l)\in S_r(i,j)} B(i,j,k,l)u_{k,l}(k) + z_{i,j}\Bigg) \tag{6.3}$$

If one assumes that the input image is permanent during the solution (i.e. $u_{k,l}(k) = u_{k,l}$) of the state equation, the calculation can be divided into two parts: the feed-forward and the feed-back part. The feed-forward part (Eq. (6.4)) contains all the calculations that has to be done only once, at the beginning of the calculation.

$$g_{i,j} = \sum_{C(k,l)\in S_r(i,j)} B(i,j,k,l)u_{k,l} + z_{i,j} \tag{6.4}$$

The feed-back (Eq. (6.5)) uses the results of the feed-forward part to perform the given number of iterations:

$$x_{i,j}(k+1) = x_{i,j}(k) + h\Bigg(-x_{i,j}(k) + \sum_{C(k,l)\in S_r(i,j)} A(i,j,k,l)y_{k,l}(k) + g_{i,j}\Bigg) \tag{6.5}$$

This way unnecessary calculations can be avoided.

The diffusion template (Eq. (6.6)) with $h = 0.2$ step size has been used to measure the performance of the presented implementations. The result of the template on a

DOI:10.15774/PPKE.ITK.2016.002

grey-scale image is shown in Fig. 6.3.

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} B = \begin{bmatrix} 0.1 & 0.15 & 0.1 \\ 0.15 & 0 & 0.15 \\ 0.1 & 0.15 & 0.1 \end{bmatrix} z = 0 \tag{6.6}$$

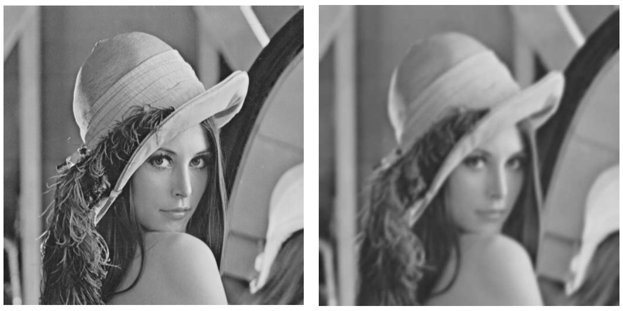


FIGURE 6.3: Example of diffuson template. Original image on the left, diffused image on the right.

6.3 GPU based CNN implementation

Earlier different implementations of CNN using GPU were reported [18, 65, 66]. In this dissertation a different approach is presented.

At first sight the spatial organization of the cores of the GPU makes the architecture the perfect choice for implementation of a CNN. The memory access pattern of the CNN simulation makes this problem a memory-bandwidth bounded problem. Fortunately, the introduced memory caching structure perfectly fits the problem and hides most of the latency and bandwidth limitations.

The following discussion details the use of texture and constant caching methods, see Fig. 6.4 for the code snippets. The array containing the image data is allocated in the main function. A texture array reference is defined in the code as global variable. The reference is bounded to the float array using the *cudaBindTexture2D()* function. In this way the array can be written using global memory access (through L2/L1 cache) and read as read-only memory through the texture cache. After the texture cache is defined it can be read using the *tex2D()* function. Similarly to the previous texture definition, the constant memory definition works in the following way. The array containing the template values is stored in a float array. A global constant memory reference is defined.

DOI:10.15774/PPKE.ITK.2016.002

This reference is bounded to the float array using the *cudaMemcpyToSymbol()* function. After the constant cache is defined it can be used as a conventional array using square brackets.

```
__constant__ float templateA_cf[9];
texture<float,2, ...> state_tex;
...
__global__ void CNNfeedback(...) {
  ...
  temp += activate(tex2D(state_tex, x,y)) * templateA_cf[4];
  ...
}
...
int main() {
  ...
  float *dev_state_f;
  cudaMalloc(&dev_state_f, allocSize);
  cudaBindTexture2D(state_tex, dev_state_f, channel, 512, 512, allocSize);
  ...
  float templateA_f[]={0.1, ... };
  cudaMemcpyToSymbol("templateA_cf", templateA_f, size);
  ...
  CNNfeedback<<<blockD,gridD>>>>(...);
  ...
}
```

FIGURE 6.4: The use of texture and constant caching - code snippets from source code.

Using these techniques the CNN state equation can be solved easily. If the input image considered static, the feed-forward part of the equation has to be calculated only once. This result can be reused in every iteration of the feed-back part of the equation. The equation is solved with the conventional forward (explicit) Euler method discussed earlier.

Two kernels are implemented to solve the state equation. One that calculates the feed-forward part and one that performs 50 iterations.

Every thread that realizes the kernel calculates one pixel of the state equation. Threads are organized in 32x32 sized blocks, and the grid size is calculated according to the size of the image. The block size has been obtained by trial and error method.

One iteration of the state equation is calculated in each kernel invocation. Thus 50+1 (feed-backs+feed-forward) kernel invocations are necessary to perform the calculations. This is an important fact, because kernel invocations are OS dependent. Under Windows

DOI:10.15774/PPKE.ITK.2016.002

each kernel invocation takes $36\mu s$. This value is significantly lower under Linux. The overall time saved by running the entire simulation under Linux is nearly $0.9ms$.

The available computing power is not utilised perfectly. The memory wall problem can not be avoided even with these techniques. Using texture and constant cache the code becomes clear and efficient.

6.4 CPU based CNN implementation

For the sake of comparison an optimized CPU code has been produced. During the tests an Intel i5 600 is used with 2 cores at 3.33 GHz clock rate, 4 MB L3, 256 KB/core L2 and 64 KB/core L1 cache. The latter is subdivided to data and instruction cache, in 32 KB and 32 KB partitions. The processor has hardware support for multi threaded applications, called Hyper-Threading Technology (HTT), which is Intel's implementation of SMT (Simulatanious MultiThreading). This processor is based on the Westmere architecture.

The optimized CNN simulator code exploits the benefits of the SSE4.2 (Streaming SIMD Extension 4, subset 2) instruction set. These instructions are operating on the 128 bit wide specialized registers and 4 single precision floating point operations can be performed in each clock cycle.

For the implementation the Intel Integrated Performance Primitives (IPP) library was used. This library contains highly optimized image and signal processing functions for Intel CPUs. It takes advantage of the SSE 4.2 instruction set and the HTT, thus providing an easy to use yet efficient tool for solving the state equation of CNN.

First, the forward part is calculated using a 2 dimensional convolution with the *ippiConvValid_32f_C1R()* function. Then 50 iterations are performed using the same convolution function and *ippsSub_32f_I*, *ippsMulC_32f_I*, *ippsAdd_32f_I* functions for imagewise subtraction, multiplication and addition. The *for* loop is initiated using multithreading. With the help of HTT and *openmp* the speed of the calculation can be increased significantly. Interprocedural optimization and level 2 optimization is used with the Intel compiler (switches used: *-openmp -O2 -ipo -xSSE4.2*).

DOI:10.15774/PPKE.ITK.2016.002

6.5 Comparison of implementations

Exhaustive comparison of the presented implementations together with the previous FPGA and GPU implementations is detailed in this section. First, the presented GPU and CPU implementation is discussed. Then these results are compared to prior implementations created by different authors.

In order to compare the two devices the theoretical FLOPS (single precision) rates provide a good basis. The test system was a desktop machine with an Intel Core i5 660 processor, 4GB of RAMs, an Intel motherboard and Asus graphics card with nVidia GeForce GTX 560 DCII Top (over-clocked by Asus to 925MHz engine and 1850 MHz shader frequency). The operating sytem was a 64bit Ubuntu 11.10 with kernel version 3, nVidia driver version 285.05.33 and CUDA 4.1 toolkit was used. According to Intel [67] the performance of the processor is 26.664 GFLOPS (at 3.33 GHz) and 29 GFLOPS (at 3.6 GHz Single Core Max Turbo). According to [68] the dual warp scheduler in the Fermi architecture is capable of issuing 2 instructions per clock cycle, thus the total number of instructions issued per second is the product of clock cycles per second and the n number of CUDA cores times the 2 instruction issued per second: $f \times n \times 2 = 1850Hz \times 336cores \times 2 = 1243.2GFLOPS$. However there are only n number of cores, the graphics processor is capable of performing only $f \times n = 1850Hz \times 336cores = 621.6GFLOPS$, i.e. each core can initiate one floating point operation (Addition, Multiplication or FMA - Fused Multiply and Add) per clock cycle [21]. FMA instructions are counted as 2 instructions, therfore theoretical peak performance is $1243.2GFLOPS$. Performance metrics of the different architectures are compared on Table 6.1.

TABLE 6.1: Comparison table

	CPU Intel i5 660	**GPU NVIDIA GTX560**	CPU IBM CELL [69]	GPU NVIDIA 8800GT [18]	FPGA Xilinx XC5VSX240T [19]
Clock rate [MHz]	**3330**	**1850**	3200	1350/1500	550
Cores [pcs]	**2(4threads)**	**336**	8	120/112	117
GFLOPS	**26.664**	**1243.2(FMA)**	25.6	504	-
TDP [W]	**73**	**150**	86	160	25
Cell iter. / s [10^6]	**397**	**4397**	3627	590	64350
Comm. time [ms]	**0**	**1.56**	-	-	4.48
Speedup	**1**	**11.07**	9.13	1.49	162.1

Results of the presented GPU and CPU implementations on 512x512 sized image compared to the previous results [18, 19, 70, 69] are summarized on Table 6.1.

DOI:10.15774/PPKE.ITK.2016.002

Using the presented GPU and CPU implementations 4397 and 397 cell iteration per second was achieved respectively. The results show that the Xilinx Virtex-5 FPGA implementation is the fastest during the solution of the CNN state equation. Taking into account that the first CELL processor was introduced in 2005, this solution has a remarkable performance even nowadays. The presented solution using the GTX560 GPU outperformed the previous GPU implementation reported by Soos et al. [18]. The current implementation is 7.45 times faster than the 8800GT implementation, if the memory transfer time is not considered. This factor is reduced to 5 if the data transfer overhead is taken into account. Partially this is explained by the facts that the number of cores, the frequency, the context switching, the FMA instructions and other minor developments have been introduced in the Fermi architecture. The main difference between the previous and the current implementations is the way state image is accessed and stored. In a previous work [18] the shared memory is used to store parts of the state image, whereas in the present case texture cache has been used for this purpose.

It has to be noted that the conversion of the 8 bit integer pixel values to 32 bit floating point values could be done on GPU after transferring the integer data to the device memory. Thus significant (factor 4) transfer time could be saved, while the conversion could be performed within microseconds.

6.6 Conclusion

A novel GPU implementation of 3x3 template sized CNN simulator together with a CPU implementation has been introduced in this chapter. A comparison with the so far published results has been discussed. The presented GPU implementation is significantly 11.07 times faster than the near optimal CPU implementation and 7.45 times an earlier 8800GT GPU implementation. It is even faster than the CELL processor implementation. The FPGA implementation is the fastest but developing proper circuitry for an FPGA takes much more effort than to prepare a GPU code. The presented GPU solution is simple and effective, thus it is a good choice for simulation of the CNN dynamics.

DOI:10.15774/PPKE.ITK.2016.002

Chapter 7

Unstructured Grid Computations

7.1 Introduction

Scientific and engineering applications require the use of parallel programming models on novel massively parallel processor architecures. Such models and programming languages are for instance CUDA, OpenCL, OpenMP, OpenACC etc., which are well established for programming multi and many-core architectures such as GPU and MIC. It is not obvious what model to use to exploit vector units of these architectures. Therefore, it is important to understand the programming models available for vectorizing codes. Single Instruction Multiple Threads (SIMT) and Single Instruction Multiple Data (SIMD) are closely related but significantly different abstractions when it comes to vectorization.

Unstructured grid applications are a key class of scientific and engineering disciplines. Therefore, understanding how the SIMT and SIMD programming abstraction map to a CPU or a MIC architecture is of high interest in the community. OpenCL is a programming language that supports the SIMT abstraction and high performance compiler from CPU and MIC vendor Intel is available.

OP2 [71] is a domain specific abstraction framework for unstructured grid applications. The reader is suggested to study [72, 73] for the details of parallelization of the OP2 framework. An OpenCL extension for OP2 was developed to benchmark the performance of OpenCL on many-core architectures like the Intel Xeon server CPU and Xeon Phi co-processor. The applications implemented with OP2 are real-world representative

DOI:10.15774/PPKE.ITK.2016.002

CFD codes: 1) Airfoil [74] - two dimensional finite volume simulation code for simulating airflow around an airfoil; 2) Volna [75] - tsunami simulation code where the geometry of the bathymetry is described by a two dimensional grid and water column hight is defined by dependent variables of a PDE above the grid points. These OpenCL implementations are compared with other (non-vectorized) parallelizations previously developed with OP2 for the same application.

Many-core co-processors such as GPUs have required new programming approaches for general purpose software development. The new languages for GPUs stem from the SIMT abstraction. NVIDIA's CUDA programming language has gained widespread popularity and has developed a large software ecosystem. However it is restricted to NVIDIA GPUs only. OpenCL, based on a very similar SIMT model, supports a wider range of hardware, but has not developed an ecosystem comparable to that of CUDA - in part because it struggles with the performance portability of OpenCL codes and the lack of proactive support from some hardware vendors. At the same time, larger vector units are being designed into CPUs, and at the lowest level they require a very explicit SIMD programming abstraction. In a SIMD abstraction, operations are carried out on a packed vector (64-512 bits) of values as opposed to scalar values, and it is more restrictive than the SIMT model. For example in the case of data-driven random memory access patterns that arise in key classes of applications, computations require explicit handling of aligned, unaligned, scatter and gather type accesses in the machine code. In contrast, in the SIMT model this is carried out implicitly by the hardware.

