OPTIMISATION OF COMPUTATIONAL FLUID DYNAMICS APPLICATIONS ON MULTICORE AND MANYCORE ARCHITECTURES

IOAN CORNELIU HADADE

Rolls-Royce Vibration UTC
Department of Mechanical Engineering
Imperial College London

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy of Imperial College London and the Diploma of Imperial College

DECLARATION OF ORIGINALITY

The work presented herein is based on the research carried out by the author at the Rolls-Royce Vibration UTC, part of the Department of Mechanical Engineering at Imperial College London. The author considers it to be original and his own unless where explicitly stated otherwise.

COPYRIGHT DECLARATION

ABSTRACT

This thesis presents a number of optimisations used for mapping the underlying computational patterns of finite volume CFD applications onto the architectural features of modern multicore and manycore processors. Their effectiveness and impact is demonstrated in a block-structured and an unstructured code of representative size to industrial applications and across a variety of processor architectures that make up contemporary high-performance computing systems.

The importance of vectorization and the ways through which this can be achieved is demonstrated in both structured and unstructured solvers together with the impact that the underlying data layout can have on performance. The utility of auto-tuning for ensuring performance portability across multiple architectures is demonstrated and used for selecting optimal parameters such as prefetch distances for software prefetching or tile sizes for strip mining/loop tiling. On the manycore architectures, running more than one thread per physical core is found to be crucial for good performance on processors with in-order core designs but not required on out-of-order architectures. For architectures with high-bandwidth memory packages, their exploitation, whether explicitly or implicitly, is shown to be imperative for best performance.

The implementation of all of these optimisations led to application speed-ups ranging between 2.7X and 3X on the multicore CPUs and 5.7X to 24X on the manycore processors.

"For from *Him* and through *Him* and to *Him* are all things. To *Him* be the glory forever. Amen."

— Romans 11:36

To my son,
Isaac

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Luca di Mare, for giving me the opportunity to undertake this PhD and for welcoming me into his research group. Despite his very busy schedule, Luca's door was always open whenever I encountered difficulties in my work or needed any sort of advice. Furthermore, he always encouraged me to pursue my own direction in research and made sure there was always funding and opportunities for further training for which I am thankful.

I would also like to thank Peter Higgs for keeping the group running smoothly by organising travel and hotels for conferences or purchasing servers and whatever else was required. I can't imagine how my PhD experience would have been without him.

I have been very fortunate to work alongside brilliant colleagues such as (in no particular order): Feng Wang, Edouard Minoux, Ahmad Nawab, Mauro Carnevale, Max Rife, Fernando Barbarossa and Gan Lu. I have come to regard all of you as family and genuinely dread the moment when each of us will go their own way.

Besides my wonderful colleagues, I have also enjoyed spending time with Teddy Szemberg O'Connor, Maria Esperanza Barrera Medrano, Nefeli Dimela and Nicolas Alferez and will most certainly miss our entertaining discussions over lunch in the Senior Common Room.

Special thanks go to all of my Romanian friends: Emanuel Pop, Paul Pop, Remus Pop, Patric Fülop, Delia Babiciu, Oana Puscas, Radu Oprescu, Ozgür Osman, Oana Marin and Adrian Draghici. You have certainly brightened up my life and made both Edinburgh and London feel a little bit closer to home.

I am also very much indebted to Christopher Dahnken, Dheevatsa Mudigere and Michael Klemm of Intel Corporation who have helped me out over the years with various things ranging from compiler intrinsics to OpenMP intricacies, Agner Fog for the support provided in using the VCL library as well as his incredibly helpful optimisation manuals and Timothy Jones from the University of Cambridge for help and discussions on software prefetching. The

work herein was also shaped by the discussions and the advice received from Paolo Adami of Rolls-Royce plc for which I am very grateful.

Many thanks go to my examiners, Paul Kelly of Imperial College London and Graham Pullan from the University of Cambridge for their constructive and excellent feedback that has shaped and significantly improved the final version of this manuscript. I have thoroughly enjoyed our discussions which enabled me to identify strengths and weaknesses of my work I had not previously considered.

My biggest thanks go to my wife, Iulia, and to my loving parents, Ioan and Cornelia Hadade as well as my "adoptive" UK parents, Melanie and Kevin Smith. I am immensely blessed to have you in my life and look forward to spending more time with all of you and less time debugging code.

This research was supported by the Engineering and Physical Sciences Research Council and Rolls-Royce plc through the Industrial CASE Award 13220161.

DISSEMINATION

Parts of the work presented in this thesis have been disseminated to a wider audience through peer-reviewed publications and oral presentations. These are listed below and are up to date as of April 2018.

JOURNAL PAPERS

[1] Ioan Hadade and Luca di Mare. "Modern multicore and manycore architectures: Modelling, optimisation and benchmarking a multiblock CFD code". In: *Computer Physics Communications* 205 (2016), pp. 32 –47. ISSN: 0010-4655. DOI: http://doi.org/10.1016/j.cpc.2016.04.006.

[2] Ioan Hadade, Feng Wang, Mauro Carnevale and Luca di Mare. "Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures". Computer Physics Communications (accepted).

CONFERENCE PAPERS

[1] Ioan Hadade and Luca di Mare. "Exploiting SIMD and Thread-Level Parallelism in Multiblock CFD". In: *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings*. Springer International Publishing, 2014, pp. 410–419. ISBN: 978-3-319-07518-1. DOI: 10.1007/978-3-319-07518-1_26.

ORAL PRESENTATIONS

[1] Ioan Hadade and Luca di Mare. "Turbomachinery CFD on Modern Multicore and Manycore Architectures". SIAM Conference on Parallel Processing for Scientific Computing, Paris, France, April 12-15. 2016. URL: http://www.siam.org/meetings/pp16/.

[2] Ioan Hadade and Luca di Mare. "Turbomachinery CFD on Modern Multicore and Manycore Architectures". 10 years of HPC at Imperial, London, UK, July 7th. 2016.

[3] Ioan Hadade and Luca di Mare. "Modern multicore and manycore architectures: Modelling, optimisation and benchmarking a multiblock CFD code". UK Turbulence Consortium Annual Review and First UKFN SIG "Multicore and Manycore Algorithms to Tackle Turbulent flow" Meeting, London, UK, Sep 5th. 2017.

CONTENTS

1 INTRODUCTION 24
1.1 Preamble . . . 24
1.2 Thesis Contributions . . . 25
1.3 Thesis Outline . . . 27
2 BACKGROUND 29
2.1 Trends in processor design . . . 29
2.1.1 Moore's Law . . . 29
2.1.2 Frequency and Power . . . 31
2.1.3 Multicore and manycore architectures . . . 32
2.1.4 Levels of parallelism . . . 33
2.1.5 Amdahl's Law . . . 39
2.1.6 The memory hierarchy . . . 40
2.2 The performance gap . . . 43
2.3 Implications for CFD codes . . . 45
2.4 Conclusions . . . 48
3 EXPERIMENTAL SETUP 49
3.1 Processor Architectures . . . 49
3.1.1 Intel Xeon Sandy Bridge . . . 49
3.1.2 Intel Xeon Haswell . . . 50
3.1.3 Intel Xeon Broadwell . . . 51
3.1.4 Intel Xeon Skylake Server . . . 51
3.1.5 Intel Xeon Phi Knights Corner . . . 52
3.1.6 Intel Xeon Phi Knights Landing . . . 53
3.2 The Roofline Model . . . 55
3.3 Conclusions . . . 59
4 OPTIMISATION OF BLOCK-STRUCTURED MESH APPLICATIONS 60
4.1 Introduction . . . 60
4.2 Related Work . . . 60
4.3 Background . . . 61
4.3.1 Governing Equations . . . 62
4.3.2 Spatial discretization . . . 63
4.3.3 Time integration . . . 65

4.3.4 Computational kernels . . . 66
4.3.5 Configuration of compute nodes . . . 68
4.4 Optimisations . . . 69
4.4.1 Flow variable update . . . 69
4.4.2 Flux computations . . . 72
4.5 Results and Discussions . . . 80
4.5.1 Flow variable update . . . 80
4.5.2 Flux computations . . . 84
4.5.3 Performance modelling . . . 86
4.5.4 Architectural comparison . . . 90
4.6 Conclusions . . . 91
5 OPTIMISATION OF UNSTRUCTURED MESH APPLICATIONS 93
5.1 Introduction . . . 93
5.2 Related Work . . . 95
5.3 Background . . . 96
5.3.1 Governing Equations . . . 97
5.3.2 Spatial Discretization . . . 97
5.3.3 Time integration . . . 99
5.3.4 Test case . . . 100
5.3.5 Computational kernels . . . 100
5.3.6 Configuration of compute nodes . . . 102
5.4 Optimisations . . . 103
5.4.1 Grid renumbering . . . 103
5.4.2 Vectorization . . . 104
5.4.3 Colouring . . . 108
5.4.4 Array of Structures . . . 108
5.4.5 Gather Scatter Optimisations . . . 112
5.4.6 Array of Structures Structure of Arrays . . . 113
5.4.7 Loop tiling . . . 118
5.4.8 Software Prefetching . . . 119
5.4.9 Multithreading . . . 120
5.5 Results and Discussions . . . 122
5.5.1 Effects of optimisations . . . 122
5.5.2 Performance scaling across a node . . . 136
5.5.3 Performance modelling . . . 143
5.5.4 Architectural Comparison . . . 147

5.6 Conclusions 149
6 CONCLUSIONS 154
6.1 Summary of findings 154
6.2 Advice for CFD practitioners 157
6.3 Limitations 159
6.4 Future work 159

BIBLIOGRAPHY 161

Appendix 171

A APPENDIX 172
A.1 Results of auto-tuning prefetch distances 172

LIST OF FIGURES

Figure 1 Trends in microprocessors over the past 46 years. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. Data between 2010-2015 collected by K. Rupp. Latest data between 2015-2017 collected and plotted by I. Hadade. 30

Figure 2 Schematics of the execution units in the Intel Sandy Bridge and Haswell microarchitectures. Updates in the Haswell architecture compared to Sandy Bridge are highlighted in orange. Figures courtesy of [82] 35

Figure 3 Comparison between peak floating point performance in double precision and peak memory bandwidth across Intel Xeon multicore CPUs (CPUs), NVIDIA Tesla GPUs (GPUs) and Intel Xeon Phi processors (Phis) spanning the last decade. Data courtesy of K. Rupp [77] with minor updates by I. Hadade 42

Figure 4 Number of floating point operations in double precision per byte from off-chip memory across Intel Xeon multicore CPUs, NVIDIA Tesla GPUs and Intel Xeon Phi processors spanning the last decade. Data courtesy of K. Rupp [77] with minor updates by I. Hadade 44

Figure 5 Example of rooflines for computational nodes based on the processor architectures presented in Section 3.1 . . . 58

Figure 6 Computational domain consisting of five blocks and corresponding stitch lines. 62

Figure 7 Schematic of the cell-centred finite volume scheme on structured grids. 67

Figure 8 Vector load operation at aligned boundary for left state and load operation at unaligned boundary for right state. 74

Figure 9 Side effects of unaligned vector load operations leading to cache line split loads. 75

Figure 10 Vector load operation at aligned boundary for both left and right states followed by inter-register shuffles for correct positioning of operands. 76
Figure 11 Effects of optimisations for the kernel performing the flow variable update across different grid sizes and running on a single-core. Results in GFLOPS on KNC 5110P for the reference runs on grid sizes 10^5 and 10^6 are 0.03 and 0.02. The symbol `(u)` stands for unaligned memory access. 82
Figure 12 Performance strong scaling of kernel executing the flow variable update on 10^6 grid cells. The symbol `(u)` stands for unaligned memory access while entries that include the symbol `(N)` implement NUMA optimisations (i.e. first touch). Results of runs with NUMA (N) optimisations are only applicable to the multicore CPUs and not to KNC. 83
Figure 13 Effects of optimisations on the kernel performing the flux computations on different grid sizes. Results in GFLOPS on KNC 5110P for the reference runs on grid sizes 10^5 and 10^6 are 0.05 and 0.04. The symbol `(u)` stands for unaligned memory access. 85
Figure 14 Performance strong scaling of kernel executing the flux computations as a result of optimisations on 10^6 grid cells. The symbol `(u)` stands for unaligned memory access. 87
Figure 15 Roofline visualisation of optimisations for single-core and full-node configurations. Key: ● Reference ● Vectorization (OMP) ● Vectorization (Shuffle) ● Memory Opt. 89
Figure 16 Schematic of the cell-centred finite volume scheme on unstructured grids. 98
Figure 17 Unstructured mesh of intake with hexahedra elements at near wall regions and prisms in the free stream domain. 101
Figure 18 Solution of ground vortex ingestion highlighting the vorticity in the normal to the ground direction. 101

Figure 19 Example of reducing the bandwidth (β) via Reverse Cuthill Mckee on `mesh 1` ($3x10^6$ elements). Figure (a) highlights the initial bandwidth where $\beta = 2808603$ while figure (b) presents the improved bandwidth where $\beta = 52319$ as a result of renumbering. 104
Figure 20 Structure of Arrays (SoA) layout for cell-centred variables such as unknowns. 110
Figure 21 Array of Structures (AoS) layout for cell-centred variables such as unknowns including padding. 111
Figure 22 On the fly transposition from AoS to SoA of the unknowns on Advanced Vector Extenstions (AVX)/Advanced Vector Extensions 2 (AVX2) architectures. Corresponding compiler intrinsics can be seen in listing 13. 117
Figure 23 AoSSoA layout for storing the normals to the face in three dimensions. 118
Figure 24 Effects of optimisations on the performance of face-based loops (flux computations) on the Sandy Bridge (SNB E5-2650) system. Results are reported as GFLOPS and Speed-up and were collected by running the application on 1 MPI rank (i.e. single-core). 123
Figure 25 Effects of optimisations on the performance of face-based loops (flux computations) on the Broadwell (BDW E5-2680) system. Results are reported as GFLOPS and Speed-up and were collected by running the application on 1 MPI rank (i.e. single-core). 124
Figure 26 Effects of optimisations on the performance of face-based loops (flux computations) on the Skylake (SKL Gold 6140) system. Results are reported as GFLOPS and Speed-up and were collected by running the application on 1 MPI rank (i.e. single-core). 125
Figure 27 Effects of optimisations on the performance of face-based loops (flux computations) on the Knights Corner (KNC 7120P) system. Results are reported as GFLOPS and Speed-up and were collected by running the application on 1 MPI rank (i.e. single-core). 126

Figure 28 Effects of optimisations on the performance of face-based loops (flux computations) on the Knights Landing (KNL 7210) system. Results are reported as GFLOPS and Speed-up and were collected by running the application on 1 MPI rank (i.e. single-core) and in Quadrant/Cache mode. 127
Figure 29 Effects of optimisations on the performance of cell-based loops (update to primitive variables) reported as GFLOPS. Results were collected by running the application on 1 MPI rank (i.e. single-core). The Knights Landing (KNL 7210) system was configured as Quadrant/Cache. 128
Figure 30 Effects of optimisations on the performance of cell-based loops (update to primitive variables) reported as speed-up relative to the reference implementation. Results were collected by running the application on 1 MPI rank (i.e. single-core). The Knights Landing (KNL 7210) system was configured as Quadrant/Cache. 129
Figure 31 Effects of optimisations on the average time per solution update within a Newton-Jacobi iteration obtained through each optimisation. Results were collected by running the application on 1 MPI rank (i.e. single-core). The Knights Landing (KNL 7210) system was configured as Quadrant/Cache. 130
Figure 32 Effects of optimisations on the speed-up per solution update within a Newton-Jacobi iteration obtained through each optimisation relative to the reference implementation. Results were collected by running the application on 1 MPI rank (i.e. single-core). The Knights Landing (KNL 7210) system was configured as Quadrant/Cache. 131
Figure 33 Performance strong scaling of face-based loops (flux computations) on the two-socket multicore nodes. 139
Figure 34 Performance strong scaling of face-based loops (flux computations) on the Intel Xeon Phi Knights Corner 7120P co-processor. 140

Figure 35 Performance strong scaling of face-based loops (flux computations) on the Intel Xeon Phi Knights Landing 7210 processor. 141
Figure 36 Peformance strong scaling of cell-based loop kernel (updates to primitive variables) on all processors. 142
Figure 37 Multicore rooflines of single core and full node configurations for both classes of computational kernels. Key: ● Reference ● Vectorization ● Gather/Scatter ● Sw. Prefetch 145
Figure 38 Manycore rooflines of single core and full node configurations for both classes of computational kernels. Key for KNC 7120P system and 1 Core KNL 7210: ● Reference ● Vectorization ● Gather/Scatter ● Sw. Prefetch ● Multi-threading. Key for KNL 7210 (64 cores): ● Reference Cache ● Opt DDR ● Opt Cache ● Opt MCDRAM ● Opt MCDRAM & DDR 146
Figure 39 Architectural comparison of the performance of face-based loops (flux computations) represented as GFLOPS at single core and full node concurrency and across all architectures (higher is better). The results presented for Knights Landing were obtained by at full concurrency by interleaving the memory access between DDR and MCDRAM (i.e. DDR+MCDRAM) 150
Figure 40 Architectural comparison of the performance of cell-based loops (update to primitive variables) represented as GFLOPS at single core and full node concurrency and across all architectures (higher is better). The results presented for Knights Landing were obtained by at full concurrency by interleaving the memory access between DDR and MCDRAM (i.e. DDR+MCDRAM). . . 151

Figure 41 Architectural comparison of the performance of the whole application represented as time per Newton-Jacobi iteration at single core and full node concurrency and across all architectures (lower is better). The results presented for Knights Landing were obtained by at full concurrency by interleaving the memory access between DDR and MCDRAM (i.e. DDR+MCDRAM). 151

LIST OF TABLES

Table 1 Evolution of Single Instruction Multiple Data (SIMD) in Intel processors 37

Table 2 Hardware and software configuration of the compute nodes used in this chapter. The SIMD Instruction Set Architecture (ISA) represents the latest vector instruction set architecture supported by the particular platform. 69

Table 3 Computational kernels representing face-based and cell-based loops used for demonstrating the implementation and impact of our optimisations. 102

Table 4 Hardware and software configuration of the compute nodes used in this chapter. The SIMD ISA represents the latest vector instruction set architecture supported by the particular platform. 102

Table 5 Jump in indices for `mesh 1` after reordering the faces based on the two algorithms in Löhner et al [49]. 109

Table 6 Comparison between time to solution update measured in seconds of baseline and best optimised implementation on the multicore CPUs. 144

Table 7 Comparison between time to solution update measured in seconds of baseline and optimised versions with 4 hyperthreads and no software prefetching, software prefetching and no hyperthreading and both software prefetching and 4 hyperthreads on the Intel Xeon Phi Knights Corner coprocessor. The speed-up is calculated relative to baseline results and the timings of the best optimisation (`prefetch x 4 threads`). On this architecture, we could only perform experimental runs on `mesh 1` at full concurrency due to the 16GB memory limit. 147
Table 8 table 148
Table 9 Speed-up of flux computation kernels and whole solver (newton-jacobi) depending on prefetch distance on Sandy Bridge E5-2650 CPU. 172
Table 10 Speed-up of flux computation kernels and whole solver (newton-jacobi) depending on prefetch distance on Broadwell 173
Table 11 Speed-up of flux computation kernels and whole solver (newton-jacobi) depending on prefetch distance on Skylake 174
Table 12 Speed-up of flux computation kernels and whole solver (newton-jacobi) depending on prefetch distance on KNC with no hyperthreading 175
Table 13 Speed-up of flux computation kernels and whole solver (newton-jacobi) depending on prefetch distance on KNL quadrant cache mode 176

LISTINGS

Listing 1 i-direction sweep 68
Listing 2 j-direction sweep 68
Listing 3 Example of extra compiler hints needed for generating aligned vector load and stores on 64-byte boundaries. . . 70

Listing 4 Unaligned SIMD loads on Knights Corner 73
Listing 5 Unaligned SIMD stores on Knights Corner 73
Listing 6 Register shuffle for aligned accesses on the *i*-face sweep with AVX. 77
Listing 7 Register shuffle for aligned accesses on the *i*-face sweep with AVX2. 77
Listing 8 Register shuffle for aligned accesses on the *i*-face sweep with IMCI. 78
Listing 9 Implementation of cache blocking by nesting both i and j sweeps together. 79
Listing 10 Example of a face-based kernel. 93
Listing 11 Example of a cell-based kernel vectorized with OpenMP 4.0 directives. 105
Listing 12 Example of a SIMD friendly implementation of face-based kernels vectorized with OpenMP 4.0 directives. . . 107
Listing 13 Example of AVX/AVX2 compiler intrinsics kernel for in-register transposition from AoS to SoA of unknowns. 114
Listing 14 Example of Advanced Vector Extensions 512 (AVX-512) compiler intrinsics kernel for in-register transposition from AoS to SoA of unknowns. 115
Listing 15 Example of Initial Many Core Instructions (IMCI) compiler intrinsics kernel for in-register transposition from AoS to SoA of unknowns. 116
Listing 16 Integration of primitives for on the fly conversion between AoS and SoA data layouts in a face-based kernel. 117
Listing 17 Implementation of software prefetching in face-based loops. 121

ACRONYMS

AoS Array of Structures

AoSSoA Array of Structures Structure of Arrays

CFD Computational Fluid Dynamics

DSL Domain Specific Language

CPU Central Processing Unit

ISA Instruction Set Architecture

NASA National Aeronautics and Space Administration

SIMD Single Instruction Multiple Data

SoA Structure of Arrays

RCMK Reverse Cuthill Mckee

MPI Message Passing Interface

TLB Translation Look-aside Buffer

OpenMP Open Multi-Processing

IMCI Initial Many Core Instructions

AVX Advanced Vector Extenstions

AVX2 Advanced Vector Extensions 2

AVX-512 Advanced Vector Extensions 512

KNC Knights Corner

KNL Knights Landing

SNB Sandy Bridge

BDW Broadwell

SKL Skylake

HSW Haswell

MUSCL Monotonic Upwind Scheme for Conservation Laws

MCDRAM Multi-Channel DRAM

NUMA Non-Uniform-Memory-Access

UMA	Uniform-Memory-Access
ILP	Instruction-level parallelism
DLP	Data-level parallelism
TLP	Thread-level parallelism
FMA	Fused-Multiply-Add
SIMT	Single-Instruction-Multiple-Thread
VSX	Vector Scalar Extension
SSE	Streaming SIMD Extensions
SMT	Simultaneous Multithreading
SPMD	Single Program Multiple Data
SM	Streaming Multiprocessor
HBM2	High-Bandwidth Memory 2
HBM3	High-Bandwidth Memory 3
VPU	Vector Processing Unit
VCL	Vector Class Library
QPI	Quick Path Interconnect
BIU	Bus Interface Unit
CHA	Cache Homing Agent

1 INTRODUCTION

1.1 PREAMBLE

The current trend in processor design is parallelism and has led to the advent of multicore and manycore processors whilst effectively ending the so called "free lunch" era in performance scaling [85]. Modern multicore and manycore processors now consist of 10-100 core numbers integrated on the same die, wide vector units with associated instruction set extensions, multiple back-end execution ports along with deeper and more complex memory hierarchies. Consequently, achieving high performance on these architectures mandates the exploitation of all of the available on-chip parallelism as well as the features of the memory system.

In the context of Computational Fluid Dynamics (CFD) applications, optimising codes to make full use of the architectural features of modern processors can result in considerable speed-ups [60],[26]. However, incorporating such optimisations in existing applications is no small task. First of all, the implementation of the underlying numerical methods may have to be reconsidered in order to make them more amenable to parallelism across different granularities. Secondly, accessing certain features of modern processors may mandate the use of specialised, non-portable code. This may in turn harm performance portability across the myriad of architectures available in high-performance computing systems unless suitable abstractions are found to address this.

For the general CFD practitioner, however, the issue remains of finding a trade-off between programming effort and gains in performance for their application, both in the short and in the long term. Thus, it is essential to expose the CFD community to the whole range of techniques available and their impact on realistic applications and modern processors.

This thesis presents such optimisations across two distinct CFD codes representing both structured and unstructured paradigms and evaluates their performance on modern multicore and manycore processors.

1.2 THESIS CONTRIBUTIONS

This thesis makes the following contributions:

- Demonstrates empirically the existence of a performance gap between CFD codes that are not optimised for the current era of multicore and manycore processors and those that are, in structured and unstructured applications. Although on the multicore CPUs, this difference is in the region of 3X, on manycore processors such as the Intel Xeon Phi, the gap in performance grows to more than an order of magnitude.

- Presents techniques useful for extracting good performance out of modern multicore and manycore processors using a single body of source code written in a traditional high-level language for structured and unstructured applications. Architectural specific optimisations are abstracted away using standard language constructs and machinery such as classes, templates and pre-processor macros whilst optimal parameters are identified for different architectures via auto-tuning.

- Demonstrates the importance of making good use of the available vector units in modern processors and presents the changes required to vectorize the computational patterns found in structured and unstructured applications such as stencil operators and gather and scatter primitives. This is achieved by exploring a wide variety of techniques ranging from compiler intrinsics to OpenMP (>=4.0) compiler directives in conjunction with re-writing various sections of the code in order to expose data-level parallelism at the algorithmic level. The results in this thesis show that the best trade-off between performance and portability is obtained via the use of OpenMP directives throughout the code with specialised architectural specific implementations limited to efficiently loading data into vector registers.

- Establishes the importance of choosing the optimal data layout format and how this may vary depending on whether the partial differential equations are discretised on structured or unstructured grids. The results in this thesis show that the Structures of Arrays (SoA) or the hybrid Array of Structures Structure of Arrays (AoSSoA) format is the most optimal for structured mesh applications due to allowing for efficient vector

load and store operations. This is in contrast to unstructured mesh applications where the Array of Structures (AoS) format was found to perform better due to its superior exploitation of the cache hierarchy in computational kernels that exhibit gather and scatter operations. Thus, due to these differences, the recommendation is for applications to implement suitable abstractions of the underlying data layout format in order to seamlessly transition between different storage types at compile time.

- Demonstrates the benefit of software prefetching in unstructured mesh applications due to the indirect memory access patterns and the utility of auto-tuning in finding the optimal distance parameters across different multicore and manycore processors.
- Presents the implementation of thread-parallelism for both structured and unstructured applications on the manycore architectures. This is shown to be highly beneficial for in-order architectures that require more than one active thread per physical core to hide memory latencies and keep the arithmetic units busy although proved to be detrimental to performance on out-of-order architectures.
- Demonstrates the utility of using performance models for appraising and analysing the performance and merits of optimisations with respect to the underlying hardware and the characteristics of the computational kernels and patterns.
- The results in this thesis clearly show that optimisations that aim to exploit parallelism across different granularities in the underlying algorithms (instruction, data, thread, core) exhibit good performance across both multicore as well as manycore processors. The same also applies to optimisations that reduce the amount of data transfers from main memory and exploit the cache hierarchies or the available high-bandwidth memory implementations. However, these may not be useful or relevant for future architectures that deviate in their design from the architectural principles currently used in multicore and manycore processors. These are:
  - 10-100 cores integrated on the same die via an on-chip interconnect (ring, mesh)
  - multiple execution ports and deep instruction pipelines

- vector units with associated instruction set extensions
- multiple execution contexts per core (multithreading)
- deep memory hierarchy based on different cache levels and including high-bandwidth memory
- multiple processors per node (i.e. multi-socket boards)

1.3 THESIS OUTLINE

The structure of this thesis is as follows:

CHAPTER 2 presents the changes that have taken place in the design of processors over the past decades and highlights the shift in paradigm which has led to the advent of multicore and manycore processors. This is followed by a description of the key architectural features of modern multicore and manycore processors and on the increasing gap in performance between applications that exploit these in their implementation and those that don't. The implications that these have on the optimisation and performance of existing CFD codes is discussed together with a review of some of the approaches that have been used in literature for mapping CFD algorithms onto modern parallel architectures.

CHAPTER 3 gives the necessary background on the processor architectures used in this work in terms of their architectural features and the implications that these have on application performance. This is followed by a description of the Roofline performance model and its application in the context of this work.

CHAPTER 4 presents optimisations for improving the performance of block-structured CFD codes on modern multicore and manycore processors and demonstrates a wide range of techniques for mapping the underlying computational patterns of structured mesh solvers such as stencil operators onto the architectural features of modern processors.

CHAPTER 5 presents a number of optimisations for improving the performance of unstructured CFD codes on modern multicore and manycore pro-

cessors. Examples of optimisations include: grid renumbering, vectorization, face colouring, data layout transformations, on-the fly transpositions, software prefetching, loop tiling and multithreading. Their implementation and impact are demonstrated in an unstructured CFD code of representative size and complexity to an industrial application and across a wide range of processor architectures such as the Intel Sandy Bridge, Broadwell and Skylake multicore CPUs and the Intel Xeon Phi Knights Corner and Knights Landing manycore processors.

CHAPTER 6 discusses the achievements of this thesis, its limitations and future work.

2

BACKGROUND

This chapter begins by presenting the changes which have taken place in the design of processors over the past decades and highlights the shift in paradigm that has led to the advent of multicore and manycore processors. This is followed by a discussion on the key architectural features of modern multicore and manycore processors and on the increasing gap in performance between applications that are able to exploit their features and those that are not. Finally, the implications with respect to the optimisation and performance of CFD codes are also discussed as well as some of the approaches that can be used for mapping CFD algorithms onto modern hardware.

2.1 TRENDS IN PROCESSOR DESIGN

Figure 1 presents trends in microprocessor characteristics over the past five decades based on data from all major Central Processing Unit (CPU) manufacturers such as Intel, AMD and IBM. These are discussed in more detail in the following sections along with the effects they have on the design of modern processors and the constraints they place on the performance of applications that run on them.

2.1.1 *Moore's Law*

Moore's law [56] is based on observations made by Gordon Moore in 1965 that the number of components (i.e. transistors) per integrated circuit will double every year as a result of continuous improvements in the fabrication of such devices. He subsequently revised the pace of transistor doubling a decade later to every two years.

The semiconductor industry has kept these predictions alive over the past decades as seen in Figure 1 translating the constant flux of additional transistors into optimisations and architectural features which made new processors significantly faster and more efficient than their predecessors. Although there

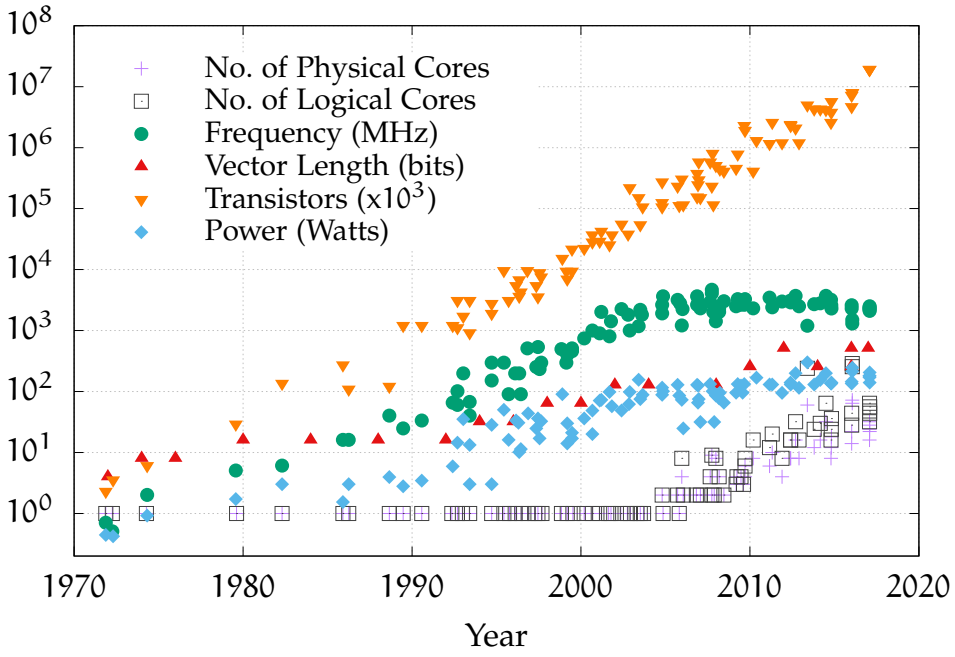


Figure 1: Trends in microprocessors over the past 46 years. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. Data between 2010-2015 collected by K. Rupp. Latest data between 2015-2017 collected and plotted by I. Hadade.

are signs that the rate at which the number of transistors is doubled is slowing down [27] as demonstrated by the increasing gap between successive generations of processors that feature smaller transistors, it is believed that Moore's law will still survive the next decade by extending photolithography into the third dimension [81] as well as by revising the pace of exponential growth to every three to five years. Therefore, in the short to medium term, it is expected that Moore's law will continue to provide chip designers with an increase in the number of transistors at their disposal for the design of processors that will make up the high-performance computing systems of the next decade.

2.1.2 *Frequency and Power*

For many years, Moore's law was accompanied by Dennard scaling [23]. Robert Dennard demonstrated in 1974 the proportional relationship between the supply of voltage and current and the linear dimensions of a transistor. Therefore, as the size of a transistor shrunk, so did the required voltage and current. This allowed for circuits to operate at higher frequencies for the same power and thus led to the increases in clock frequencies that we see in Figure 1. The trend of frequency scaling continued until 2004 and reached its peak during the 1990s when clock rates doubled on average every 18 months. The combination of extra transistors on the back of Moore's law coupled with the continuous increase in clock frequencies at which they operated allowed chip designers to double the performance of processors with every new generation. The majority of hardware innovations at that time were based on increasing instruction throughput in order to exploit the high clock rates such as complex branch predictors, out of order execution and deeper instruction pipelines. Consequently these improvements would automatically translate into faster execution of applications while requiring no changes to the programming paradigms. As performance and the number of transistors increased at a similar rate over this period, this led people to equate Moore's law with ever increasing performance [90].