The shared memory parallel programming model for multi-threading has been the dominant model for programming traditional CPU based systems, where task or thread level parallelism is used for multi-threading (OpenMP, POSIX threads). But the emergence of accelerators such as GPUs (e.g. NVIDIA GPUs [22]), the Intel Xeon Phi [76] and similar co-processors (DSPs [77], FPGAs [78]) have complicated the way in which parallel programs are written to utilize these systems. Even traditional x86 based processor cores now have increasingly larger vector units (e.g. AVX), and require special programming techniques in order to get the best performance.

In previous generations of CPUs vectorization of computations was of lesser importance, due to the shorter vector units, however the 256 bit and 512 bit long vectors have become key features in the latest architectures, their utilization is increasingly necessary to

DOI:10.15774/PPKE.ITK.2016.002

achieve high performance. This has drawn the auto-vectorization capabilities of modern compilers into focus; such compilers at best could only vectorize a few classes of applications with regular memory access and computation patterns, such as structured grids or multimedia. While the Xeon Phi is designed as an accelerator with processor cores based on a simpler x86 architecture it has the largest vector lengths currently found on any processor core. However, it requires very explicit programming techniques specific to the hardware to gain maximum performance.

To make efficient use of today's heterogeneous architectures, a number of parallelization strategies have to be employed, often combined with each other to enable execution in different systems utilizing multiple levels of parallelism. It has been shown that very low level assembly implementations of algorithms can deliver performance on the Xeon Phi, usually as part of software packages such as MKL [50] or Linpack [79]. Several studies have been published that discuss porting existing applications to the Phi by relying on higher-level language features, usually compiler auto-vectorization, and have shown success in the fields of structured grid computations [80, 81] molecular dynamics [82, 83] and finance [84]. Most of the computations in these applications were engineered to lend themselves easily to auto-vectorization due to the structure of the underlying problems. Even then, the use of low level vector programming was still required in many cases. Irregular computations have been notoriously hard to vectorize, due to dependencies driven by data [85].

The focus of this chapter is to present research into gaining higher performance through OpenCL-based vectorization on CPUs and the Xeon Phi and using GPS minimal matrix bandwidth reordering to improve mini-partitioning in real-world applications. The domain of unstructured mesh based applications are targeted, a key class of applications that have very irregular access patterns. The OP2 [71] domain specific abstraction framework was used to develop specific vectorizing implementations. OP2 is an "active" library framework for the solution of unstructured mesh applications.

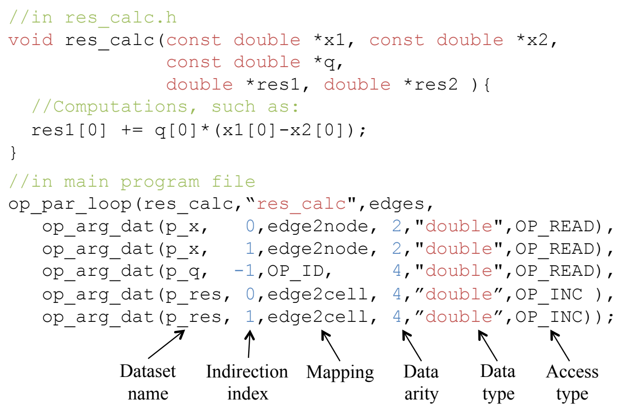
7.2 The OP2 Library

Recently, domain specific languages (DSLs) and similar high-level abstraction frameworks have been utilized to reduce the complexity of programming parallel applications.

DOI:10.15774/PPKE.ITK.2016.002

With DSLs, the application programmer describes the problem at a higher level and leaves the details of the implementation to the library developer. Given the right abstraction, this enables high productivity, easy maintenance and code longevity for the domain scientist, permitting them to focus on the problem at hand. At the same time, it enables the library developers to apply radical and platform-specific optimizations that help deliver near-optimal performance.

OP2 is such a high-level framework for the solution of unstructured mesh applications [71]. Its abstraction involves breaking up the problem into four distinct parts: (1) sets, such as vertices or edges, (2) data on sets, such as coordinates and flow variables, (3) connectivity between sets and (4) operations over sets. These form the OP2 API that can be used to fully and abstractly define any unstructured mesh. Unstructured grid algorithms tend to iterate over different sets, accessing and modifying data indirectly on other sets via mappings; for example flux computations often loop over edges accessing data on edges and neighboring cells, updating flow variables indirectly on these cells. In a parallel setting this leads to data races, the efficient handling of which is paramount for high performance.



```
//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));
```

FIGURE 7.1: Demonstrative example of the OP2 API in Airfoil: parallel execution of the `res_calc` kernel function by the `op_par_loop` for each elements in the `edges` set.

The OP2 abstraction is designed to implicitly describe parallelism; the basic assumption is that the order in which elements are processed does not affect the final result, to within the limits of finite precision floating point arithmetic. This allows for parallelization of the execution over different set elements, however, potential data races have to be recognized and dealt with. Therefore the API is designed to explicitly express access types and patterns, based on the following building blocks:

1. `op_set`: basic components of the mesh, such as edges and cells

DOI:10.15774/PPKE.ITK.2016.002

2. `op_dat`: data on each element of a set, with a given arity (number of components)

3. `op_map`: connectivity from one set to an other, with a given arity, e.g. each edge connects to to vertices

4. `op_par_loop`: a parallel loop over a given set, executing an elementary kernel function on each element of the set passing in pointers to data based on arguments described as follows:
   `op_arg_dat(op_dat,idx,op_map,dim,"typ",access)`,
   where a given dataset with `dim` arity and `type` datatype is to be accessed through a specific index in a mapping to another set (or no mapping if it is defined on the same dataset), describing the type of access, which can be either read, write, increment or read-write.

This API allows OP2 to use a combination of code generation and run-time execution planning in order to support a wide range of contrasting hardware platforms, using a number of parallel execution models. For multi-threading on a pre-processing step is used to split up the computational domain into mini-partitions, or blocks, that are colored based on potential data races [86]. See Figure 7.2 for a demonstrative example on block coloring. Blocks of the same color can then be executed by different threads concurrently without any need for synchronization. The same technique is used to assign work to OpenMP threads, CUDA thread blocks or OpenCL work-groups.

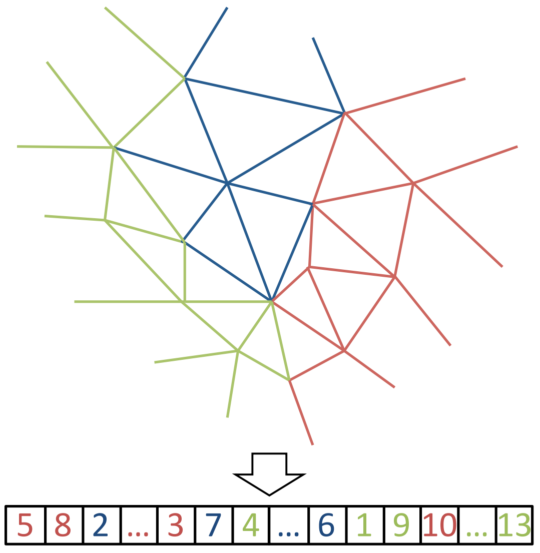


FIGURE 7.2: Demonstrative example of block coloring: edges are partitioned into three blocks colors. Indices of edges belonging to different colors are shuffled in the linear memory.

DOI:10.15774/PPKE.ITK.2016.002

7.3 Mini-partitioning and coloring

In unstructured grid computations such as CFD simulations the low level parallelization of computation is a critical issue to improving performance and especially important to improving parallel scalability of the simulation. The OP2 [71] framework uses a naïve coloring method (see Section 3.3 of [73] for details) to resolve conflicting computational regions in the mesh by identify potential WAR (Write After Read) hazards. Using graph coloring the mesh is partitioned into independent set of elements which are executed independently and therefore the WAR operations can be performed safely.

In addition to the two level coloring in OP2 blocks (in [73] it is also called mini-partition) are also created. These blocks represent a coarser level of parallelism and they are colored in the second level of the coloring mechanism. The blocks are created by grouping a pre-defined number of consecutive elements. In the example on Figure 7.3 the block size is 20, which creates blocks with elements 0-19, 20-39, 40-59 etc. As the mesh elements are connected, blocks have neigboring elements in other blocks. In the case of an improperly ordered mesh these connections establish too many connections between blocks which lead to high number of colors in the the second level of coloring and results in high number of block colors. Therefore, meshes are reordered using a minimum bandwidth reordering such as the Reverse Cuthill-McKee [87, 88] or the GPS [89, 88] matrix bandwidth minimization algorithms. The reordered mesh lends itself to mini-partitioning with better locality and less connections to neighboring blocks, see Figure 7.3. This has a significant positive direct effect on block coloring as it decreases the number of block colors. Moreover, Figure 7.3 also shows, that if the block size is larger than the bandwidth of the adjacency matrix describing the connectivity of the mesh, than two colors are sufficient to color the blocks. The bandwidth B of a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is defined as $B = 2k + 1$, where $k = \max\{|i - j| : a_{ij} \neq 0\}$. This criteria provides extremely good parallel scalability for such applications, that allow block sizes in the order of the bandwidth of the adjacency matrix.

7.3.1 Mesh reordering in Volna OP2

As part of an unpublished research the Volna [75] tsunami simulation code was ported to the OP2 framework. Volna is a representative simulation code for simulating real-world

DOI:10.15774/PPKE.ITK.2016.002

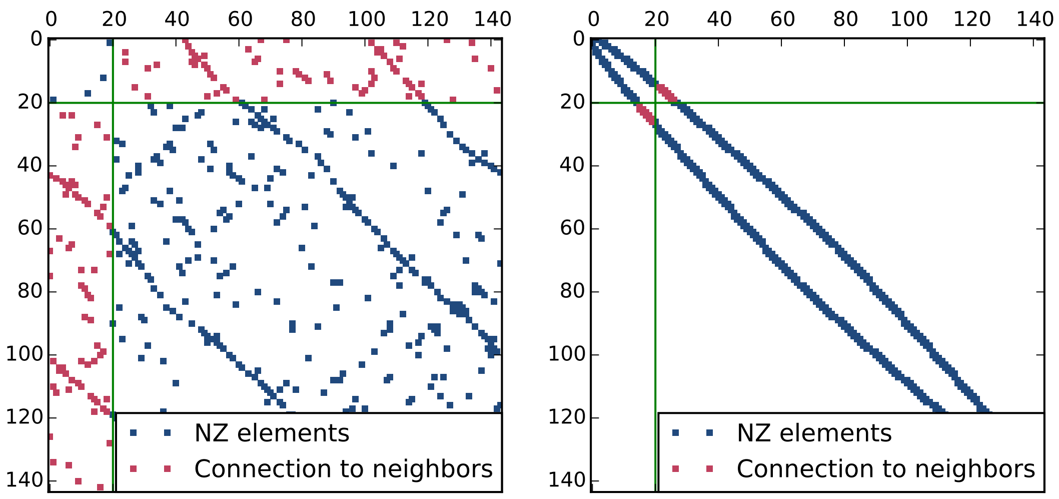
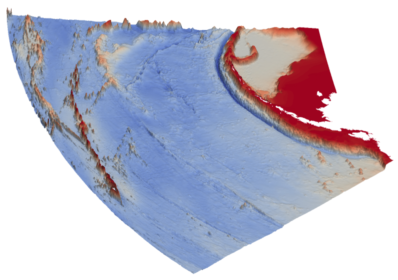
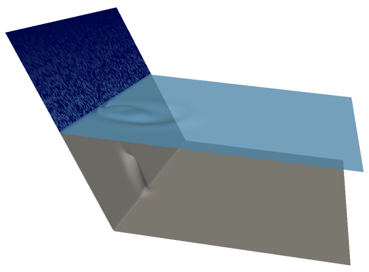


Figure 7.3: Example of sparse symmetric matrix or adjacency matrix representing an unstructured mesh (left) and the same matrix after matrix bandwidth minimization reordering (right). Elements connecting the vertices of the first block (elements 0-19) of the mesh with other blocks of the mesh (20-39, 40-59, etc.) are denoted with red color. On the left matrix the first block of the mesh is connected to other blocks. On the right matrix the first block is only connected to the second block.

tsunami wave propagation and inundation to existing geographical terrain. The simulation uses geometries, meshes and initialization data from measurements and therefore it is a good representative case to demonstrate the effectivity of GPS reordering for improved mini-partitioning and coloring on real-world unstructured grid applications. See Figure 7.4a for a representative test case and Figure 7.4b for a validation test case.



(a) Real-world bathymetry for the simulation of a potential tsunami near Rat Islands, Alaska, US. Blue - deep ocean floor, red - shallow ocean floor.



(b) Artificial bathymetry for studying tsunami initiated by a gaussian-shaped landslide in the bathymetry. Grey - ocean floor, light blue - water level, dark blue - shore.

Figure 7.4: Demonstrative examples of the Volna OP2 test cases.

In the Volna OP2 code the Scotch [90] implementation of the GPS bandwidth minimization algorithm is used and the reordering is done based on the adjacency matrix of cells created from the cells-to-cells, cells-to-edges or cells-to-nodes connectivity data defined by the Volna meshes. The cell adjacency matrix created from the:

DOI:10.15774/PPKE.ITK.2016.002

- *cells-to-cells* mapping considers two cells adjacent if they are indeed next to each other;
- *cells-to-edges* mapping considers two cells adjacent if they have common edges;
- *cells-to-nodes* mapping considers two cells adjacent if they have common nodes.

Cells were chosen as the set for reordering since every time consuming computation in the finite volume numerical method is based data associated to cells, edges bounding the cell or nodes defining the cell geometry. As a consequence all mesh components such as the cells, edges and nodes as well as their mapping and associated data need to be separately reordered. See Figure 7.5 for the details.

```
// Stage 1: Reorder cells and associated data
// Obtain new permutation (GPS ordering) of cells based on cells-to-cells connectivity
// Invers permutation of cell reordering is also retrieved
op_get_permutation(&cells_perm, &cells_iperm, cellsToCells, cells);
// Reorder op_maps according to direct or inverse permutation
op_reorder_map(cellsToCells, cells_perm,  cells_iperm, cells);
op_reorder_map(cellsToNodes, cells_perm,  cells_iperm, cells);
op_reorder_map(cellsToEdges, cells_perm,  cells_iperm, cells);
op_reorder_map(edgesToCells, cells_perm,  cells_iperm, cells);
// Reorder op_dats according to inverse permutation
op_reorder_dat(cellCenters,  cells_iperm, cells);
op_reorder_dat(cellVolumes,  cells_iperm, cells);
op_reorder_dat(values,       cells_iperm, cells);
op_reorder_dat(initial_z,    cells_iperm, cells);
op_reorder_dat(initEtadat,   cells_iperm, cells);

// Stage 2: Reorder edges and associated data
// Obtain new permutation (GPS ordering) of edges based on cells-to-edges connectivity
// Invers permutation of edge reordering is also retrieved
op_get_permutation(&edges_perm, &edges_iperm, cellsToEdges, edges);
// Reorder op_maps according to direct or inverse permutation
op_reorder_map(cellsToEdges, edges_perm, edges_iperm, edges);
op_reorder_map(edgesToCells, edges_perm, edges_iperm, edges);
// Reorder op_dats according to inverse permutation
op_reorder_dat(edgeNormals, edges_iperm, edges);
op_reorder_dat(edgeLength,  edges_iperm, edges);
op_reorder_dat(isBoundary,  edges_iperm, edges);
op_reorder_dat(edgeCenters, edges_iperm, edges);

// Stage 3: Reorder nodes and associated data
// Obtain new permutation (GPS ordering) of nodes based on cells-to-nodes connectivity
// Invers permutation of node reordering is also retrieved
op_get_permutation(&nodes_perm, &nodes_iperm, cellsToNodes, nodes);
// Reorder op_maps according to direct or inverse permutation
op_reorder_map(cellsToNodes, nodes_perm,  nodes_iperm, nodes);
// Reorder op_dats according to inverse permutation
op_reorder_dat(nodeCoords,   nodes_iperm, nodes);
```

FIGURE 7.5: Reordering the mesh and associated data of Volna OP2 based on GPS reordering. Mesh component such as the cells, edges and nodes as well as their associated data are separately reorder. **op_get_permutation** internally creates an adjacency matrix of cells based on the cells-to-cells, cells-to-edges or cells-to-nodes connectivity.

DOI:10.15774/PPKE.ITK.2016.002

7.3.2 Benchmark

The effect of mesh reordering on the block colors and the execution time is benchmarked in this section. Three representative test cases were chosen for benchmarking the performance:

1. *bump*: The seashore geometry is based on the Molène archipelago (ie. island group) in Brittany, France. The simulation start with a cylinder shaped elevated column above the resting sea level. As the water column spreads it initiates tsunami waves.
2. *gaussian*: A theoretical benchmark setup where a gaussian-shaped landslide in the bathymetry initiates the tsunami wave, see Figure 7.4b.
3. *catalina*: The mesh and bathymetry data is based on the geometry of Santa Catalina Island, California, US.
4. *matane*: The mesh and bathymetry data is based on the geometry of Matane, Quebec, Canada.
5. *rat*: The mesh and bathymetry data is based on the geometry of Rat Islands, Alaska, US, see Figure 7.4a. The tsunami is initiated with the elevation of the water level according to a gaussian shape.

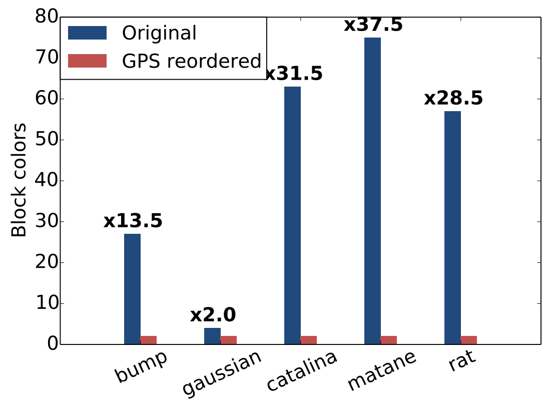
Details on the size of the mesh and the bandwidth of the adjacency matrix created from *cell-to-cell* mapping is shown on Table 7.1. Simulations are executed on a dual socket Intel Xeon server processor using the OpenMP backend of OP2. The OpenMP threads were pinned to the CPU cores using the `KMP_AFFINITY=compact,0` environment variable to avoid the caching and NUMA effects of thread relocation among CPU cores. The block size was chosen to be 2048, which allows enough parallelism and cache reuse on the CPU for efficient execution for these mesh sizes. The particular system used for the benchmark is detailed in Appendix A.3.

TABLE 7.1: Details of the representative tsunami test cases. BW - Bandwidth

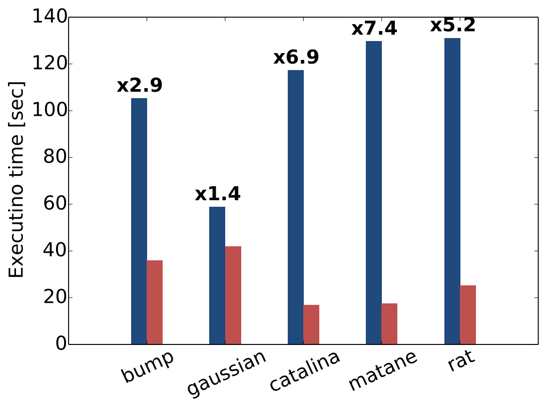
Name	Location	No. nodes	No. cells	No. edges	Orig. BW	BW after GPS
bump	Molène archipelago, France	12828	23873	36709	46877	467
gaussian	Gaussian landslide	1002001	2000000	3002000	3999	2001
catalina	Catalina Island, CA, US	49456	98198	147653	196295	723
matane	Matane, Quebec, Canada	117555	232992	350547	465935	1149
rat	Rat Islands, AK, US	87946	171289	259254	342531	1329

DOI:10.15774/PPKE.ITK.2016.002

Figure 7.6 shows the effect of reordering the mesh based on the connectivity data. The original Volna code used single precision arithmetics, therefore the simulations are only performed for single precision. One may note that the predefined block size of 2048 is sufficient to color the mesh with only two colors for all five test cases. As the *gaussian* example is an artificially generated mesh, even the initial block coloring was significantly better that in the rest of the test cases, which is also true for the execution time. One may note that the overall speedup only due to the mesh reordering is significant, as it is shown on Figure 7.6b. For real-world geometries $\times 2.9 - 7.4$ speedup is reached.



(A) The effect of mesh reordering on the block colors is significant.



(B) The effect of mesh reordering on the execution time is significant.

FIGURE 7.6: The effect of mesh reordering on the block colors and execution time.

7.4 Vectorization with OpenCL

Recently, many-core processors and co-processors gained widespread popularity. These consist of a large number of low power, low frequency compute cores and rely on high throughput to achieve performance by allowing to execute a massive number of threads in parallel. This is in contrast to speeding up execution of a few threads on traditional CPUs. For many-core processors and co-processors a popular programming model is the Single Instruction Multiple Thread (SIMT) model, where a number of lightweight threads execute the same instructions at the same time. From a programming perspective, one implements code for a single thread, where data or control flow usually depends on a thread index, and then this code is executed by different threads concurrently. While it is possible for different threads to have divergent flow paths, in these cases the execution is serialized because there is only a single instruction being executed at the same time; threads not active in the currently executed branch are deactivated

DOI:10.15774/PPKE.ITK.2016.002

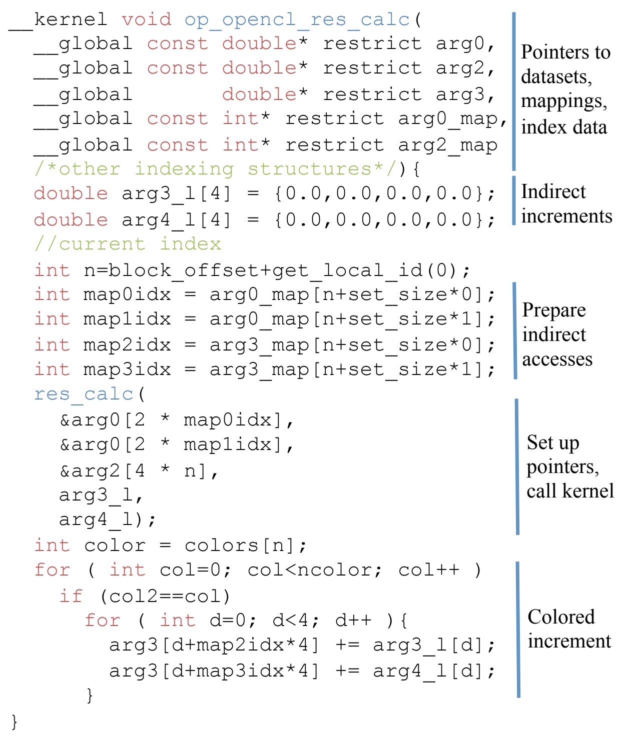
by the hardware. Threads may also access different memory addresses when executing the same instruction; the hardware is responsible for collecting data and feeding it to the threads. The relative position of these memory addresses between adjacent threads has an important impact on performance: addresses packed next to each other may be accessed with a single memory transaction, while gather and scatter type accesses may require multiple transactions. This, combined with the comparatively small amount of cache per thread often has a non-trivial effect on performance that is difficult to predict. CUDA and OpenCL are based on the SIMT model, and the latter maps to both CPU vector units and GPUs.

Finally, the Single Instruction Multiple Data (SIMD) execution and programming model is used by the vector units on Intel CPUs and the Xeon Phi. While some programming models (such as OpenCL) do support a SIMT programming model and compilation for these architectures, the hardware and therefore the generated assembly code has to use SIMD. Vector instructions operate on vector registers that are 256 bits (AVX) or 512 bits (IMCI) long; they can contain 8 or 16 integers/floats, or 4 or 8 doubles. There is also support for masked instructions, where specific vector lanes can be excluded from an instruction, thereby facilitating branching. This execution model implies that data has to be explicitly packed into vector registers that can then be passed as arguments to an operation. Explicit management of data movement requires differentiation between loading a contiguous chunk of data from memory that is (1) aligned, (2) not aligned to the vector length, or (3) that has to be collected from different addresses and packed; the same patterns apply for store operations.

While many structured grid computations are amenable to pragma-driven compiler auto-parallelization, most unstructured grid computations require a lot of additional effort to parallelize. Developing and maintaining large-scale codes with multiple different combinations of parallel programming approaches in order to support today's diverse hardware landscape is clearly not feasible; doing so would reduce the productivity of application developers and would also require intimate knowledge of different hardware.

When using a SIMT programming model, adjacent set elements are assigned to adjacent threads, gather operations and potential branching is automatically handled by the programming model and the hardware, and in case of indirect writes data access is serialized based on the second level of coloring and block-level synchronization constructs.

DOI:10.15774/PPKE.ITK.2016.002



```
__kernel void op_opencl_res_calc(
  __global const double* restrict arg0,
  __global const double* restrict arg2,
  __global       double* restrict arg3,
  __global const int* restrict arg0_map,
  __global 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] += arg3_l[d];
        arg3[d+map3idx*4] += arg4_l[d];
      }
}
```

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

Figure 7.7: Simplified example of code generated for the OpenCL vectorized backend.

OpenCL is an open standard language based on the SIMT abstraction, targeting a wide variety of heterogeneous computing platforms, including CPUs, GPUs, DSPs, FPGAs etc. Since OpenCL is a standard, the implementation of the driver and the runtime environment relies on the support of the hardware manufacturers. Following the SIMT abstraction, the workload is broken up into a number of work-groups, each consisting of a number of work-items. The exact mapping of these is influenced by the parallel capabilities of the device and the possible optimizations that can be carried out by the compiler or synthesizer. As with most portable programming abstractions, performance portability is an issue, therefore we first have to understand how it maps to the target hardware.

The OpenCL abstraction fully describes any concurrency issues that may arise, therefore vectorization of work-items would always be valid. However similar to the case of compiler auto-vectorization of conventional C/C++ code, this can be prevented by a number of factors: control flow branching, irregular memory access patterns etc. Since Intel's IMCI instruction set is more extensive than AVX it gives more flexibility for the compiler to produce vector code.