However, Dennard scaling failed to predict that as transistors shrink to smaller and smaller scales, it becomes increasingly challenging to offset the increase in current leakage as well as dissipating heat efficiently without affecting the integrity of the device. Consequently, the increase in clock frequencies was

deemed to be no longer profitable as evidenced by the plateau for both power and frequency in Figure 1 from circa 2004 onwards.

2.1.3 *Multicore and manycore architectures*

The end of clock frequency scaling due to power consumption has forced the semiconductor industry into a shift in paradigm. This has led to the advent of multicore processors where instead of utilising the additional transistors guaranteed by Moore's law to build a single monolithic processor, they were used to replicate and integrate multiple cores running at a lower frequency on the same die. The advantages of the multicore approach are based on the fact that reducing the clock frequency of a single core by 20% results in 50% less power consumption at a cost of a 13% drop in performance [75]. Consequently, if one were to divide the work equally among two processors both of which operate at 80% of the original frequency, the performance would be 73% higher than on a single core for the same power usage. Moreover, dissipating the heat across multiple discrete cores is also more efficient than across a single one. As a consequence, from 2004 onwards (Figure 1), hardware designers started integrating more and more physical cores on the same die, a trend that continues to this day where a multicore CPU can implement as many as 28 physical cores per chip [3].

However, the problem with multicore processors is that translating their superior performance and energy efficiency into palpable application acceleration mandates the explicit exploitation of parallelism. Whereas the previous paradigm of increasing clock frequencies and improving serial performance required modest code interventions on the application-side, the multicore trend only favours parallel applications to the exclusion of serial ones. As a result, the burden is placed on the shoulders of application developers to revisit and rethink their existing implementations so as to fully exploit the available parallelism as more and more cores are integrated unto a die [66].

This burden is further exacerbated by the emergence of manycore processors. These architectures push the boundaries of the multicore approach even further by integrating tens to hundreds of cores on the same die. This integration is made possible by reducing the clock rates of cores at more than half the frequency of their multicore counterpart and by discarding architectural features geared towards serial performance which require a significant amount

of on-die logic [65] such as out-of-order execution or complex branch predictors in favour of more space and energy efficient features such as wide vector units. Thus, although the "slower" core in a manycore architecture will operate at a lower performance compared to the more powerful core in multicore processors, the total compute throughput of the manycore processor when all of them and their features are exploited in unison will be higher than that of the multicore system [12]. However, for this to be true in practical terms depends on whether the application can not only scale linearly with the number of cores on the device but also across all other existing levels of parallelism (i.e. thread, data and instruction). Otherwise, applications will see significantly worse performance on manycore devices than on multicore systems due to the slower cores that lack features oriented towards single thread performance. The one saving grace is that multicore and manycore processors share much of the same architectural features. As a result, optimisations for one architecture will most likely be of benefit on the other.

Thus, one returns to the issue beforehand where making effective use of both multicore and manycore processors requires the exploitation of parallelism across different granularities. Therefore, understanding what these levels are and ways through which one can exploit them will see an application perform well in the era of multicore and manycore architectures which, judging by the trends in Figure 1, is here to stay.

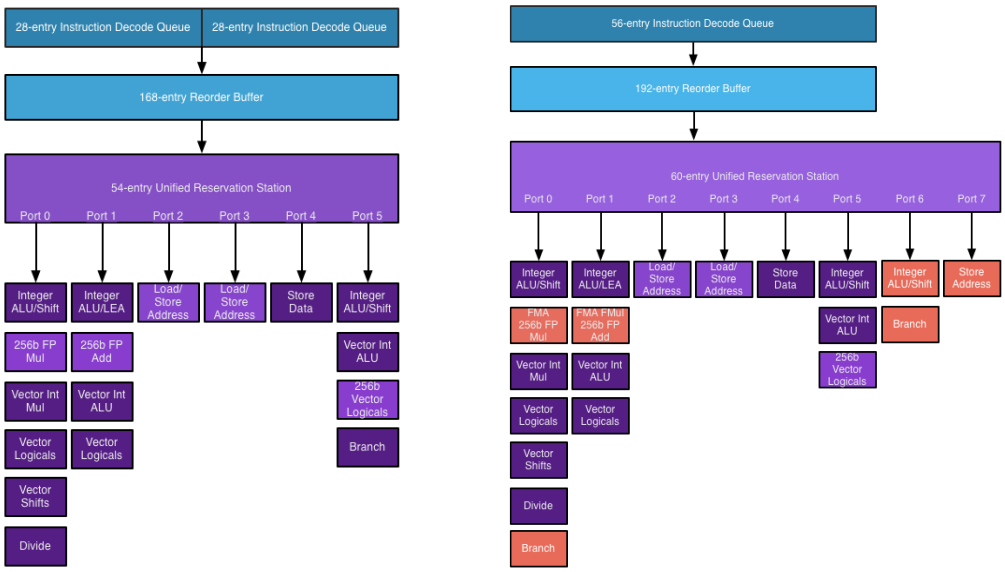
2.1.4 *Levels of parallelism*

INSTRUCTION Instruction-level parallelism (ILP) is implemented by the underlying architecture transparently via techniques such as instruction pipelining, out-of-order execution, register renaming, speculative execution or branch prediction [65]. Consequently, in the majority of cases, these features will be exploited at a lower level of abstraction than that of the program code. However, some forms of ILP can be addressed by the application depending on the choice of algorithms and their implementation. For instance, Figure 2 presents the execution units of two successive multicore architectures, namely the Intel Sandy Bridge (2011) and Haswell (2013). The changes to the execution unit that have been implemented in the newer Haswell architecture can be seen in orange. A close inspection of both reveals several things. On Sandy Bridge, in order to fully exploit the available floating point units in ports 0-1, the compu-

tations would need to exhibit a near perfect balance between multiplications and additions. Otherwise, in the case of an imbalance represented by a significantly larger number of additions over multiplications or vice-versa, one of the execution ports would be oversubscribed while the other would sit idle therefore reducing the maximum attainable performance by a factor of two. On Haswell, both ports 0 and 1 implement Fused-Multiply-Add (FMA) capabilities. As a result, two separate FMA instructions can be executed on both in parallel. However, applications that have an imbalance between multiplications and additions will also suffer from imbalance on Haswell since these won't be cast to FMA operations. Consequently, a vector multiplication would only execute on port 0 while an addition only on port 1. Another example is floating point division which is only implemented on port 0 across both microarchitectures. As these operations are non-pipelined and can only execute on 128-bit lanes [40], an algorithm that translates to a high number of divisions will degrade the performance significantly since it will prevent any other operation to utilise port 0 leading to an instruction bottleneck.

Some of the issues above can be addressed either through a different choice of algorithms or through programming techniques such as casting all multiplications or additions to FMA operations by either multiplying by a constant or zero addition. This would in theory exploit both ports on Haswell although it would lead to worse performance on Sandy Bridge due to the higher number of instructions that will execute on different ports. Moreover, these types of optimisations are not recommended since the compiler would most likely remove these transformations at certain optimisation levels. Therefore, the best approach and advice for application developers is to consider the operational balance of algorithms and where possible, implement functionality that takes advantage of parallelism in the execution units by avoiding instructions that can lead to port pressure such as floating point divisions and favouring implementations that exhibit a larger degree of parallelism (i.e. vector blend instructions vs. shuffles).

DATA Data-level parallelism (DLP) is exposed in conventional multicore and manycore architectures through the SIMD execution model. This is based on performing the same instruction be it a floating point, integer or a logical operation across successive elements of a vector simultaneously. The number of operands which can be processed by one instruction is limited to the width



(a) Intel Sandy Bridge (b) Intel Haswell

Figure 2: Schematics of the execution units in the Intel Sandy Bridge and Haswell microarchitectures. Updates in the Haswell architecture compared to Sandy Bridge are highlighted in orange. Figures courtesy of [82]

of the vector registers. On modern multicore and manycore architectures, these vector registers vary between 128, 256 and 512-bits in length and are exploited via extensions to the ISA such as AVX [30] or variants thereof on x86 processors from Intel and AMD, NEON [4] on ARM processors or Vector Scalar Extension (VSX) on the latest IBM POWER architectures.

Although the implementation of DLP through the SIMD taxonomy was popular during the 1970s in specialised high-performance computing systems such as the CRAY-1 [78], its adoption in commodity processors only came later during the end of the 1990s when Intel introduced the MMX instruction set on 64-bit wide registers. This was followed by the Streaming SIMD Extensions (SSE) instruction set which increased the width of vector registers to 128-bits and provided support for single precision floating point operations. However, subsequent improvements were based only on incremental updates while the vector register width remained fixed for over a decade. Thus, the slow pace of development as well as the lack of support for double precision floating point operations limited the usability and adoption of SIMD in scientific codes.

The turning point came in 2010 with the introduction of AVX and 256-bit vector registers in the Intel Sandy Bridge architecture. From then onwards,

the pace of improvements to SIMD capabilities in Intel architectures gained significant momentum as seen in Table 1 and led to further increases to the width of vector registers to 512-bits as well as extensions to the instruction set such as AVX2, IMCI and AVX-512.

This recent trend of increasing SIMD capabilities in modern processors stems from the consideration that vector-based architectures are more area and energy efficient than scalar-based architectures [46],[84]. Consequently, we can attribute this to the same trends that contributed to the shift to multicore processors, that of energy efficiency. However, as with multicore architectures, making best use of vector-based architectures requires the explicit exploitation of significant amounts of parallelism in the application, albeit at at a different granularity. This is either done manually through the utilisation of assembly, compiler intrinsics or libraries implementing the former or by the compiler based on its internal heuristics or guided through directives. The conversion from a scalar to a vector implementation is called `vectorization` which, if applied automatically by the compiler with limited or no external intervention becomes known as `auto-vectorization`. In the best case scenario, the latter is always true and the compiler is able to cast the majority of instructions in the program code as vector operations. However, this is only the case in applications that exhibit regular access patterns and where the safety of vectorization can be guaranteed by the compiler. Consequently, in the majority of cases, the vectorization of any scientific code of significant size and complexity is a laborious endeavour which is made even more complicated in the presence of irregular and complex access patterns.

On specialised hardware such as NVIDIA GPUs, data parallelism is implemented through Single-Instruction-Multiple-Thread (SIMT) execution whereby all threads within a warp execute the same instruction. As a result, these architectures behave similarly to the general purpose SIMD architectures of conventional multicore and manycore processors with the only difference being the number of operands that can be processed simultaneously. Consequently, optimisations that expose a large degree of data parallelism on vectors of variable lengths will perform well across all architectures.

THREAD Thread-level parallelism (TLP) is based on the ability of modern multicore and manycore architectures to maintain architectural state for more than one execution context. In multicore architectures such as Intel CPUs,

year	vector width	instruction set	architecture
1997	64-bit	MMX	P5
1999	128-bit	SSE	P6
2001	128-bit	SSE2	NetBurst
2004	128-bit	SSE3	NetBurst
2006	128-bit	SSE4	Core
2007	128-bit	SSE4.2	Nehalem
2010	256-bit	AVX	Sandy Bridge
2013	256-bit	AVX2	Haswell
2013	512-bit	IMCI	Knights Corner
2016	512-bit	AVX-512F/CD/ER/PF	Knights Landing
2017	512-bit	AVX-512F/CD/BW/DQ/VL	Skylake Server

Table 1: Evolution of SIMD in Intel processors

TLP is implemented via the Simultaneous Multithreading (SMT) model where every physical core on the die can appear to the operating system as two logical cores. This is achieved by duplicating features that maintain execution state but not the execution units. Therefore, when two threads (i.e. execution contexts) are scheduled to run on the same physical core, if one of them is stalled due to a cache miss, its state can be saved while the other thread is given control and ownership of the execution unit assuming that its instruction operands are available. Thus, one can hide latencies incurred from dependencies in the instruction pipeline or misses in the cache hierarchy by overlapping the execution of two threads.

On manycore architectures such as the Intel Xeon Phi, each core is capable of running up to four concurrent threads. On the Knights Corner architecture, the execution of more than one thread per physical core is required in order to compensate for the in-order nature of the core where a miss in the L1 cache leads to a complete execution stall. Therefore, by running two, three or four threads concurrently, context can be switched to any thread that has data available when a miss in L1 occurs therefore circumventing a complete execution stall. In essence, TLP on the Knights Corner (KNC) coprocessor is used as a means of accomplishing out-of-order execution. Consequently, although a KNC coprocessor might integrate up to 61 physical cores, best performance mandates the exploitation of at least 120 independent execution contexts (i.e. threads).

This is similar to the approach taken by NVIDIA GPUs where the SIMT execution model utilises a number of threads grouped in warps which are ex-

ecuted in a lock-step fashion on a Streaming Multiprocessor (SM). If one group of threads waits on a memory transaction, context can be quickly switched to another one thereby hiding the latency incurred by the prior group.

The effectiveness as well as the exploitation of TLP is therefore highly dependent on the application as well as the underlying architecture. On multicore architectures that feature powerful out-of-order cores, running more than one thread per core can be detrimental to performance depending on whether the application is compute or memory bound. On the other hand, on manycore architectures, the exploitation of TLP is required for good performance due to the "slow" in-order cores. For GPUs, the SIMT model somewhat blends data and thread parallelism together however, on the Intel Xeon Phi architecture, thread parallelism differs from data parallelism and requires separate consideration such as another level of domain decomposition for grid-based applications.

CORE Parallelism at core granularity is based on the exploitation of the available physical cores that are integrated on the same die. This can be explicitly addressed on conventional multicore and manycore architectures either via exploiting the shared memory nature of multicore systems where the discrete cores of a multicore processor usually exhibit Uniform-Memory-Access (UMA) properties in respect to main memory or via the Single Program Multiple Data (SPMD) model. In the latter, copies of the same application are executed independently across the physical cores while communication among them is performed explicitly via messages using programming models such as Message Passing Interface (MPI). To complicate matters further, it is becoming common practice for scientific codes to implement both types of techniques whereby one or more cores in a multicore or manycore processor runs in a SPMD fashion whereas the rest take the form of threads spawned from the SPMD context that communicate via shared memory rather than explicit messages due to the more favourable latencies. Such programming models are defined as hybrid and are particularly encouraged on manycore architectures such as the Intel Xeon Phi due to the need of running more than one thread context per core.

The equivalent of a discrete core on GPUs differs depending on the manufacturer. For NVIDIA GPUs, this would be the SM. However, compared to the physical cores in conventional multicore and manycore systems, the functionality and execution of an SM on NVIDIA GPUs is performed exclusively by the hardware. A group of threads (warp) can be scheduled to run on any available

SM. As a result, parallelism at such level cannot be exploited explicitly by the application.

NODE Compute nodes that form the building block of contemporary high-performance systems usually host two or more multicore CPUs that are integrated on the same circuit board in separate sockets and connected together via proprietary high bandwidth interconnects. Alternatively, they can usually contain a single self-hosted manycore processor such as the Intel Xeon Phi Knights Landing or a multicore CPU as well as one or more accelerators such as GPUs or the Intel Xeon Phi KNC coprocessor due to the inability of such platforms to host their own operating system. Consequently, exploiting node parallelism can mean different things depending on the architectural composition of the node.

For heterogeneous nodes containing both a conventional multicore CPU and an accelerator or coprocessor, node parallelism can mean performing symmetric computations on both where some computations are offloaded to the accelerator while another stream of independent calculations is performed on the CPU. The difficulty in this approach is that of load balancing the execution on two distinct architectures with different levels of throughput and latencies. Another type of parallelism at the granularity of an heterogeneous node involves the exploitation of locality when more than one accelerator or coprocessor are connected to the same circuit board.

For homogeneous node architectures comprising of a number of multicore CPUs or a manycore processor, parallelism at this granularity is usually the same as that at the core granularity with the only consideration that communication via shared memory across different sockets will most likely involve Non-Uniform-Memory-Access (NUMA) effects as each socket is affine to the memory nearest to then.

2.1.5 *Amdahl's Law*

Since achieving high performance on both multicore and manycore processors mandates the explicit exploitation of parallelism across different levels, the maximum speed-up that an application can exhibit on these architectures will be limited by Amdahl's law [38],[13].

Amdahl's law [7] states that the speed-up that can be attained by an application that is executed in parallel is limited by the percentage of execution that is performed serially such that:

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

where $S(N)$ is the speed-up as a function of N processors, P the fraction of code that is executed in parallel and N the number of processors this is parallelised on. Therefore, as $N \to \infty$, $S(N) \to \frac{1}{(1-P)}$

In practical terms, the limitations imposed by Amdahl's law are that the maximum speed-up that is attainable for an application that spends 25% of its overall runtime in sequential execution is a factor of four, irrespective of the number of processors (i.e $\frac{1}{(1-0.75)} = 4$).

Consequently, applications that will perform best on multicore and manycore processors are those that can maximize P as well as make good use of the underlying architectural features. Furthermore, the limitations in speed-up imposed by Amdahl's law are applicable across all of the available levels of parallelism in modern processors where not exploiting one granularity such as data parallelism leads to a drastic limitation of the maximum attainable performance.

2.1.6 *The memory hierarchy*

A trend that is not featured in Figure 1 although as important is that of the growing disparity between the performance of the processor and that of the memory system. Wulf et al [93] observed in 1995 that although both microprocessor speeds as well as memory bandwidth grew exponentially, they did so at different exponents while the difference between them also grew at an exponential rate. The authors coined this trend as "the memory wall" and argued that if this were to continue at the same pace, it would limit the performance of almost all applications to that of the memory system, thus making future advancements in microprocessors redundant.

As a consequence, chip designers began devoting large amounts of die area and transistors for the integration of on-chip caches. This improved memory performance albeit for applications that addressed the principles of spatial and

temporal locality in their memory access patterns. Nevertheless, the integration of larger caches across deeper hierarchies combined with their exploitation by the application proved to be effective in bridging the gap in performance between the processor and the memory system.

However, with the emergence of multicore and manycore processors, the disparity between the performance of the processor and that of the memory system began to grow again. This was due to the fact that as the number of cores per processor increased, so did floating point performance as arithmetic units were replicated as well. In contrast, other components such as the number of memory channels connecting the memory system to the processor did not scale at a similar pace [57]. Consequently, although peak floating point performance almost doubled with every processor generation, the ability of the memory system to keep all cores and their arithmetic units busy with data lagged further and further behind.

This is evidenced in Figure 3 which presents a comparison between peak floating point performance and peak memory bandwidth of high-end Intel multicore CPUs, NVIDIA Tesla GPUs and Intel Xeon Phi processors released between 2007 and 2017. The highest imbalance between floating point performance and memory bandwidth is seen in the Intel multicore CPUs. We can explain the reasons behind this by analysing and comparing the last two processors which were released in 2016 and 2017 respectively. These are represented in Figure 3 by the 22-core Intel Broadwell E5-2699 v4 processor and its direct successor, the 28-core Intel Skylake Platinum 8180 processor. Although the latter has more than double the floating point performance of its predecessor due to AVX-512 and the increase in size of vector registers from 256 to 512-bits as well as six additional cores, the number of memory channels between the two architectures only increased by 50% from the previous four to six. As a result, the increases in floating point performance on the Skylake-based processor have not been matched with a similar increase in memory bandwidth therefore leading to a higher imbalance between the arithmetic capabilities of the processor and the performance of the memory system.

On manycore architectures such as NVIDIA Tesla GPUs or Intel Xeon Phi processors, this imbalance is alleviated due to the implementation of on-package high bandwidth memory such as the Multi-Channel DRAM (MCDRAM) in the Intel Xeon Phi Knights Landing architecture. This is integrated using 3D-stacked technology and is connected to the Knights Landing die via eight high-

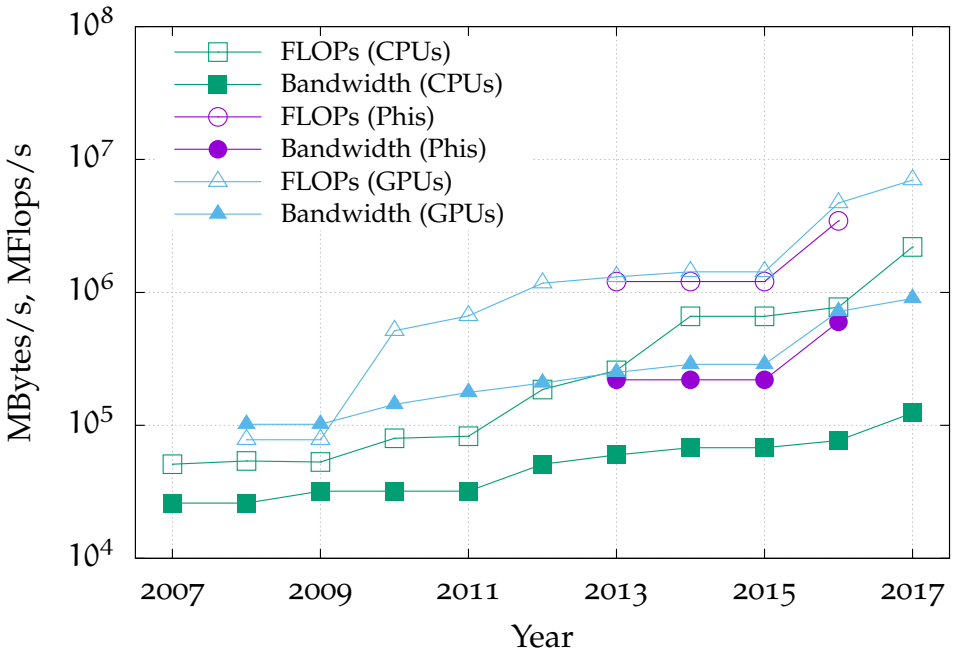


Figure 3: Comparison between peak floating point performance in double precision and peak memory bandwidth across Intel Xeon multicore CPUs (CPUs), NVIDIA Tesla GPUs (GPUs) and Intel Xeon Phi processors (Phis) spanning the last decade. Data courtesy of K. Rupp [77] with minor updates by I. Hadade

bandwidth controllers offering a factor of four more bandwidth than the traditional DDR4 memory [44]. Therefore, although the Knights Landing architecture has increased floating point performance by more than a factor of three compared to its Knights Corner predecessor, the performance of MCDRAM over the previous GDDR5 implementation in Knights Corner has scaled at a similar rate.

A similar approach to that of the Knights Landing architecture is also taken by NVIDIA Tesla GPUs with the integration of High-Bandwidth Memory 2 (HBM2) and High-Bandwidth Memory 3 (HBM3) in the P100 and V100 GPUs based on the Pascal and Volta architectures. However, an important difference between the two is that the MCDRAM implementation on the Knights Landing architecture can be configured in a variety of memory modes by the user whereas the behaviour of both HBM2 and HBM3 is hard wired on the GPUs.

A consequence of the growing disparity in performance between the processor execution speed and the bandwidth and latency of main memory is

that the number of floating point operations required to keep all of the available arithmetic units busy for every byte of data retrieved from main memory has increased considerably. This is highlighted in Figure 4. While this ratio is dependent on the machine balance and varies across architectures, it is still growing on the multicore CPUs as previously seen as well as on the NVIDIA GPUs, albeit at a different rate. The implication that this has on application performance is that algorithms that were previously bound by the computational capability of the processor, such as dense matrix-vector operations, are now limited by the performance of the memory. Consequently, techniques for improving the exploitation of the cache hierarchy and which limit the exposure of an application to the latency and bandwidth of main memory will have a high impact on modern architectures and especially on multicore processors. On the other hand, while the Intel multicore CPUs seem to suffer the most from this growing imbalance, this can be rectified by integrating on-package high bandwidth memory systems similar to the one in the Knights Landing architecture to counter balance the "memory wall". Thus, it may be only a matter of time until such implementations are also found in high-end Intel multicore CPUs. This may somewhat complicate matters for the application developer as these will potentially require explicit attention for best performance however, without them, applications will see little benefit from the continuous increases in floating point performance if these are not corroborated with increases in memory performance.

2.2 THE PERFORMANCE GAP

The multicore and manycore trend has led to processors which exhibit parallelism at various granularities along with deeper and more complex memory hierarchies. Consequently, translating the performance of these modern architectures into palpable application speed-ups mandates the exploitation of all of their architectural features.

According to Satish et al [80], this has directly contributed to the so called "ninja performance gap" where only a limited number of expert programmers are able to extract the full power of modern processors. In contrast, the average programmer can only extract a fraction of this or even worse, sees his performance drop on the more unforgiving manycore processors.

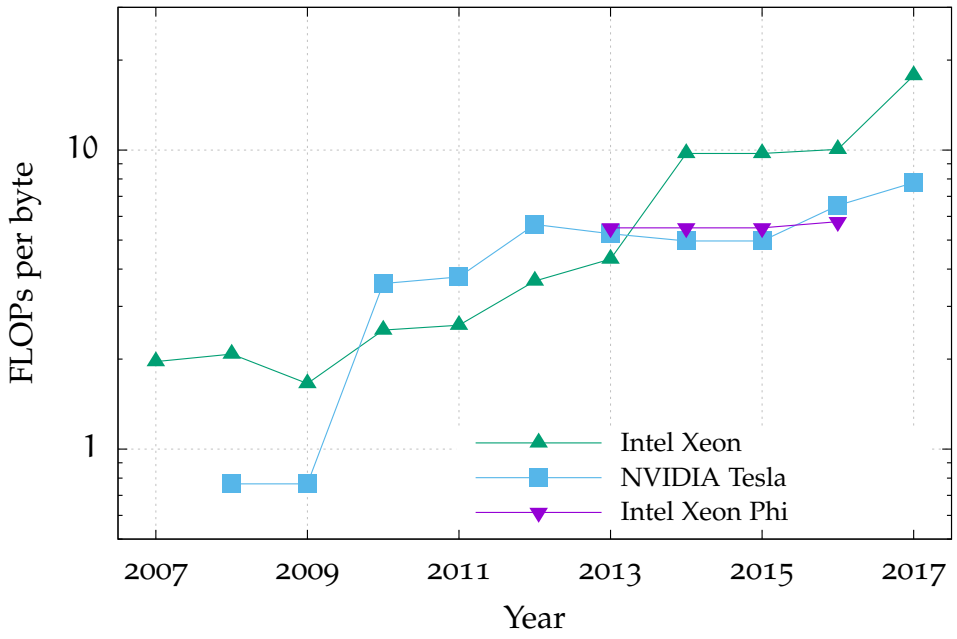


Figure 4: Number of floating point operations in double precision per byte from off-chip memory across Intel Xeon multicore CPUs, NVIDIA Tesla GPUs and Intel Xeon Phi processors spanning the last decade. Data courtesy of K. Rupp [77] with minor updates by I. Hadade

The authors refute the claim that traditional approaches to programming are at fault for this state of affairs and that the only panacea are radical alternatives such as new programming languages. Instead, they demonstrate that while a gap in performance does exist, this can be mitigated by exploiting the low hanging fruits such as vectorization via compiler directives and thread-level parallelism using application interfaces such as Open Multi-Processing (OpenMP). Through these techniques, the average difference in performance between an application that was hand tuned for the underlying architecture and its naively written counterpart dropped from the previous 24X to 3.5X. Moreover, they demonstrated that more involved algorithmic improvements such as data layout transformations for making better use of the underlying vector units as well as memory optimisations for alleviating memory bandwidth pressure via cache blocking or loop tiling have further reduced this difference to an average of 1.4X across 11 real-world applications.

Consequently, although the authors envisage that the "ninja gap" will invariably grow as emerging architectures will only improve the architectural features that can only be exploited via explicit hierarchical parallelism, they also claim that traditional approaches to programming are more than able to extract performance out of these modern processors. However, they argue that higher performance will also translate to a considerably higher programming effort. Nevertheless, without optimisations focused on exploiting the multi-level parallelism available on modern processors and the features of the memory system, scientific applications will see their execution speed stagnate or drop even as newer processors will claim increases in their peak performance. The latter will most likely be reserved exclusively for codes that are able to map to the architectural features of these processors.

2.3 IMPLICATIONS FOR CFD CODES

The majority of existing CFD codes used in both industry and academia were developed prior to the multicore and manycore era. As these codes were primarily designed for distributed-memory environments via programming models such as MPI with one rank per uniprocessor, they already exhibit parallelism at the granularity of a core or compute node. However, they do not exploit the parallelism available at instruction, vector or thread-level which, as demonstrated in the previous sections, is essential for extracting a high de-

gree of performance out of modern architectures. Furthermore, on modern processors, these codes are predominantly limited by memory bandwidth due to the growing disparity between the peak floating point performance and that of the available memory bandwidth. Consequently, optimisations that alleviate the pressure placed on the memory system such as cache blocking and loop tiling for structured grid applications or grid renumbering for unstructured grids will most likely deliver palpable speed-ups [31].

On manycore architectures such as the Intel Xeon Phi Knights Landing, exploiting the on-package MCDRAM is imperative for getting the most out of the hardware and to that extent, the processor supports a variety of configurations that are traditionally hard wired in architectures such as GPUs. Finding the best configuration therefore requires an empirical evaluation and depends on various aspects related to the application. Moreover, on manycore architectures that integrate simple in-order cores such as the Intel Xeon Phi Knights Corner and NVIDIA GPUs, hiding the latency incurred by the lack of out-of-order execution as well as complex hardware prefetchers is possible through multithreading and an optimal software prefetching strategy.

Finally, as the number of cores continue to increase in multicore and manycore processors, running one MPI rank per core might not be the best solution especially for manycore architectures and their significantly slower cores. Consequently, the implementation of hybrid parallelism with multiple threads per MPI rank could be beneficial not only for manycore architectures but also for multicore systems due to the ability of exploiting the shared memory nature that is present in multi-socket compute nodes.

Incorporating these optimisations in existing large scale CFD codes is no small task and is highly dependent on both the underlying processor architectures as well as the CFD algorithms. In the context of finite volume methods, such optimisations would vary depending on whether the governing equations are discretised on structured or unstructured meshes as these exhibit different computational patterns which mandate different optimisations. As for the underlying processor architectures, the majority of multicore CPUs from Intel, AMD, ARM and IBM share architectural similarities in that they all require the exploitation of the underlying vector units and of the memory hierarchies. As a result, the optimisations highlighted above will be transferable to a large extent among all of them unless architectural specific implementations are used. This is also applicable to the Intel Xeon Phi processors which can be

exploited using the same programming models as their multicore counterpart and where optimisations for the Xeon Phi processors are also beneficial to the multicore CPUs and vice-versa.

On the other hand, although this is also applicable to more specialised architectures such as NVIDIA GPUs that also rely on data and thread-level parallelism in their implementation albeit at larger scales, the programming models required to exploit these architectures differ significantly from those used for conventional hardware. To that extent, porting a large scale legacy CFD code to GPUs is almost synonymous to writing the entire code from scratch which, from the point of view of an industrial practitioner, is not feasible.

One approach to mitigate these portability issues is through the development of a Domain Specific Language (DSL) or high level framework. This can express the underlying algorithms in a form accessible to the CFD practitioner and generate efficient code across both conventional multicore processors as well as specialised GPU hardware. Examples of these are SBLOCK [14] developed by Brandvik et al for optimising the stencil computations arising from structured CFD mesh applications. Another example in the context of structured mesh algorithms is OPS [70] which focuses more on the implementation of optimisations for reducing the memory traffic of such algorithms via the implementation of loop tiling and cache blocking techniques as well as their mapping on both multicore as well manycore architectures such as Xeon Phi and NVIDIA GPUs. Examples of DSLs for unstructured mesh computations are OP2 [32],[59] and Liszt [22] and more specialised implementations with application to finite element codes such as FEniCS [48] and Firedrake [69] which integrates features from both the former as well as the OP2 project.

While the benefits of "writing once, run everywhere" are obvious in the context of current processor trends, and this is certainly achievable with DSL implementations such as the ones presented above, as Giles et al [31] remarked, the utilisation of DSLs also brings forth a number of challenges. First of all, there is the issue of expressiveness and whether a high level abstraction can fully capture the requirements of a CFD practitioner in an industrial or research context. Secondly, maintaining the software is challenging as many of these implementations arise from highly technical research groups with finite amounts of funding. Thirdly, there is also the issue of porting an existing large scale application to such high level implementation which, is not very different to that of re-writing the application from the very beginning.

Consequently, in the context of existing codes, the only viable approach is to revisit and optimise the underlying CFD algorithms so as to map them onto the architectural features of modern processors. For such endeavours, the optimisations and techniques presented in this thesis will act as valuable resource for establishing the return on investment in terms of programming effort over application speed-up that each particular optimisation can offer on conventional multicore and manycore processors.

2.4 CONCLUSIONS

This chapter presented current trends in the development of modern processors and the effects these have on the performance of scientific applications and CFD solvers in particular. A discussion on how best to mitigate these in CFD applications was also given which serves as the main motivation of this work along with previous efforts that have been implemented for this purpose.

3

EXPERIMENTAL SETUP

This chapter provides the necessary background on the processor architectures used in this thesis followed by a discussion on the Roofline performance model and its application in our work.