Spreading the work-load on CPUs is done by mapping work-groups onto CPU hardware

DOI:10.15774/PPKE.ITK.2016.002

```
// Divide for loop of work-groups across CPU hardware threads equivalent to
// #pragma omp parallel for
for(wg=0; wg < get_num_groups(2)*get_num_groups(1)*get_num_groups(0); wg++)
  for(k=0; k < get_local_size(2); k++)
    for(j=0; j < get_local_size(1); j++)
      // Run vectorized for loop equivalent to
      // #pragma simd
      for(i=0; i < get_local_size(0); i+=SIMD_LENGTH)
        // Execute work-item
        kernel(...);
```

FIGURE 7.8: Pseudo-code for OpenCL work distribution and vectorization.

threads. Each work-group then is executed sequentially by mapping the 3D work-group to three nested loops over the length in the different dimensions. If possible, the inner-most loop is then vectorized. A pseudo-code on Figure 7.8 explains the implementation in more details.

Task parallelism at the level of the work-groups in Intel OpenCL is implemented using Intel's TBB (Thread Building Blocks) library [91]. Although TBB offers an efficient solution for multi-threading, the overall overhead of scheduling work-groups is larger in OpenCL than that of static OpenMP parallel loops; however this keeps improving. Since CPUs - as opposed to GPUs - don't have dedicated local and private memory units, using these memory spaces introduces unnecessary data movement and may lead to serious performance degradation. Thus in the OpenCL based CPU implementation OP2 does not use local memory. OpenCL distinguishes between global, local and private address spaces.

A key observation when optimizing code for the CPU is that work-groups are executed sequentially. One can be certain that there are no data-races between work-items in a work-group if the compiled code is scalar. Even if the code is implicitly vectorized the bundled work-items execute in lock-step; synchronization between them is implicit, and these bundles are still executed sequentially. Recognizing this, it is possible to remove work-group level synchronizations which would otherwise be very expensive. This of course is a platform-specific optimization that violates the OpenCL programming model, but it is necessary to obtain good performance. It is possible to remove barriers from the following operations:

- Sharing information specific to a block: since every work-group processes one block (mini partition) of the whole domain, some information about the block data structure can be shared amongst the work-items of the work-group. Either

DOI:10.15774/PPKE.ITK.2016.002

this data is read by one work-item and shared with other work-items through a local variable or every work-item reads the same data and then no local memory is necessary. The barrier is not necessary in the first case, because work-item 0 reads the data and stores it in local memory, and it will be available to every other work-item, since work-item 0 is executed before any other work-item in the group, due to the sequential execution model.

- Reductions: doing a reduction is straightforward due to the sequential execution even in the case of vectorized kernels, first the reduction is carried out on vectors and at the end values of the accumulator vector are added up.
- Indirect increments (or indirect scatter): use of barriers is not needed here either, since 1) the work-group is executed sequentially and 2) even if the kernel is vectorized, a sequential colored increment is used to handle WAR (Write After Read) conflicts.

Figure 7.7 shows a simplified example of the code generated for the loop in Figure 7.1, the "host-side" setup of parameters and the kernel launch are omitted for brevity. This code is very similar to the CUDA code generated for GPU execution, except for CPU-specific optimizations as discussed above.

7.5 Performance analysis

7.5.1 Experimental setup

The goal is to benchmark the performance characteristics of OpenCL on a multi-core Xeon server CPU and a many-core Xeon Phi (also known as Intel MIC) coprocessor [76], both relative to each other and in absolute terms, calculating computational performance and memory throughput. Details of test hardware can be found in Table A.5. In addition to the theoretical performance figures, as advertised by the vendors results of standard benchmarks (STREAM for memory bandwidth and generic matrix-matrix multiply, GEMM for arithmetic throughput) are shown. These figures serve to illustrate how much of the theoretical peak can be realistically achieved. It is clear that achieving anything close to the theoretical peak - especially on accelerators - is quite difficult, and often requires low-level optimizations and parameter tuning. The FLOP/byte metric of processors is depicted in Table A.5 in Appendix A.4, which is essentially a metric for the

DOI:10.15774/PPKE.ITK.2016.002

Table 7.2: Properties of Airfoil and Volna kernels; number of floating point operations and numbers transfers

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

balance of computational throughput and memory bandwidth on different architectures. A higher figure means that the architecture is more suited for computationally expensive applications, while a smaller figure means that the architecture is more suited for memory intensive applications. This is especially important for generally bandwidth-bound applications, such as the ones being investigated, because if their FLOP/byte ratio is much lower than that of the hardware, then a lot of computational power will be wasted.

For benchmarking two applications are chosen, both implemented in OP2; the first is Airfoil, a non-linear 2D inviscid airfoil code that uses an unstructured grid [74], implemented in both single and double precision, and the second is Volna, a shallow-water tsunami simulation code [75], implemented only in single precision. Both simulations use a finite volume numerical algorithm, which is generally considered bandwidth-bound. Table 7.2 details the communication and computational requirements of individual parallel loops in Airfoil and Volna, in terms of useful data transferred (ignoring e.g. mapping tables) as number of floating-point values and useful computations (ignoring e.g. indexing arithmetic) for each grid point during execution. Transcendental operations (sin, cos, exp, sqrt) are counted as one, these are present in `adt_calc` and `compute_flux`. Note, that these figures do not account for caching, therefore, in case of indirectly accessed data, off-chip transfers are reduced by data reuse, and the FLOP to byte ratio goes up.

Comparing the ratio of floating point operations per bytes transferred to those in Table A.5 it is clear that most of these kernels are theoretically bandwidth-bound, however, we have to account for the fact that several kernels may not be auto-vectorizing, therefore, for a fair comparison, the FLOP/byte ratios of CPU architectures have to be divided

DOI:10.15774/PPKE.ITK.2016.002

by the vector length (4 in double and 8 in single precision with 256 bit vectors). This pushes several kernels much closer to being compute-limited, which suggests that by applying vectorization, the code can be potentially further accelerated to the point where its entirely bandwidth-limited, eliminating any computational bottlenecks. In the following analysis the primary focus is on the achieved bandwidth, calculated based on the minimal (useful) amount of data moved; this assumes an infinite cache size for the duration of a single loop, which is of course unrealistic, but it gives a good idea of the efficiency of execution.

Table 7.3: Test mesh sizes and memory footprint in double(single) precision

Mesh	cells	nodes	edges	memory
Airfoil small	720000	721801	1438600	94(47) MB
Airfoil large	2880000	2883601	5757200	373(186) MB
Volna	2392352	1197384	3589735	n/a(355) MB

For Airfoil, performance is evaluated on two problem sizes; a 720k cell mesh and its quadrupled version, a 2.8M cell mesh, to investigate the sensitivity of the hardware to load-balancing issues. For Volna, a real-world mesh with 2.5M cells is used, 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. Mesh sizes and memory footprints are detailed in Table 7.3. These meshes are halved for OpenCL benchmarks on CPU, because execution is restricted to a single socket due to limitations in the runtime discussed in Section 7.5.2.

7.5.2 OpenCL performance on CPU and the Xeon Phi

Due to a limitation on the tested Intel CPUs, neither AMD's nor Intel's OpenCL 1.2 driver (both support compilation for Intel CPUs) is currently able to select - using the device fission feature - a subset of processor cores to execute on a single NUMA socket. Since a fully operational MPI+OpenCL back-end is not yet available, the presented benchmark is limited to single socket performance comparisons. Scheduling threads to a single socket is enforced by the *numactl* utility. Based on the first touch memory allocation policy in the Linux kernel, it is certain that the master thread and the child threads - placed on the same socket - get memory allocated to the same NUMA memory region.

DOI:10.15774/PPKE.ITK.2016.002

In the presented performance measurements the one-time cost of run-time compilation is not counted. Only the time spent on effective computation and synchronization is shown. Figure 7.9 shows that OpenCL execution time in the CPU case is close to the plain OpenMP time. As opposed to conventional auto-vectorization, where segments of a code can be vectorized, OpenCL either vectorizes a whole kernel or none of it. Even though `adt_calc`, `bres_calc`, `compute_flux` and `numerical_flux` kernels are vectorized, the overall performance of the OpenCL implementation is not significantly better.

The Intel Offline Compiler [91] has been used to test whether kernels have been vectorized or not. Results are shown in Table 7.5. The extended capabilities of the instruction set on the Xeon Phi, including the gather and scatter instructions, allow the compiler to vectorize more complex code. The AVX instruction set is more restrictive and although the compiler could produce vector code, it refuses to do so if the heuristics predict worse performance. Even though the Intel OpenCL compiler can handle some branching in the control flow, optimization decisions may override these.

Table 7.4: 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 the K40 GPU

Kernel	MPI CPU		
	Time	BW	Comp
save_soln	4	46	3.2
adt_calc	24.6	13	14.6
res_calc	25.2	27	32
bres_calc	0.09	29	12
update	14.05	56	8
RK_1	3.24	53	4
RK_2	2.88	59	5
compute_flux	23.34	14	42
numerical_flux	4.68	29	4
space_disc	16.86	21	9

The kernel level breakdown of OpenCL in Table 7.5 shows that the largest difference between the OpenMP and implicitly vectorized OpenCL comes from the `adt_calc` and `res_calc` kernels in Airfoil and from the `compute_flux` and `numerical_flux` in Volna. Even though `adt_calc` is vectorized by OpenCL, and is indeed faster than the non-vectorized version shown in Table 7.4. The same is true for `compute_flux`. In the case of `numerical_flux`, the kernel was vectorized but the performance degraded. On the other hand `space_disc` was not vectorized by OpenCL but the performance still increased.

DOI:10.15774/PPKE.ITK.2016.002

The OpenCL performance on the Xeon processors is satisfying compared to the non-vectorized OpenMP performance and even better in the case of the Xeon Phi coprocessor. The results achieved with the coprocessor are shown on Figure 7.10. Thus OpenCL, as an environment implementing SIMT programming model that is amenable to vectorized execution, does deliver some performance improvement over previous non-vectorized implementations, but profiling shows that significant performance is lost due to scheduling. Furthermore, in comparison to results in the next section, we see that while some vectorization is achieved, especially on the Xeon Phi, it is currently not capable of creating efficient vector code.

TABLE 7.5: Timing and bandwidth breakdowns for the Airfoil benchmarks in double(single) precision on the 2.8M cell mesh and Volna using the OpenCL backend on a single socket of CPU and Xeon Phi. Also, kernels with implicit OpenCL vectorization are marked in the right columns.

Kernel	CPU		Xeon Phi		OpenCL vec	
	Time	BW	Time	BW	CPU	Phi
save_soln	4.15(2.18)	44(42)	2.6(1.57)	71(59)	–	✔
adt_calc	18.27(13.23)	17.7(12.2)	12.1(7.2)	27(22)	✔	✔
res_calc	31.43(29.91)	22(11.6)	46(29.76)	15(12)	–	✔
update	14.65(7.34)	53.5(53.4)	12(6.5)	65(60)	–	✔
RK_1	1.37	42	0.89	64	–	✔
RK_2	1.18	49	0.76	75	–	✔
compute_flux	6.4	51	4.91	67	✔	✔
numerical_flux	7.48	18	3.28	42	✔	✔
space_disc	9.24	40	7.95	45	–	✔

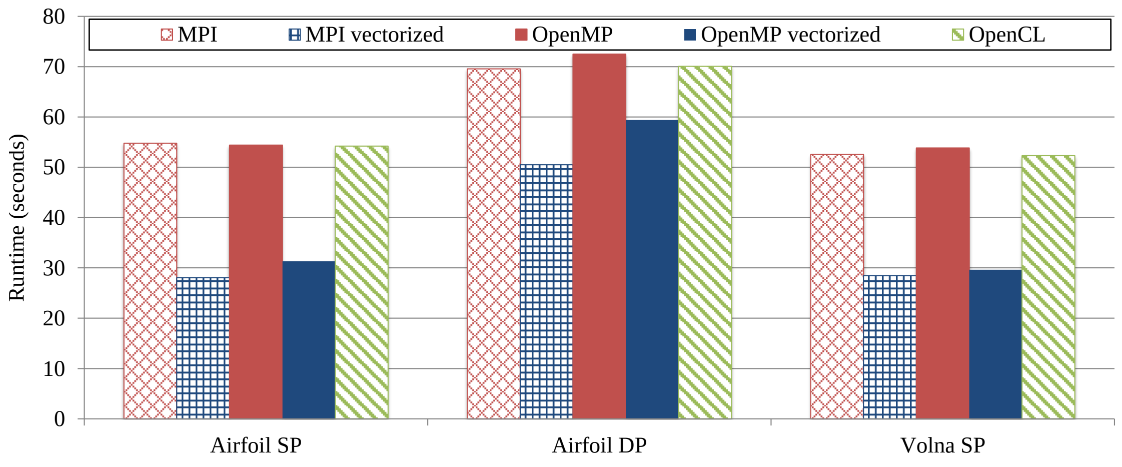


FIGURE 7.9: Vectorization with OpenCL compared to previous implementations. Performance results from Airfoil in single (SP) and double (DP) precision on the 2.8M cell mesh and from Volna in single precision on CPU.