3.1 PROCESSOR ARCHITECTURES

The processor architectures described in this section and used throughout this thesis as experimental platforms have been selected because: (i) they make up a very large proportion of current and likely future node architectures in high-performance computing systems; (ii) they incorporate architectural features such as: multiple execution ports, out of order and in-order core designs, wide vector units, multi-threading capabilities, medium to large core counts as well as high-bandwidth memory implementations representative of current multicore and manycore processor designs. Consequently, presenting and evaluating optimisations that exploit any or all of these features will make them applicable for any processor architecture whether multicore or manycore that implement one or all of them in their design.

3.1.1 *Intel Xeon Sandy Bridge*

The Intel Sandy Bridge (SNB) microarchitecture was the first to introduce 256-bit vector registers and an extension to the SIMD instruction set via AVX [30]. The AVX instruction set maintains compatibility with its predecessors and is implemented on top of SSE by fusing two 128-bit SSE registers. The latter design consideration has an impact on certain operations that target data elements across 128-bit lanes such as shuffle operations and permutations. In theory, the load ports in Sandy Bridge can perform 256-bit loads via AVX, however, achieving this bandwidth requires the simultaneous usage of the two available load ports. This limitation leads to situations where vectorization might not deliver the expected performance boost due to load port pressure [40].

For arithmetic purposes, and as already mentioned in Chapter 2, the Sandy Bridge architecture can execute one vector multiplication and one vector addition in the same clock cycle thus requiring a balance of such operations for best performance.

In terms of the cache hierarchy, the L1 is 32KB and 8-way associative and can sustain two 128-bit loads and a single 128-bit store per cycle. The L2 cache is 256KB in size and 8-way associativity and a 12 cycle load-to-use latency as well as a write-back design. The L3 cache is 20MB in size and shared across the cores on the die. Integration of all physical cores on the chip is done via a ring-based interconnect [40].

3.1.2 *Intel Xeon Haswell*

The Intel Haswell (HSW) microarchitecture is based on a 22nm design and is a successor to the Ivy Bridge architecture. Whereas Ivy Bridge did not contain any major architectural changes compared to its Sandy Bridge predecessor, Haswell provides a number of modifications through a new core design, improved execution unit based on the AVX2 extensions and a revamped memory subsystem [68].

The major improvements to the execution unit in the Haswell microarchitecture regard the addition of two ports, one for memory and one for integer operations which aids instruction-level parallelism. To that end, there are two execution ports for performing FMA operations as part of AVX2 with peak performance of 16 double precision floating-point operations per cycle, double that of Sandy Bridge and Ivy Bridge. Further improvements provided by AVX2 are gather operations which are particularly useful for packing non-contiguous elements within a vector register with application to unstructured mesh calculations and whole register cross-lane permutations and shuffles with the implementation of the `vpermpd` instruction.

From a memory standpoint, the Haswell fixes the back-end port width issue by providing true 64 byte load (2x256-bits) and 32 byte store (1x256-bits) per cycle functionality. As a result, vectorization of codes that contain a large amount of load and store operations should perform significantly better than on the Sandy Bridge and Ivy Bridge architectures.

3.1.3 *Intel Xeon Broadwell*

The Intel Broadwell (BDW) microarchitecture is the successor to Haswell to which it brings a number of enhancements such as latency improvements for floating-point multiply operations and FMA as well as increased throughput of gather instructions [40]. The architecture is based on a 14 nm die shrink from the previous 22 nm on Haswell which allows high-end multicore CPUs based on the Broadwell architecture to integrate as many as 24 physical cores on the same die, six more than on Haswell.

3.1.4 *Intel Xeon Skylake Server*

The Intel Skylake (SKL) Server microarchitecture was released in July 2017 and is the successor to Broadwell [41]. At core level, Skylake Server increases the vector register size to 512-bits and adds support for AVX-512 instructions. Although the execution unit has the same number of ports as Haswell and Broadwell, port 0 and 1 can either perform AVX/AVX2 vector computations on 256-bit lanes or fused 512-bit AVX-512 computations. Port 5 is exclusive to AVX-512 execution. Therefore, to take full advantage of this architecture, vector computations should target the wider vector lanes via AVX-512 as it utilises the highest available throughput. Skylake Server can perform 32 double precision computations per cycle when utilising both AVX-512 FMA units, twice more than Broadwell and Haswell.

In terms of the cache subsystem, the L1 cache on Skylake offers similar latency and size to Broadwell, 32KB at 4-6 cycles [40]. The major difference however is based on the increase in bandwidth to 128 bytes (2x512-bits) for loads and 64 bytes (1x512-bits) for stores which are required by AVX-512 operations. In essence, the L1 cache can service up to two entire cache lines (64 bytes each) to the load ports if AVX-512 is used and the underlying data is aligned to 64 byte boundary which make these considerations crucial for achieving best performance.

Radical changes are also present in the L2 cache which sees a factor of four increase in size compared to Broadwell (1MB vs. 256KB)[40]. The L3 cache is marginally smaller in size than on Broadwell and is configured as a victim cache (non-inclusive) to the higher levels. The effect of these changes is that applications making use of communication avoiding algorithms such as loop

tiling or cache blocking should target the L2 cache rather than L3 with the same considerations applying for software prefetching.

Access to main memory can be serviced by up to 6 memory channels on Skylake Server versus 4 on Broadwell which should provide a 50% increase in available memory bandwidth. This figure has been corroborated with our own STREAM[53] benchmarks. Consequently, the difference in performance between Broadwell and Skylake Server depends on whether the application is compute or memory bound as previously discussed in Chapter 2.

Further changes are also present in regard to the the on-chip interconnect topology where the previous ring implementation is replaced with a 2D mesh interconnect which was initially implemented on the Intel Xeon Phi Knights Landing architecture [40]. This enables the Skylake Server microarchitecture to scale to as many as 28 cores per die.

3.1.5 *Intel Xeon Phi Knights Corner*

The Intel Xeon Phi KNC coprocessor can be classed as an x86-based Shared-Memory-Multiprocessor-on-a-chip [43] with more than 60 physical cores on the die each supporting four concurrent hardware threads.

A Knights Corner core contains one Vector Processing Unit (VPU) that can operate on 32 512-bit wide registers and 8 mask registers based on the IMCI extensions. The IMCI extensions however are not compatible with the SIMD extensions on x86 multicore CPUs. The functionality offered by the VPU is heavily geared towards floating point computations with support for FMA, gather and scatter operations useful for unstructured mesh computations. Theoretically, the VPU can execute one FMA operation per cycle (16 double precision floating-point operations) however, due to the in-order execution nature of the core, either software prefetching or more than one in-flight thread is required to keep the VPU busy. This is due to the fact that every miss in the L1 cache leads to a complete execution stall unless context can be switched to another in-flight thread. The cores on the die are connected via a bi-directional ring interconnect that offers full cache coherence.

Communication with the host CPU is performed via the PCI-Express bus in a similar fashion to GPUs. Although Knights Corner can be used in a number of modes such as offload, symmetric or native, the native mode has been used throughout this work as it removes any unnecessary synchronization

constructs thus enabling for a more accurate assessment of the platform's computational characteristics and performance.

3.1.6 *Intel Xeon Phi Knights Landing*

The Knights Landing (KNL) architecture is the successor of Knights Corner and second iteration of the Intel Xeon Phi family series. The Knights Landing is the first self-boot manycore processor able to run a standard operating system [44] and therefore differentiates itself among all other coprocessor and accelerator platforms. Another important distinction of the KNL architecture compared to its predecessor is binary compatibility with x86 multicore CPUs and the integration of a high bandwidth on-package memory (MCDRAM)[83] alongside standard DDR4 main memory.

The building block of the Knights Landing architecture is the tile which is replicated across the entire chip on a 2D lattice interconnect. A tile is further composed of two cores sharing a 1MB L2 cache and associated memory control agents. The KNL core is based on the Intel Atom (Silvermont) architecture and includes additional features targeting floating point workloads while also supporting up to four in-flight threads [44].

For the purpose of compute, the core integrates two VPUs each supporting AVX-512 execution. Implementation of SSE/AVX/AVX2 instructions [44] is present on one of the two vector units and not on both. Therefore, similarly to the Skylake Server architecture, full throughput can only be achieved via AVX-512 calculations as SSE/AVX/AVX2 can only utilise half of the available vector units.

Due to the out of order nature, one thread per KNL core can saturate all available core resources. The L1 cache can perform two 64 byte (2x512-bits) loads and one 64 byte (1x512-bits) stores per cycle and supports unaligned and split load accesses across multiple cache lines which was not previously possible in KNC. Furthermore, the L1 cache also implements special logic for gather and scatter operations whereby a single instruction can access multiple memory locations at once without the need of a blocking loop implementation as found in KNC [67]. This is particularly important for applications that exhibit indirect and irregular memory access patterns such as unstructured mesh solvers.

The L2 cache is shared by the two cores in a tile via the Bus Interface Unit (BIU) and has a bandwidth of 64 bytes (1x512-bits) for reads and 32 bytes (1x256-bits) for writes per cycle. This can therefore become a bottleneck for memory bound workloads when both cores issue AVX-512 vector loads and store operations.

The Knights Landing architecture can support various clustering and memory mode configurations which were traditionally hard-wired during the chip manufacturing process but are now exposed as boot-able options [44]. Clustering modes target ways in which data is routed on the 2D mesh in the event of a cache miss in the shared L2 cache [83]. An L2 miss in Knights Landing involves the interaction of three actors: the tile from where the miss originated, the Cache Homing Agent (CHA) that tracks the location and state of the memory address for which a miss was generated and the actual memory location (i.e. MCDRAM or DDR4) [44].

In the *All-to-All* configuration, all memory addresses are distributed uniformly across all CHA's without any locality considerations. This mode provides the lowest performance out of all available configurations and is usually used as fall-back in the event of memory imbalance or errors.

In the *Quadrant* configuration, the entire die is divided into four distinct regions and memory addresses are mapped to the caching agents which reside in the same quadrant. This creates affinity between the memory location and the CHA which reduces latency and provides an increase in bandwidth.

Finally, in the *Sub-Numa-Clustering* mode, the die is divided in either four or two distinct sections analogous to a four or two socket multicore CPU node where affinity exists between all agents: tile, CHA and memory. This configuration introduces NUMA considerations that have to be exploited on the application-side as accessing data owned by a tile from a different region will lead to increased latency due to the longer path traversed on the 2D mesh. However, if implemented correctly, this mode of operation can provide the best performance as all communications are localised.

With regard to memory modes, the Knights Landing architecture supports three distinct memory configurations. The first option is *Cache* mode in which the 16GB MCDRAM acts as a transparent direct mapped memory side cache. In *Flat* mode, the MCDRAM is exposed as an explicit memory buffer that must be managed by the application and which has a different address space than standard DDR4 memory. *Hybrid* mode combines both of the previous options

where MCDRAM can be used as cache and explicit high bandwidth memory based on pre-defined ratios.

In this work, experimental results for the Knights Landing processor were obtained using the ARCHER-KNL [2] evaluation platform. Consequently, all runs were performed in the *Quadrant* cluster mode as this was the global configuration across all nodes. With regard to memory modes, separate queues allowed for the evaluation of both *Cache* as well as *Flat* modes.

3.2 THE ROOFLINE MODEL

In order to determine the effects of our code optimisations rigorously, performance models are built and correlated based on the characteristics of the processor architectures presented in Section 3.1. A good performance model can highlight how well the implementation of an algorithm uses the underlying architectural features of the processor as well as the degree of performance that can still be achieved with further optimisations.

In this thesis, all performance modelling activities were performed using the Roofline [91],[92],[62] performance model. The model is based on the assumption that the principal vectors of performance in numerical algorithms are computation, communication and locality [92]. The Roofline model defines computation as number of floating point operations per second, communication as units of data from memory required by the computations and locality as the distance in memory from where the data is retrieved i.e. (cache, DRAM, MCDRAM etc). However, due to the fact that modelling the performance of the entire cache system is a complex task for one architecture, let alone six, the locality component in this work is set to DRAM main memory for the multicore CPUs, GDDR5 for the Intel Xeon Phi Knights Corner and MCDRAM for the Intel Xeon Phi Knights Landing processor.

The correlation between computation and communication is defined by the model as the arithmetic intensity of a given kernel which can be expressed as the ratio of useful floating point operations to the corresponding number of bytes requested from memory. As the units of measurement for both flops and byte transfers in current CPU architectures are given as GigaFLOPS per second and GigaBYTES per second, the maximum attainable performance of a

computational kernel, measured in GigaFLOPS per second, can be calculated as:

$$\text{Max. GFlops/sec} = \min \begin{cases} \text{Peak floating-point performance} \\ \text{Max. Memory bandwidth} \times \text{Kernel flops/bytes} \end{cases} \quad (2)$$

Peak floating point is obtained from multiplying the number of floating point operations per cycle to the nominal clock frequency together with the number of physical cores while maximum memory bandwidth is obtained using the STREAM[54],[53] benchmark.

The model visualises these metrics in a diagram [91] where the attainable performance of a kernel is plotted as a function of its arithmetic intensity. This acts as the main roofline of the model and exhibits a slope followed by a plateau when peak FLOPS is reached. The position at which the arithmetic intensity coupled with the available memory bandwidth equals peak FLOPS is called the "ridge" and signifies the transition from memory to compute bound.

Achieving either maximum memory bandwidth or peak FLOPS on modern multicore and manycore processors requires the exploitation of a plethora of architectural features even though the algorithm might exhibit a favourable arithmetic intensity. For example, achieving peak FLOPS requires a balance of additions and multiplications for fully populating all functional units or for all such operations to be cast as FMA operations on architectures that support such primitives. Moreover, even where the computational kernels exhibit a perfect balance of operations or can cast all of them into FMA instructions, they will not automatically extract peak performance if data parallelism through SIMD and instruction level parallelism through loop unrolling are not exploited as well.

Achieving performance close to the maximum memory bandwidth of the system requires that memory accesses are performed at unit stride and include some form of prefetching either by the hardware or by the software. Furthermore, on two-socket systems, extracting the maximum performance out of the memory system also requires that NUMA effects are catered for when running at full node concurrency [92],[11]. Consequently, best attainable performance will vary significantly between applications that exhibit a high degree of irregular and indirect access patterns and those with regular and unit stride memory access patterns such as structured and unstructured CFD codes.

To demonstrate this, roofline diagrams are presented in Figure 5 for the processor architectures described in Section 3.1 and which are used throughout this thesis.

The vertical ceilings in Figure 5 that are parallel to the horizontal line represented as peak floating point performance are optimisations that are required to improve computational performance. The ceilings that are parallel to the diagonal represented by peak main memory bandwidth are memory optimisations that are necessary for efficiently exploiting the memory system of that particular architecture. The order of each ceiling or roofline is set based on how likely it is for the compiler to automatically apply such optimisations without external interventions [92].

For the Sandy Bridge node, a balance between additions and multiplications is required due to the design of the execution ports where one multiplication and one addition can be performed every clock cycle as previously discussed. An imbalanced kernel will fully utilise the addition or multiplication execution unit whilst the other sits idle therefore reducing throughput by a factor of two. Furthermore, lack of SIMD computations via AVX on 256-bit vector registers further decreases performance by a factor of four from the previous ceiling in the context of double precision computations.

On the Haswell and Broadwell nodes, lack of FMA operations reduces the maximum attainable performance by a factor of four since two SIMD FMA operations can be performed in one cycle on both architectures. Lack of balance between floating point operations is also detrimental on these architectures since these can only be performed in one of the two ports, same as on Sandy Bridge. As a result, this leads to another factor of two drop in the maximum performance that is attainable. Finally, absence of AVX/AVX2 execution on Haswell and Broadwell results in a further reduction by a factor of four as a result of the same considerations as on Sandy Bridge.

On the Skylake Server architecture, lack of FMA operations has a similar effect to that on Haswell and Broadwell. However, if the wide SIMD units are not exploited via AVX-512, the maximum attainable performance is reduced further by a factor of eight due to the wide 512-bit vector registers.

The two-socket compute nodes based on the multicore CPU architectures share the same ceiling for memory optimisations such as NUMA optimisations when crossing socket boundaries followed by larger drops in performance in cases where the kernel exhibits irregular access patterns.

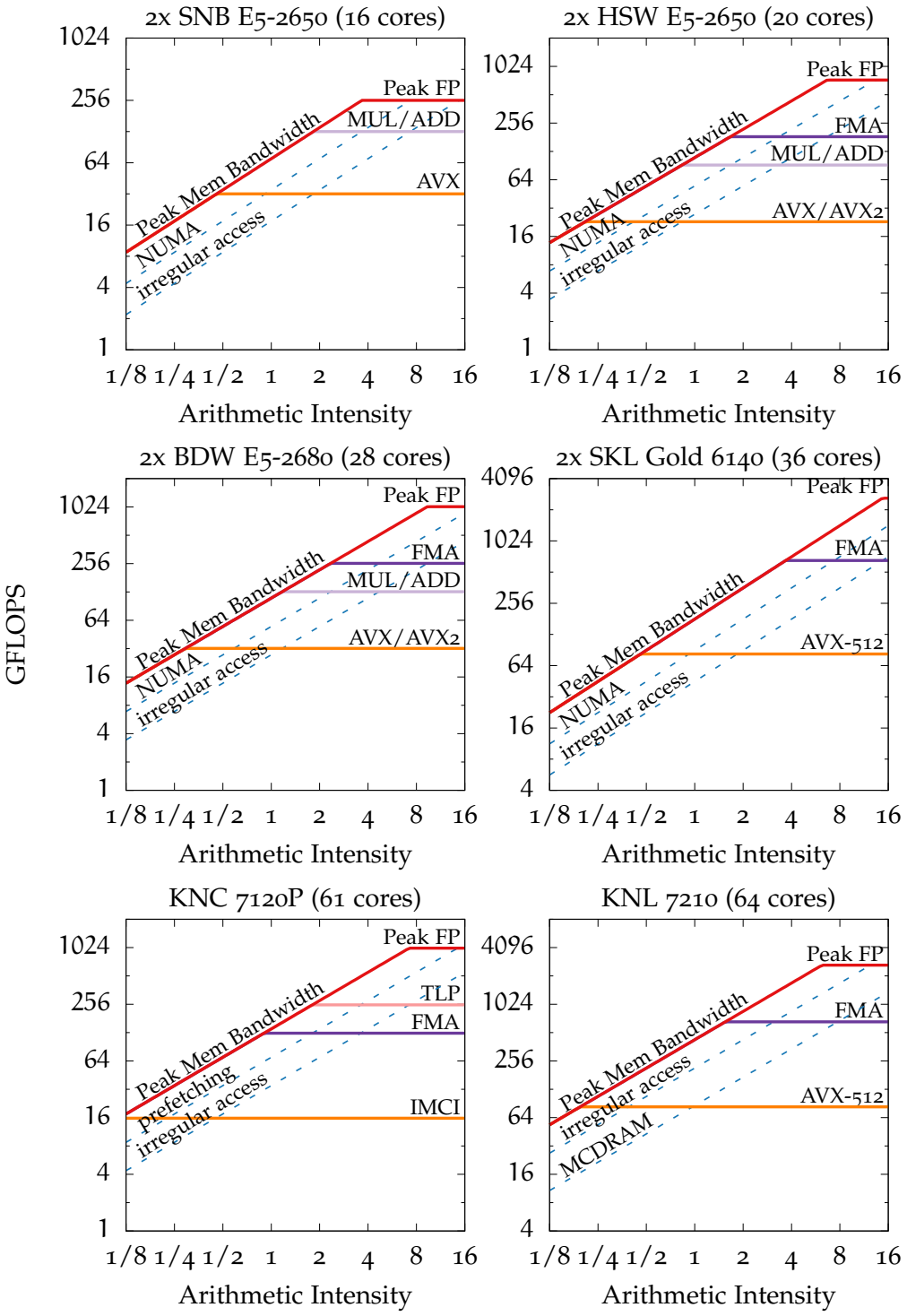


Figure 5: Example of rooflines for computational nodes based on the processor architectures presented in Section 3.1

On the Knights Corner 7120P coprocessor, the in-order cores require the use of multiple threads to compensate for lack of out-of-order execution. In theory, the VPU unit can be filled by running between two or three threads on the core. Running with only one thread in-flight reduces the attainable peak performance by a factor of four in the worst case scenario. Furthermore, similarly to Haswell and Broadwell, an imbalanced kernel that cannot fully exploit FMA operations on Knights Corner will suffer a factor of two drop in performance. Lack of SIMD results in a factor of eight decrease due to the wide 512-bit vector registers.

Since the Knights Corner architecture does not exhibit NUMA effects due to the ring topology and interleaved memory bank placement, the first ceiling for obtaining performance close to maximum memory bandwidth is prefetching. This is due to the lack of hardware prefetchers. The second ceiling is similar to the multicore architectures and represents lack of contiguous and unit stride memory access patterns which results in a factor of four drop in performance relative to the maximum attainable memory bandwidth.

On the Knights Landing architecture, the first in-core ceiling is based on lack of FMA operations. As there are two VPUs in a Knights Landing core, if the kernel's floating-point operations cannot be executed as FMA instructions, performance will drop by a factor of four. Furthermore, lack of AVX-512 leads to a factor of eight drop in performance.

With regard to memory ceilings, the Knights Landing architecture integrates both an L1 as well as L2 hardware prefetcher. As a result, the first ceiling is represented by lack of contiguous memory access patterns. The second one is based on not exploiting the MCDRAM implementation which leads to a factor of five decrease of the maximum attainable memory bandwidth.

3.3 CONCLUSIONS

This chapter presented the necessary background on the processor architectures used in this thesis followed by a discussion on the Roofline performance model and its application in this work.

4

OPTIMISATION OF BLOCK-STRUCTURED MESH APPLICATIONS

4.1 INTRODUCTION

This chapter presents a number of optimisations for improving the computational performance of block-structured CFD solvers across different processors such as the Intel Sandy Bridge and Haswell multicore CPUs and the Intel Xeon Phi Knights Corner manycore coprocessor. Code optimisations are demonstrated on two computational kernels exhibiting different computational patterns: regular and streaming access patterns represented by the update of flow variables and the stencil-based operations arising from the computation of fluxes. A discussion on the code transformations required for achieving efficient exploitation of the available vector units, threads and cores for both kernels and across the selected processors is also given while performance results are correlated with the Roofline performance model.

The remaining chapter is structured as follows. Section 4.2 discusses related work, Section 4.3 presents the test vehicle for this study, a description of the two computational kernels and the configuration of the hardware. Section 4.4 describes the code optimisations and their implementation, Section 4.5 presents results and discussions while Section 4.6 gives concluding remarks.

4.2 RELATED WORK

There have been a number of previous efforts concentrated on optimising the performance of structured grid applications on modern processors and which are also applicable to block-structured CFD codes.

Henretty et al [37] applied data transpositions for optimising the SIMD execution of stencil operations. Rostrup et al [76] demonstrated similar techniques for exploiting data parallelism in structured grid applications with the addition of implementing hand-tuned primitives for shuffling vector operands in stencil operations in order to allow for aligned vector load and store operations on

the Cell processor. Rosales et al. [74] determined the effect of data layout transformations for improving the performance of vector operations and the issues of exploiting a large degree of thread-parallelism on the manycore Intel Xeon Phi Knights Corner processor using a Lattice Boltzmann code as a test vehicle. Datta et al [21] studied the problem of modelling and auto-tuning the performance of stencil operations across a wide range of modern processors in order to exploit their distinct architectural characteristics and therefore extract a higher degree of performance portability. With regards to the latter, Mcintosh-Smith et al [55] and Curran et al [19] presented the need for a performance portable approach to the optimisation of structured grid applications. According to them, the solution to this problem is the utilisation of a framework that is compatible among the majority of manycore processors such as OpenCL [63]. They demonstrated that performance portability across a variety of structured grid codes can be achieved by re-writing the main computational kernels in the OpenCL framework. Their results indicate that good performance portability can be achieved across a wide range of manycore architectures such as NVIDIA and AMD GPUs as well as Intel Xeon Phi processors although this does differ significantly depending on the complexity of the application.

Other approaches for ensuring performance portability on both multicore as well as manycore processors is through auto-tuning as presented by Williams et al [92] or the implementation of DSLs and source to source code generators such as the already mentioned SBLOCK [14] and OPS [70] implementations.

The contribution of this chapter is the incorporation of some of the techniques presented in literature together with more novel optimisations in a block-structured solver of complexity and structure representative of industrial CFD applications. Their impact is demonstrated on two multicore CPU architectures as well as on the Intel Xeon Phi Knights Corner coprocessor and shows that performance in structured grid applications can be exploited across such architectures using traditional programming models coupled with portable APIs such as OpenMP.

4.3 BACKGROUND

The test vehicle for this study is an Euler solver used in an industrial setting for performing quick calculations of transonic turbo-machinery flows. The solver computes inviscid flow solutions in the $m' - \theta$ coordinate system [87] where θ

is the angular position around the annulus, m is the arclength evaluated along a stream surface $dm = \sqrt{dx^2 + dr^2}$ and m' is a normalized (dimensionless) curvilinear coordinate defined from the differential relation $dm' = \frac{dm}{r}$. The $m' - \theta$ system is used in turbomachinery computations because it preserves aerofoil shapes and flow angles. A typical computational domain is shown in Figure 6 and consists of a number of blocks connected by stitch lines.

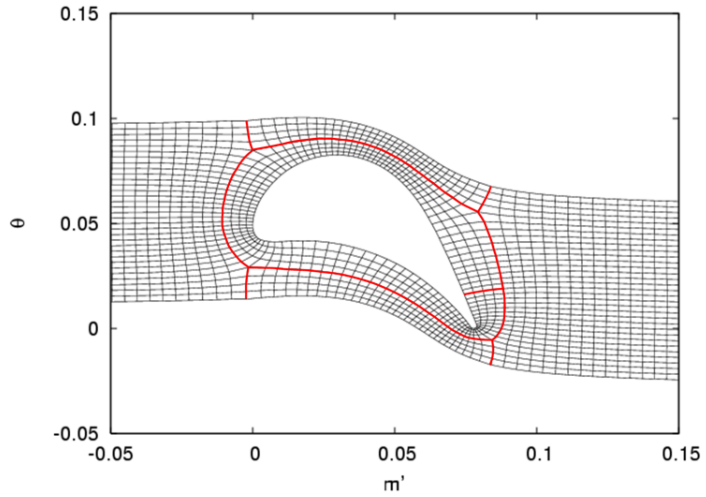


Figure 6: Computational domain consisting of five blocks and corresponding stitch lines.

4.3.1 *Governing Equations*

The Euler equations are solved in semi-discrete form

$$\frac{d}{dt} W_{i,j} \mathbf{U}_{i,j} = \mathbf{F}_{i-1/2,j} - \mathbf{F}_{i+1/2,j} + \mathbf{G}_{i,j-1/2} - \mathbf{G}_{i,j+1/2} + \mathbf{S}_{i,j} = \mathbf{RHS}_{i,j} \quad (3)$$

In equation (3), $W_{i,j}$ is the volume of cell i,j, $\mathbf{U}_{i,j}$ is the vector of conserved variables and the vectors $\mathbf{F}_{i-1/2,j}$, $\mathbf{G}_{i,j-1/2}$ and $\mathbf{S}_{i,j}$ denote fluxes through i-faces, j-faces and source terms, respectively and are evaluated as follows:

$$\mathbf{F}_{i-1/2,j} = s^{\zeta}_{i-1/2,j} \begin{bmatrix} \rho w_{\zeta} \\ \rho w_{\zeta} u_m + p n^{\zeta}_m \\ \rho w_{\zeta} u_{\theta} + p n^{\zeta}_{\theta} \\ \rho w_{\zeta} h - p w_{\zeta} \end{bmatrix}_{i-1/2,j} \tag{4}$$

$$\mathbf{G}_{i,j-1/2} = s^{\eta}_{i,j-1/2} \begin{bmatrix} \rho w_{\eta} \\ \rho w_{\eta} u_m + p n^{\eta}_m \\ \rho w_{\eta} u_{\theta} + p n^{\eta}_{\theta} \\ \rho w_{\eta} h - p w_{\eta} \end{bmatrix}_{i,j-1/2} \tag{5}$$

$$\mathbf{S}_{i,j} = W_{i,j} \rho_{i,j} \begin{bmatrix} 0 \\ u^2_{\theta} \sin\phi \\ u_m u_{\theta} \cos\phi \\ 0 \end{bmatrix}_{i,j} \tag{6}$$

For the purpose of flux and source term evaluation, the contravariant velocities $w_{\zeta/\eta}$, the normals $n^{\zeta}_{m,\theta}$ and $n^{\eta}_{m,\theta}$ and the radial flow angle ϕ are also needed.

4.3.2 *Spatial discretization*

The physical fluxes are approximated with second order TVD-MUSCL [6][39][73] numerical fluxes

$$\mathbf{F}^{*}_{i-1/2,j} = \frac{1}{2}\left(\mathbf{F}_{i-1,j} + \mathbf{F}_{i,j}\right) \quad -\tfrac{1}{2}\mathbf{R}|\mathbf{\Lambda}|\mathbf{L}\left(\mathbf{U}_{i,j} - \mathbf{U}_{i-1,j}\right) \\ -\tfrac{1}{2}\mathbf{R}|\mathbf{\Lambda}|\mathbf{\Psi}\mathbf{L}\Delta\mathbf{U}_{i-1/2,j} \tag{7}$$

The term $\mathbf{R}|\mathbf{\Lambda}|\mathbf{\Psi}\mathbf{L}\Delta\mathbf{V}_{i-1/2,j}$ represents the second order contribution to the numerical fluxes. $\mathbf{\Psi}$ is the limiter and the flux eigenvectors and eigenvalues $\mathbf{R}$,$\mathbf{\Lambda}$, $\mathbf{L}$ are evaluated at the Roe-average [73] state

$$\boldsymbol{\Psi}\mathbf{L}\Delta\begin{bmatrix}\rho\\ u_m\\ u_\theta\\ p\end{bmatrix}_{i-1/2,j} = \boldsymbol{\Psi}^{+}\mathbf{L}l_{i-1,j}\begin{bmatrix}\frac{\partial\rho}{\partial m} & \frac{\partial\rho}{\partial r\theta}\\ \frac{\partial u_m}{\partial m} & \frac{\partial u_m}{\partial r\theta}\\ \frac{\partial u_\theta}{\partial m} & \frac{\partial u_\theta}{\partial r\theta}\\ \frac{\partial p}{\partial m} & \frac{\partial p}{\partial r\theta}\end{bmatrix}^{+}_{i-1,j}\begin{bmatrix}n_m^\zeta\\ n_\theta^\zeta\end{bmatrix} + \boldsymbol{\Psi}^{-}\mathbf{L}l_{i,j}\begin{bmatrix}\frac{\partial\rho}{\partial m} & \frac{\partial\rho}{\partial r\theta}\\ \frac{\partial u_m}{\partial m} & \frac{\partial u_m}{\partial r\theta}\\ \frac{\partial u_\theta}{\partial m} & \frac{\partial u_\theta}{\partial r\theta}\\ \frac{\partial p}{\partial m} & \frac{\partial p}{\partial r\theta}\end{bmatrix}^{-}_{i,j}\begin{bmatrix}n_m^\zeta\\ n_\theta^\zeta\end{bmatrix} \tag{8}$$

where $\mathbf{L}$ is the matrix of the left eigenvectors of the Roe-averaged flux Jacobian.

$$\mathbf{L} = \begin{bmatrix}1 & 0 & 0 & -\frac{1}{\tilde{a}^2}\\ 0 & t_m & t_\theta & 0\\ 0 & n_m & n_\theta & -\frac{1}{\tilde{\rho}\tilde{a}}\\ 0 & n_m & n_\theta & -\frac{1}{\tilde{\rho}\tilde{a}}\end{bmatrix} \tag{9}$$

The evaluation of the numerical fluxes also requires the Roe-averaged eigenvalues and right-eigenvectors of the flux Jacobian, which are evaluated as follows:

$$\boldsymbol{\Lambda} = \mathrm{diag}\,(\tilde{u}_n, \tilde{u}_n, \tilde{u}_n + \tilde{a}, \tilde{u} - \tilde{a}) + \varepsilon \tag{10}$$

$$\mathbf{R} = \begin{bmatrix}1 & 0 & 1 & 1\\ 0 & t_m & (\tilde{u}_n + \tilde{a})n_m & (\tilde{u}_n - \tilde{a})n_m\\ 0 & t_\theta & (\tilde{u}_n + \tilde{a})n_\theta & (\tilde{u}_n - \tilde{a})n_\theta\\ \tilde{k} & \tilde{u}_t & \tilde{h} + \tilde{u}_n\tilde{a} & \tilde{h} - \tilde{u}_n\tilde{a}\end{bmatrix} \tag{11}$$

where ε is Harten's entropy correction [36]. The Roe-averaged state is defined by

$$\tilde{u}_m = \omega u_m^{i+1,j} + (1-\omega) u_m^{i,j} \tag{12}$$
$$\tilde{u}_\theta = \omega u_\theta^{i+1,j} + (1-\omega) u_\theta^{i,j} \tag{13}$$
$$\tilde{h} = \omega h^{i+1,j} + (1-\omega) h^{i,j} \tag{14}$$
$$\tilde{a}^2 = (\gamma-1)(\tilde{h} - \tilde{k}) \tag{15}$$
$$\tilde{\rho} = \sqrt{\rho^{i+1,j}\rho^{i,j}} \tag{16}$$
$$\omega = \frac{\sqrt{\rho^{i+1,j}}}{\sqrt{\rho^{i+1,j}} + \sqrt{\rho^{i,j}}} \tag{17}$$

Similar definitions are applied for the $G^*_{i,j-1}$ numerical flux vector.

4.3.3 *Time integration*

Convergence to a steady state is achieved by a matrix free, implicit algorithm. At each iteration, a correction to the primitive variable vector $\mathbf{V}$ is determined as solution to the linear problem [34]

$$\frac{\delta\left(W_{i,j}\mathbf{U}_{i,j}\right)}{\delta t} = \mathbf{RHS}_{i,j} + \mathbf{J}_{i,j}^{h,k}\delta\mathbf{V}_{h,k} \tag{18}$$

or, equivalently

$$(W_{i,j}\mathbf{K}_{i,j} - \mathbf{J}_{i,j}^{h,k})\delta\mathbf{V}_{i,j} = -\left(\delta W_{i,j}\right)\mathbf{IU}_{i,j} + \mathbf{RHS}_{i,j} \tag{19}$$

where $\mathbf{V}_{i,j}$ is the vector of primitive variables at the cell i,j and $\mathbf{K}_{i,j}$ is the transformation Jacobian from primitive variables to conserved variables. The linear problem in equation (19) can be approximated by a diagonal problem if the assembled flux Jacobian $\mathbf{J}_{i,j}^{h,k}$ is replaced by a matrix bearing on the main diagonal the sum of the spectral radii $|\tilde{\Lambda}|$ of the flux Jacobian contributions for each cell

$$\mathbf{J}_{i,j}^{h,k} \approx -\mathrm{diag}\left(\sum s\left|\tilde{\Lambda}\right|_{max}\right)_{i,j} \tag{20}$$

At a fixed Courant number $\sigma = \frac{\delta t_{i,j} \sum s|\tilde{\lambda}|_{max}}{W_{i,j}}$ this approximation yields the following update

$$\delta \mathbf{V}_{i,j} = \frac{1}{W_{i,j}\,(1+\sigma)} \mathbf{K}_{i,j}^{-1} \left(-\mathbf{U}_{i,j} \delta W_{i,j} + \mathbf{RHS}_{i,j} \right) \tag{21}$$

The solver has been validated against MISES [50] for turbine testcases. For the purpose of implementation, the baseline solver stores the primitive variables $\mathbf{V}_{i,j}$ at each cell as well as auxiliary quantities such as speed of sound and total enthalpy while all computations are carried out in double precision. The application is implemented as a set of C++ classes where Eqs. (3)-(21) are implemented as methods in distinct classes representing an abstraction of a gas model. These methods are called at appropriate times during application execution to compute the numerical fluxes, updates etc. The remaining elements of the application in charge of pre-processing or post-processing are handled by a different set of classes.

4.3.4 *Computational kernels*

Inspection of equations (3)-(21) reveals the existence of two types of computational kernels and patterns. The first type can be defined as a cell-based loop which evaluates cell attributes and is performed by looping over the cells in the domain. The evaluation of $\mathbf{S}_{i,j}$ in equation 3 or the block diagonal inversion in equation 21 fall within this category. The second type evaluates stencil operators and needs to be performed by looping over the neighbours of each cell or by looping over the cell boundaries. The evaluation of the numerical fluxes $\mathbf{F}_{i\pm1/2,j}$ and $\mathbf{G}_{i,j\pm1/2}$ in equation 3 is an example of such stencil kernels.

The solver spends 75% of time in computing the numerical fluxes and approximately 15% in evaluating cell-centred attributes, the most time consuming of which is the update of flow variables. Therefore, optimising both of these kernels will have the highest impact on overall application performance.

FLUX COMPUTATIONS The flux computations kernel iterates over a set of cell interfaces in separate `i` and `j` sweeps and computes numerical fluxes using Roe's approximate Riemann solver as seen in Listings 1 and 2. The kernel accepts pointers to the left and right states (`q`), to the corresponding residuals

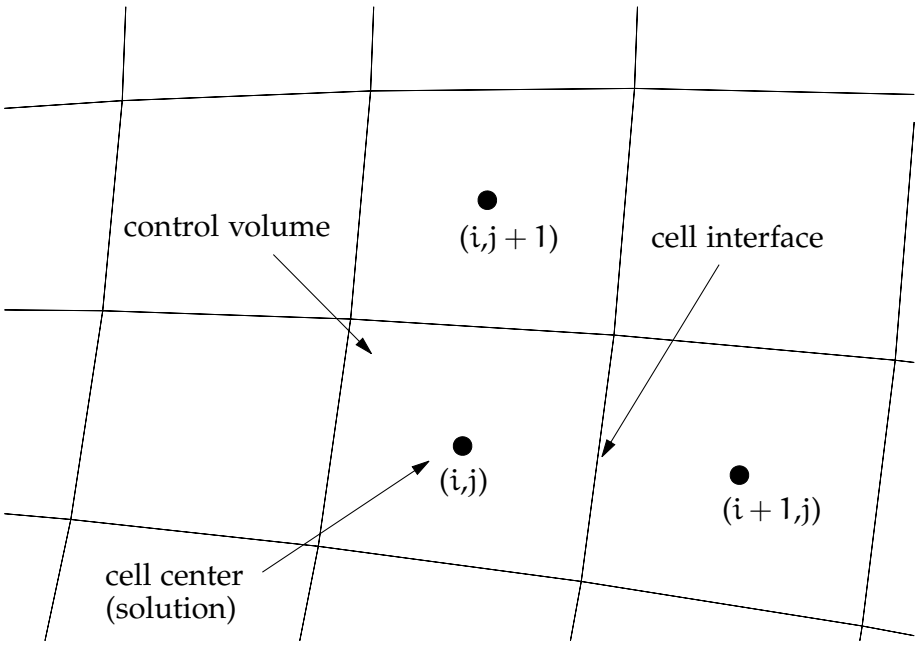


Figure 7: Schematic of the cell-centred finite volume scheme on structured grids.

(`rhs`), and to the normals of the interfaces. The computations performed at each interface are: the evaluation of Euler fluxes for the left and right state - equations (4,5), evaluation of Roe averages and corresponding eigen-vectors and eigen-values - equations (9)-(17), assembly of the numerical fluxes, and accumulation of the residuals. The kernel computes two external products (left and right Euler flux), and one 4x4 GAXPY. A judicious implementation requires 191 FLOPS and loads 38 double precision values per direction sweep.

```
// fluxes in i-direction
 for( j=0;j<nj;j++ )
{
    for( i=0;i<ni-1;i++ )
   {
       iq= j*ni+i;
       ql[0]=q[0][iq];
       qr[0]=q[0][iq+1];
       ...
       riem(ql,qr,...,f,&lmax);
       rhs[0][iq]-= f[0];
       rhs[0][iq+1]+= f[0];
    }
}
```

Listing 1: i-direction sweep

```
// fluxes in j-direction
for( j=0;j<nj-1;j++ )
{
    for( i=0;i<ni;i++ )
    {
        iq = j*ni+i;
        ql[0]=q[0][iq];
        qr[0]=q[0][iq+ni];
        ...
        riem(ql,qr,...,f,&lmax);
        rhs[0][iq]-= f[0];
        rhs[0][iq+ni]+= f[0];
    }
}
```

Listing 2: j-direction sweep

FLOW VARIABLE UPDATE The kernel computes the primitive variables updates, based on the residuals of the discretized Euler equations, as defined in equation (21). The kernel accesses the arrays storing the flow variables, the residuals, an array storing auxiliary variables and an array storing the spectral radii of the flux Jacobians. The arrays are passed to the function through their base pointers. The flow and auxiliary variables are used to compute the entries of the transformation Jacobian $K^{-1}_{i,j}$ between conserved variables and primitive variables. The kernel performs 40 FLOPS per cell and loads 35 double precision values giving it a 0.14 flops/byte ratio.

4.3.5 *Configuration of compute nodes*

Table 2 presents details with respect to the configuration of the compute nodes used in this chapter such as processor architecture and model, memory configuration and compiler version.

	SNB	HSW	KNC
Version	E5-2650	E5-2650	5110P
Sockets	2	2	1
Cores	8	10	60
Threads	2	2	4
Clock (GHz)	2.0	2.3	1.053
SIMD ISA	AVX	AVX2	IMCI
SIMD width	256-bit	256-bit	512-bit
L1 Cache (KB)	32	32	32
L2 Cache (KB)	256	256	512
L3 Cache (MB)	20	25	-
DRAM (GB)	32	64	8
DRAM type	DDR3	DDR4	GDDR5
Stream (GB/sec)	71	110	140
Compiler	icpc 15.0		

Table 2: Hardware and software configuration of the compute nodes used in this chapter. The SIMD ISA represents the latest vector instruction set architecture supported by the particular platform.

4.4 OPTIMISATIONS

This section presents in detail a number of optimisation techniques that have been applied to the selected computational kernels introduced in Section 4.3.

4.4.1 *Flow variable update*

VECTORIZATION Vectorization of the flow variable update kernel has been achieved through the use of OpenMP 4.0 [64] compiler directives i.e. `#pragma omp simd`. The use of OpenMP 4.0 directives is found preferable to compiler-specific directives such as the ones available in Intel compilers because they are portable and supported across other compilers. In order to achieve efficient vectorization, the qualifier `restrict` needs to be added to the function arguments. This guarantees to the compiler that arrays represented by the arguments do not overlap. In absence of further provisions, the compiler generates code for unaligned loads and stores. This is a safety precaution, as aligned SIMD load/store instructions on unaligned addresses lead to bus errors on the Intel Xeon Phi coprocessor. In contrast, Sandy Bridge and Haswell processors can deal with aligned access instructions on unaligned addresses, albeit with

some performance penalty due to the inter-register data movements that are required.

Aligned vector load and store operations can be achieved by issuing the `aligned` qualifier to the original directive and by allocating all the relevant arrays using the `_mm_malloc` wrapper function. `_mm_malloc` takes an extra argument representing the required alignment (32 bytes for AVX/AVX2 and 64 bytes for the IMCI on Knights Corner). A number of additional directives may be needed to persuade the compiler that aligned loads can be issued safely, as shown in the snippet below for a pointer storing linearly four variables in the SoA arrangement, at offsets `id0`, `id1`:

```
__assume_aligned(rhs, 64);
__assume(id0%8==0);
__assume(id1%8==0);
__assume((id0*sizeof(rhs[0]))%64==0);
__assume((id1*sizeof(rhs[0]))%64==0);
```

Listing 3: Example of extra compiler hints needed for generating aligned vector load and stores on 64-byte boundaries.

The `assume_aligned` construct indicates that the base pointer is aligned with the SIMD register boundary. The following `__assume` statements indicate that subscripts and indices accessing the four sections of the array are also aligned on the SIMD register boundary. The need for such additional directives is due to the fact that the OpenMP 4.0 implementation can only generate aligned load and store operations if the argument passed to the aligned clause i.e. `aligned(rhs:64)` is a single pointer and not a pointer to pointer. This somewhat complicates the issue of generating aligned vector load and store operations even for kernels with regular access patterns since the additional directives presented in Listing 3 are specific to Intel compilers and therefore not portable.

Another way of achieving SIMD execution with aligned load and stores operations is by invoking compiler intrinsics or by using a higher abstraction layer such as a library. In this work, versions of the cell-based flow variable update kernel have been implemented using both compiler intrinsics specific to each architecture as well as Agner Fog's Vector Class Library (VCL) [16]. The main advantage of the latter is that ugly compiler intrinsics are encapsulated away allowing for a more readable code. On Knights Corner, an extension of

the VCL library was used which was developed by Przemyslaw Karpiński at CERN [86].

SOFTWARE PREFETCHING Software prefetching can also be used to improve the performance of memory-bound kernels. This has been implemented in the kernel version based on compiler intrinsics and using the functionality of the `_mm_prefetch` intrinsics. The latter can take as arguments the cache level where the prefetch is to appear in. When using `_mm_prefetch` it is advisable to disable the compiler prefetcher using the `-no-opt-prefetch` compilation flag (Intel 15.0 compilers).

DATA LAYOUT TRANSFORMATIONS The previous data layout used to store both cell-centred as well as face quantities such as normals was based on the SoA format. The SoA layout is advisable for SIMD since it allows for contiguous SIMD load and stores. However, this can lead to performance penalties such as Translation Look-aside Buffer (TLB) misses for very large arrays while also requiring that multiple memory streams are kept in-flight for each individual vector. A hybrid approach such as the Array of Structures Structure of Arrays (AoSSoA) can alleviate these issues by offering the recommended SIMD grouping in sub vectors coupled with improved intra and inter structure locality.

Therefore, a version of the kernel using the AoSSoA data layout has also been studied. Four sub vector lengths were tested: 4,8,16,32 double precision values on the multicore processors and 8,16,32,64 on the coprocessor. The best performing sub vector length of AoSSoA proved to be the one equal to the SIMD vector width of the processor (i.e. 4 and 8 respectively).

THREAD PARALLELISM Thread parallelism of the flow variable update kernel was exploited using the OpenMP 4.0 construct for the auto-vectorized version which complements the initial `#pragma omp simd` construct and further decomposes the loop iterations across all of the available threads in the environment. The kernel versions based on compiler intrinsics used the generic OpenMP loop-level construct. The work decomposition has been carried out via a static scheduling clause, whereby each thread receives a fixed size chunk of the entire domain. Additionally, the first touch policy has been applied for all data structures utilised in this method in order to alleviate NUMA effects

when crossing over socket boundaries. The first touch is a basic technique which consists in performing trivial operations - e.g. initialization to zero - on data involved in parallel loops before the actual computations. This forces the relevant sections of the arrays to be loaded on the virtual pages of the core and socket where the thread resides. The first touch needs to be performed with same scheduling and thread pinning as the subsequent computations. In terms of thread pinning, the `compact` process affinity was used for Sandy Bridge and Haswell running only with one thread per physical core. On Knights Corner, the `scatter` affinity brought forth better results compared to `compact`.

4.4.2 *Flux computations*

VECTORIZATION Code vectorization of the kernel computing Roe's numerical fluxes has been achieved by evaluating a number of avenues such as auto-vectorization via compiler directives, compiler intrinsics and the VCL vector class, similar to the cell-based kernel.

In order for the code to be vectorized by the compiler, a number of modifications had to be performed. First of all, in the original version, the face neighbours were obtained using indirect references in order to also allow for calculations on unstructured meshes. These were removed and their position explicitly computed for each iteration. As these increase in a linear fashion by unit stride once their offset is computed, the compiler was then able to replace gather and scatter instructions with unaligned vector loads and stores, albeit with the help of the OpenMP 4.0 `linear` clause. This had a particularly large effect on Knights Corner where performance of unaligned load and stores was a factor of two higher compared to gather and scatter operations. Furthermore, the `restrict` qualifier was also necessary for any degree of vectorization to be attempted by the compiler.

In structured and block-structured codes, it is natural to group the faces according to the transformed coordinate that stays constant on their surface, e.g. i-faces and j-faces. As a result, fluxes can be evaluated in two separate sweeps visiting i-faces or the j-faces, respectively. Each operation consists of nested i- and j-loops as seen in Section 4.3. When visiting i-faces, if the left and right states are stored at unit stride, they cannot be simultaneously aligned on vector register boundary thus preventing efficient SIMD execution as demonstrated in Figure 8. When visiting j-faces this problem does not appear, as padding is

sufficient to guarantee that left and right states of all j-faces can be simultaneously aligned on the vector register boundary.

As a result, vectorized versions of the flux computation kernel were implemented based on compiler intrinsics and the VCL library that utilise unaligned loads and stores when visiting the i-faces but aligned operations for the sweeps along the j-faces. On the Knights Corner architecture, the VPU ISA does not contain any native instructions for performing vector load and stores operations at unaligned addresses. To this extent, the programmer has to rely upon the `unpacklo`, `unpackhi`, `packstorelo` and `packstorehi` instructions and their corresponding intrinsics to emulate such behaviour. Examples of methods used in this work for performing unaligned load and store operations on Knights Corner can be seen in Listings 4 and 5. As one can observe, emulating an unaligned load on Knights Corner requires two aligned load/store and register manipulations hence the penalty in performance compared to their aligned counterparts.

```
inline __m512d loadu(double *src)
{
        __m512d ret;
        ret = _mm512_loadunpacklo_pd(ret, src);
        ret = _mm512_loadunpackhi_pd(ret, src+8);
        return ret;
}
```

Listing 4: Unaligned SIMD loads on Knights Corner

```
inline void storeu(double *dest, __m512d val)
{
        _mm512_packstorelo_pd(dest, val);
        _mm512_packstorehi_pd(dest+8,val);
}
```

Listing 5: Unaligned SIMD stores on Knights Corner

The lack of alignment for the loop over the i-faces can be resolved by two techniques: shuffle operations or via transposition. Shuffling consists in performing the loads using register-aligned addresses, and then modifying the vector registers to position correctly all the operands. On AVX, performing these shuffle operations requires three instructions as register manipulations

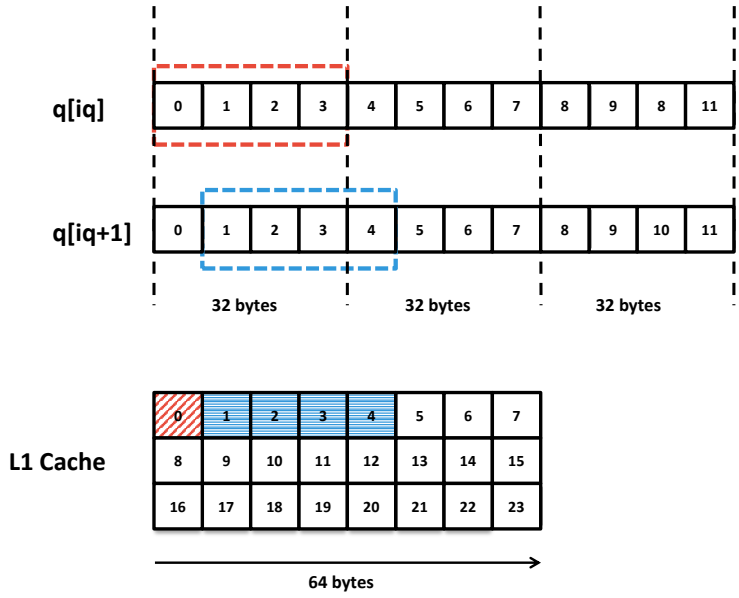


Figure 8: Vector load operation at aligned boundary for left state and load operation at unaligned boundary for right state.

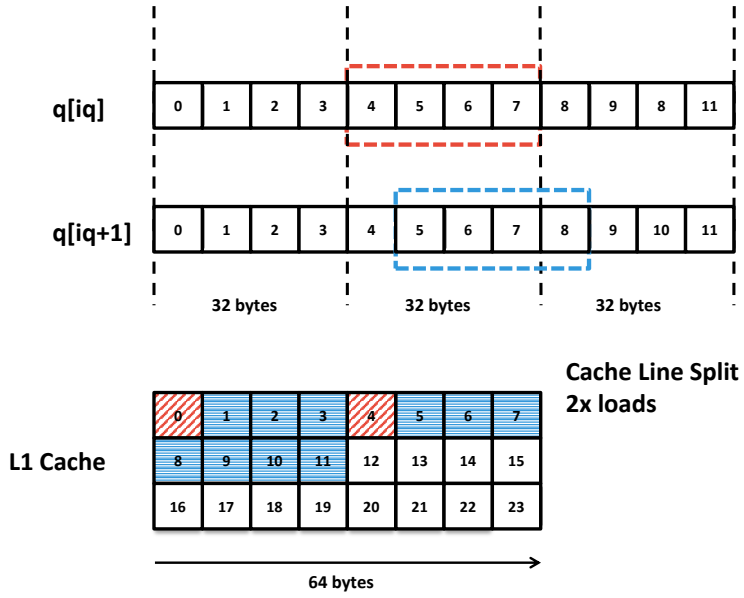


Figure 9: Side effects of unaligned vector load operations leading to cache line split loads.

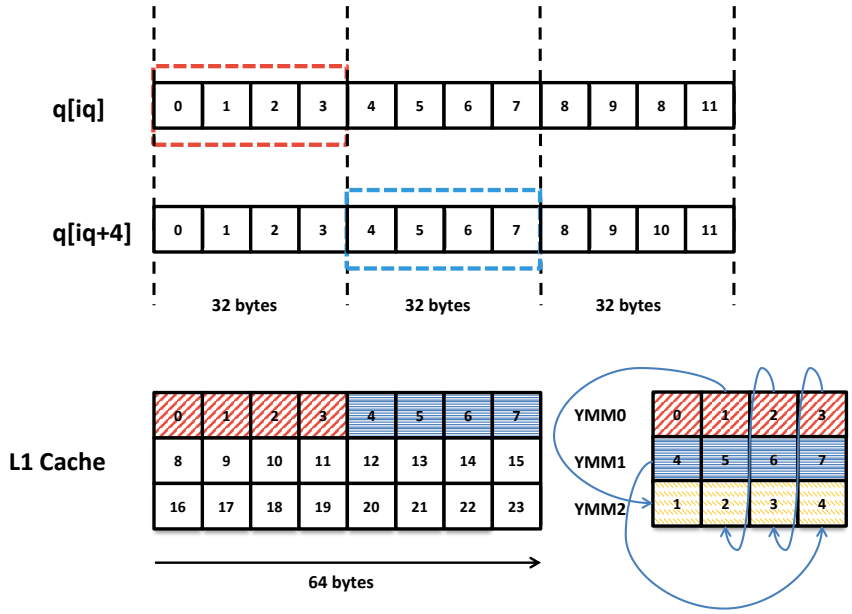


Figure 10: Vector load operation at aligned boundary for both left and right states followed by inter-register shuffles for correct positioning of operands.

can only be performed on 128-bit lanes. The implementation using compiler intrinsics and register shuffles issues aligned load and stores combined with shuffle operations when handling i-faces. The use of register shuffling also saves a load for each iteration, as the current right state can be re-used as left state when evaluating the following face therefore allowing for register blocking. The issue of unaligned loads for the i-sweep direction as well as the remedy of performing inter-register shuffling can be seen in Figures 8, 9 an 10. For best performance, the optimisation of the vector register rotations and shifts (i.e. the shuffle operations) is critical. On AVX, this was achieved by using the `vperm2f128` instruction followed by a shuffle. AVX2 supports the `vpermd` instruction which can perform cross-lane permutations, therefore a more efficient approach can be used which requires fewer instructions. Knights Corner also supports `vpermd` although the values have to be cast from `epi32` (integers) to the required format (doubles in the present case). The choice of compiler intrinsics for each operation is based on their latency and throughput: instructions with throughput larger than one can be performed by more than one execution port, if these are available, leading to improved instruction level parallelism. As an example, the AVX/AVX2 `_mm256_blend_pd` compiler intrinsic which converts to a `vblendpd` instruction has a latency of one cycle and a throughput of 3 on Haswell, and one cycle latency and throughput of 2 on Sandy Bridge. Consequently, it is preferable to the AVX `_mm256_shuffle_pd` intrinsic (`vshufpd`), which has one cycle latency, but unit throughput on both architectures. The code snippets below show how shuffling can be achieved across the three architectures.

```
// vl= 3 2 1 0
// vr= 7 6 5 4
__m256d t1=_mm256_permute2f128_pd(vl,vr,33);  // 5 4 3 2
__m256d t2=_mm256_shuffle_pd(vl,t1,5);        // 4 3 2 1
```

Listing 6: Register shuffle for aligned accesses on the *i*-face sweep with AVX.

```
// vl= 4,3,2,1
// vr= 8,7,6,5
__m256d blend = _mm256_blend_pd(vl,vr,0x1); // 4,3,2,5
__m256d res =_mm256_permute4x64_pd(blend,_MM_SHUFFLE(0,3,2,1)); // 5,4,3,2
```

Listing 7: Register shuffle for aligned accesses on the *i*-face sweep with AVX2.

```
// vl= 8,7,6,5,4,3,2,1
// vr= 16,15,14,13,12,11,10,9
__m512i idx = {2,3,4,5,6,7,8,9,10,11,12,13,14,15,0,1};
//  8,7,6,5,4,3,2,9
__m512d blend = _mm512_mask_blend_pd(0x1,vl,vr );
//  9,8,7,6,5,4,3,2
__m512d res = _mm512_castsi512_pd(_mm512_permutevar_epi32(idx,
    _mm512_castpd_si512(blend)));
```

Listing 8: Register shuffle for aligned accesses on the *i*-face sweep with IMCI.

Generating aligned load/stores in the VCL library can be performed by using the `blend4d` method which requires as argument an index shuffle map. However, this was found to be less efficient than the implementation based on compiler intrinsics as the `blend4d` primitive did not utilise the instructions with the best latency and throughput for each distinct architecture.

Another technique for addressing the stream alignment conflict issue is by transposing the cell-centred variables before the sweep over the i-faces and transposing them back prior to the sweep over the j-faces. This is somewhat similar to Henretty's dimension-lifted transposition [37] however, the version implemented in this work has the disadvantage of requiring that data be re-transposed prior to the sweep on the j-faces due to the independent evaluation in the i and j directions.

CACHE BLOCKING In the implementations up to this point, the loops visiting i-faces and j-faces have been kept separate. Spatial and temporal locality can be improved by fusing (blocking) the evaluation of fluxes across the j-face and the i-face of each cell. This technique is known as cache blocking. The cache blocking kernel fuses both i- and j-passes and it is based on the kernel versions that perform inter-register shuffles as described above for allowing aligned loads and stores on the i-faces. The cache blocking kernel further benefits from the fact that it can save an extra load/store operation when writing back results for each evaluated face. The blocking factor should be a multiple of the SIMD vector length as to allow for efficient vectorization.

DATA LAYOUT TRANSFORMATIONS Discussion so far for the flux computations has assumed a SoA data layout format. A kernel using the AoSSoA data layout and based on inter-register shuffling for aligned SIMD load operations

```
 for( j=1;j<min(j+jBF,nj);j+=jBF )
{
    for( i=1;i<min(i+iBF,ni);i+=iBF )
   {
        fluxi(...);
        fluxj(...);
   }
}
```

Listing 9: Implementation of cache blocking by nesting both i and j sweeps together.

was also tested. The best performing sub-vector size was the size of the SIMD register, similar to the flow variable update kernel.

THREAD PARALLELISM For the purpose of studying performance at full chip and node concurrency, domain decomposition is performed at block-level. This can be performed in two ways. One option is to split a block into slabs along the i- and j-planes depending on sweep, the number of slabs being equal to the number of available threads. The second option is to split each block into tiles. This method increases locality and complements the cache-blocking processing of the kernel computing fluxes as discussed above. For the non-fused kernels where separate plane sweeps are performed, the first decomposition method was chosen due to the elimination of possible race conditions. For the fused kernels, the latter decomposition was required together with the handling of race conditions at the tile boundary. In the second option, the decomposition mimics that on distributed-memory machines via MPI and requires treatment for updating boundaries in a serial fashion once the fluxes are computed in the inner regions.

Thread and process affinity have been applied in a similar fashion to the flow variable update kernel i.e. `compact` for the multicore CPUs and `scatter` for the coprocessor. Task and chunk allocation have been performed manually as required by the custom domain decomposition within the OpenMP parallel region. Another reason for doing the above and not relying on the OpenMP runtime was the high overhead that this caused on the Knights Corner processor when running on more than 100 threads. Furthermore, all parallel runs of the optimised flux computation kernels implement NUMA-aware placement via applying the first touch technique.

4.5 RESULTS AND DISCUSSIONS

This section presents the results of the optimisations on both computational kernels and a discussion on their effectiveness.

4.5.1 *Flow variable update*

Figure 11 presents the effect that the optimisations had on the performance of the flow variable update kernel when running on a single core. Results at full-chip and full-node concurrency for each optimisation are presented in Figure 12.

VECTORIZATION On Sandy Bridge and Haswell, even the auto-vectorized kernel performs twice and three times faster than the baseline kernel. On Knights Corner, the improvement is almost one order of magnitude as the VPU and therefore wide SIMD lanes are exploited whereas they were not previously.

Aligned versus unaligned load and stores see no benefit on Sandy Bridge and for smaller problem sizes a marginal improvement on Haswell, due to better L1 cache utilisation. The aligned access version outperforms its unaligned counterpart on Knights Corner as a 512-bit vector register maps across an entire cache line therefore allowing for efficient load and store operations to/-from the L1 cache. The reason there is no significant difference between aligned and unaligned SIMD load and store operations on the multicores is possibly due to the fact that the kernel is already limited by memory bandwidth on these platforms. On the other hand, on Knights Corner, the difference in instruction count as well as throughput between aligned and unaligned load and store operations leads to a noticeable difference in performance.

The utilisation of compiler intrinsics or the VCL implementation did not bring forth any speed-ups over the directive based vectorization for this particular kernel. This would indicate the fact that the compiler is able to vectorize kernels with regular access patterns competitively provided it is aided with additional hints such as the ones presented in Listing 3 and compiler directives.

SOFTWARE PREFECTHING Software prefetching for the cell-based flow update kernel sees no benefit across any of the three architectures.

THREAD PARALLELISM The kernel under consideration is memory-bound on all three architectures and is therefore very sensitive to NUMA effects. For Haswell, not using the first touch technique can lead to a large performance degradation as soon as the active threads spill outside a single socket. This can be seen by the behaviour of the data sets that lack NUMA optimisations in Figure 12. A reason for why the discrepancies between NUMA and non-NUMA runs are larger on Haswell compared to Sandy Bridge is the larger core count on the former which means that the Quick Path Interconnect (QPI) system linking the two sockets gets saturated more quickly due to the increase in cross-socket transfer requests as more cores are serviced.

On Knights Corner there are no NUMA effects due to the ring-based interconnect and the interleaved memory controller placement. However, aligned loads play an important role on this architecture: the SIMD length is 64 bytes (8 doubles) which maps to the size of an L1 cache line. If unaligned loads are issued, the thread is required to go across cache boundaries, loading two or more cache lines and performing inter register movements for packing up the necessary data. This wastes memory bandwidth and forms a very damaging bottleneck. This can be seen from the drop in performance above 60 cores in the version written in compiler intrinsics with unaligned accesses and the auto-vectorized version data set in Figure 12. Consequently, although unaligned accesses did not produce such issues when run across a single core, the overhead of additional memory traffic across the cache hierarchies as a result of unaligned memory accesses grew as the number of cores increased as well.

The best performing version was based on the hybrid AoSSoA format. The version based on aligned load and store operations via compiler intrinsics and combined with manual prefetching performed second best followed by the version implementing aligned compiler intrinsics as well as compiler issued prefetches.

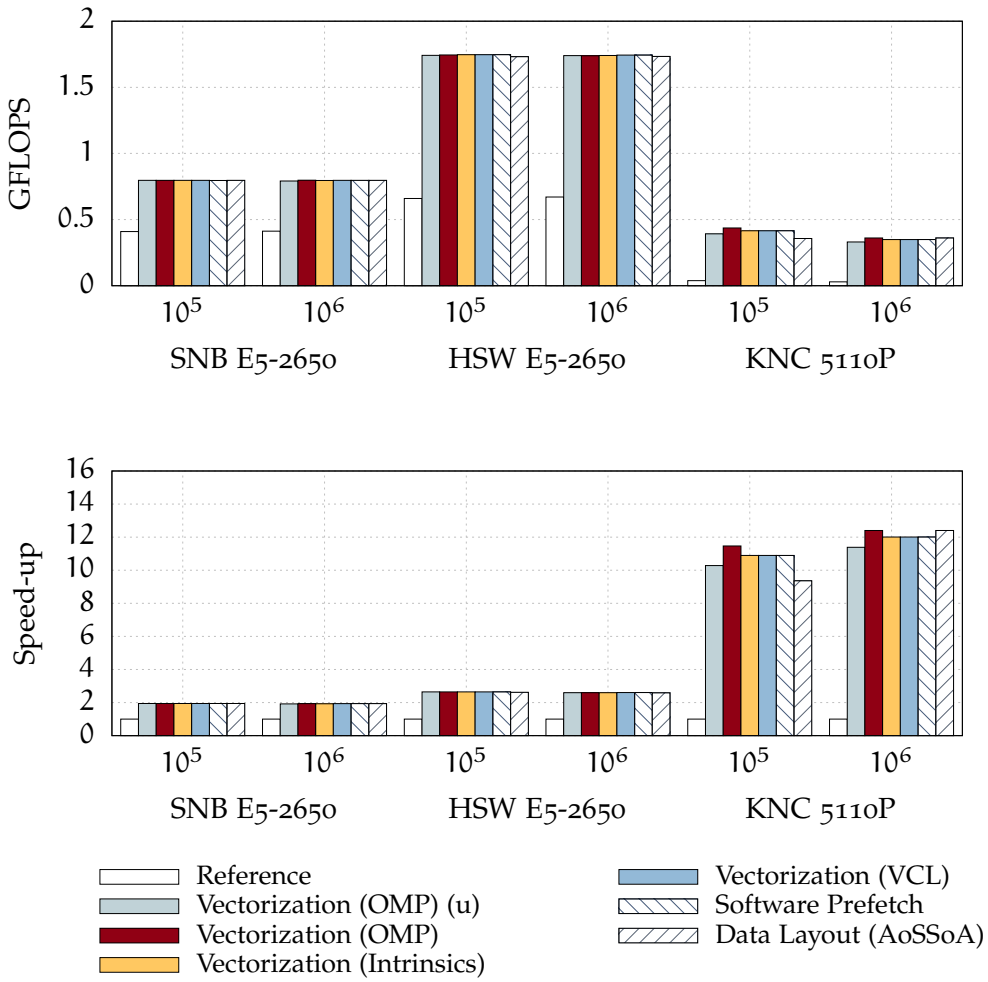


Figure 11: Effects of optimisations for the kernel performing the flow variable update across different grid sizes and running on a single-core. Results in GFLOPS on KNC 5110P for the reference runs on grid sizes 10^5 and 10^6 are 0.03 and 0.02. The symbol `(u)` stands for unaligned memory access.