DOI:10.15774/PPKE.ITK.2016.002

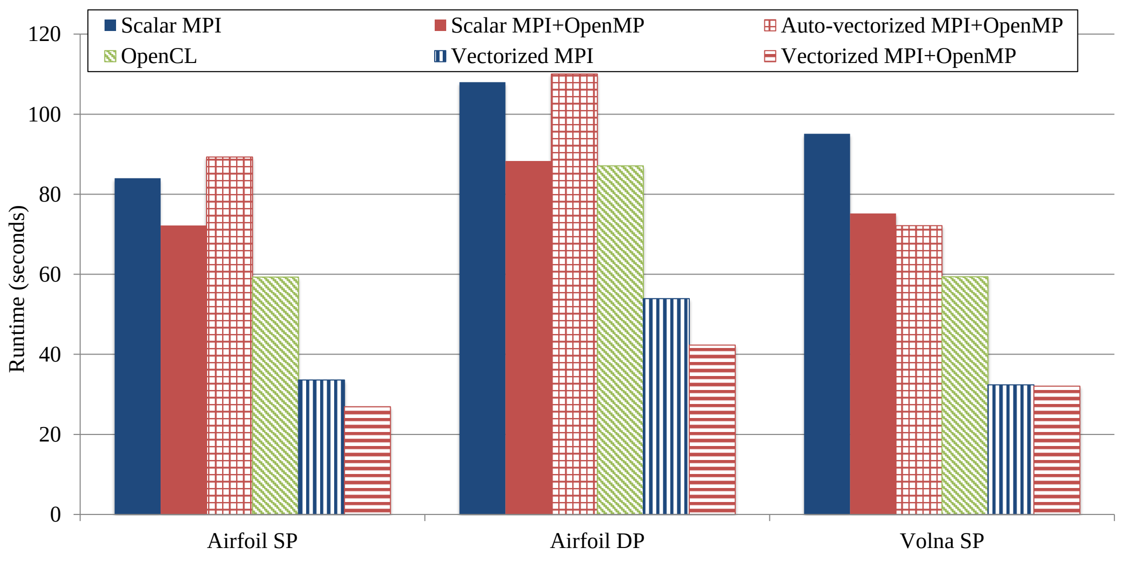


FIGURE 7.10: Performance of OpenCL compared to non-vectorized, auto-vectorized and explicitly vectorized (using intrinsic instructions) versions on the Xeon Phi using Airfoil (2.8M cell mesh) and Volna test cases.

7.6 Conclusion

The efficiency of the parallel colored indirect incrementation in real-world unstructured grid computations can be significantly improved by reordering the elements of the mesh in OP2 using the GPS minimal bandwidth reordering algorithm. This approach dramatically decreases the connectivity between the blocks and allows the mini-partitioning and naïve coloring method for decreasing the number of block colors. The fewer colors means less sequential execution and leads to better parallel execution performance and better parallel scalability.

In this chapter OpenCL vectorization capabilities were also evaluated to achieving efficient vectorized execution on modern Intel CPUs and the Xeon Phi. A key question is whether OpenCL SIMT model for expressing parallelism is sufficient for achieving good performance on multi- and many-core architectures. It has been shown how OP2 can map execution to multi-level parallel (multi-threading and vectorzation) setting using OpenCL. It is shown that the OpenCL extension developed for OP2 is capable of vectorizing certain kernels for unstructured grid computations on architectures based on SIMD type ISA. The SIMT parallel programming paradigm can be used to create vectorized SIMD machine code through Intel's support for the OpenCL language. The results show that OpenCL is adequately portable, but at the time of writing the driver and the runtime are not very efficient. When compared to the simple (non-vectorized)

DOI:10.15774/PPKE.ITK.2016.002

OpenMP execution, runtime is only slightly better; even though some degree of vectorization is carried out, there is a large overhead coming from the runtime system. This is expected to improve with time and with the introduction of new instruction sets.

DOI:10.15774/PPKE.ITK.2016.002

Chapter 8

Conclusion

Each chapter describing the particular problem that is solved is concluded with its own conclusion section. In this chapter the new scientific results are highlighted.

8.1 New scientific results

The new scientific results are grouped into thesis groups according to their classification among the 13 dwarves. Results regarding the new solutions proposed for solving tridiagonal system of equations can be categorized as the "Sparse Linear Algebra" dwarf which are detailed in Thesis group I. Image processing and PDE solution using the ADI method is categorized as the "Structured Grid" dwarf which is detailed in Thesis group II. Finally, results regarding CFD computations are categorized as the "Unstructured Grid" dwarf which is detailed in Thesis group III. The relations between thesis groups, parallel problem classification and parallel processor architectures are summarized in Figure 8.1.

New scientific results are published in journals(marked as [J]), conference (marked as [C]) proceedings and conference talks (marked as [CT]). Publications corresponding to the thesis groups are noted below.

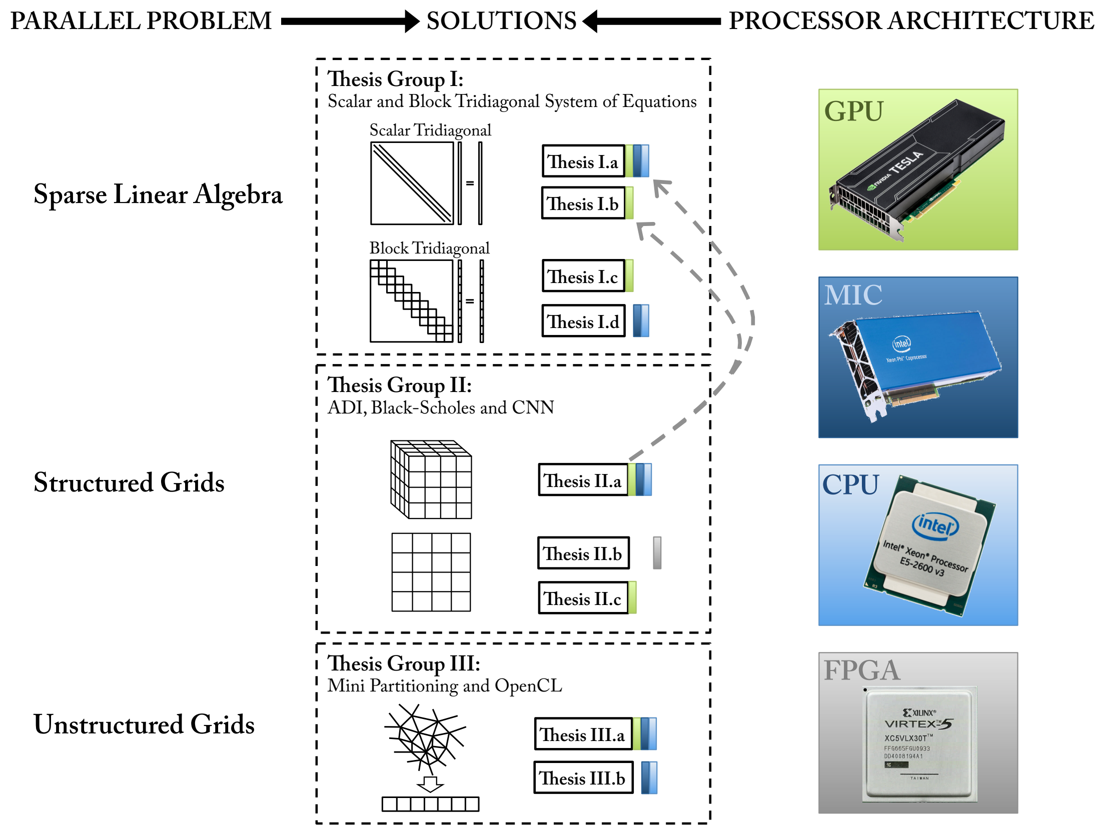


FIGURE 8.1: Thesis groups: relation between thesis groups, parallel problem classification and parallel processor architectures. Colored vertical bars right to thesis number denote the processor architecture that is used in that thesis. Dashed grey arrow represent relation between theses.

Thesis Group I. Efficient algorithms for sparse linear algebra

Many times PDEs arising in the scientific, engineering and financial applications require an extensive use of sparse linear algebra which needs to be efficiently parallelised for current and upcoming parallel processor architectures. In particular, the numerical solution of some special parabolic, diffusion type PDEs with implicit solvers boil down to the solution of tridiagonal system of equations where the elements of the tridiagonal matrix are either scalar values or blocks with size $M \times M$, where $M \in [2, 8]$. New parallel algorithms for the acceleration of such tridiagonal solvers is therefore essential to these scientific, engineering and financial communities to accelerate research and innovation. Theses in this group contribute to the parallelisation and acceleration of such methods on CPU, GPU and MIC architectures.

Corresponding publications: [J1], [C1], [CT1], [C2], [C3]

DOI:10.15774/PPKE.ITK.2016.002

Thesis I.a *I have developed new register blocking based local data transposition algorithms for multi- and many-core parallel processor architectures to improve the memory access pattern of the Thomas algorithm when solving tridiagonal system of equations where the coefficients are stored in consecutive order in the memory. The overall performance gain is: 1) up to* $\times 4.3$ *on the GPU compared to the NVIDIA cuSPRASE library; 2) up to* $\times 1.5$ *on the CPU compared to the Intel MKL library and 3) up to* $\times 1.9$ *on the MIC compared to the Intel MKL library.*

A tridiagonal system of equation is composed of three coeffcient vectors **a**, **b** and **c**, one unknown vector **u** and the right hand side vector **d**. All these vectors are element of $\mathbb{R}^N$ and they have identical data layout in the memory. The data layout of the coefficients of a tridiagonal system of equations may depend on the grid and the numerical method where it is applied, this can be consecutive or stride-N. Consecutive or stride-1 means that elements a_k and $a_{(k+1)}$ of vector **a** are at consecutive memory addresses in the linear memory, ie. at address k and $(k+1)$. Stride-N means that elements a_k and $a_{(k+1)}$ of vector **a** are at address $k \times N$ and $(k+1) \times N$ in the linear memory. In many cases the consecutive data layout is used which results in poor cache-line utilization and therefore requires memory access pattern optimization. I have developed two algorithms for GPUs, to perform data load (and store) of multiple values at once and transpose these data for calculation. I have also developed similar transposition based solvers for the CPU and MIC architectures. My solutions allow the transposition to be done using registers and the use of data directly from registers.

Thesis I.b *I have developed an efficient implementation of a new hybrid algorithm for the solution of tridiagonal system of equations on GPU architectures using the conventional Thomas and PCR algorithms in the case when the varying coefficients are stored in the main memory of the GPU. This GPU specific solution allows the utilization of the large number of registers on GPU architectures and local transposition of data when reading the coefficients are done in registers. The resulting solver is up to* $\times 9$ *times faster than the solver in the NVIDIA cuSPARSE library and up to* $\times 2.1$ *times faster than the transposition based GPU solver introduced in Thesis I.a.*

I have created the implementation of a new hybrid algorithm which is a combination of the Thomas and the PCR algorithms, which is optimal in the sense that is limited

DOI:10.15774/PPKE.ITK.2016.002

either by the memory bandwidth or the compute capacity of the GPU, depending on the floating point precision. This hybrid GPU specific algorithm allows the utilization of registers which leads to extreme efficiency when the system size is sufficiently small to fit into the registers of the GPU. Consecutive or stride-N data layout can both be handled with this algorithm. When needed transposition in register can be performed without the use of shared memory. As there is no need for storing intermediate values in global or local memory as in the case of the regular Thomas algorithm the ratio of floating point operations per byte is much higher for the new hybrid algorithm which results in better performance on a high compute intensity architecture like the GPU.

Thesis I.c *I have developed a new algorithm for the solution of block tridiagonal system of equations on GPU using a new thread work sharing and register blocking approach, when the block sizes are $M \times M$ with $M = \in 2, 8]$. The achieved computational performance of this GPU specific work sharing approach is superior compared to the know algorithms and their implementation. I have experimentally showed that it is up to $\times 27$ times faster than the Intel MKL banded solver and up to $\times 9.8$ faster than the PCR-based GPU solver proposed by [54].*

Block tridiagonal system of equations arise in CFD (Computational Fluid Dynamic) applications [8, 7], where block sizes vary according to the multi-variable PDE. I gave a new thread work-sharing algorithm to parallelise the Thomas algorithm. I have also shown that this approach is computationally more efficient than the CR or PCR algorithms and that sufficient parallelism can be exploited with work-sharing to exceed the performance of the CR and PCR algorithms when block size is in the range $M = \in 2, 8]$.

Thesis I.d *I have developed a new implementation for the solution of block tridiagonal system of equations on multi-core CPU and many-core MIC with vector instructions, which outperforms the banded solver of the Intel MKL library in terms of computational performance by a factor of $\times 6.5$ and $\times 5$ in the case of the CPU and MIC repectively.*

I have restructured the code and data of the standard block Thomas algorithm with C++ templates and code transformation to achieve better data locality and guide the compiler in the vectorization procedure. The result is a highly efficient SIMD based CPU and MIC implementation of the block Thomas algorithm.

DOI:10.15774/PPKE.ITK.2016.002

Thesis Group II. Efficient algorithms for strucutred grid computations

The solution of parabolic, diffusion type PDEs on a structured grid domain can be efficiently solved using the ADI (Alternating Direction Implicit) method which – in higher dimensions – boils down to the solution of multiple tridiagonal system of equations. The memory access pattern of the tridiagonal solves along different dimensions varies and results in significant performance loss if conventional multi- and many core implementations of numerical library functions are used. Therefore new efficient algorithms are needed. Also, the solution of diffusion type PDEs like the Black-Scholes PDE arising in financial applications and other PDEs related to CNN based image processing require efficient parallelization solutions for parallel processor architectures like FPGAs and GPUs.

Corresponding publications: [J1], [CT1], [C3], [C6]

Thesis II.a *I have elaborated and implemented new parallel algorithms to accelerate the calculation of the ADI (Alternating Direction Implicit) method for the solution of PDEs in higher spatial dimensions on CPU, MIC and GPU. The resulting ADI solver utilizes new solvers of tridiagonal system of equations with stride-1 and stride-N access patterns.*

When solving parabolic, diffusion type multi-dimensional PDEs many times it is possible to decouple the solution of the higher spatial N dimensions into N number of one-dimensional problems. I have developed ways to optimize the efficiency of the solution in each dimension as the memory access pattern changes significantly along each dimension for arbitrary dimensions. This result is also a representative use case for the tridiagonal solvers of Thesis I.a and Thesis I.b.