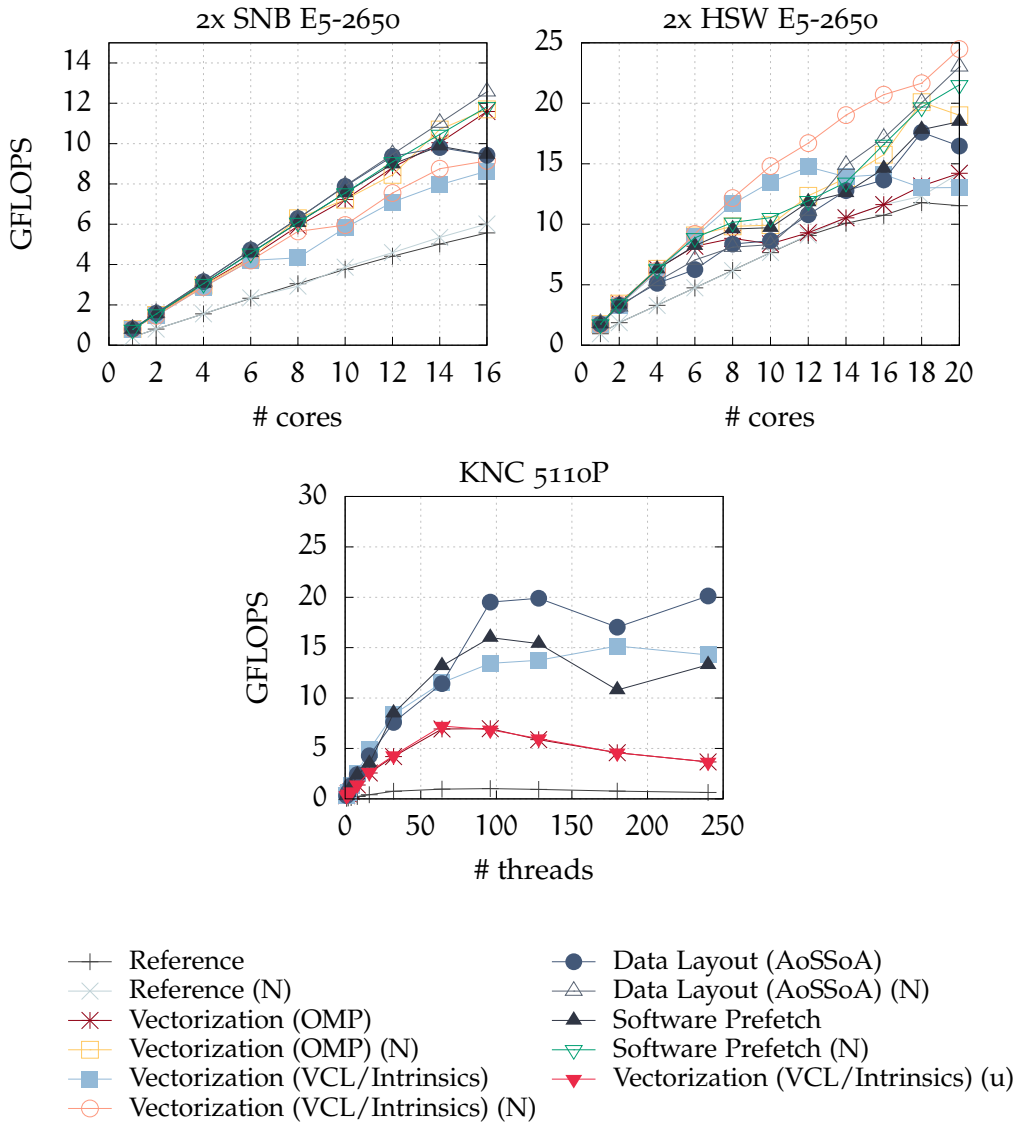


Figure 12: Performance strong scaling of kernel executing the flow variable update on 10^6 grid cells. The symbol `(u)` stands for unaligned memory access while entries that include the symbol `(N)` implement NUMA optimisations (i.e. first touch). Results of runs with NUMA (N) optimisations are only applicable to the multicore CPUs and not to KNC.

4.5.2 *Flux computations*

Figure 13 presents the effects that the optimisations had on the single-core performance of the flux computations kernel. Results at full-chip and full-node concurrency are shown in Figure 14.

VECTORIZATION For the kernel computing Roe's numerical fluxes, the version based on auto-vectorization via compiler directives brought forth a 2.5X improvement on Sandy Bridge, 2.2X on Haswell and 13X on Knights Corner when compared to their respective baseline. The penalty for unaligned access on the i-faces is better tolerated on the multicore processors, where inter-register movements can be performed with small latency, but is very damaging on Knights Corner, as already seen.

The kernel version based on unaligned load and store operations and compiler intrinsics delivers a 4X speed-up over the baseline implementation on Sandy Bridge and Haswell and 13.5X on Knights Corner. Comparisons with the auto-vectorized version sees a 60% speed-up on Sandy Bridge and 50% on Haswell although no noticeable improvements on the coprocessor. The increase in performance for the compiler intrinsics version on the multicore processors is due to the manual inner loop unrolling when assembling fluxes which allows for more efficient instruction level parallelism.

The compiler intrinsics and shuffle kernel performs similarly on Sandy Bridge compared to the unaligned version due to AVX cross-lane shuffle limitations whilst performing 30% and 10% faster on Haswell and Knights Corner. The kernel versions based on the VCL library perform worse on both multicore CPUs compared to the version using compiler intrinsics due to the inefficient implementation of shuffling in the `blend4d` routine. On Knights Corner, these were replaced with the hand-tuned compiler intrinsics primitives presented in Listing 4.4.2 as they were not implemented in the VLCKNC library. For this reason, the aligned VCL version on the coprocessor delivers very similar performance compared to the version based on compiler intrinsics.

The kernel version based on aligned vector load and store operations on both sweeps thanks to transpositions performs worse than the version that is based on inter-register shuffles. This is due to the fact that transposing the cell-centred variables prior to the sweep on the i-faces and back for the following sweep on the j-faces can have a degrading effect on performance due to the

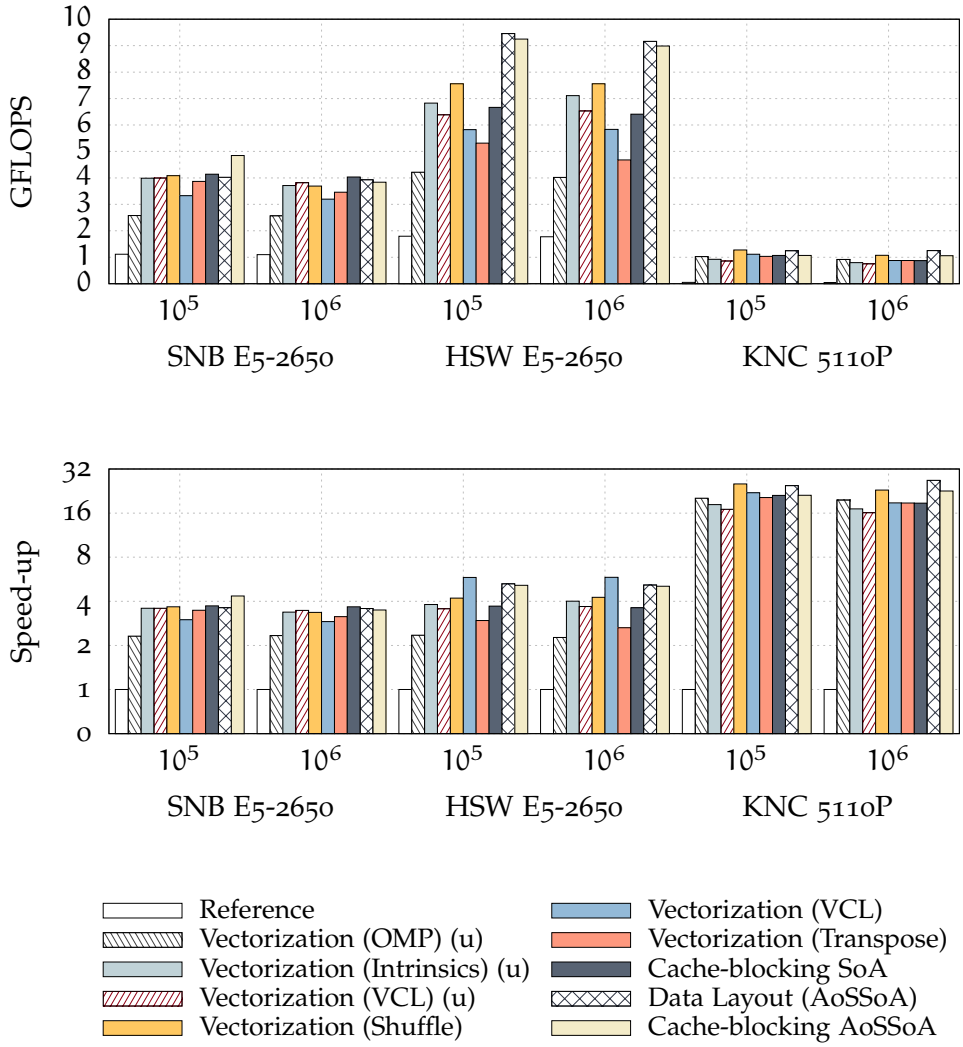


Figure 13: Effects of optimisations on the kernel performing the flux computations on different grid sizes. Results in GFLOPS on KNC 5110P for the reference runs on grid sizes 10^5 and 10^6 are 0.05 and 0.04. The symbol `(u)` stands for unaligned memory access.

working set not fitting into the cache. As a result, this technique is discouraged due to the overhead it produces and which is not offset by the difference in performance between unaligned and aligned vector operations. However, it is important to note that such transpositions might be beneficial if the data is only transposed once as seen in the work of Henretty et al [37]. In hindsight, this could have been accomplished by fusing both i and j sweeps (i.e. in the cache-blocked version).

CACHE BLOCKING Across all three platforms, cache blocking did not bring forth any palpable benefits for single core runs. The reason for this is that although data cache reuse is increased, fusing both loops brings forth an increase in register pressure and degrades performance as the fused kernel performs 382 FLOPS for every iteration.

THREAD PARALLELISM The performance on both Sandy Bridge and Haswell nodes at full concurrency is 53 GFLOPS and 101 GFLOPS respectively. On Knights Corner, the best performing version achieves 94 GFLOPS when running on 180 threads. While on the multicore CPUs, the best performing kernels were the versions based on compiler intrinsics, cache-blocking as well as the hybrid AoSSoA layout, the scaling on the coprocessor of this particular kernel was second best. This can be attributed to register pressure as more threads get scheduled on the same core which compete for available resources. This assumption can be validated by examining the scaling behaviour of the cache-blocked kernel without the hybrid AoSSoA data layout which suffers from poor scalability as more than one thread gets allocated per core.

Comparing best performing results to their respective baselines reveals a 3.1X speed-up on Sandy Bridge and Haswell and 24X on the Knights Corner coprocessor at full concurrency.

4.5.3 *Performance modelling*

A visualisation of the Roofline model based on the results of the applied optimisations can be seen in Figure 15. For brevity, the main classes of optimisations are grouped as:

- **Reference**: for the initial baseline

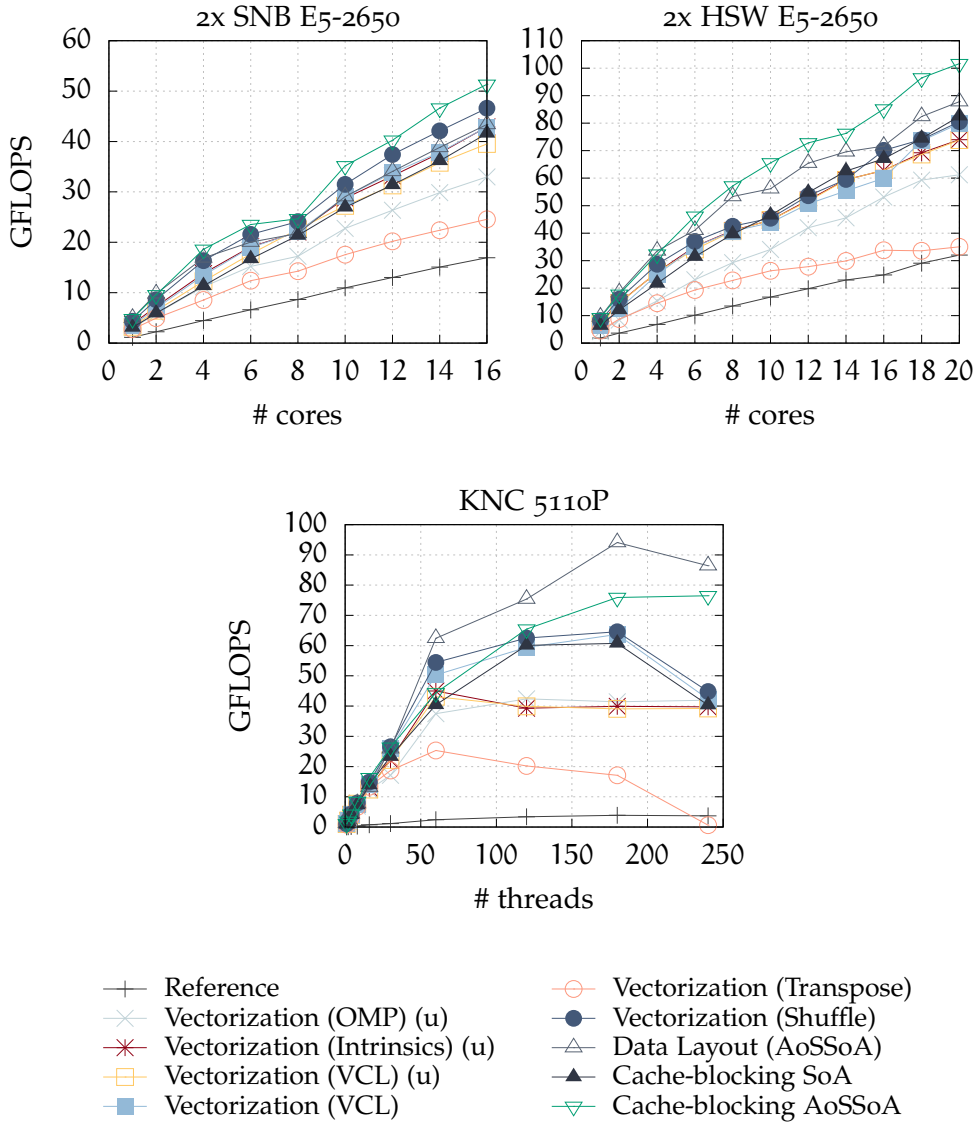


Figure 14: Performance strong scaling of kernel executing the flux computations as a result of optimisations on 10^6 grid cells. The symbol `(u)` stands for unaligned memory access.

- **Vectorization (OMP)**: for versions vectorized using compiler directives;
- **Vectorization (Shuffle)**: assuming best performing version based on compiler intrinsics on that particular kernel and platform (i.e. shuffle, transposition etc)
- **Memory Opt.**: for best performing version implementing aligned vector load and store operations, compiler intrinsics and specific memory optimisation (cache-blocking, AoSSoA etc)

SINGLE CORE Observations on the single core diagrams highlight interesting aspects on the two computational kernels and their evolution in the optimisation space.

For the cell-based flow variable update kernel, vectorization using compiler directives pierces through the IMCI roofline on the coprocessor. This is due to the fact that vectorization on Knights Corner allows for instructions to be routed towards the more efficient VPU unit. Furthermore, this kernel performs as well as the hand-tuned kernels based on compiler intrinsics across all three platforms (see Figures 11 and 12) as its performance is limited by memory bandwidth.

For flux computations, vectorization using compiler directives pierces through their respective SIMD wall whilst hand tuned compiler intrinsics and subsequent memory optimisations deliver close to the attainable performance prediction. On Haswell and Knights Corner however, speed-up is limited by the fact that the FMA units are not fully utilised due to the algorithmic nature of stencils operators and their inherent lack of FMA operations. Even so, for single core runs, the best optimised version incorporating SIMD as well as memory optimisations achieves between 80-90% of the available performance compared to the baseline which delivers 20% on the multicore CPUs and only 1.67% on the coprocessor.

FULL NODE At full node concurrency, the sizeable increase in memory bandwidth permits the reference implementation to obtain a larger degree of performance across all architectures.

For the cell-based flow variable update kernel, NUMA first touch optimisations applied to all of the optimised kernel versions allows them to bypass the NUMA wall. On the coprocessor, software prefetching only works when

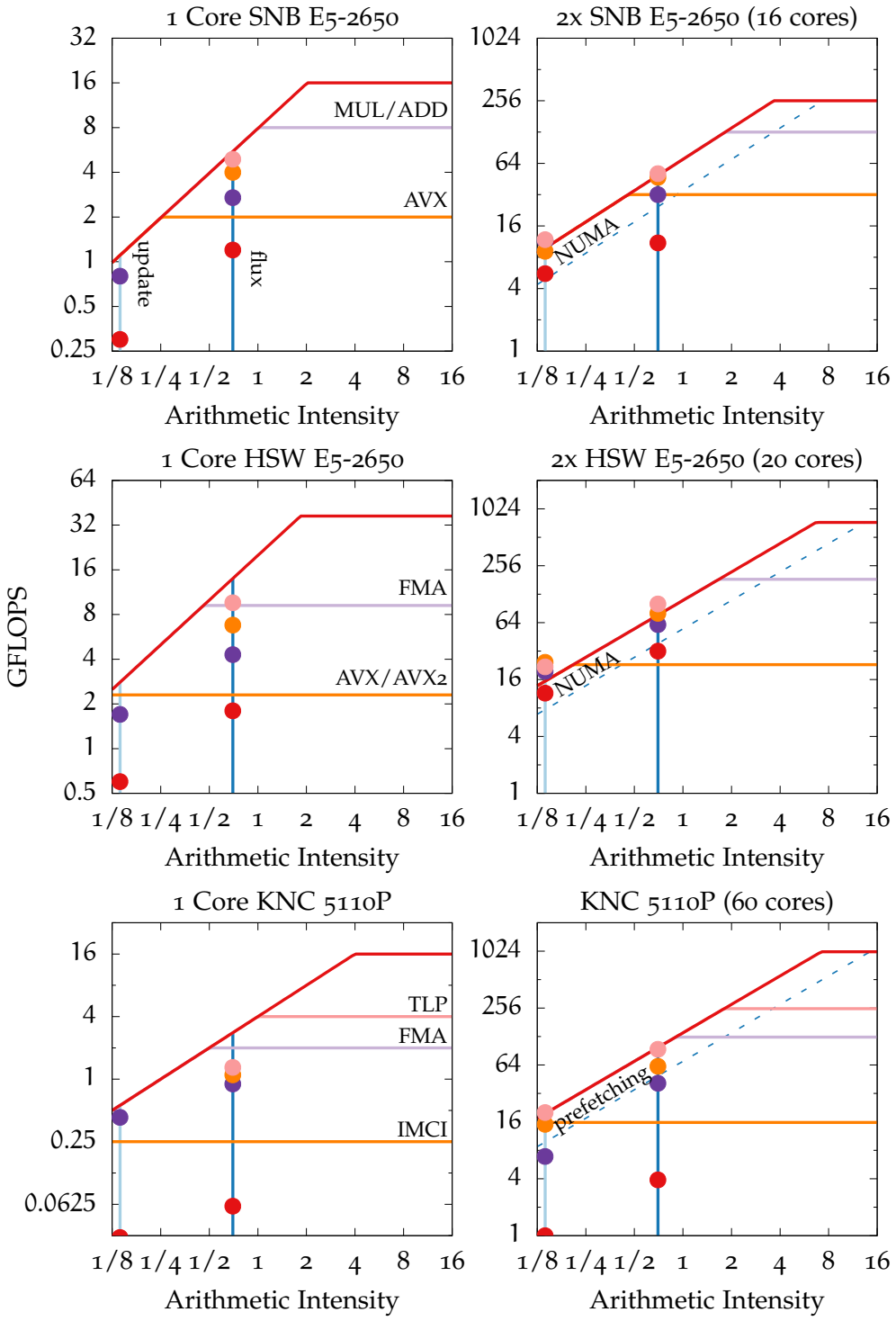


Figure 15: Roofline visualisation of optimisations for single-core and full-node configurations. Key: ● Reference ● Vectorization (OMP) ● Vectorization (Shuffle) ● Memory Opt.

coupled with SIMD optimisations due to the poor performance of the scalar processing unit compared to the VPU. As more memory bandwidth becomes available, subsequent optimisations such as the hybrid data layout transformation that prevents TLB misses and provides better data locality performs better than the vectorized SoA-based hand tuned compiler intrinsics version.

For flux computations, vectorization through compiler directives and intrinsics results in performance that is close to the main roofline represented by peak DRAM memory bandwidth while subsequent memory optimisations bypass the model's prediction. This is due to the fact that some of the data in the optimised kernels is reused from the higher cache levels as a result of memory optimisations such as cache-blocking and is therefore not transferred from main memory. This in effect improves the arithmetic intensity of the kernel since fewer loads are in fact serviced by main memory and is a limitation of this analysis that is worth considering. An alternative would be to also update the arithmetic intensity of each kernel as optimisations are implemented that improve upon the locality of the data. However, the difficulty for such an approach is how to accurately determine the proportion of loads that are serviced by the higher cache levels and those serviced by main memory.

4.5.4 *Architectural comparison*

On a core-to-core comparison among the three processors, the Haswell-based Xeon E5-2650 core performs on average 2X and 4-5X faster compared to a single Sandy Bridge Xeon E5-2650 and Xeon Phi Knights Corner 5110P core across both kernels. However, at full node and chip concurrency, the two socket Haswell Xeon E5-2650 node is approximately on par with the Knights Corner coprocessor for flux computations and 25% faster for updating the flow variables. The Xeon Phi Knights Corner coprocessor and Haswell node outperform the two socket Sandy Bridge Xeon E5-2650 node by approximately a factor of two on flux computations and 50% to 2X on the flow variable update. However, on a flop per watt and flop per dollar metric, the Knights Corner 5110P coprocessor delivers superior performance compared to both multicore CPUs at the cost of higher development and optimisation time needed for exploiting its underlying features in numerical computations. The increase in time spent on fine tuning the code on the coprocessor is attenuated by the fact that the majority of optimisations targeting fine and coarse grained levels of parallelism

such as SIMD and threads are transferable between both types of platforms including GPGPUs.

4.6 CONCLUSIONS

In this chapter, a variety of optimisation techniques have been implemented in two distinct computational kernels within a block-structured CFD code and across three modern architectures. A detailed description was given on the exploitation of all levels of parallelism available in modern multicore and manycore processors through efficient code vectorization and thread parallelism. Memory optimisations described in this work included software prefetching, data layout transformations through hybrid data structures such as the AoSSoA layout and cache blocking.

The practicalities of enabling efficient vectorization were discussed which established that for relatively simple kernels such as cell-based loops for updating state vectors, the compiler can generate efficient SIMD code with the aid of portable OpenMP 4.0 directives. This approach however does not fully extend to more complex kernels such as flux computations involving a stencil-based access pattern where best SIMD performance is mandated through the use of aligned vector load and store operations made possible via inter-register shuffles and permutations. Implementations of such operations were performed in this work using compiler intrinsics and the VCL framework and included bespoke optimisations for each architecture. Vectorized and non-vectorized computations exhibit a 2X performance gap for the flow variable update kernel and up to 5X for flux computations on the Sandy Bridge and Haswell multicore CPUs. The difference in performance is significantly higher on the manycore Knights Corner coprocessor where vectorized code outperforms the non-vectorized baseline by 13X in updating the flow variables and 23X for computing the numerical fluxes. These figures, when correlated with projections that future multicore and manycore architectures will contain further improvements to the width and latency of SIMD units, mandates efficient code SIMDization as a crucial avenue for attaining performance in structured grid applications on current and future architectures.

Modifying the data layout from SoA to a hybrid AoSSoA led to improvements for the vectorized kernels by minimizing TLB misses when running on large grids and by increasing data locality. Performance gains were particu-

larly noticeable when running at full concurrency and performed best on the Knights Corner coprocessor. Cache-blocking the two separate sweeps within the flux computation kernel coupled with the hybrid data layout delivered best results on the multicore CPUs when running across all available cores however performed second best on the coprocessor due to increased register pressure.

Core and thread parallelism has been achieved through the use of OpenMP 4.0 directives for the cell-based flow variable update kernel which offers mixed parallelism at SIMD and thread granularities through common constructs. For the numerical fluxes, the domain was decomposed into slabs on the i and j faces for implementations that performed separate plane sweeps and into tiles for the fused sweep implementation with cache blocking. The flow variable update kernel achieved a 2X speed-up on the Sandy Bridge and Haswell nodes and 20X on the Knights Corner coprocessor when compared to their respective baselines. Computations of the numerical fluxes at full concurrency also obtained a 3X speed-up on the multicore CPUs and 24X on the coprocessor over the baseline implementation.

The Roofline performance model has been used to appraise the optimisation process with respect to the algorithmic nature of the two computational kernels and the three architectures. For single core execution, the optimised flow variables update kernel achieved approximately 80% efficiency on the multicore CPUs and 90% on the coprocessor. Flux computations obtained 90% efficiency on Sandy Bridge, 80% on Haswell and approximately 60% on Knights Corner. The reason for the relatively low efficiency on the coprocessor is due to the in-order core design which requires at least two threads for fully populating the coprocessor VPU. The computational efficiency at full concurrency outperforms the model's predictions for both kernels across the multicore platforms due to the fact that some of the data in the optimised kernels is retrieved from the higher cache levels and not main memory.

5

OPTIMISATION OF UNSTRUCTURED MESH APPLICATIONS

5.1 INTRODUCTION

The solution of fluid flow problems in the vicinity of complex geometries mandate the utilisation of unstructured grids. However, compared to structured and block-structured mesh applications, computations on unstructured grids are known to exhibit unsatisfactory performance on cache-based architectures [9]. This is due in great part to the data structures required for expressing grid connectivity and the resulting indirect and irregular access patterns.

In finite volume discretizations, these data structures and access patterns appear when iterating over the faces or edges of the computational domain for the purpose of evaluating fluxes, gradients and limiters. These computational kernels are usually structured as a sequence of gather, compute and scatter operations where variables are gathered from pairs of cells or vertices sharing a face or edge followed by the calculation and scatter of results to the respective face or edge end-points.

An example of a typical face-based loop can be seen in Listing 10 where unknowns are gathered from `q` and used together with mesh geometrical attributes in `geo` to compute the flux residuals `f` that are subsequently accumulated and scattered back to `rhs`. For edge-based solvers, one can simply replace faces with edges and cells with vertices or nodes.

```
 for( ic=0;ic<num_faces;ic++ )
{
     u1= q[ifq[0][ic]];
     u2= q[ifq[1][ic]];
     f= geo[ic]*(u2-u1);
     rhs[ifq[0][ic]]-= f;
     rhs[ifq[1][ic]]+= f;
}
```

Listing 10: Example of a face-based kernel.

The gather and scatter operations arising from the indirect access in `q` and `rhs` via the `ifq` index array can operate across large and irregular strides in memory and is determined by the mesh topology. As a result, these face-based kernels are an example of a class of computational patterns that do not naturally map to the architectural features of modern cache-based multicore and manycore architectures. First of all, accessing memory in an irregular fashion is detrimental to performance as it fails to exploit both spatial and temporal locality required for efficient utilisation of the memory hierarchy. Secondly, the existence of indirect addressing when accessing cell-centred variables inhibits vectorization by the compiler as well as the exploitation of thread parallelism due to potential data dependencies when scattering back the results in `rhs`. Finally, accessing memory indirectly also has a detrimental impact on the operation of hardware prefetchers thus reducing any opportunity for memory parallelism.

A typical unstructured finite volume code will spend more than two thirds of its execution time in face-based loops for computing fluxes across cell interfaces and boundaries. Consequently, addressing the limitations that prevent optimal use of the vector units, threads and memory hierarchy will have the highest impact in achieving improved performance on modern processors. However, optimisations that are beneficial to face-based kernels might be to the detriment of cell-based loops where the remaining execution time is spent. As a result, this could potentially offset any performance gains with respect to the whole application. Moreover, as face-based loops are optimised, the computational bottleneck will invariably shift towards cell-based loops. Consequently, for best full application performance, one must optimise both face-based and cell-based kernels and assess the impact this has on improving overall performance.

This chapter presents a wide range of such optimisations in an unstructured finite volume CFD code typical in size and complexity of an industrial application. Their implementation and impact are demonstrated on kernels computing inviscid, viscous and linearised fluxes as an example of face-based loops and on a kernel computing updates to primitive variables as an example of a cell-based loop. The benefits of each distinct optimisation are evaluated across a wide range of multicore and manycore compute nodes based on architectures such as the Intel Sandy Bridge, Broadwell, Skylake and the Intel Xeon

Phi Knights Corner and Knights Landing processors and across two different computational domain sizes.

The remaining chapter is structured as follows. Section 5.2 presents related work, Section 5.3 provides details of the code, test case and hardware setup. Section 5.4 presents an in-depth description of each optimisation followed by results and discussions in Section 5.5 and concluding remarks in Section 5.6.

5.2 RELATED WORK

The flexibility of unstructured mesh CFD solvers in dealing with complicated geometries have led to their wide adoption in industry and across the CFD community as a whole [52]. Consequently, there has always been significant interest in the optimisation and acceleration of unstructured mesh solvers on the latest computer architectures.

Anderson et al [8] presented the optimisation of FUN3D [28], a tetrahedral vertex-centered unstructured mesh code developed at the National Aeronautics and Space Administration (NASA) Langley Research Center for the solution of the compressible and incompressible Euler and Navier-Stokes equations and for which they received the 1999 Gordon Bell Prize [1]. Their optimisations were based on the concept of memory centric computations whereby the aim was to minimize the number of memory references as much as possible in the recognition that flops are cheap relative to memory load and store operations. The authors achieved this by increasing spatial locality with the help of interlacing in which data items that are required in close succession such as unknowns are stored contiguously in memory using data structures such as AoS. They also reduced the impact of the underlying gather and scatter operations by renumbering the mesh vertices using the Cuthill-Mckee [20] sparse matrix bandwidth minimizer. Their work was subsequently extended in the context of the FUN3D code by a number of studies such as Gropp et al [35] which introduced performance models in order to guide the optimisation process by classifying the operational characteristics of the computational kernels and their interaction with the underlying hardware, Mudigere et al [60] who demonstrated shared memory optimisations on modern parallel architectures including vectorization and threading through a hybrid MPI/OpenMP implementation, Al Farhan et al [29] who presented optimisations specific to the Intel Xeon Phi Knights Corner processor as well as Duffy et al [25] who ported

FUN3D for execution on graphical processing units obtaining a factor of two speed-up as a result.

More recently, Economon et al [26] presented the performance optimisation of the open-source SU2 [79] unstructured CFD code on Intel architectures. Their work demonstrated the impact of a number of optimisations such as vectorization, edge reordering, data layout transformations for improving single core performance on modern multicore architectures as well as optimisations of the linear solver in order to remove the impact of performing collective operations at large scales. As a result, they obtained speed-ups of more than a factor of two at single node and multi node granularities.

A different approach of optimising unstructured grid applications is by implementing such optimisations at a higher level of abstraction through a DSL. Examples of such initiatives with respect to unstructured CFD solvers can be found by examining the work in the OP2 framework [32],[59],[71],[72] as well as the other examples that are discussed in Chapter 2.

The work presented herein complements the above in that it presents the impact of some of the described optimisations on new architectures such as the Intel Xeon Skylake and Intel Xeon Phi Knights Landing processors while also demonstrating a number of improvements and novel approaches for exploiting the architectural features across both multicore and manycore platforms in a large scale unstructured CFD application.

5.3 BACKGROUND

The test vehicle for this study is the in-house CFD solver AU3X[24],[88]. AU3X uses a cell-centred finite volume approach to solve the unsteady Favre-averaged Navier-Stokes equations on unstructured meshes. Steady solutions are obtained by pseudo time marching and time accurate solutions by dual time stepping [42]. The governing equations, spatial discretization and time integration schemes are briefly described in the following sections in order to complement the implementation details and code optimisation study although some similarities with the numerical algorithms presented in Chapter 4 will be evident.

5.3.1 *Governing Equations*

The Favre-averaged Navier-Stokes equations for compressible flows in the differential form read:

$$\begin{aligned}
\frac{\partial \overline{\rho}}{\partial t} + \frac{\partial(\overline{\rho}\tilde{v_i})}{\partial x_j} &= 0 \\
\frac{\partial(\overline{\rho}\tilde{v_i})}{\partial t} + \frac{\partial(\overline{\rho}\tilde{v_i}\tilde{v_j})}{\partial x_j} &= -\frac{\partial \overline{p}}{\partial x_i} + \frac{\partial}{\partial x_j}(\tilde{\tau_{ij}} + \tau^t_{ij}) \\
\frac{\partial(\overline{\rho}\tilde{E})}{\partial t} + \frac{\partial(\overline{\rho}\tilde{v_j}\tilde{H})}{\partial x_j} &= -\frac{\partial}{\partial x_j}(\kappa\frac{\partial \tilde{T}}{\partial x_j} + \tilde{v_i}(\tilde{\tau_{ij}} + \tau^t_{ij}))
\end{aligned} \tag{22}$$

Where,

$$\tilde{\tau_{ij}} = 2\mu(\tilde{S_{ij}} - \frac{1}{3}\frac{\partial \tilde{v_k}}{\partial x_k}\delta_{ij}), \quad \overline{p} = (\gamma-1)\overline{\rho}(\tilde{E} - \frac{1}{2}\tilde{u_j}\tilde{u_j}), \quad \kappa\frac{\partial T}{\partial x_j} = \frac{\gamma}{\gamma-1}\frac{\mu}{P_r}\frac{\partial}{\partial x_j}(\frac{\overline{p}}{\overline{\rho}}), \tag{23}$$

The tilde "~" and overbar "−" represent Favre averaging and Reynolds averaging respectively. The working fluid is air and it is treated as calorically perfect gas while γ and the Prandtl number P_r are held constant at 1.4 and 0.72 respectively. μ is evaluated by Sutherland's law and is based on a reference viscosity of $1.7894 \times 10^{-5}\,\frac{kg}{ms}$ together with a reference temperature of 288.15K and Sutherland's constant at 110K. If the Boussinesq assumption holds, the Reynolds stress τ^t_{ij} can be written as a linear function of the mean flow gradient:

$$\tau^t_{ij} = 2\mu_t(\tilde{S_{ij}} - \frac{1}{3}\frac{\partial \tilde{v_k}}{\partial x_k}\delta_{ij}) \tag{24}$$

Turbulent viscosity μ_t is computed by turbulence models. In this work, the Wilcox $k-\omega$ turbulence model is used and additional equations are required for k and ω. Readers can refer to Wilcox [89] for more details.

5.3.2 *Spatial Discretization*

The flow variables are stored at the cell centres and the boundary conditions are applied at the ghost cells, the positions of which are generated by mirroring

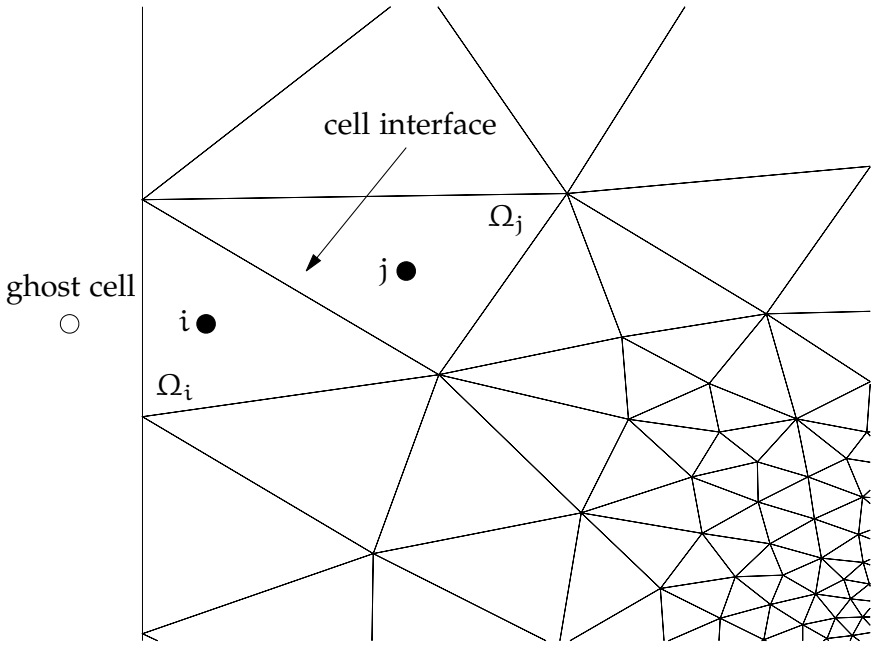


Figure 16: Schematic of the cell-centred finite volume scheme on unstructured grids.

the positions of the cells immediately adjacent to the boundary. The inviscid and viscous fluxes are evaluated at the cell-to-cell and boundary-to-cell interfaces. The schematic of the finite volume scheme is shown in Figure 16. Flow gradient is computed at the cell centre using the weighted least square procedure [51]. The matrix of the weighted least square gradient is evaluated once at pre-processing for static grids and at every nonlinear iteration for moving meshes.

The inviscid fluxes are computed by the upwind scheme using the approximated Riemann solver of Roe [73]. Second order spatial discretization is obtained by extrapolating the values from the cell centre to the interface via the Monotonic Upwind Scheme for Conservation Laws (MUSCL) [47] with the van Albada limiter [39]. The viscous fluxes at the interface are computed by using the inverse of the distance weighting from the ones evaluated at the cell centres on both sides of the interface while source terms are evaluated at the cell centres and are assumed to be piecewise constant in the cell.

5.3.3 *Time integration*

After inviscid, viscous fluxes and source terms are computed for each cell, the coupled system in Equation 22 can be described as the following:

$$\Omega_i \frac{dU_i}{dt} = -\sum_{j=1}^{N} R_i(U_j) \tag{25}$$

Where U_i are the conservative variables of cell i, namely $(\overline{\rho}, \overline{\rho}\tilde{v_i}, \overline{\rho}\tilde{E})^T$, Ω_i the cell volume, U_j the conservative variables of the neighbouring cells of U_i, N the number of neighbouring cells and R_i the right hand side of cell i, which are the fluxes evaluated at each cell. Here we assume no mesh motion and Ω_i remains a constant for each cell in the computation.

The system in Equation 22 is solved implicitly by first applying the backward Euler scheme:

$$\Omega_i \frac{\Delta U_i}{\Delta t} = -\sum_{j=1}^{N} R_i(U_j^{n+1}) \tag{26}$$

Where n is the solution at the current level, $n+1$ is the solution to be solved in the next level and $\Delta U_i = U_i^{n+1} - U_i^n$

Expanding $R_i(U_j^{n+1})$ in Taylor series, Equation 26 becomes:

$$\Omega_i \frac{\Delta U_i}{\Delta t} = -\sum_{j=1}^{N} R_i(U_j^n) - \sum_{j=1}^{N} \frac{\partial(R_i(U_j^n))}{\partial U_j} \Delta U_j^n \tag{27}$$

Where $\frac{\partial(R_i(U_j^n))}{\partial U_j}$ is the flux Jacobian. Equation 27 can be re-arranged and the flux Jacobian is approximated by its spectral radius. The resulting linear system reads:

$$[J_i^n(\frac{\Omega_i}{J_i^n \Delta t} + 1)]\Delta U_i^n = -\sum_{j=1}^{N} R_i(U_j^n) \tag{28}$$

Equation 28 is the resulting linear system to march the solution from time level n to $n+1$, and it is solved by the Newton-Jacobi method where J_i^n is the spectral radius of the flux Jacobian matrix which is accumulated across the cell interfaces. Linearised fluxes $\frac{\partial(R_i(U_j^n))}{\partial U_j}\Delta U_j^n$ are required to update the solutions at each Newton-Jacobi iteration and they are evaluated exactly for the inviscid and viscous fluxes. For the Wilcox $k-\omega$ turbulence models, an approximation is used for linearising the governing equations for k and ω. The Newton-Jacobi is executed for user-specified iterations to march the solution from n to $n+1$, the right and left hand sides are then updated, and the Newton-Jacobi is invoked again. This process proceeds until a user-specified convergence criteria is met.

5.3.4 *Test case*

The numerical test case used for the optimisation study represents an aero-engine intake operating near ground. Validation and numerical investigation using the AU3X code have been previously presented by Carnevale et al [17],[18] using experimental data provided by Murphy et al [61]. The computational domain based on an unstructured mesh can be seen in Figure 17. Near wall regions have been discretized with hexahedral elements for boundary layer prediction whilst prismatic elements have been used in the free stream domain. Furthermore, two mesh sizes have been utilised throughout this work, namely `mesh 1` which contains $3.3x10^6$ elements and `mesh 2` with $6x10^6$ elements respectively.

5.3.5 *Computational kernels*

Table 3 presents details of the face-based and cell-based kernels used to demonstrate the implementation and the impact of optimisations. Face-based loops are represented by four kernels computing the inviscid, viscous and linearised fluxes while cell-based loops are represented by linearised updates to the primitive variables. Face-based loops are characterised by a relatively high arithmetic intensity as they involve a large number of floating point operations per pair of adjacent cells whereas cell-based loops tend to exhibit a modest number of calculations per memory load operation. Therefore, it is expected that

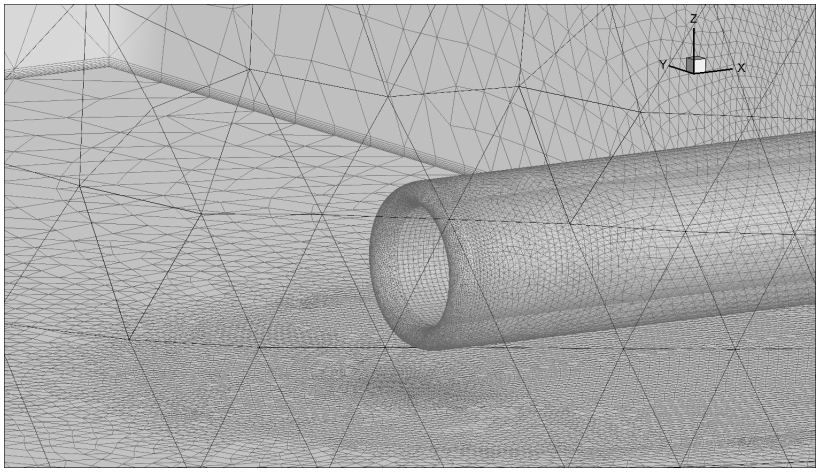


Figure 17: Unstructured mesh of intake with hexahedra elements at near wall regions and prisms in the free stream domain.

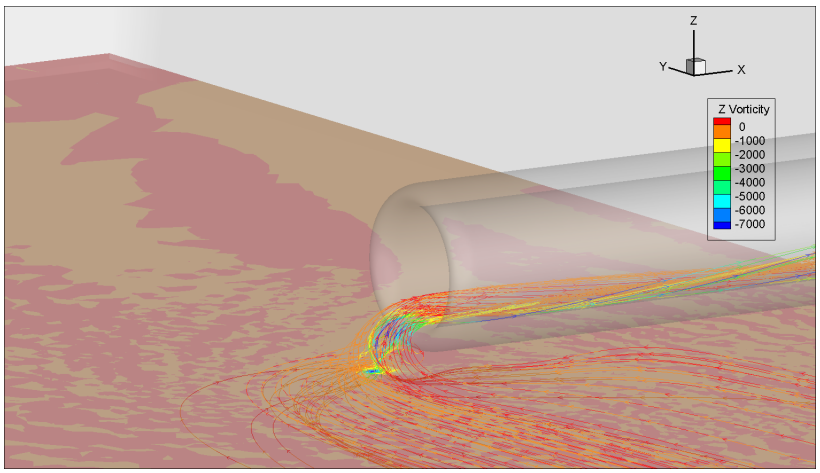


Figure 18: Solution of ground vortex ingestion highlighting the vorticity in the normal to the ground direction.

face-based loops, and especially the second order inviscid MUSCL fluxes, will benefit the most from optimisations that improve the throughput of floating point computations such as vectorization and to scale with the available number of computational cores. For cell-based loops, it is expected that they scale with the available memory bandwidth although it will be of interest to assess the impact that other optimisations have on their performance such as vectorization or data layout transformations and whether optimisations for face-based kernels have any negative impact on their performance.

The optimisations presented for the kernels in Table 3 have been implemented across all other face-based and cell-based kernels in the application. This is reflected in the results presenting whole application performance as time per solution update (Newton-Jacobi iteration).

kernel	loop	runtime (%)	flops/bytes	description
iflux	faces	13	1.30	second order TVD MUSCL fluxes
vflux	faces	8	0.80	viscous fluxes
diflux	faces	32	0.84	linearised inviscid fluxes
dvflux	faces	30	0.80	linearised viscous fluxes
dvar	cells	5	0.18	update to primitive variables

Table 3: Computational kernels representing face-based and cell-based loops used for demonstrating the implementation and impact of our optimisations.

5.3.6 *Configuration of compute nodes*

Table 4 presents details with respect to the configuration of the compute nodes used in this chapter in terms of processor model, memory configuration and software environment. A broader discussion on the characteristics and architectural features of each processor is given in Chapter 3.

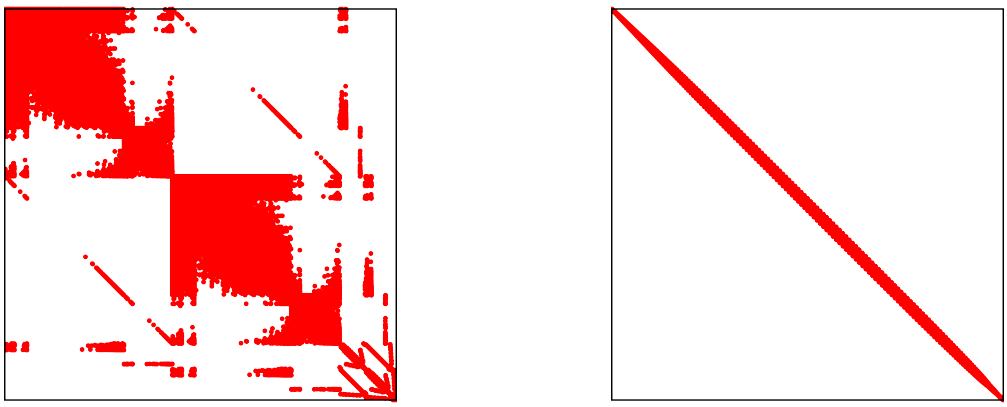
	SNB	BDW	SKL	KNC	KNL
Version	E5-2650	E5-2680	Gold 6140	7120P	7210
Sockets	2	2	2	1	1
Cores	8	14	18	61	64
Threads	2	2	2	4	4
Clock (GHz)	2.0	2.4	2.3	1.2	1.3
SIMD ISA	AVX	AVX2	AVX-512	IMCI	AVX-512
SIMD width	256-bit	256-bit	512-bit	512-bit	512-bit
L1 Cache (KB)	32	32	32	32	32
L2 Cache (KB)	256	256	1024	512	1024
L3 Cache (MB)	20	35	25	-	-
DRAM (GB)	32	128	196	16	96/16
DRAM type	DDR3	DDR4	DDR4	GDDR5	DDR4/MCDRAM
Stream (GB/sec)	71	118	186	181	82/452
Compiler			icpc 17.0		
MPI Library			Intel MPI 2017		

Table 4: Hardware and software configuration of the compute nodes used in this chapter. The SIMD ISA represents the latest vector instruction set architecture supported by the particular platform.

5.4 OPTIMISATIONS

5.4.1 *Grid renumbering*

In order to improve the exploitation of the cache hierarchy in face-based loops, the distance between memory references when gathering and scattering cell-centred data has to be minimized. In this work, this is achieved using the Reverse Cuthill Mckee (RCMK) [20] sparse matrix bandwidth minimizer. The RCMK algorithm reorders the non-zero elements in the adjacency matrix derived from the underlying mesh topology so as to cluster them as close as possible to the main diagonal [15]. An example of the resulting bandwidth reduction when applying RCMK on `mesh 1` can be seen in Figure 19 where the maximum distance to the diagonal has been reduced by 53X. Following the renumbering, the list of faces is sorted in ascending order based on the first index which results in a sequence of first indices that increase monotonically. A subsequent sort is performed on the second index so that in groups of consecutive faces where the first index is constant, the second index reference will be visited in ascending order. This leads to improvements in both spatial and temporal locality as the cells referenced by the first index will be quasi contiguous in memory and therefore able to better exploit the cache hierarchy and hardware prefetchers. Furthermore, minimizing the stride in memory accesses can also reduce the number of TLB misses which are particularly expensive on manycore architectures based on more simple core designs that cannot handle a page walk as efficiently as conventional multicore CPUs. The final sort on the second index further improves memory performance since hardware prefetchers operate best on streams with ordered accesses whether in a forward or backward direction. The mesh reordering is performed immediately after solver initialisation and across all MPI ranks. Each rank is in charge of renumbering its local cells after which it traverses its list of halos in order to relabel the corresponding internal cells with their new value. Since the reordering is performed only once at solver start-up and scales linearly with the number of processors, its effect on the overall application execution time is negligible.



(a) $\beta = 2808603$ (b) $\beta = 52319$

Figure 19: Example of reducing the bandwidth (β) via Reverse Cuthill Mckee on `mesh 1` ($3x10^6$ elements). Figure (a) highlights the initial bandwidth where $\beta = 2808603$ while figure (b) presents the improved bandwidth where $\beta = 52319$ as a result of renumbering.

5.4.2 *Vectorization*

The continuous increase in size of the underlying vector registers combined with steady improvements to the associated SIMD ISA have seen vectorization become indispensable for exploiting the computational power of modern processors. Although compilers have evolved over the years and are generally better at vectorizing non-trivial kernels, they can only do so when they can guarantee safety. Consequently, computational kernels such as face-based loops are not vectorized due to the existence of indirection and potential dependencies when accumulating and scattering back to the cell-centres. Furthermore, even in the case of kernels with regular access patterns such as cell-based loops, compiler auto-vectorization is not always possible due to various reasons such as pointer aliasing, inner function calls or conditional branching. As a result, relying on the compiler to generate vector instructions based only on its internal analysis is not recommended as the success rate will invariably differ across different compilers, programming languages and computational kernels.

The alternative to compiler auto-vectorization is explicit vectorization via the utilisation of directives, lower level intrinsics or inline assembly. In this work, the predominant approach for achieving vectorization was based on OpenMP 4.0 [64] directives. The use of compiler directives has been preferred over other alternatives since they offer far greater flexibility and portability across SIMD architectures and compilers at a small cost in performance compared to lower level implementations such as compiler intrinsics or assembly.

In cell-based loops, vectorization was made possible through the addition of directives either at loop level or at function declaration which not only forces the compiler to vectorize the construct but also enables the generation of efficient vector code based on auxiliary data attributes such as alignment, variable scoping and vector lane length. An example of this can be seen in Listing 11 based on a simplified example of a cell-based loops for updating the unknowns.

```
#pragma omp simd simdlen(VECLEN) safelen(VECLEN) \
                 aligned(q:ALIGN,dq:ALIGN)
for( iq=0;iq<num_cells;iq++ )
{
    q[iq]+=dq[iq];
}
```

Listing 11: Example of a cell-based kernel vectorized with OpenMP 4.0 directives.

For face-based loops however, a complete re-write was necessary in order to switch to a vector programming paradigm. Listing 12 presents the improved vector-friendly layout which replaces the original example in Listing 10 (Section 5.1).

In the new implementation, the original loop is divided into three distinct stages which naturally map to the underlying gather, compute and scatter pattern. In essence, each main loop iteration will process a number of consecutive faces in parallel by exploiting the available vector lanes as defined by the `VECLEN` macro. The first nested loop gathers the unknowns into local short vector arrays. Depending on the underlying architecture, the compiler will either generate SIMD gather instructions or serial load sequences if the architecture does not support such operations. Once all data is loaded into the short vectors, the second nested loop performs the computation. Similarly to the previous stage, computations are carried out in parallel on the available

vector lanes. In AVX/AVX2, four faces would be processed concurrently or eight for IMCI and AVX-512 architectures in double precision. Since all intermediate short vectors are allocated on the stack as static arrays of sizes known at compile time as defined by the pre-processor macros, the compiler can easily generate efficient vector code specific to the underlying architecture as all dependencies have been eliminated. This includes loading contiguous data such as face geometrical properties at aligned addresses as expressed through the aligned clause in the directive. The last stage involves the accumulation and scatter of the residuals to their respective cells. This is done sequentially since generating vector code for this section would lead to incorrect results due to data dependencies as multiple successive faces are processed in parallel.

Grouping the operations specific to the gather, compute and scatter patterns in distinct sections enabled the vectorization of the first two where the majority of instructions are issued while the scatter, which prevented vectorization in the first place, is still performed sequentially. Vectorizing the gather and the computation stages is important since finite volume discretizations involve a large number of floating point operations per pair of cells which if vectorized, can lead to significant speed-ups. The additional nested loops in the new implementation are unrolled automatically at compilation as they decay into single or multiple SIMD instructions therefore removing potential overhead as long as the iteration space (`VECLEN`) is equal or a relatively small multiple of the underlying vector lane size. The promotion from scalar to short vectors is performed only for variables that appear in at least one of the three distinct stages (i.e. `f` representing the flux residual, computed in stage two and written back in stage three).

In addition to re-writing all face-based kernels in the solver following the principles set out in Listing 12, a number of other optimisations were also performed at this stage such as: (i) allocating face and cell data structures on aligned 32 or 64 byte boundaries using `_mm_malloc` depending on the SIMD architecture. (ii) padding of the list of faces through the addition of redundant entries up to a size that is a multiple of `VECLEN`. (iii) the replacement of divisions with reciprocal multiplications in places where the divisors were geometrical variables as SIMD division operations are non-pipelined across the majority of architectures and suffer from very high latencies.

```
# if defined __MIC__
    # define VECLEN 8
    # define ALIGN  64
# elif defined __AVX512F__
    # define VECLEN 8
    # define ALIGN  64
# elif defined __AVX__
    # define VECLEN 4
    # define ALIGN 32
# elif defined __SSE3__
    # define VECLEN 2
    # define ALIGN 16
# else
    # define VECLEN 1
    # define ALIGN 16
# endif

double u1[VECLEN];
double u2[VECLEN];
double f[VECLEN];

 for( ic=0;ic<num_faces;ic+=VECLEN )
{
   // gather data from adjacent cells
    #pragma omp simd simdlen(VECLEN)
    for( iv=0;iv<VECLEN;iv++ )
   {
       u1[iv]= q[ifq[0][ic+iv]];
       u2[iv]= q[ifq[1][ic+iv]];
   }
   // computation
    #pragma omp simd simdlen(VECLEN)  \
    aligned(geo:ALIGN,u1:ALIGN,u2:ALIGN)
    for( iv=0;iv<VECLEN;iv++ )
   {
      f[iv]= geo[ic+iv]*(u2[iv]-u1[iv]);
   }
   // scatter, serially due to dependencies
    for( iv=0;iv<VECLEN;iv++ )
   {
      rhs[ifq[0][ic+iv]]-= f[iv];
      rhs[ifq[1][ic+iv]]+= f[iv];
   }
 }
```

Listing 12: Example of a SIMD friendly implementation of face-based kernels vectorized with OpenMP 4.0 directives.

5.4.3 *Colouring*

The dependencies that prevent vectorization when scattering back to the face end-points can be removed by further colouring and reordering the list of faces. Löhner et al [49] presented two algorithms for this purpose and which were implemented in this work. The first algorithm (Löhner 1) uses a simple colouring approach whereby it reorders the list of faces such that groups of consecutive faces of size equal to the specified vector length (i.e. `VECLEN`) have no dependencies across their end-points. The second algorithm (Löhner 2) is an extension of the first algorithm with the difference that it also attempts to minimize the jumps in the second index within the vector groups. This is achieved by performing additional iterations as defined by a search distance parameter in order to find suitable faces that exhibit no dependencies and keep the one that contributes to the smallest jump in the second index when compared with the first entry of the vector group.

Table 5 presents the jumps in both the first and the second index after reordering the faces using both algorithms on `mesh 1`. The entries `jump1` and `jump1a` represent maximum and average jumps across the first index while `jump2` and `jump2a` have the same meaning but for the second index. It can be observed that Löhner 2 decreases the jumps in the second index as the search distance is increased although these lead to significant increases in the first index. As a result, one has to choose a combination that provides the best trade off between the two (i.e. largest decrease in jumps in the second index over the smallest increases in the first). Consequently, throughout this work, the Löhner 2 algorithm was used with a vector length of 4 and a search distance of 16 for AVX/AVX2 architectures and a vector length of 8 and and a search distance of 16 for the IMCI and AVX-512 platforms.

After removing the dependencies at the face end-points in groups of faces equal to the underlying SIMD length (i.e. `VECLEN`), the scatter sections across all face-based loops were also vectorized using OpenMP 4.0 directives in a similar fashion to the gather and computation sections.

5.4.4 *Array of Structures*

The reference implementation used the SoA layout to store both face and cell-centred variables. For example, unknowns were stored in individual vectors as

algorithm	vector length	search distance	jump1	jump1a	jump2	jump2a
Original	-	-	7	0.4	79490	18869
Löhner 1	4	-	12	1.4	79490	18869
Löhner 2	4	16	69	11.5	79235	6540
Löhner 2	4	32	116	19	79023	6064
Löhner 2	4	64	201	31	79028	5853
Löhner 1	8	-	25	1.9	79028	17296
Löhner 2	8	16	85	11.7	79261	4708
Löhner 2	8	32	158	21.3	79977	3541
Löhner 2	8	64	272	35	79030	3153
Löhner 1	16	-	48	2.3	79039	16258
Löhner 2	16	16	89	10.6	78801	5235
Löhner 2	16	32	224	20.3	79051	2858
Löhner 2	16	64	376	37.5	78922	1941
Löhner 1	32	-	94	2.6	79490	15683
Löhner 2	32	32	208	18.6	79274	3267
Löhner 2	32	64	428	35	79453	1767
Löhner 2	32	128	791	67.3	78750	1104
Löhner 1	64	-	173	2.9	79670	15399
Löhner 2	64	64	412	31.7	97813	2143
Löhner 2	64	128	948	63.2	79521	1119
Löhner 2	64	256	1607	126.1	79287	698.5

Table 5: Jump in indices for `mesh 1` after reordering the faces based on the two algorithms in Löhner et al [49].

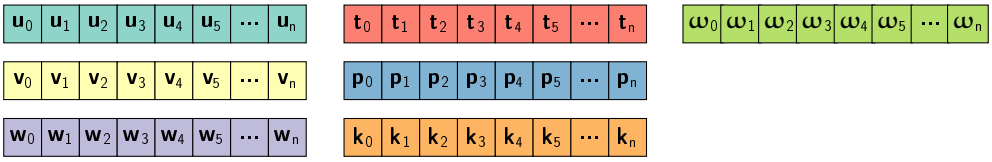


Figure 20: SoA layout for cell-centred variables such as unknowns.

seen in Figure 20 where n is equal to the number of cells in the computational domain.

As faces are processed in consecutive order within face-based loops, accessing face variables such as normals, coordinates and frame speed is done via contiguous vector load operations which map effectively to the CPU vector registers. However, the access for cell-centred variables is indirect and irregular since cells are traversed non-consecutively according to the correspondence array. As a result, for each cell, all cell-centred variables have to be gathered from their respective vectors and into the SIMD registers. For SIMD architectures that do not support vector gather operations, the compiler will generate sequential loads in order to fill in the vector registers. Similarly, scattering values such as residuals back to the corresponding arrays translates into sequential stores if scatter vector instructions are not available. On architectures with available gather and/or scatter support, the number of issued instructions is reduced by a factor equal to the underlying number of vector lanes. However, when compared to regular SIMD load and store operations, gather and scatter primitives are known to suffer from significantly higher latencies [40] especially on the Knights Corner architecture [67]. Consequently, the data structures storing cell-centred variables have been modified to an AoS implementation whilst face data structures were kept in the SoA format.

In the AoS layout, cell-centred variables are stored contiguously in short arrays for every cell as seen in Figure 21 rather than being stored in individual vectors. As a result, although the AoS format does not fully remove all gather and scatter operations, irregular load and stores are only executed once for each cell vector as subsequent successive elements are loaded automatically at cache line granularity as long as the short arrays are padded. This improves locality and minimizes cache misses although the disadvantage compared to SoA is that variables have to be transposed into their respective vector register and lane positions.

Figure 21: AoS layout for cell-centred variables such as unknowns including padding.

The advantage of using AoS over SoA in loops with gather and scatter operations can be demonstrated by analysing the number of cache lines required to gather the unknowns in a single vector iteration for both layout formats. For the purpose of such analysis, VLEN represents the vector register size, NVAR the number of variables to be gathered per cell, NCELL the number of cells and CLEN the number of variables that can be stored in one cache line. Therefore, VLEN is equal to either 4 or 8 depending on the underlying SIMD architecture (i.e. AVX/AVX2 or IMCI/AVX-512), $\text{NVAR} = 7$ for the unknowns since there are 5 variables for the flow and 2 for the turbulence model, $\text{CLEN} = 8$ in double precision since eight variables will occupy an entire 64-byte cache line and finally, $\text{NCELL} = 2 \times \text{VLEN}$ as each face is shared by two cells and the indices referencing the cells are unique across every iteration due to colouring. In the best case scenario, the indices referencing the pairs of neighbouring cells within a vector iteration are ordered consecutively among themselves while in the worst case, the cell indices will be at a distance greater than the size of a cache line (i.e. $> \text{CLEN}$). As a result, the number of cache lines accessed by the SoA layout in the best case scenario would be $\text{NVAR} \times (\text{NCELL} \div \text{CLEN})$ while for AoS, this would be $\text{NCELL} \times (\text{NVAR} \div \text{CLEN})$. In the worst case scenario, SoA would require $\text{NCELL} \times \text{NVAR}$ cache lines whereas for AoS this would remain as $\text{NCELL} \times (\text{NVAR} \div \text{CLEN})$ due to the fact that all of the successive unknowns within a cell fit across a single cache line after padding. Therefore, for AVX/AVX2, the SoA layout would require 7 cache lines in the best case scenario and 56 in the worst. In the AoS layout, both the best and worst case would require 8 cache lines. For IMCI/AVX-512, one can simply multiply the above numbers by a factor of two where the best and worst case for SoA would be 14 and 112 respectively and 16 across both cases for AoS.

Consequently, although SoA is superior in the best case scenario where the visited cells are contiguous in memory, it performs worse by a factor equal to the number of variables that are gathered per cell compared to AoS when the distance between the face end-points is larger than the size of the cache line. Judging by the average indices in Table 5, one can clearly observe that for both the first and the second index, the average jumps resulting from reordering the

list of faces using the Löhner 2 algorithm are larger than the size of a single cache line (>8). As a result, the AoS layout is in theory the optimal choice since it significantly reduces the number of cache lines required per vector iteration and should therefore improve the performance of the cache hierarchy.

The transition from SoA to AoS was by no means a trivial endeavour as it required significant changes to the message passing interface, array access semantics and switching the order of all nested loop constructs that manipulated cell centred data. It is therefore recommended that some form of abstraction is implemented with regards to the layout in memory of such data structures so that a switch between different implementations can be performed at compile time which would be a useful design trait. This is important when considering that for structured codes, SoA or a hybrid thereof, is the better performing implementation as demonstrated in Chapter 4 since cells are traversed consecutively following an i,j,k indexing system which therefore allows for efficient vector load and store operations. Finally, with regard to unstructured grid applications, modifying cell data to the AoS layout has an impact on cell-based kernels. Whereas previously with SoA, these kernels could exploit SIMD contiguous load store operations as cells were traversed in successive order, switching to AoS means that variables for each cell have to be loaded in vector registers and subsequently transposed into the SoA format which adds extra latency and decreases performance. It is therefore important to empirically evaluate the impact of this optimisation as performance will be lost in the cell-based loops although improvements are expected for the face-based constructs which are the main bottleneck in the application.

5.4.5 *Gather Scatter Optimisations*

The AoS memory layout for cell-centred variables requires that gathers, arising from indirect addressing, are only executed once per structure and not for every variable as it is the case with SoA if the distance between cells is larger than that of the underlying cache line. However, successive elements in AoS although contiguous in memory, need to be transposed into the correct vector register and lane positions. If one considers the seven unknowns in the AoS layout with padding in the last position at a given cell index i, depending on the underlying SIMD architecture (i.e 256-bit or 512-bit wide register), all of the eight double precision values can be loaded from the structure with

either one or two aligned vector load instructions for each cell end-point. As vectorization is applied across consecutive faces, the unknowns of all cell end-points need to be re-arranged on the fly and packed into the SoA format. This process is described in Figure 22 which presents the on the fly transposition from AoS to SoA on AVX/AVX2 architectures with 256-bit registers. In this work, the transpose primitives were implemented using compiler intrinsics for each individual SIMD architecture by using instructions that exhibit the lowest latency and highest degree of instruction parallelism as presented in Listings 13,14 and 15 for AVX/AVX2, AVX-512 and IMCI respectively. This is relevant since the SIMD implementations across processors differ significantly and a generic solution would leave performance on the table. For example, the Knights Corner architecture provides swizzle operations which can perform on the fly data multiplexing prior to execution from the register. Consequently, some permutations and shuffles can be done with "zero" penalty whereas on all other architectures, these will be forwarded to an execution port (port 5 on multicores) which can lead to port pressure. Similarly, scatters for writing back results to the face end-points are implemented by transposing back from SoA to AoS and utilising aligned vector stores which is possible since faces have been coloured. Listing 16 presents the integration of the AoS to SoA conversion primitives in a simplified face-based loop. The conversion primitives are also used in cell-based kernels such as the candidate for updating flow variables in order to convert to and from AoS and SoA as efficiently as possible with the distinction that the indices for performing the gather and scatters in cell-based loops are linear and at unit stride.