Thesis II.b *I have developed a power-efficient parallel FPGA based solver in HLS (High Level Synthesis) to accelerate the numerical solution of the 1-factor Black-Scholes PDE. The resulting FPGA solver is on par with the CPU solver in terms of computational performance, but it outperforms the CPU in terms of computational efficiency (GFLOPS/W) by a factor of 4 in the case of the explicit solution and by a factor of 1.3 when solving the implicit problem.*

DOI:10.15774/PPKE.ITK.2016.002

Solving the 1-factor Black-Scholes PDE for pricing financial derivatives with one underlying asset requires an explicit or implicit solution of the PDE. I have proposed an efficient stencil operation type solutions for the explicit method and an efficient Thomas algorithm implementation for the implicit method.

Thesis II.c *I have experimentally proven that the utilization of the texture cache by a double buffer approach is an efficient way to implement a GPU based accelerator for the solution of the CNN state equation as it increases the cache hit rate of the two dimensional texture cache due to spatial locality and reduces the number of integer operation involved during the memory index calculations through the built in functionality of the texture cache.*

When solving the CNN state equation to perform a diffusion operation on an image, a heat diffusion PDE is solved. The solution of such a PDE is memory bandwidth limited. As such, a tiling or cache blocking optimizations amortize the data transfer and allow good performance. I have shown that this performance can be overcome by utilizing the built-in texture cache capability of the GPU. With this approach the solution of the two dimensional CNN state equation can take advantage of the significant cache reuse of the texture cache when fetching data. Also, significant integer operations and latency can be saved by utilizing the built-in coordinate calculation units of the GPU.

Thesis Group III. Efficient algorithms for unstructured grid computations

Increasing the efficiency of parallel computations on unstructured grid applications in the OP2 framework is of high importance to the scientific community. Improving the single node scalability of the indirect increment of values in the OP2 framework is of great importance for performance. One component of the scalability is the mini-partitioning. The current naive mini-partitioning solution can be improved by exploiting the use minimal matrix bandwidth reordering techniques. New methods for an OpenCL backend of the OP2 framework can improve the performance of unstructured grid applications on CPU and MIC, and provide code portability.

Corresponding publications: [C4], [J2], [C5]

DOI:10.15774/PPKE.ITK.2016.002

Thesis III.a *I have experimentally proven that the efficiency of the parallel colored indirect incrementation in unstructured grid computations can be improved by extending the mini-partitioning in OP2 using the GPS (Gibbs-Poole-Stockmeyer) minimal bandwidth reordering algorithm on real-world CFD simulation problems. This approach dramatically decreases the number of block colors used in the parallel increment and therefore increases the number of blocks within a color which can be incremented in parallel. The reduction in the number of block colors is up to* $\times 37.5$ *for real-world test cases and the speedup resulting from this improvement is* $\times 7.4$.

In order to increase the efficiency of parallel incrementation on unstructured grids OP2 utilizes a two level coloring scheme to identify the set of elements and blocks which can be incremented in parallel. The number of colors determines the number of sequential steps during the incrementation process. In real world applications the number of block colors can be high due to excessive number of connections between the mini-partitions. I have proposed the use of the GPS bandwidth minimization algorithm to reorder meshes so that the naïve mini-partitioning creates blocks with less neighboring connections which helps the two level coloring algorithm to get better block coloring.

Thesis III.b *I have analysed and proposed new heuristic code transformation techniques to improve the vectorization capabilities of OpenCL on CPU and MIC architectures. The resulting OpenCL kernels within OP2 lend themselves to better parallelization properties on real world simulation codes. In case of the CPU the achieved performance is on par with the OpenMP and MPI parallelization. However, the OpenCL implementation is* $\times 1.5$ *faster on the MIC compared to the MPI+OpenMP solution.*

I have exploited the capabilities of the Intel OpenCL implementation for CPU and MIC to increase their performance by implementing the OpenCL backend of the OP2 framework. The key to this improvement is the matching of multi-threading and SIMD vectorization features of the CPU (or MIC) to the SIMT type kernel abstraction of the OpenCL standard.

DOI:10.15774/PPKE.ITK.2016.002

8.2 Application of the results

As noted in the Section 1.2 the problems selected for parallelization originate from the scientific, engineering and financial disciplines and therefore they have a direct application in these fields. The results concerning scalar and block tridiagonal solvers were presented at the GPU Technology Conference in San Jose in 2014, the largest GPU conference by today and on conferences and journal papers..

The integration of the results from the research of scalar tridiagonal solver to atmospheric simulations is an ongoing work by Eike Mueller and his collegues at the Department of Mathematical Sciences at the University of Bath, UK. In atmospheric modelling due to the structured meshing of the atmosphere the cell geometries are too distorted which results in numerical stability problems, also known as highly anisotropic problem. Line-smoother in the multigrid preconditioner requires the frequent inversion of a tridiagonal matrix and therefore the Thomas-PCR algorithm may have a high impact on the performance.

My results are used to improve the performance of a Navier-Stokes CFD solver called Turbostream. Turbostream is developed by Dr. Tobias Brandvik and Dr. Graham Pullan in the Whittle Laboratory at the University of Cambridge, UK. It is aimed at solving CFD problems arising in the design of turbomachinery where the numerical solution of the Navier-Stokes equations are done using a line-implicit or semi-implicit Runge-Kutta method on an unstructured grid with stretched regions. Along these stretched regions an (block tridiagonal) implicit solution is required for unconditional stability, as the grid cells are highly anisotropic.

Dr. Serge Guillas and his colleagues at the University College London, UK used the VolnaOP2 tsunami simulation code – written by István Reguly and me – to perform measurements for the papers [92] and [93]. This implementation make large scale tsunami simulations feasible which result in statistically relevant results.

DOI:10.15774/PPKE.ITK.2016.002

Appendix A

Appendix

A.1 Hardware and Software for Scalar and Block Tridiagonal Solvers

The most salient properties of the CPU, MIC and GPU used in the present study are shown in Table A.1. The Intel Composer XE Suite 2015.2.164 with compiler version 15.0.2 compiler and Intel Math Kernel Library version 11.2 Update 2 for Linux was used to perform the benchmarks. For the NVIDIA GPU benchmark the CUDA 7.0 runtime and driver version 346.46 was used.

DOI:10.15774/PPKE.ITK.2016.002

Table A.1: Details of Intel Xeon Sandy Bridge server processor, Intel Xeon Phi coprocessor and the NVIDIA Tesla GPU card. *Estimates are based on [23], [44] and [94]. Both Intel architectures have 8-way, shared LLC with ring-bus topology. HT - Hyper Thread, MM - Multi Media, RO - Read Only

	Intel Xeon E5-2680	Intel Xeon Phi 5110P	NVIDIA K40m
Microarchitecture	Sandy Bridge	Knights Corner	GK110b
No. sockets	2	1	1
CPU core / SMX unit	2x8	59 (+1 for OS)	15
HT / core or CUDA core/SMX	2	4	192
MM register width	256 bits	512 bits	32 bits
Registers / thread	16 YMM	32 ZMM	255
L1 data cache / core	32 KB	32 KB	64 KB + 48 KB RO
L2 data cache / core	256 KB	30 MB	1.5 MB
L3 data cache / socket	2x20 MB	-	-
Cache line	64 bytes	64 bytes	32 bytes
Cache latency L1/L2/L3*	4/11/21*	3/22/-*	38/-/- *
Virtual page size	4KB-2MB	2MB	4KB*
Clock rate	2.7 (3.5 Turbo) GHz	1053 MHz	745 MHz
fp32 perf.	2x172.8 GFLOPS	2.022 TFLOPS	4.29 TFLOPS
fp64 perf.	2x86.4 GFLOPS	1.011 TFLOP	1.43 TFLOPS
Installed memory	64 GB DDR3	8 GB GDDR5	12 GB GDDR5
Memory bandwidth	2x51.2 GB/s	320 GB/s	288 GB/s
PCI bus	PCI-E x40 Gen3	PCI-E x16 Gen2	PCI-E x16 Gen3
TDP / Cooling	2x130 W / Active	225W / Passive	235 W / Passive
Recommended price	2x1727 USD	2649 USD	5499 USD

DOI:10.15774/PPKE.ITK.2016.002

A.2 System size configuration for benchmarking block tridiagonal solvers.

TABLE A.2: Parameter N - Length of a system used for benchmarking a processor architecture and solver. The length of the system is chosen such that the problem fits into the memory of the selected architecture.

	Block size	2	3	4	5	6	7	8
SP	CPU	128	128	128	128	128	96	96
	CPU MKL 11.2.2	128	128	128	128	128	96	96
	MIC	128	128	128	128	128	96	96
	MIC MKL 11.2.2	128	128	128	128	128	96	96
	GPU Shuffle	128	128	128	128	128	96	96
	GPU Shared	128	128	128	128	128	96	96
DP	CPU	128	128	128	128	128	96	96
	CPU MKL 11.2.2	128	128	128	128	128	96	96
	MIC	128	128	128	128	128	64	64
	MIC MKL 11.2.2	64	64	64	64	64	48	48
	GPU Shuffle	128	128	128	128	128	96	96
	GPU Shared	128	128	128	128	128	96	96

TABLE A.3: Parameter P - Number of systems used for benchmarking a processor architecture and solver. The number of systems is chosen such that the problem fits into the memory of the selected architecture.

	Block size	2	3	4	5	6	7	8
SP	CPU	65536	65536	65536	65536	65536	32768	32768
	CPU MKL 11.2.2	65536	65536	65536	65536	65536	32768	32768
	MIC	32768	32768	32768	32768	32768	16384	16384
	MIC MKL 11.2.2	32768	32768	32768	32768	32768	16384	16384
	GPU Shuffle	65536	65536	65536	65536	65536	32768	32768
	GPU Shared	65536	65536	65536	65536	65536	32768	32768
DP	CPU	65536	65536	65536	65536	65536	32768	32768
	CPU MKL 11.2.2	65536	65536	65536	65536	65536	32768	32768
	MIC	32768	32768	32768	32768	32768	16384	16384
	MIC MKL 11.2.2	32768	32768	32768	32768	32768	16384	16384
	GPU Shuffle	65536	65536	65536	65536	65536	32768	32768
	GPU Shared	65536	65536	65536	65536	65536	32768	32768

DOI:10.15774/PPKE.ITK.2016.002

A.3 Volna OP2 benchmark setup

The system used for benchmarking Volna OP2 is based on a dual socket Intel Xeon server processor. Ubuntu 14.04 LTS server edition operating system with Intel Composer XE 2015.3.187 and compiler version 15.0.3 was used. Software threads are pinned to CPU cores using the `KMP_AFFINITY=compact,0` environment variable.

Table A.4: Details of Intel Xeon Haswell server processor used in the benchmarks of Volna OP2. HT - Hyper Thread, MM - Multi Media

Intel Xeon E5-2695v3	
Microarchitecture	Haswell
ISA	AVX2
No. sockets	2
CPU core	2x14
Threads / core	2
MM register width	256 bits
Registers / thread	16 YMM
L1 data cache / core	32 KB
L2 data cache / core	256 KB
L3 data cache / socket	2x35 MB
Cache line	64 bytes
Clock rate	2.3 (3.3 Turbo) GHz
fp32 perf.	2x512.2 GFLOPS
fp64 perf.	2x257.6 GFLOPS
Installed memory	64 GB DDR4
Memory bandwidth	2x68 GB/s
TDP / Cooling	2x120 W / Active
Recommended price	2x2424 USD

A.4 OpenCL benchmark setup

Table A.5: Benchmark systems specifications

System	CPU	Xeon Phi
Architecture	2×Xeon E5-2640	Xeon Phi 5110P
Clock frequency	2.4 GHz	1.053 GHz
Core count	2×6	61 (60 used)
Last level cache	2×15MB	30MB
Peak bandwidth	2×42.6 GB/s	320 GB/s
Peak GFLOPS DP(SP)	2×120(240)	1010 (2020)
Stream bandwidth	66.8 GB/s	171 GB/s
D/SGEMM GFLOPS	229(433)	833(1729)
FLOP/byte DP(SP)	3.42(6.48)	4.87(10.1)
Recommended price	2x889 USD	2649 USD

DOI:10.15774/PPKE.ITK.2016.002

The Author's Publications

[J1] **Endre Laszló**, Mike Giles, and Jeremy Appleyard. "Manycore algorithms for batch scalar and block tridiagonal solvers (Accepted)". In: *ACM Transactions on Mathematical Software (TOMS)* (2015).

[C1] **Endre Laszló**, Zoltán Nagy, Mike Giles, István Reguly, Jeremy Appleyard, and Péter Szolgay. "Analisys of Parallel Processor Architectures for the Solution of Tridiagonal System of Equations". In: *International Symposium on Circuits and Systems*. Lisbon, Protugal: IEEE Press, 24-27 May 2015.

[CT1] **Endre László** and Mike Giles. "Efficient Solution of Multiple Scalar and Block-Tridiagonal Equations". In: *GPU Technology Conference*. San Jose, CA: NVIDIA, 2014.