Finally, similar work to the above has been presented by Pennycook et al [67] for optimising the gather/scatter patterns in molecular dynamics applications. The solution presented here extends their work by supporting additional SIMD architectures and by applying them to unstructured computational fluid dynamics solvers.

5.4.6 *Array of Structures Structure of Arrays*

While the SoA format is ideal for mapping face data to SIMD registers, it requires that multiple memory streams are serviced in parallel for each vector of variables. Since prefetchers can only operate on a limited number of streams, maintaining a large number of them in flight wastes memory bandwidth and

```
 inline void aos2soa(AOS_t *src, int *pos, double dest[NVAR][VECLEN])
{

    __m256d v[NVAR],s[NVAR],p[NVAR];

   // load variables in 256-bit registers
   v[0] = _mm256_load_pd(&src[pos[0]].var[0]);  // u0,v0,w0,t0
   v[4] = _mm256_load_pd(&src[pos[0]].var[4]);  // p0,k0,o0
   v[1] = _mm256_load_pd(&src[pos[1]].var[0]);  // u1,v1,w1,t1
   v[5] = _mm256_load_pd(&src[pos[1]].var[4]);  // p1,k1,o1
   v[2] = _mm256_load_pd(&src[pos[2]].var[0]);  // u2,v2,w2,t2
   v[6] = _mm256_load_pd(&src[pos[2]].var[4]);  // p2,k2,o2
   v[3] = _mm256_load_pd(&src[pos[3]].var[0]);  // u3,v3,w3,t3
   v[7] = _mm256_load_pd(&src[pos[3]].var[4]);  // p3,k3,o3

   // 64-bit wide interleave
   s[0] = _mm256_shuffle_pd(v[0],v[1],0x0);  // u0,u1,w0,w1
   s[4] = _mm256_shuffle_pd(v[4],v[5],0x0);  // p0,p1,o0,o1
   s[1] = _mm256_shuffle_pd(v[0],v[1],0xF);  // v0,v1,t0,t1
   s[5] = _mm256_shuffle_pd(v[4],v[5],0xF);  // k0,k1
   s[2] = _mm256_shuffle_pd(v[2],v[3],0x0);  // u2,u3,w2,w3
   s[6] = _mm256_shuffle_pd(v[6],v[7],0x0);  // p2,p3,o2,o3
   s[3] = _mm256_shuffle_pd(v[2],v[3],0xF);  // v2,v3,t2,t3
   s[7] = _mm256_shuffle_pd(v[6],v[7],0xF);  // k2,k3

   // 128-bit wide interleave
   p[0] = _mm256_permute2f128_pd(s[0],s[2],0x20);  // u0,u1,u2,u3
   p[1] = _mm256_permute2f128_pd(s[1],s[3],0x20);  // v0,v1,v2,v3
   p[2] = _mm256_permute2f128_pd(s[0],s[2],0x31);  // w0,w1,w2,w3
   p[3] = _mm256_permute2f128_pd(s[1],s[3],0x31);  // t0,t1,t2,t2
   p[4] = _mm256_permute2f128_pd(s[4],s[6],0x20);  // p0,p1,p2,p3
   p[5] = _mm256_permute2f128_pd(s[5],s[7],0x20);  // k0,k1,k2,k3
   p[6] = _mm256_permute2f128_pd(s[4],s[6],0x31);  // o0,o1,o2,o3

   // store in SoA format
   _mm256_store_pd(&dest[0][0],p[0]);
   _mm256_store_pd(&dest[1][0],p[1]);
   _mm256_store_pd(&dest[2][0],p[2]);
   _mm256_store_pd(&dest[3][0],p[3]);
   _mm256_store_pd(&dest[4][0],p[4]);
   _mm256_store_pd(&dest[5][0],p[5]);
   _mm256_store_pd(&dest[6][0],p[6]);

}
```

Listing 13: Example of AVX/AVX2 compiler intrinsics kernel for in-register transposition from AoS to SoA of unknowns.

```
 inline void aos2soa(AOS_t *src, int *pos, double dest[NVAR][VECLEN])
{
    __m512d v[NVAR],u[NVAR],p[NVAR],s[NVAR];

    // load variables in 512-bit registers
    v[0] = _mm512_load_pd(&src[pos[0]].var[0]); // u0,v0,w0,t0,p0,...
    v[1] = _mm512_load_pd(&src[pos[1]].var[0]); // u1,v1,w1,t1,p1,...
    // truncated, repeat for rows 2-7

    // 64-bit wide interleave
    u[0] = _mm512_unpacklo_pd(v[0],v[1]); // u0,u1,w0,w1,p0,p1,o0,o1
    u[1] = _mm512_unpacklo_pd(v[2],v[3]); // u2,u3,w2,w3,p2,p3,o2,o3
    u[2] = _mm512_unpacklo_pd(v[4],v[5]); // u4,u5,w4,w5,p4,p5,o4,o5
    u[3] = _mm512_unpacklo_pd(v[6],v[7]); // u6,u7,w6,w7,p6,p7,o6,o7
    u[4] = _mm512_unpackhi_pd(v[0],v[1]); // v0,v1,t0,t1,k0,k1
    u[5] = _mm512_unpackhi_pd(v[2],v[3]); // v2,v3,t2,t3,k2,k3
    u[6] = _mm512_unpackhi_pd(v[4],v[5]); // v4,v5,t4,t5,k4,k5
    u[7] = _mm512_unpackhi_pd(v[6],v[7]); // v6,v7,t6,t7,k6,k7

    // 128-bit wide interleave
    p[0] = _mm512_mask_permutex_pd(u[0],0xCC,u[1],0x44); // u0-3,p0-3
    p[2] = _mm512_mask_permutex_pd(u[4],0xCC,u[5],0x44); // v0-3,k0-3
    p[4] = _mm512_mask_permutex_pd(u[1],0x33,u[0],0xEE); // w0-3,o0-3
    p[6] = _mm512_mask_permutex_pd(u[5],0x33,u[4],0xEE); // t0-3
    p[1] = _mm512_mask_permutex_pd(u[2],0xCC,u[3],0x44); // u4-7,p4-7
    p[3] = _mm512_mask_permutex_pd(u[6],0xCC,u[7],0x44); // v4-7,k4-7
    p[5] = _mm512_mask_permutex_pd(u[3],0x33,u[2],0xEE); // w4-7,o4-7
    p[7] = _mm512_mask_permutex_pd(u[7],0x33,u[6],0xEE); // t4-7

    // 256-bit wide interleave
    s[0] = _mm512_shuffle_f64x2(p[0],p[1],0x44); // u0,u1,u2,u3,u4,...
    s[1] = _mm512_shuffle_f64x2(p[2],p[3],0x44); // v0,v1,v2,v3,v4,...
    s[2] = _mm512_shuffle_f64x2(p[4],p[5],0x44); // w0,w1,w2,w3,w4,...
    s[3] = _mm512_shuffle_f64x2(p[6],p[7],0x44); // t0,t1,t2,t3,t4,...
    s[4] = _mm512_shuffle_f64x2(p[0],p[1],0xEE); // p0,p1,p2,p3,p4,...
    s[5] = _mm512_shuffle_f64x2(p[2],p[3],0xEE); // k0,k1,k2,k3,k4,...
    s[6] = _mm512_shuffle_f64x2(p[4],p[5],0xEE); // o0,o1,o2,o3,o4,...

    // store in SoA format
    _mm512_store_pd(&dest[0][0],s[0]);
    _mm512_store_pd(&dest[1][0],s[1]);
    // truncated, repeat for dest[2-6] <- s[2-6]
}
```

Listing 14: Example of AVX-512 compiler intrinsics kernel for in-register transposition from AoS to SoA of unknowns.

```
 inline void aos2soa(AOS_t *src, int *pos, double dest[NVAR][VECLEN])
{

    __m512d tmp0,tmp1;
    __m512d v[NVAR],b[NVAR],p[NVAR];

    // load variables in 512-bit registers
    v[0] = _mm512_load_pd(&src[pos[0]].var[0]); // u0,v0,w0,t0,p0,...
    v[1] = _mm512_load_pd(&src[pos[1]].var[0]); // u1,v1,w1,t1,p1,...

    // u0,u1,t0,t1,u0,u1,t0,t1
    tmp0 = _mm512_mask_blend_pd(0x22,v[0],
           _mm512_swizzle_pd(v[1],_MM_SWIZ_REG_CDAB));
    // u2,u3,t2,t3,u2,u3,t2,t3
    tmp1 = _mm512_mask_blend_pd(0x88,
           _mm512_swizzle_pd(v[2],_MM_SWIZ_REG_BADC),
           _mm512_swizzle_pd(v[3],_MM_SWIZ_REG_AAAA));
    // u0,u1,u2,u3,t0,t1,t2,t3
    b[0] = _mm512_mask_blend_pd(0xCC,tmp0,tmp1);
    // u4,u5,t4,t5,u4,u5,t4,t5
    tmp0 = _mm512_mask_blend_pd(0x22,v[4],
           _mm512_swizzle_pd(v[5],_MM_SWIZ_REG_CDAB));
    // u6,7,t6,t7,u6,u7,t6,t7
    tmp1 = _mm512_mask_blend_pd(0x88,
           _mm512_swizzle_pd(v[6],_MM_SWIZ_REG_BADC),
           _mm512_swizzle_pd(v[7],_MM_SWIZ_REG_AAAA));
    // u4,u5,u6,u7,t4,t5,t6,t7
    b[1] = _mm512_mask_blend_pd(0xCC,tmp0,tmp1);
    // u0,u1,u2,u3,u4,u5,u6,u7
    p[0] =_mm512_castsi512_pd(
           _mm512_mask_permute4f128_epi32(
           _mm512_castpd_si512(b[0]), 0xFF00,
           _mm512_castpd_si512(b[1]),_MM_PERM_BABA));
    // t0,t1,t2,t3,t4,t5,t6,7
    p[3] =_mm512_castsi512_pd(
          _mm512_mask_permute4f128_epi32(
          _mm512_castpd_si512(b[1]), 0xFF,
          _mm512_castpd_si512(b[0]),_MM_PERM_DCDC));
   // repeat for v-k,w-o,p-0

    _mm512_store_pd(&dest[0][0],u0u1u2u3u4u5u6u7);
    _mm512_store_pd(&dest[4][0],t0t1t2t3t4t5t6t7);
    // repeat for other v,w,p,k,o
}
```

Listing 15: Example of IMCI compiler intrinsics kernel for in-register transposition from AoS to SoA of unknowns.

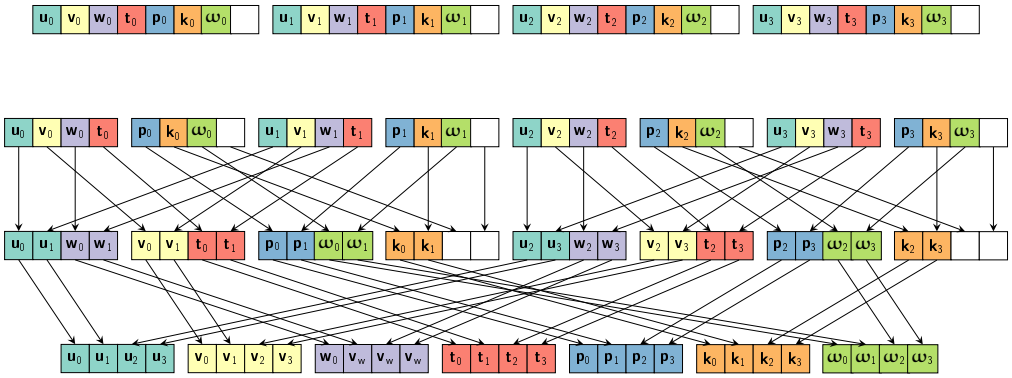


Figure 22: On the fly transposition from AoS to SoA of the unknowns on AVX/AVX2 architectures. Corresponding compiler intrinsics can be seen in listing 13.

```
double u1[NVAR][VECLEN];
double u2[NVAR][VECLEN];
double r1[NVAR][VECLEN];
double r2[NVAR][VECLEN];
double f[NVAR][VECLEN];

 for( ic=0;ic<num_faces;ic+=VECLEN )
{
    // gather
    aos2soa(q,&(ifq[0][ic]),u1);
    aos2soa(q,&(ifq[1][ic]),u2);
    aos2soa(rhs,&(ifq[0][ic]),r1);
    aos2soa(rhs,&(ifq[1][ic]),r2);

    // compute
    #pragma omp simd simdlen(VECLEN)  \
    aligned(geo:ALIGN,u1:ALIGN,u2:ALIGN)
    for( iv=0;iv<VECLEN;iv++ )
   {
        for( jv=0;jv<NVAR;jv++ )
       {
            f[jv][iv]= geo[ic+iv]*(u2[jv][iv]-u1[jv][iv]);
            r1[jv][iv]-= f[jv][iv];
            r2[jv][iv]+= f[jv][iv];
       }
   }
    // scatter
    soa2aos(r1,&(ifq[0][ic]),rhs);
    soa2aos(r2,&(ifq[1][ic]),rhs);
}
```

Listing 16: Integration of primitives for on the fly conversion between AoS and SoA data layouts in a face-based kernel.

other valuable resources. Consequently, the SoA layout can be replaced with the hybrid AoSSoA layout. Essentially, in the AoSSoA layout, face attributes such as normals and coordinates are clubbed together in short vectors equal to the underlying vector register size or a multiple of it. As an example, if one considers the normals to the face in three dimensions: x,y,z on AVX/AVX2 with 256-bit wide registers, the hybrid AoSSoA implementation would map these to the layout presented in Figure 23. This increases locality since distinct variables are stored at a stride equal to one or more vector load and store operations. Furthermore, whereas before, three independent streams were required to load normals in each dimension, the new layout merges this into a single stream whilst still allowing for aligned vector load and store operations.

Figure 23: AoSSoA layout for storing the normals to the face in three dimensions.

5.4.7 *Loop tiling*

The transfer of data to and from the underlying memory hierarchy is the main bottleneck in unstructured mesh applications based on finite volume discretizations. Thus, the implementation of techniques and algorithms that minimize this movement will most likely yield improvements in performance. In this work, this was accomplished via loop tiling, also known as loop blocking or strip mining, with the purpose of reusing data across multiple kernels before writing the results back to main memory. The benefits of such optimisations in unstructured mesh applications have previously been demonstrated by Giles et al [33] via an analytical study.

The implementation of loop tiling in this work was limited to the linear solver, and more specifically, to the computation of linearised fluxes where the largest amount of time was being spent. This consisted in evaluating both linearised inviscid and viscous flux computations over a unified iteration space. In essence, the inviscid fluxes would be computed first for a pre-set amount of faces followed by viscous computations. This sequence would be repeated until all faces of the domain have been processed. The number of faces to be processed in each set of iterations was determined after telescoping through

various multiples of the underlying register size in order to block the data in the L2 cache on each architecture.

5.4.8 *Software Prefetching*

In structured codes with contiguous and regular access patterns, memory parallelism is exploited by the hardware transparently via the available hardware prefetchers across the different cache levels. In unstructured codes, the irregular and indirect access patterns make prefetching more difficult to accomplish by the hardware. In the majority of cases, hardware prefetchers are unable to anticipate which data is required in the upcoming iterations of face-based loops due to indirection and the non-consecutive traversal of the face endpoints. However, this limitation can be addressed through the programmer's domain knowledge since the order of traversal is known at run-time and can be deduced from the associated connectivity arrays.

The prefetch strategy implemented in this work is based on compiler intrinsics and auto-tuning. Prefetch instructions are executed inside the routines that gather and transpose cell-centred data from AoS to SoA as demonstrated in Listing 17. In this manner, as data is gathered and transposed from the group of cells traversed in vector iteration $\mathtt{i}$, prefetches are also issued for the data of all the cells that will be visited in the vector iteration $\mathtt{i} + \mathtt{d}$ where $\mathtt{d}$ is a distance parameter.

For software prefetching to be effective, the value of $\mathtt{d}$ has to be chosen carefully. If too small, it will fetch data into the higher cache levels that is no longer needed resulting in unnecessary memory traffic and therefore consuming valuable memory bandwidth. On the other hand, if $\mathtt{d}$ is too large, the prefetched data will be evicted by the time it is referenced therefore increasing the number of cache misses and cache line replacements. The challenge of selecting the optimal value of $\mathtt{d}$ is exacerbated by the fact that this will invariably differ across computational kernels and processors since it depends on architectural specific metrics such as latencies of caches and main memory as well as kernel specific characteristics such as number of instructions executed per loop iteration [58],[45]. As a result, finding the optimal value of $\mathtt{d}$ across all kernels and for every distinct architecture can only be achieved by means of auto-tuning. In this work, an auto-tuning phase is executed once on every processor in order to telescope through a range of values for $\mathtt{d}$ as well as for assessing the

optimal placement of prefetch instructions across the available cache levels (i.e. L1 only, L1 and L2, L2 only).

In face-based loops, prefetch instructions are issued for both cell-centred data and for the indices in the connectivity arrays that are used for referencing the face end-points. Otherwise, cache misses that result from accessing the connectivity array would offset any benefits of prefetching the actual data. Furthermore, Ainsworth et al [5] demonstrated that the optimal distance for prefetching the indices is twice the distance used for prefetching the data. Thus, within the auto-tuning phase, the auto-tuner telescopes through multiple ranges of index and data prefetch distances whilst maintaining this ratio and selects the distances that result in the highest speed-up for every face-based kernel. This is demonstrated more clearly in Listing 17.

In cell-based loops, prefetch instructions are issued only for cell-centred data as there is no indirection. The optimal prefetch distance is obtained during the same auto-tuning phase that is performed for the face-based loops.

The efficiency of the software prefetching implementation is also improved by the choice of data structures for the cell-centred variables. A side effect of using the AoS data layout is that all successive variables within the structure are loaded at cache line granularity. This means that compared to SoA, the AoS layout only requires that a single prefetch instruction is issued per structure using the address of the first variable since consecutive entries will be loaded as well in the same cache line.

5.4.9 *Multithreading*

Thread-level parallelism has been exploited in the application via the utilisation of OpenMP directives and specifically targets the Intel Xeon Phi manycore architectures. For the multicore CPUs, multi-threading in conjunction with MPI did not bring forth any performance improvement and was therefore abandoned. However, on the Intel Xeon Phi architecture, running more than one thread context per physical core is highly encouraged especially for Knights Corner where it can hide memory latencies due to the in-order core execution engine. Although exposing parallelism in loops over cells is trivial, exposing thread parallelism in face loops is more challenging and requires colour concurrency. There are a number of approaches which can be utilised in respect to the latter [10]. In this work, the available colouring algorithms

```
# if defined L1_INDEX
    # define L1_DATA (L1_INDEX >> 1)
# endif
# if defined L2_INDEX
    # define L2_DATA (L2_INDEX >> 1)
# endif

 inline void prefetchi( int *pos )
{
# if defined L2_INDEX
   _mm_prefetch((char *)&(pos[L2_INDEX]),_MM_HINT_T1);
# endif
# if defined L1_INDEX
   _mm_prefetch((char *)&(pos[L1_INDEX]),_MM_HINT_T0);
# endif
}

 inline void prefetchd( AOS_t *data, int *pos )
{
# if defined L2_INDEX
    for( int i=0;i<NVAR;i++ )
       _mm_prefetch((char *)&(data[pos[L2_DATA]].var[i]),_MM_HINT_T1);
# endif
# if defined L1_INDEX
    for( int i=0;i<NVAR;i++ )
       _mm_prefetch((char *)&(data[pos[L1_DATA]].var[i]),_MM_HINT_T0);
# endif
}

 inline void aos2soa(AOS_t *src, int *pos, double dest[NVAR][VECLEN])
{
   ...
   // prefetch the data
   prefetchd(src,pos);
   ...
}
 for( ic=0;ic<num_faces;ic+=VECLEN )
{
    // prefetch the indices once per iteration
    prefetchi(&(ifq[0][ic]));
    prefetchi(&(ifq[1][ic]));
    // gather
    aos2soa(q,&(ifq[0][ic]),u1);
    ...
}
```

Listing 17: Implementation of software prefetching in face-based loops.

were used to reorder the list of faces such that the number of dependency-free faces was not only a multiple of the underlying vector register size but could also be divided so that each active thread per core can process an equal amount of vector iterations. The iteration space was then scheduled as static at a chunk equal to one vector iteration whereby each thread executes vector iterations in a round robin fashion. The main rationale behind this approach is that it guarantees correctness and integrates well with MPI whereby each rank is pinned to a physical core with subsequent threads spawned within the same core domain. This maintains data locality and affinity to the underlying cache hierarchy, reduces traffic caused from the protocols in charge of cache coherence and mitigates against the risk of false sharing. Most importantly, this approach hides latency best especially for Knights Corner since it guarantees that each thread will access different cell-centred data in face-based loops due to colouring and that every miss and stall in L1 can be circumvented by switching between active threads that have data available.

5.5 RESULTS AND DISCUSSIONS

5.5.1 *Effects of optimisations*

Figures 24, 25, 26, 27, 28, 29, 30 and 31 present the impact that each optimisation has on the performance of both classes of computational kernels and on the overall application run-time. Results are averaged across 10 Newton-Jacobi iterations while running on 1 MPI rank. For the Knights Landing system, results in this section were obtained while running in the Quadrant/Cache configuration.

GRID RENUMBERING Renumbering the grid via the RCMK algorithm led to minor improvements in performance for face-based kernels. Unsurprisingly, the highest impact is observed in kernels with a lower flop per byte ratio represented by the computation of linearised inviscid and viscous fluxes and where the speed-up is as high as 10%.

VECTORIZATION Vectorization results in the largest increases in performance across all architectures and computational patterns. In face-based kernels, these increases range between 2-5X on the multicore CPUs and 2-3X on

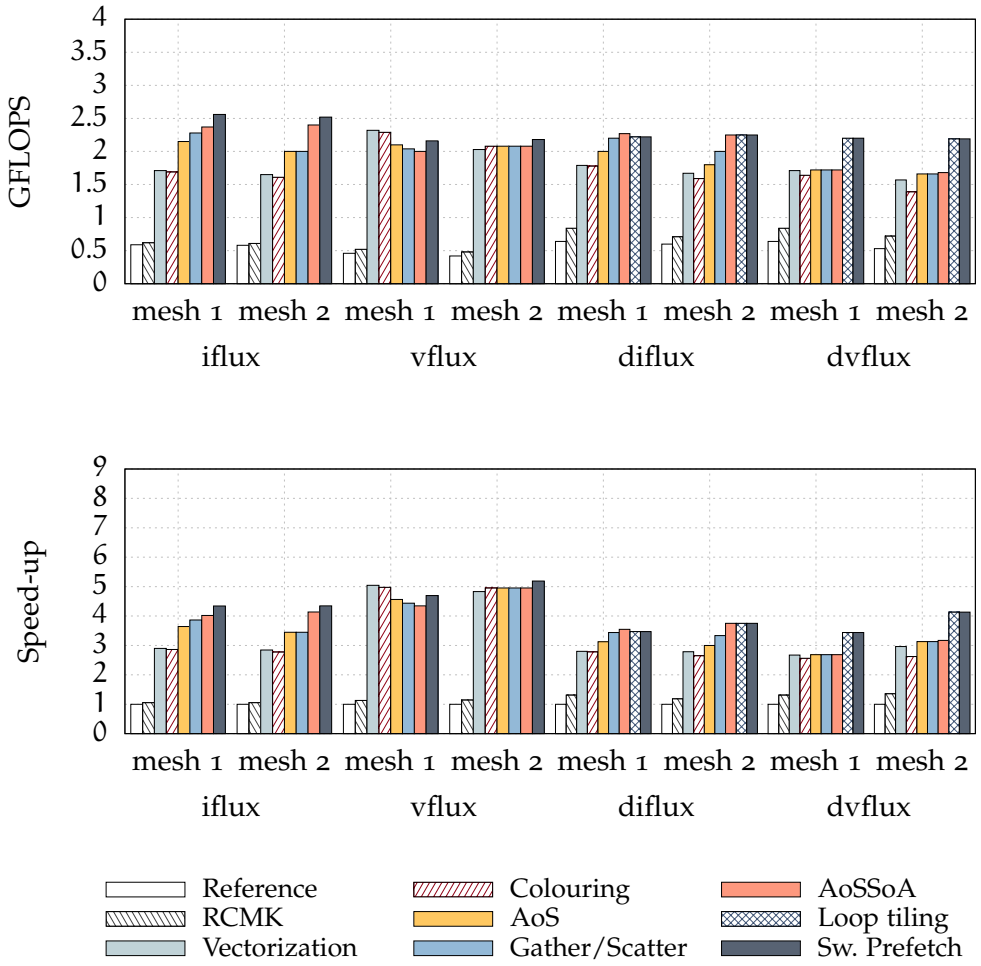


Figure 24: Effects of optimisations on the performance of face-based loops (flux computations) on the Sandy Bridge (SNB E5-2650) system. Results are reported as GFLOPS and Speed-up and were collected by running the application on 1 MPI rank (i.e. single-core).

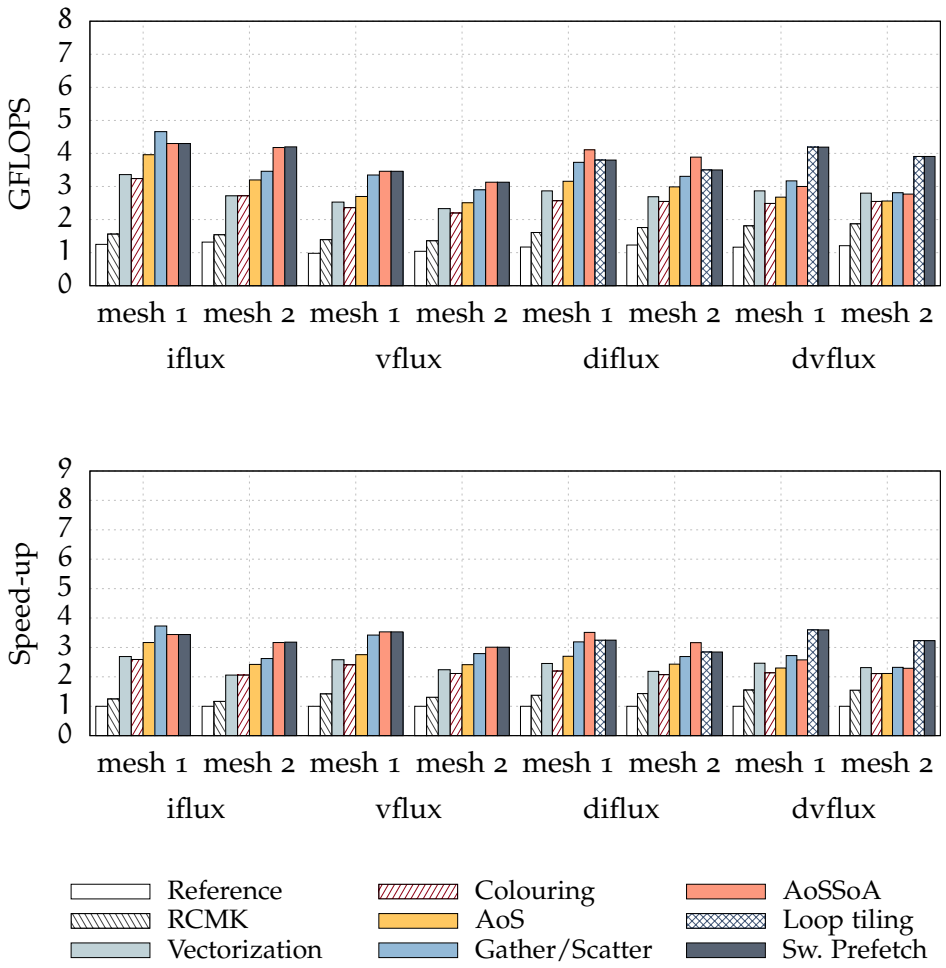


Figure 25: Effects of optimisations on the performance of face-based loops (flux computations) on the Broadwell (BDW E5-2680) system. Results are reported as GFLOPS and Speed-up and were collected by running the application on 1 MPI rank (i.e. single-core).

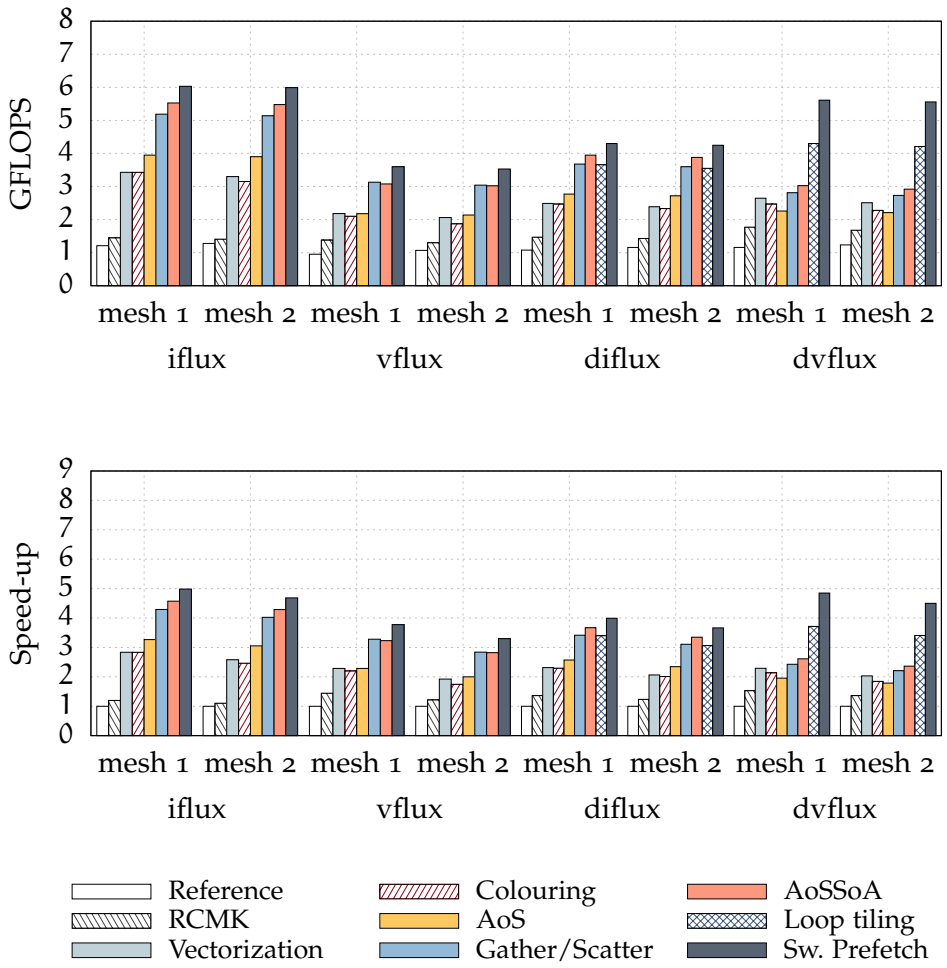


Figure 26: Effects of optimisations on the performance of face-based loops (flux computations) on the Skylake (SKL Gold 6140) system. Results are reported as GFLOPS and Speed-up and were collected by running the application on 1 MPI rank (i.e. single-core).

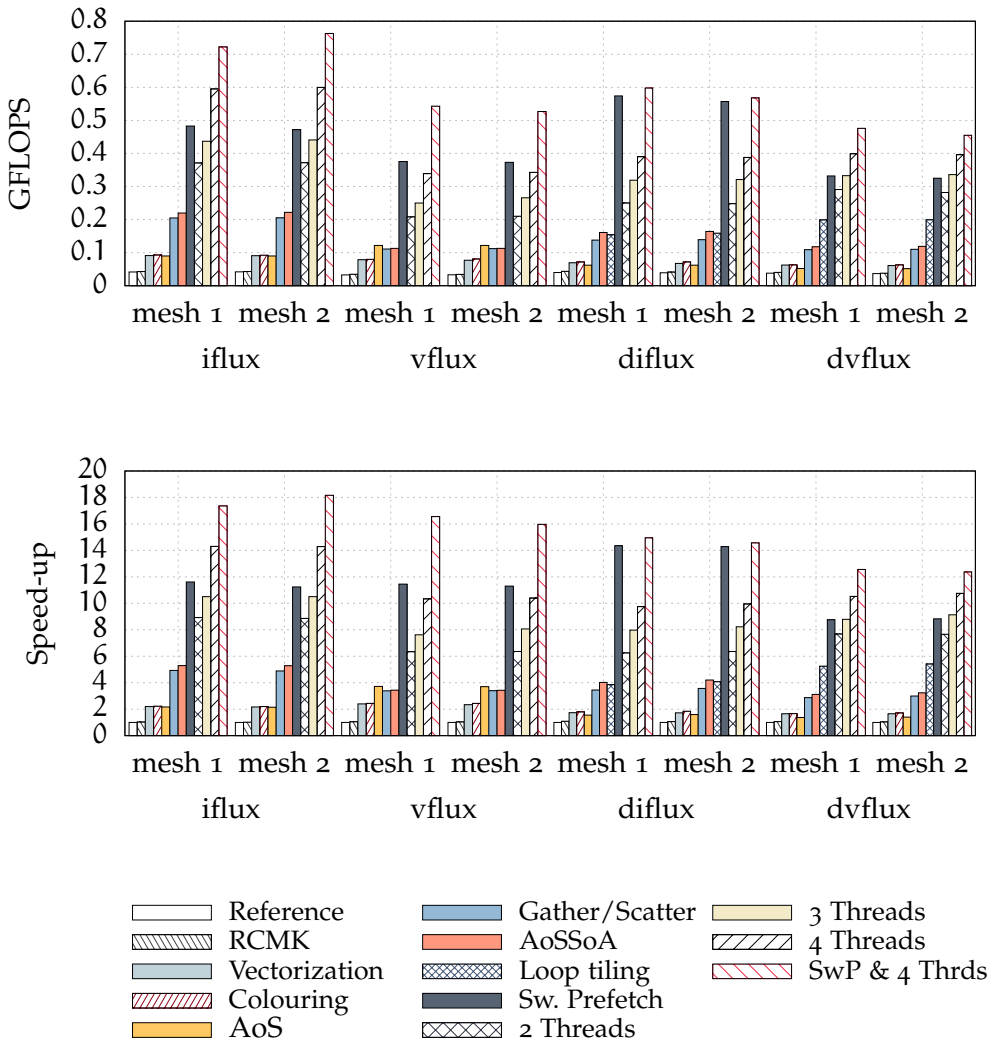


Figure 27: Effects of optimisations on the performance of face-based loops (flux computations) on the Knights Corner (KNC 7120P) system. Results are reported as GFLOPS and Speed-up and were collected by running the application on 1 MPI rank (i.e. single-core).