[C2] Mike Giles, **Endre László**, István Reguly, Jeremy Appleyard, and Julien Demouth. "GPU Implementation of Finite Difference Solvers". In: *Proceedings of the 7th Workshop on High Performance Computational Finance*. WHPCF '14. New Orleans, Louisiana: IEEE Press, 2014, pp. 1–8. ISBN: 978-1-4799-7027-8. DOI: 10.1109/WHPCF.2014.10. URL: http://dx.doi.org/10.1109/WHPCF.2014.10.

[C3] **Endre László**, Michael B Giles, Jeremy Appleyard, and Péter Szolgay. "Methods to utilize SIMT and SIMD instruction level parallelism in tridiagonal solvers". In: *Cellular Nanoscale Networks and their Applications (CNNA), 2014 14th International Workshop on*. IEEE. 2014, pp. 1–2.

[C6] **Endre László**, Péter Szolgay, and Zoltán Nagy. "Analysis of a GPU based CNN implementation". In: *Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on*. IEEE. 2012, pp. 1–5.

[C4] István Z Reguly, **Endre László**, Gihan R Mudalige, and Mike B Giles. "Vectorizing Unstructured Mesh Computations for Many-core Architectures". In: *Proceedings of Programming Models and Applications on Multicores and Manycores*. ACM. 2014, p. 39.

DOI:10.15774/PPKE.ITK.2016.002

[J2] I Z. Reguly, **Endre László**, Gihan R. Mudalige, and Mike B. Giles. "Vectorizing unstructured mesh computations for many-core architectures". In: *Concurrency and Computation: Practice and Experience* (2015). ISSN: 1532-0634. DOI: 10.1002/cpe.3621. URL: http://dx.doi.org/10.1002/cpe.3621.

[C5] Mike B Giles, Gihan R Mudalige, Carlo Bertolli, Paul HJ Kelly, **Endre László**, and I 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:* IEEE. 2012, pp. 477–482.

DOI:10.15774/PPKE.ITK.2016.002

The Author's Other Publications

Publications not strictly related to the doctoral theses

[J3] Béla Szentpáli, Gábor Matyi, Péter Fürjes, **Endre László**, Gábor Battistig, István Bársony, Gergely Károlyi, and Tibor Berceli. "Thermopile-based THz antenna". In: *Microsystem technologies* 18.7-8 (2012), pp. 849–856.

[C7] Béla Szentpáli, Gábor Matyi, Péter Fürjes, **Endre László**, Gábor Battistig, István Bársony, Gergely Károlyi, and Tibor Berceli. "THz detection by modified thermopile". In: *SPIE Microtechnologies* (2011).

[C8] **Endre László**, Kálmán Tornai, Gergely Treplán, and János Levendovszky. "Novel load balancing scheduling algorithms for wireless sensor networks". In: *CTRQ 2011, The Fourth International Conference on Communication Theory, Reliability, and Quality of Service.* 2011, pp. 54–59.

[C9] Janos Levendovszky, **Endre László**, Kalman Tornai, and Gergely Treplan. "Optimal pricing based resource management". In: *Proceedings of the International Conference on Operations Research* (2010), p. 169.

[LN1] Zoltán Nagy, Péter Szolgay, András Kiss, and **Endre László**. "GPU architektúrák". In: *Párhuzamos számítógép architektúrák, processzortömbök.* Pázmány Egyetem eKiadó, 2015. Chap. 3, pp. 34–59.

DOI:10.15774/PPKE.ITK.2016.002

References

[1] Krste Asanovic et al. “A View of the Parallel Computing Landscape”. In: *Commun. ACM* 52.10 (Oct. 2009), pp. 56–67. ISSN: 0001-0782. DOI: 10.1145/1562764.1562783. URL: http://doi.acm.org/10.1145/1562764.1562783.

[2] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[3] John L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Communications of the ACM* 31 (1988), pp. 532–533.

[4] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785.

[5] Yushan Wang, Marc Baboulin, Jack Dongarra, Joël Falcou, Yann Fraigneau, and Olivier Le Maître. “A Parallel Solver for Incompressible Fluid Flows”. In: *Procedia Computer Science* 18 (2013). 2013 International Conference on Computational Science, pp. 439–448. ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2013.05.207. URL: http://www.sciencedirect.com/science/article/pii/S1877050913003505.

[6] Tobias Brandvik and Graham Pullan. “An Accelerated 3D Navier–Stokes Solver for Flows in Turbomachines”. In: *Journal of Turbomachinery* 133.2 (2011), pp. 021025+. DOI: 10.1115/1.4001192. URL: http://dx.doi.org/10.1115/1.4001192.

[7] Thomas H. Pulliam. “Implicit solution methods in computational fluid dynamics”. In: *Applied Numerical Mathematics* 2.6 (1986), pp. 441–474. ISSN: 0168-9274. DOI: http://dx.doi.org/10.1016/0168-9274(86)90002-4. URL: http://www.sciencedirect.com/science/article/pii/0168927486900024.

DOI:10.15774/PPKE.ITK.2016.002

[8] Thomas H Pulliam. “Solution methods in computational fluid dynamics”. In: *Notes for the von Kármán Institute For Fluid Dynamics Lecture Series* (1986).

[9] I.J.D. Craig and A.D. Sneyd. “An alternating-direction implicit scheme for parabolic equations with mixed derivatives”. In: *Computers and Mathematics with Applications* 16.4 (1988), pp. 341–350. ISSN: 0898-1221. DOI: http://dx.doi.org/10.1016/0898-1221(88)90150-2. URL: http://www.sciencedirect.com/science/article/pii/0898122188901502.

[10] Duy M. Dang, Christina Christara, and Kenneth R. Jackson. “Parallel Implementation on GPUs of ADI Finite Difference Methods for Parabolic PDEs with Applications in Finance”. In: *Social Science Research Network Working Paper Series* (Apr. 3, 2010). URL: http://ssrn.com/abstract=1580057.

[11] Craig C. Douglas, Sachit Malhotra, and Martin H. Schultz. *Parallel Multigrid with ADI-like Smoothers in Two Dimensions*. 1998.

[12] B. Düring, M. Fournié, and A. Rigal. “High-Order ADI Schemes for Convection-Diffusion Equations with Mixed Derivative Terms”. English. In: *Spectral and High Order Methods for Partial Differential Equations - ICOSAHOM 2012*. Ed. by Mejdi Azaiez, Henda El Fekih, and Jan S. Hesthaven. Vol. 95. Lecture Notes in Computational Science and Engineering. Springer International Publishing, 2014, pp. 217–226. ISBN: 978-3-319-01600-9. DOI: 10.1007/978-3-319-01601-6_17. URL: http://dx.doi.org/10.1007/978-3-319-01601-6_17.

[13] Samir Karaa and Jun Zhang. “High order ADI method for solving unsteady convection–diffusion problems”. In: *Journal of Computational Physics* 198.1 (2004), pp. 1–9. ISSN: 0021-9991. DOI: http://dx.doi.org/10.1016/j.jcp.2004.01.002. URL: http://www.sciencedirect.com/science/article/pii/S002199910400018X.

[14] D. W. Peaceman and Jr. Rachford H. H. “The Numerical Solution of Parabolic and Elliptic Differential Equations”. English. In: *Journal of the Society for Industrial and Applied Mathematics* 3.1 (1955), ISSN: 03684245. URL: http://www.jstor.org/stable/2098834.

[15] J. Douglas and H. H. Rachford. “On the numerical solution of heat conduction problems in two and three space variables”. In: *Transaction of the American Mathematical Society* 82 (1956), pp. 421–489.

[16] Jr. Douglas Jim and JamesE. Gunn. “A general formulation of alternating direction methods”. English. In: *Numerische Mathematik* 6.1 (1964), pp. 428–453. ISSN: 0029-599X. DOI: 10.1007/BF01386093. URL: http://dx.doi.org/10.1007/BF01386093.

DOI:10.15774/PPKE.ITK.2016.002

[17] T. Roska and L.O. Chua. “The CNN universal machine: an analogic array computer”. In: *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on* 40.3 (Mar. 1993), pp. 163–173. ISSN: 1057-7130. DOI: 10.1109/82.222815.

[18] Balázs Gergely Soós, Ádám Rák, József Veres, and György Cserey. “GPU Boosted CNN Simulator Library for Graphical Flow-based Programmability”. In: *EURASIP J. Adv. Signal Process* 2009 (Jan. 2009), 8:1–8:11. ISSN: 1110-8657. DOI: 10.1155/2009/930619. URL: http://dx.doi.org/10.1155/2009/930619.

[19] Zsolt Vörösházi, András Kiss, Zoltán Nagy, and Péter Szolgay. “Implementation of embedded emulated-digital CNN-UM global analogic programming unit on FPGA and its application”. In: *International Journal of Circuit Theory and Applications* 36.5-6 (2008), pp. 589–603. ISSN: 1097-007X. DOI: 10.1002/cta.507. URL: http://dx.doi.org/10.1002/cta.507.

[20] *CUDA C Programming Guide*. NVIDIA. Mar. 2015. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3aTPUq4Jo.

[21] *Whitepaper, NVIDIA's Next Generation CUDA Compute Architecture: Fermi v1.1*. NVIDIA. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf.

[22] *Whitepaper, NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, v1.0*. NVIDIA. URL: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[23] Subhash Saini, Johnny Chang, and Haoqiang Jin. *Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications*. Tech. rep. Moffett Field, California 94035-1000, USA: NASA Advanced Supercomputing Division, NASA Ames Research Center.

[24] Agner Fog. “The microarchitecture of Intel, AMD and VIA CPUs”. In: *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering* (Aug. 7, 2014). URL: http://www.agner.org/optimize/microarchitecture.pdf.

[25] *Intel Architecture Instruction Set Extensions Programming Reference*. Intel. Aug. 1989. URL: https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf.

[26] *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*. Intel. Sept. 2012. URL: https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf.

DOI:10.15774/PPKE.ITK.2016.002

[27] Xilinx. “7 Series FPGAs Overview - Product Specification v1.17”. In: (May 27, 2015). URL: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[28] Omer Egecioglu, Cetin K. Koc, and Alan J. Laub. “Special Issue on Parallel Algorithms for Numerical Linear Algebra A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors”. In: *Journal of Computational and Applied Mathematics* 27.1 (1989), pp. 95–108. ISSN: 0377-0427. DOI: http://dx.doi.org/10.1016/0377-0427(89)90362-2. URL: http://www.sciencedirect.com/science/article/pii/0377042789903622.

[29] Yao Zhang, Jonathan Cohen, and John D. Owens. “Fast Tridiagonal Solvers on the GPU”. In: *SIGPLAN Not.* 45.5 (Jan. 2010), pp. 127–136. ISSN: 0362-1340. DOI: 10.1145/1837853.1693472. URL: http://doi.acm.org/10.1145/1837853.1693472.

[30] Li-Wen Chang, John A. Stratton, Hee-Seok Kim, and Wen-Mei W. Hwu. “A Scalable, Numerically Stable, High-performance Tridiagonal Solver Using GPUs”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* SC ’12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 27:1–27:11. ISBN: 978-1-4673-0804-5. URL: http://dl.acm.org/citation.cfm?id=2388996.2389033.

[31] Harold S. Stone. “An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations”. In: *J. ACM* 20.1 (Jan. 1973), pp. 27–38. ISSN: 0004-5411. DOI: 10.1145/321738.321741. URL: http://doi.acm.org/10.1145/321738.321741.

[32] H. H. Wang. “A Parallel Method for Tridiagonal Equations”. In: *ACM Trans. Math. Softw.* 7.2 (June 1981), pp. 170–183. ISSN: 0098-3500. DOI: 10.1145/355945.355947. URL: http://doi.acm.org/10.1145/355945.355947.

[33] Henk A van der Vorst. “Large tridiagonal and block tridiagonal linear systems on vector and parallel computers”. In: *Parallel Computing* 5.1–2 (1987). Proceedings of the International Conference on Vector and Parallel Computing-Issues in Applied Research and Development, pp. 45–54. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/0167-8191(87)90005-6. URL: http://www.sciencedirect.com/science/article/pii/0167819187900056.

[34] Stefan Bondeli. “Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations”. In: *Parallel Computing* 17.4–5 (1991), pp. 419–434. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/S0167-8191(05)80145-0. URL: http://www.sciencedirect.com/science/article/pii/S0167819105801450.

DOI:10.15774/PPKE.ITK.2016.002

[35] Nathan Mattor, Timothy J. Williams, and Dennis W. Hewett. “Algorithm for solving tridiagonal matrix problems in parallel”. In: *Parallel Computing* 21.11 (1995), pp. 1769–1782. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/0167-8191(95)00033-0. URL: http://www.sciencedirect.com/science/article/pii/0167819195000330.

[36] G Spaletta and D.J Evans. “The parallel recursive decoupling algorithm for solving tridiagonal linear systems”. In: *Parallel Computing* 19.5 (1993), pp. 563–576. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/0167-8191(93)90006-7. URL: http://www.sciencedirect.com/science/article/pii/0167819193900067.

[37] L. H. Thomas. *Elliptic Problems in Linear Differential Equations over a Network*. Tech. rep. New York: Columbia University, 1949.

[38] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. New York, NY, USA: Cambridge University Press, 2007. ISBN: 0521880688, 9780521880688.

[39] O. Buneman. *COMPACT NON-ITERATIVE POISSON SOLVER*. Jan. 1969.

[40] Harold S. Stone. “Parallel Tridiagonal Equation Solvers”. In: *ACM Trans. Math. Softw.* 1.4 (Dec. 1975), pp. 289–307. ISSN: 0098-3500. DOI: 10.1145/355656.355657. URL: http://doi.acm.org/10.1145/355656.355657.

[41] Walter Gander and Gene H. Golub. “Cyclic Reduction - History and Applications”. In: *Proceedings of the Workshop on Scientific Computing*. Oct. 1997.

[42] Nikolai Sakharnykh. “Efficient Tridiagonal Solvers for ADI methods and Fluid Simulation”. In: Presented at the GPU Technology Conference, San Jose, CA, 2010. URL: http://on-demand.gputechconf.com/gtc/2010/presentations/S12015-Tridiagonal-Solvers-ADI-Methods-Fluid-Simulation.pdf.

[43] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel. Apr. 2012. URL: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[44] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. “Demystifying GPU microarchitecture through microbenchmarking”. In: *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*. 2010, pp. 235–246. DOI: 10.1109/ISPASS.2010.5452013.

DOI:10.15774/PPKE.ITK.2016.002

[45] I. Z. Reguly, E. László, G. R. Mudalige, and M. B. Giles. “Vectorizing Unstructured Mesh Computations for Many-core Architectures”. In: *Proceedings of Programming Models and Applications on Multicores and Manycores*. PMAM’14. Orlando, FL, USA: ACM, 2014, 39:39–39:50. ISBN: 978-1-4503-2657-5. DOI: 10.1145/2560683.2560686. URL: http://doi.acm.org/10.1145/2560683.2560686.

[46] Rami Al Na’mneh, W. David Pan, and Seong-Moo Yoo. “Efficient Adaptive Algorithms for Transposing Small and Large Matrices on Symmetric Multiprocessors”. In: *Informatica* 17.4 (Dec. 2006), pp. 535–550. ISSN: 0868-4952. URL: http://dl.acm.org/citation.cfm?id=1413878.1413883.

[47] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. “NUMA-aware algorithms: the case of data shuffling”. In: *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. Jan. 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=258798.

[48] Lars Bergstrom. “Measuring NUMA effects with the STREAM benchmark”. In: *CoRR* abs/1103.3225 (2011). URL: http://arxiv.org/abs/1103.3225.

[49] Nikolai Sakharnykh. “Tridiagonal Solvers on the GPU and Applications to Fluid Simulation”. In: Presented at the GPU Technology Conference, San Jose, CA, 2009. URL: http://www.nvidia.com/content/gtc/documents/1058_gtc09.pdf.

[50] Intel. *Math Kernel Library*. 2015. URL: http://software.intel.com/en-us/articles/intel-mkl/.

[51] *CUSPARSE LIBRARY v7.0*. NVIDIA. Mar. 2015. URL: http://docs.nvidia.com/cuda/cusparse/index.html#axzz3aTPUq4Jo.

[52] S.P. Hirshman, K.S. Perumalla, V.E. Lynch, and R. Sanchez. “BCYCLIC: A parallel block tridiagonal matrix cyclic solver”. In: *Journal of Computational Physics* 229.18 (2010), pp. 6392–6404. ISSN: 0021-9991. DOI: http://dx.doi.org/10.1016/j.jcp.2010.04.049. URL: http://www.sciencedirect.com/science/article/pii/S0021999110002536.

[53] Sudip K. Seal, Kalyan S. Perumalla, and Steven P. Hirshman. “Revisiting parallel cyclic reduction and parallel prefix-based algorithms for block tridiagonal systems of equations”. In: *Journal of Parallel and Distributed Computing* 73.2 (2013), pp. 273–280. ISSN: 0743-7315. DOI: http://dx.doi.org/10.1016/j.jpdc.2012.10.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731512002535.

DOI:10.15774/PPKE.ITK.2016.002

[54] Christopher P Stone, Earl PN Duque, Yao Zhang, David Car, John D Owens, and Roger L Davis. “GPGPU parallel algorithms for structured-grid CFD codes”. In: *Proceedings of the 20th AIAA Computational Fluid Dynamics Conference*. Vol. 3221. 2011.

[55] *TESLA C2050 / C2070 GPU Computing Processor*. NVIDIA. July 2010. URL: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf.

[56] Mike Giles, Endre László, István Reguly, Jeremy Appleyard, and Julien Demouth. “GPU Implementation of Finite Difference Solvers”. In: *Proceedings of the 7th Workshop on High Performance Computational Finance*. WHPCF ’14. New Orleans, Louisiana: IEEE Press, 2014, pp. 1–8. ISBN: 978-1-4799-7027-8. DOI: 10.1109/WHPCF.2014.10. URL: http://dx.doi.org/10.1109/WHPCF.2014.10.

[57] Qiwei Jin, D.B. Thomas, and W. Luk. “Exploring reconfigurable architectures for explicit finite difference option pricing models”. In: *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. Aug. 2009, pp. 73–78. DOI: 10.1109/FPL.2009.5272549.

[58] G. Chatziparaskevas, A. Brokalakis, and I. Papaefstathiou. “An FPGA-based parallel processor for Black-Scholes option pricing using finite differences schemes”. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*. Mar. 2012, pp. 709–714. DOI: 10.1109/DATE.2012.6176562.

[59] T. Becker, Qiwei Jin, W. Luk, and Stephen Weston. “Dynamic Constant Reconfiguration for Explicit Finite Difference Option Pricing”. In: *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. Nov. 2011, pp. 176–181. DOI: 10.1109/ReConFig.2011.29.

[60] Samuel Palmer and David Thomas. “Accelerating Implicit Finite Difference Schemes Using a Hardware Optimised Implementation of the Thomas Algorithm for FPGAs”. In: *arXiv preprint arXiv:1402.5094* (2014).

[61] Fischer Black and Myron S Scholes. “The Pricing of Options and Corporate Liabilities”. In: *Journal of Political Economy* 81.3 (May 1973), pp. 637–54. URL: http://ideas.repec.org/a/ucp/jpolec/v81y1973i3p637-54.html.

[62] John D. McCalpin. “Memory Bandwidth and Machine Balance in Current High Performance Computers”. In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

DOI:10.15774/PPKE.ITK.2016.002

[63] Zoltán Nagy, Zsolt Vörösházi, and Péter Szolgay. "Emulated Digital CNN-UM Solution of Partial Differential Equations: Research Articles". In: *Int. J. Circuit Theory Appl.* 34.4 (July 2006), pp. 445–470. ISSN: 0098-9886. DOI: 10.1002/cta.v34:4. URL: http://dx.doi.org/10.1002/cta.v34:4.

[64] L.O. Chua and L. Yang. "Cellular neural networks: applications". In: *Circuits and Systems, IEEE Transactions on* 35.10 (Oct. 1988), pp. 1273–1290. ISSN: 0098-4094. DOI: 10.1109/31.7601.

[65] A. Fernandez, R. San Martin, E. Farguell, and G.E. Pazienza. "Cellular Neural Networks simulation on a parallel graphics processing unit". In: *Cellular Neural Networks and Their Applications, 2008. CNNA 2008. 11th International Workshop on.* July 2008, pp. 208–212. DOI: 10.1109/CNNA.2008.4588679.

[66] R. Dolan and G. DeSouza. "GPU-based simulation of cellular neural networks for image processing". In: *Neural Networks, 2009. IJCNN 2009. International Joint Conference on.* June 2009, pp. 730–735. DOI: 10.1109/IJCNN.2009.5178969.

[67] *Intel microprocessor export compliance metrics.* Intel. June 2011. URL: http://www.intel.com/support/processors/sb/CS-032815.htm.

[68] *Looking Beyond Graphics, White Paper.* NVIDIA. Sept. 2009. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/T.Halfhill_LookingBeyondGraphics.pdf.

[69] Z. Nagy, L. Kek, Z. Kincses, and P. Szolgay. "CNN model on cell multiprocessor array". In: *Circuit Theory and Design, 2007. ECCTD 2007. 18th European Conference on.* Aug. 2007, pp. 276–279. DOI: 10.1109/ECCTD.2007.4529590.

[70] L. Füredi, Zoltán Nagy, András Kiss, and Péter Szolgay. "An improved emulated digital CNN architecture for high performance FPGAs". In: *NOLTA 2010. International symposium on nonlinear theory and its applications. Krakow, 2010.* Krakow: IEICE, Sept. 2010, pp. 103–106. URL: http://eprints.sztaki.hu/6378/.

[71] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H.J. Kelly. "Performance analysis of the OP2 framework on many-core architectures". In: *SIGMETRICS Perform. Eval. Rev.* 38.4 (Mar. 2011), pp. 9–15. ISSN: 0163-5999. DOI: 10.1145/1964218.1964221. URL: http://doi.acm.org/10.1145/1964218.1964221.

[72] M. B. Giles, G. R. Mudalige, and I. Reguly. *OP2 C++ User's Manual.* Dec. 2013. URL: http://www.oerc.ox.ac.uk/sites/default/files/uploads/ProjectFiles/OP2/OP2_Users_Guide.pdf.

DOI:10.15774/PPKE.ITK.2016.002

[73] *OP2 Developers Guide*. University of Oxford. Apr. 19, 2012. URL: http://www.oerc.ox.ac.uk/sites/default/files/uploads/ProjectFiles/OP2/dev.pdf.

[74] 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.

[75] Denys Dutykh, Raphaël Poncet, and Frédéric 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). Special Issue: Nearshore Hydrodynamics, pp. 598–615. ISSN: 0997-7546. DOI: 10.1016/j.euromechflu.2011.05.005. URL: http://www.sciencedirect.com/science/article/pii/S0997754611000574.

[76] K. Skaugen. *Petascale to Exascale: Extending Intel's HPC Commitment*. ISC 2010 keynote. June 2011. URL: http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.p%20df.

[77] *Texas Instruments Multi-core TMS320C66x Processor*. URL: http://www.ti.com/c66multicore.

[78] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and Haohuan Fu. "Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications". In: *Micro, IEEE* 31.2 (Mar. 2011), pp. 41–49. ISSN: 0272-1732.

[79] A. Heinecke et al. "Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor". In: *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. 2013, pp. 126–137. DOI: 10.1109/IPDPS.2013.113.

[80] Jim Rosinski. *Porting, validating, and optimizing NOAA weather models NIM and FIM to Intel Xeon Phi*. Tech. rep. NOAA, 2013.

[81] R. G. Brook, B. Hadri, V. C. Betro, R. C. Hulguin, and R. Braby. "Early Application Experiences with the Intel MIC Architecture in a Cray CX1". In: *Cray User Group (CUG), 2012*. 2012.

[82] A. Vladimirov and V. Karpusenko. *Test-driving Intel Xeon Phi coprocessors with a basic N-body simulation*. Tech. rep. Colfax International, 2013. URL: http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx.

DOI:10.15774/PPKE.ITK.2016.002

[83] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis. "Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors". In: *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on.* 2013, pp. 1085–1097. DOI: 10.1109/IPDPS.2013.44.

[84] M. Smelyanskiy et al. "Analysis and Optimization of Financial Analytics Benchmark on Modern Multi- and Many-core IA-Based Architectures". In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* 2012, pp. 1154–1162. DOI: 10.1109/SC.Companion.2012.139.

[85] Seonggun Kim and Hwansoo Han. "Efficient SIMD Code Generation for Irregular Kernels". In: *SIGPLAN Not.* 47.8 (Feb. 2012), pp. 55–64. ISSN: 0362-1340. URL: http://doi.acm.org/10.1145/2370036.2145824.

[86] Eugene L. Poole and James M. Ortega. "Multicolor ICCG Methods for Vector Computers". In: *SIAM J. Numer. Anal.* 24.6 (1987), ISSN: 00361429.

[87] E. Cuthill and J. McKee. "Reducing the bandwidth of sparse symmetric matrices". In: *Proceedings of the 1969 24th national conference.* ACM '69. New York, NY, USA: ACM, 1969, pp. 157–172. DOI: 10.1145/800195.805928. URL: http://doi.acm.org/10.1145/800195.805928.

[88] Norman E. Gibbs, William G. Poole Jr., and Paul K. Stockmeyer. "A Comparison of Several Bandwidth and Profile Reduction Algorithms". In: *ACM Trans. Math. Softw.* 2.4 (Dec. 1976), pp. 322–330. ISSN: 0098-3500. DOI: 10.1145/355705.355707. URL: http://doi.acm.org/10.1145/355705.355707.

[89] Norman E. Gibbs, Jr. Poole William G., and Paul K. Stockmeyer. "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix". English. In: *SIAM Journal on Numerical Analysis* 13.2 (1976), ISSN: 00361429. URL: http://www.jstor.org/stable/2156090.

[90] *Scotch and PT-Scotch.* 2013. URL: http://www.labri.fr/perso/pelegrin/scotch/.

[91] *Intel SDK for OpenCL Applications.* 2013. URL: http://software.intel.com/en-us/vcsource/tools/opencl-sdk.

[92] Joakim Beck and Serge Guillas. "Sequential design with Mutual Information for Computer Experiments (MICE): emulation of a tsunami model". In: *arXiv preprint arXiv:1410.0215* (2014).

[93] Dimitra Makrina Salmanidou, Aggeliki Georgiopoulou, Serge Guillas, and Frederic Dias. "Numerical Modelling of Mass Failure Processes and Tsunamigenesis on the Rockall Trough, NE Atlantic Ocean". In: *Proceedings of the Twenty-fifth (2015) International Ocean and Polar Engineering Conference* (2015).

DOI:10.15774/PPKE.ITK.2016.002

[94] *AMD64 Architecture Programmer's Manual Volumes 1-5*. 2013. URL: http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/%5C#manuals.