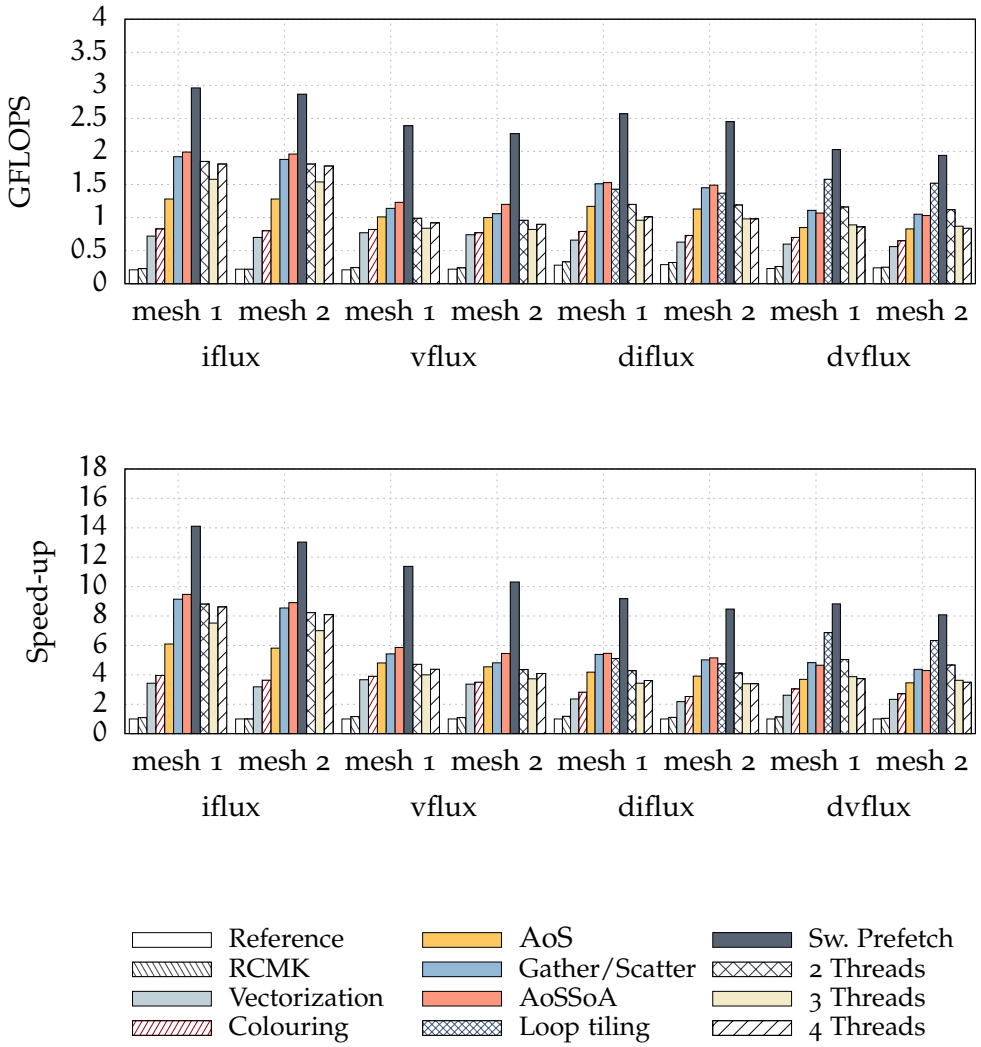


Figure 28: Effects of optimisations on the performance of face-based loops (flux computations) on the Knights Landing (KNL 7210) system. Results are reported as GFLOPS and Speed-up and were collected by running the application on 1 MPI rank (i.e. single-core) and in Quadrant/Cache mode.

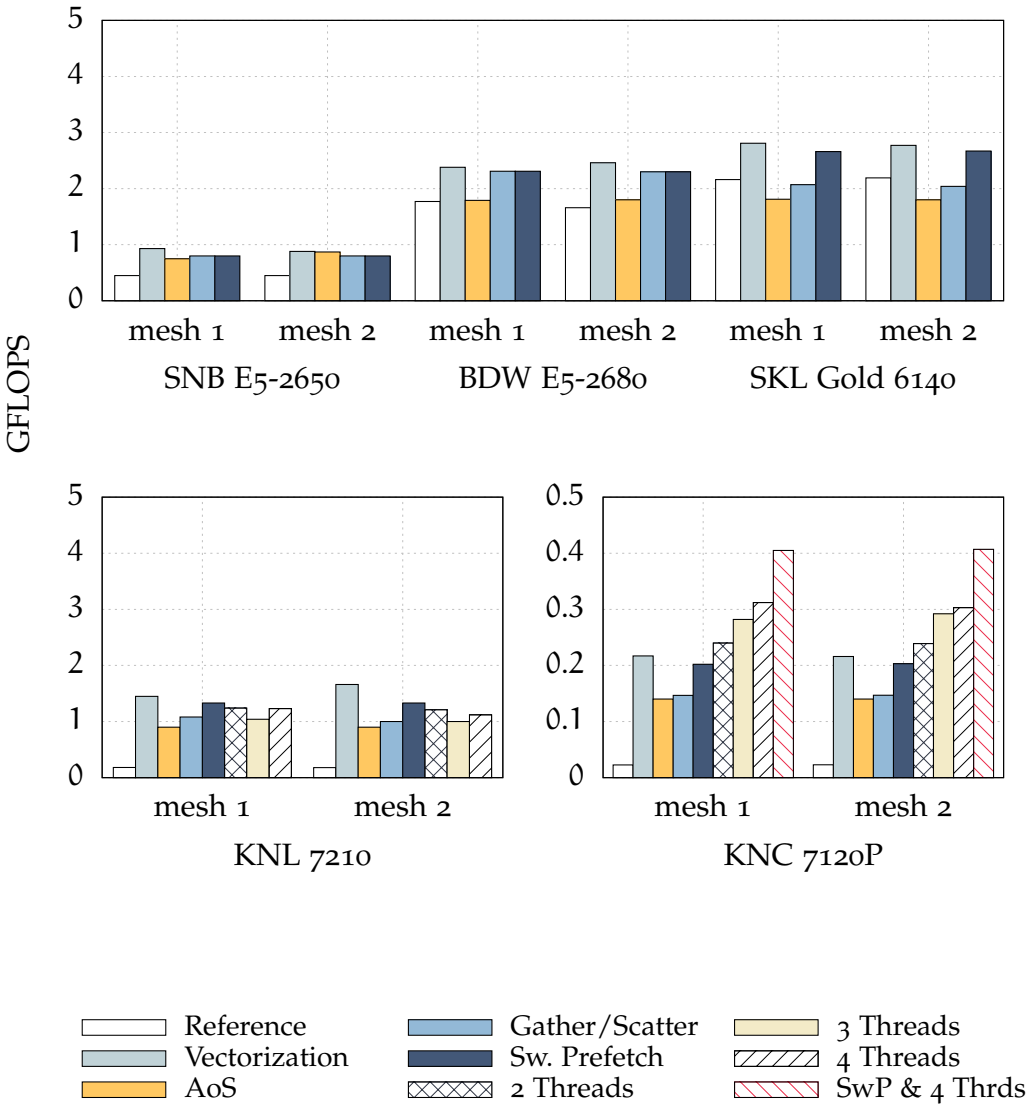


Figure 29: Effects of optimisations on the performance of cell-based loops (update to primitive variables) reported as GFLOPS. Results were collected by running the application on 1 MPI rank (i.e. single-core). The Knights Landing (KNL 7210) system was configured as Quadrant/Cache.

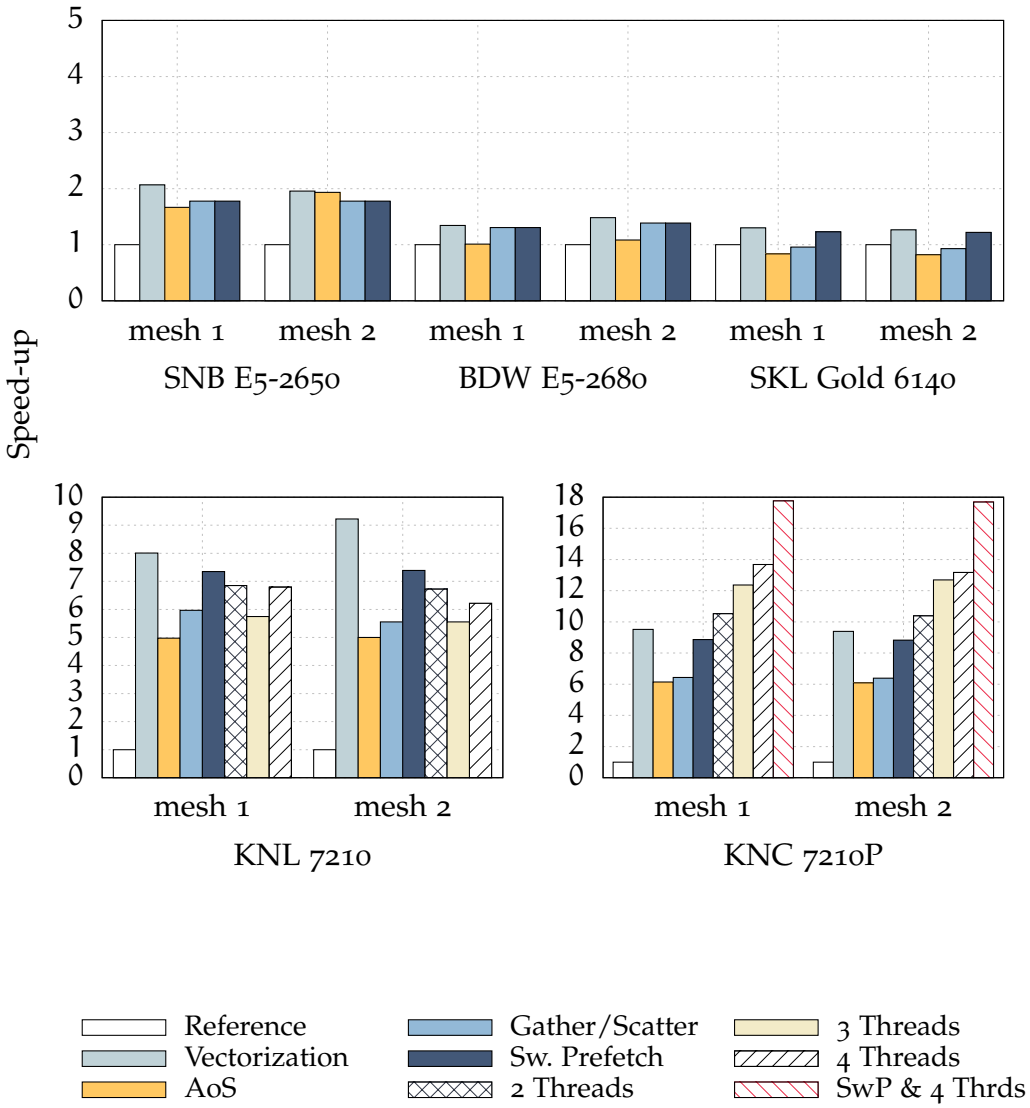


Figure 30: Effects of optimisations on the performance of cell-based loops (update to primitive variables) reported as speed-up relative to the reference implementation. Results were collected by running the application on 1 MPI rank (i.e. single-core). The Knights Landing (KNL 7210) system was configured as Quadrant/Cache.

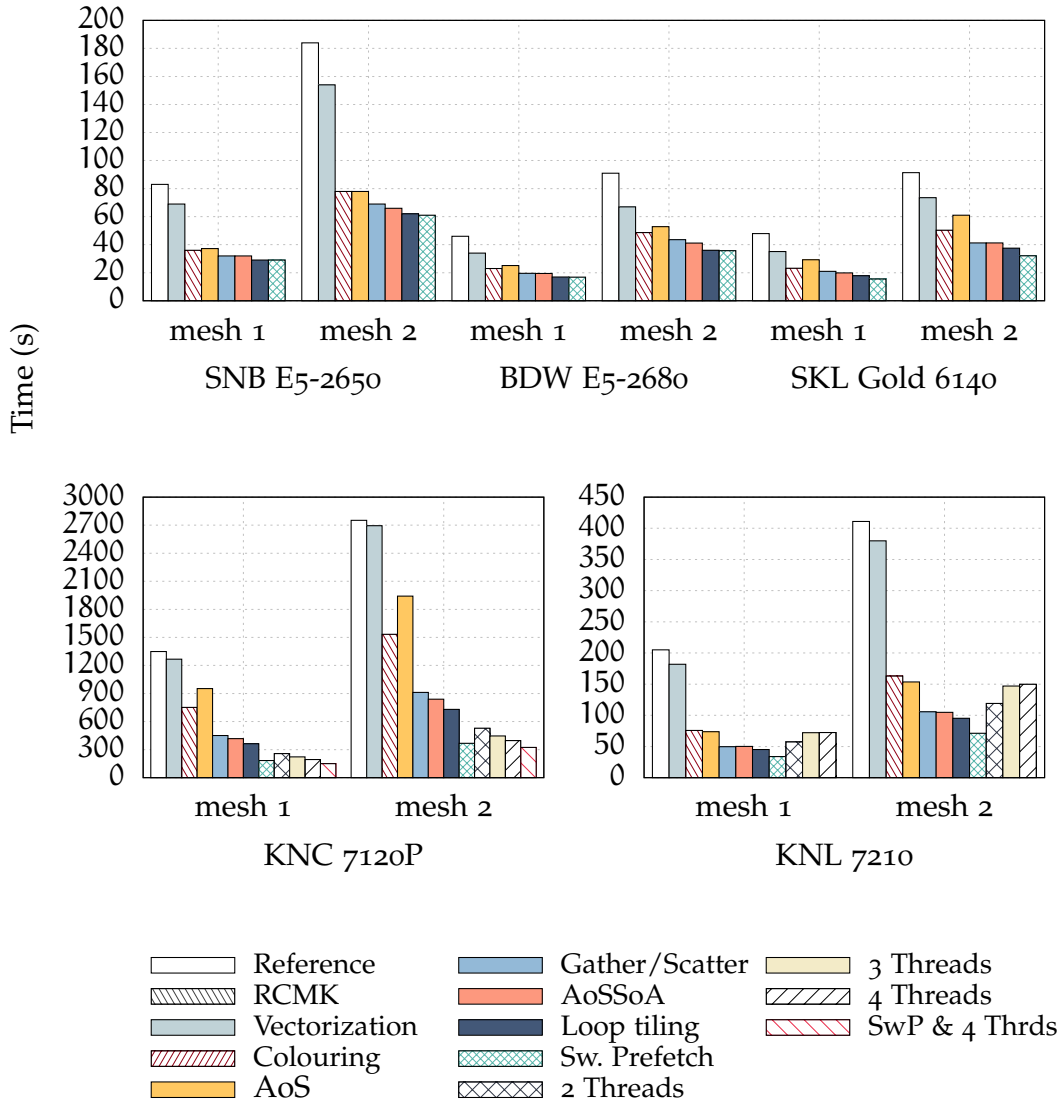


Figure 31: Effects of optimisations on the average time per solution update within a Newton-Jacobi iteration obtained through each optimisation. Results were collected by running the application on 1 MPI rank (i.e. single-core). The Knights Landing (KNL 7210) system was configured as Quadrant/Cache.

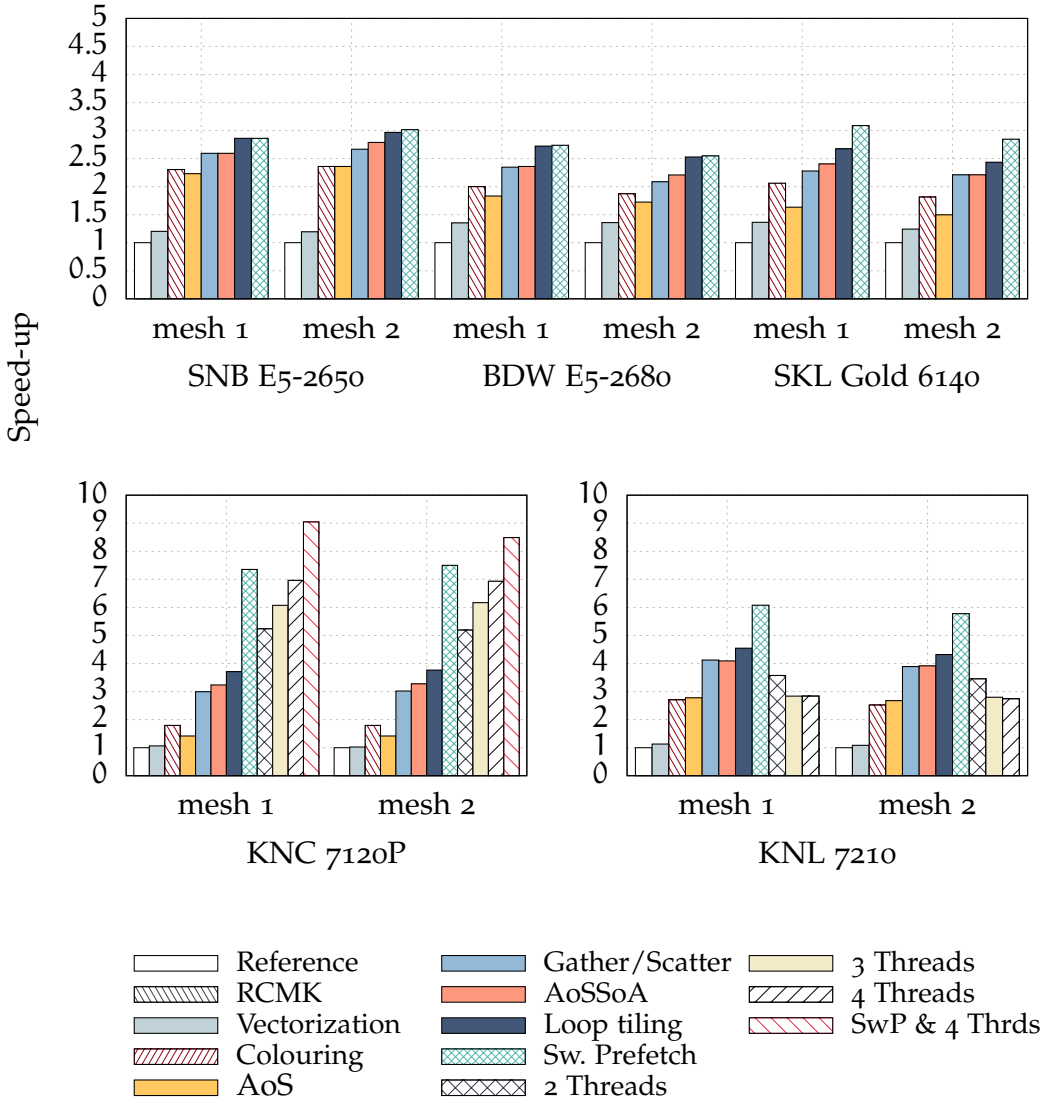


Figure 32: Effects of optimisations on the speed-up per solution update within a Newton-Jacobi iteration obtained through each optimisation relative to the reference implementation. Results were collected by running the application on 1 MPI rank (i.e. single-core). The Knights Landing (KNL 7210) system was configured as Quadrant/Cache.

the Knights Corner and Knights Landing processors. As one can observe, the gains are considerably higher in kernels with a high arithmetic intensity such as the computation of the inviscid second order MUSCL fluxes (iflux) since efficient vector computations are mandatory for achieving a high instruction throughput on modern processors. A similar impact is seen for cell-based kernels where vectorization provided the highest performance impact compared to all other optimisations. Furthermore, all of these translate to an overall application speed-up between 2-3X relative to the original baseline.

COLOURING The colouring/reordering of faces for enabling vectorized scatters of data to the face end-points does not yield any benefits on the multicore CPUs where it is in fact detrimental to performance. In contrast, this is beneficial on the manycore processors and leads to marginal speed-ups although not higher than 10%. This difference might stem from the fact that non-vectorized serial stores are significantly slower on the manycore processors than the SIMD scatter instruction. On the multicore CPUs, the Sandy Bridge and Broadwell architectures implement AVX/AVX2 which do not provide vector scatter functionality. On these processors, scatters are performed as a series of vector insert and store that have a higher cumulative latency than their serial store counterpart. On Skylake, the AVX-512F implementation does provide support for vector scatters. Thus, the small decrease in performance is attributed on this architecture to the increase in the distance across the first and second index due to face reordering which leads to an increase in cache misses due to the limitations of the SoA layout in kernels with irregular access patterns. Nevertheless, colouring is important further down the line once additional optimisations are implemented since it exposes parallelism in writing back values across all face-based loops and allows for the exploitation of thread parallelism.

ARRAY OF STRUCTURES The conversion of cell centred data structures from SoA to AoS provides improvements in performance for some of the face-based kernels on the multicore CPUs. In face-based kernels with a high arithmetic intensity, the speed-up as a result of this conversion is as much as 25% although this is not replicated across the board. On Knights Landing, the AoS layout sees significant speed-ups of as much as 50% across the majority of studied face-based loops. This is opposite to Knights Corner where there is no noticeable improvement. Furthermore, modifying the memory layout of cell-centred data

structures has a negative effect in cell-based loops. This is important when bearing in mind that the overall solver performance decreases after conversion from SoA to AoS. However, as with colouring, this optimisation is required for further optimisations which will exploit the additional data locality available within each vector of cell-centred variables.

GATHER SCATTER OPTIMISATIONS The on the fly transposition from AoS to SoA using specialised intrinsics-based functions improves performance across all processors. The impact is significantly higher on architectures with wider vector registers such as Skylake and the Intel Xeon Phi processors. This is due to the fact that consecutive variables are gathered via a single aligned vector load and subsequently transposed and arranged into the correct lane via architectural specific permute instructions and the fact that the number of SIMD instructions required for the transposition scales logarithmically compared to the serial implementation. It is worth mentioning that on Knights Landing and Skylake with AVX-512F, permute instructions using a single source operand (`vpermpd`) perform better compared to the newly available two source operand instructions (`vpermi2pd`) since they exhibit superior throughput. On Sandy Bridge and Broadwell with AVX/AVX2, the best version is also the one presented in Listing 13 even when compared with other implementations that performed the first step of the shuffle on the load ports with the utilisation of the `vinsertf128` instruction on a memory operand. This would indicate that the bottleneck on these architectures is not port 5 pressure where all of the interleave and shuffle operations are executed.

In cell-based loops, performing the conversion between AoS and SoA on the fly via compiler intrinsics leads to a moderate improvement in performance for the candidate kernel and in some cases, it offsets the performance dropped from switching to AoS from SoA in the first place. More importantly, the implementation of these primitives leads to whole application speed-up on all processors between a few percentages on the multicore CPUs where it amends for the loss in performance due to the switch to AoS in cell-based loops to as much as 50% on the Xeon Phi processors on top of the previous optimisations.

ARRAY OF STRUCTURES STRUCTURE OF ARRAYS The conversion of face data structures to the hybrid AoSSoA layout concerns only face-based loops and yields minor improvements in performance. However, these are not signi-

ficant enough to warrant its usefulness as it did not generate any substantial speed-ups on any architectures. As a result, the effort of implementing such hybrid memory layout within a code of representative size might not be warranted.

LOOP TILING Loop tiling yields a large increase in performance for the computation of the linearised viscous fluxes since groups of consecutive faces processed during the computation of the linearised inviscid fluxes are then passed to the viscous routine and blocked in L2. The best performing tile sizes were 128 for Sandy Bridge, Broadwell and Knights Corner and 512 for Skylake and Knights Landing due to the larger L2 cache. It is therefore believed that extending the scope of such optimisations across the entire application and not just the linear solver would be beneficial across all cache-based architectures. This could be implemented by encapsulating the entire solver over a unified iteration space at a stride equal to the vector register size or a multiple of it. However, the difficulty in such implementation is resolving the dependencies between face-based and subsequent cell-based loops which is mesh dependent and can only be done at run-time.

SOFTWARE PREFETCHING Software prefetching exhibits substantial speed-ups on the Knights Corner architecture due to the in-order core design and on the Skylake and Knights Landing processors as a result of the larger L2 cache. On Knights Corner, the best performing strategy was to issue prefetch instructions at a distance of 32 and 64 for the indices in L1 and L2 respectively and half of that for the actual data as described previously in the methodology. For Knights Landing and Skylake, the best strategy was to only issue prefetch instructions for the L2 cache at a distance of 32 for the index and therefore 16 for the data as any prefetch instructions for L1 were in fact detrimental to performance. The reason for this is most likely related to the smaller size of the L1 cache (32KB) which remained unchanged from Sandy Bridge and Broadwell while the L2 was increased by a factor of four. Furthermore, in the case of Knights Landing, this approach bears fruit since the L1 prefetcher can operate on irregular streams however the L2 is not able to hence why issuing prefetch instructions for L2 in software is advisable. On Sandy Bridge and Broadwell, the advantage of software prefetching is minimal and virtually non existent. This is probably due to the smaller L2 caches on these architectures

which leads to capacity misses. The effect that various combinations of prefetch distances have on the performance of face-based kernels and the overall solver across all architectures obtained as a result of auto-tuning can be inspected in A.1.

For the cell-based loop kernel, software prefetching has no effect on Sandy Bridge and Broadwell but is beneficial on Skylake, Knights Corner and Knights Landing. This is not surprising for Knights Corner where the lack of an L1 hardware prefetcher mandates the utilisation of prefetch instructions either in software or through the compiler even for regular unit stride access patterns. The prefetch distances for cell-based kernels with linear unit stride loads have been selected via auto-tuning as well. For L1, the best linear prefetch distance was found to be 32 and 64 for L2 respectively across Knights Corner, Knights Landing and Skylake.

Finally, an important consideration to take into account is that if software prefetching is implemented explicitly as described above, compiler prefetching should be switched off in order to avoid for competing prefetch instructions to be issued which can degrade performance.

MULTITHREADING Multithreading via OpenMP on Knights Corner yields significant speed-ups with performance increasing almost linearly with the number of threads. The reason for this lies again in the in-order nature of the Knights Corner core where a missed load in the L1 cache leads to a complete pipeline stall. Having more than one thread in flight can help minimize memory latency as context can be switched to a thread for which data is available. Since both software prefetching and multithreading on Knights Corner have the same purpose, the runs with 2,3,4 threads per MPI rank and physical core have been done with software prefetching disabled while for the `SwP & 4 Thrds` analysis, both software prefetching and 4 active threads were utilised per MPI rank. The results indicate that best performance on Knights Corner is obtained when both threading and prefetching are enabled, however, if required to choose between the two, careful software prefetching based on an auto-tuning approach yields better performance than multithreading alone. This is an important aspect to take into account since the implementation of software prefetching is less intrusive and error prone than exposing another level of parallelism in the application for multithreading within an MPI rank. Surprisingly, on Knights Landing, multithreading was actually detrimental to performance.

This most likely due to the fact that the core architecture of Knights Landing is out of order and therefore capable of utilising all core resources with a single thread per core. Running more than one thread per core leads to the division of core resources among the in-flight threads.

5.5.2 *Performance scaling across a node*

Performance strong scaling is performed for each computational kernel and the whole solver across all architectures. For the multicore-based two-socket compute nodes, the baseline implementation is compared across both mesh sizes with the best optimised version. On the Knights Corner co-processor, strong scaling studies are performed for three different versions besides the baseline in order to assess their scalability across the entire chip. These are the optimised version with software prefetching enabled and multithreading disabled, the optimised version with four threads per MPI rank and no software prefetching and lastly, the optimised version with both software prefetching and four threads per rank enabled. On Knights Landing, the difference in performance between baseline and optimised versions is studied with the additional exploration of the memory configurations that are available on this architecture. In cache mode, the MCDRAM is configured as a direct mapped cache and acts as a memory side buffer transparent to the user. In flat mode, all memory allocations are performed explicitly in MCDRAM using the `numactl` utility. For the larger `mesh 2`, approximately 10% of the allocations were re-routed to DRAM memory after utilising all of the available 16GB in MCDRAM. In DRAM mode, the allocation of memory has been done only in DDR memory and MCDRAM has not been utilised at all. The last option was a hybrid evaluation where the storage of cell-centred data structures is allocated in MCDRAM whilst face data is allocated in DDR together with MPI buffers in order to interleave the memory access and therefore exploit the bandwidth of both memories. This has been implemented in the code using the libmemkind interface.

5.5.2.1 *Face-based Loops*

The performance scaling of face-based loops on the multicore nodes (Figure 33) is dependent on the arithmetic intensity of each individual kernel. Best

performance is exhibited by the kernel computing the inviscid second order fluxes (`iflux`) due to its high arithmetic intensity (i.e 1.3) and which scales almost linearly with the number of cores on each node. Good performance is also exhibited by the kernel computing the linearised viscous fluxes (`dvflux`) due to loop tiling which improves its arithmetic intensity since data is accessed from the higher cache levels rather than main memory. Therefore, it can be seen how optimisations that reduce the amount of data movements from the lower memory hierarchies to the processor are beneficial when scaling is performed across the entire device as well.

On the manycore processors (Figures 34 and 35), the difference between baseline and best optimised version for the face-based loops and at full concurrency varies between 12-16X on Knights Corner (60 cores x 4 threads) and 7.3-11.7X on Knights Landing (64 cores) and remain similar to the results on 1 MPI rank.

On Knights Corner (Figure 34), for the compute bound TVD MUSCL fluxes (`iflux`), the best version as one scales across all physical cores is the one that performs both software prefetching and runs with four threads per core. This is followed by the version with no software prefetching and multithreading while software prefetching only is last by a significant margin. However, for the more memory bound face-based loops such as the computation of linearised inviscid fluxes (`diflx`) or the non-linear viscous fluxes (`vflux`), the second best version is software prefetching only while the worst out of the optimised versions is the one consisting of multithreading only (i.e. four threads per MPI rank). This demonstrates that for compute bound kernels, the best approach for exploiting the performance of an in-order architecture is via running more threads whereas for memory bound kernels, a good software prefetching strategy will be more beneficial.

With regard to the performance of the various memory modes on Knights Landing, not exploiting the MCDRAM either as a cache or explicitly as a memory side buffer leads to a factor of three drop in performance once the application scales past 16 cores on the processor. This is to be expected since the difference in bandwidth between MCDRAM and DDR4 is more than a factor of four as measured with STREAM. The best performing version is the interleaved implementation where both MCDRAM and DDR4 are utilised together and where allocation of specific data structures is performed across both interfaces. This is due to the fact that through this approach, one is able

to not only utilise all eight MCDRAM memory channels but also the four available DDR4 controllers. As DDR4 exhibits lower bandwidth but better latency compared to MCDRAM, allocating latency sensitive data structures in DDR4 (i.e. MPI buffers) and all other data structures in MCDRAM allows for the exploitation of both interfaces simultaneously therefore increasing the available memory bandwidth.

5.5.2.2 *Cell-Based Loops*

For the cell-based kernel (Figure 36), the baseline implementation slightly outperforms the optimised version on Skylake and Broadwell as the kernel scales across the node. This is most likely due to the extra latency incurred by the transposition from AoS to SoA in the optimised implementations which becomes more of a bottleneck as memory bandwidth is saturated. The optimised implementation outperforms the baseline on Sandy Bridge and the Xeon Phi processors since serial instructions are significantly slower on the manycore architectures than their vector counterparts and the Sandy Bridge node is more balanced compared to Broadwell and Skylake. As cell-based kernels are memory bound with low flop per byte ratios, they tend to scale with the available memory bandwidth. Therefore, optimisations focused on improving floating-point performance such as vectorization are not as effective as the number of cores are increased.

The negative effect that the conversion from SoA to AoS had on the performance of cell-based loops is made evident in the results for the Broadwell and Skylake systems where the overhead of performing the transposition between the two formats are the main bottleneck at full node concurrency.

On Knights Corner, as with the face-based kernels, best performance was obtained by a large margin when running with 4 threads per MPI rank and software prefetching.

On Knights Landing, the DDR4 interface scales better than with face-based kernels due to the regular unit stride access pattern however, it is almost 3X worse than the other alternatives that exploit the MCDRAM.

5.5.2.3 *Full Application*

Results for whole application strong scaling can be seen in Tables 6, 7 and 8. At full concurrency and on both mesh sizes, the best optimised version is

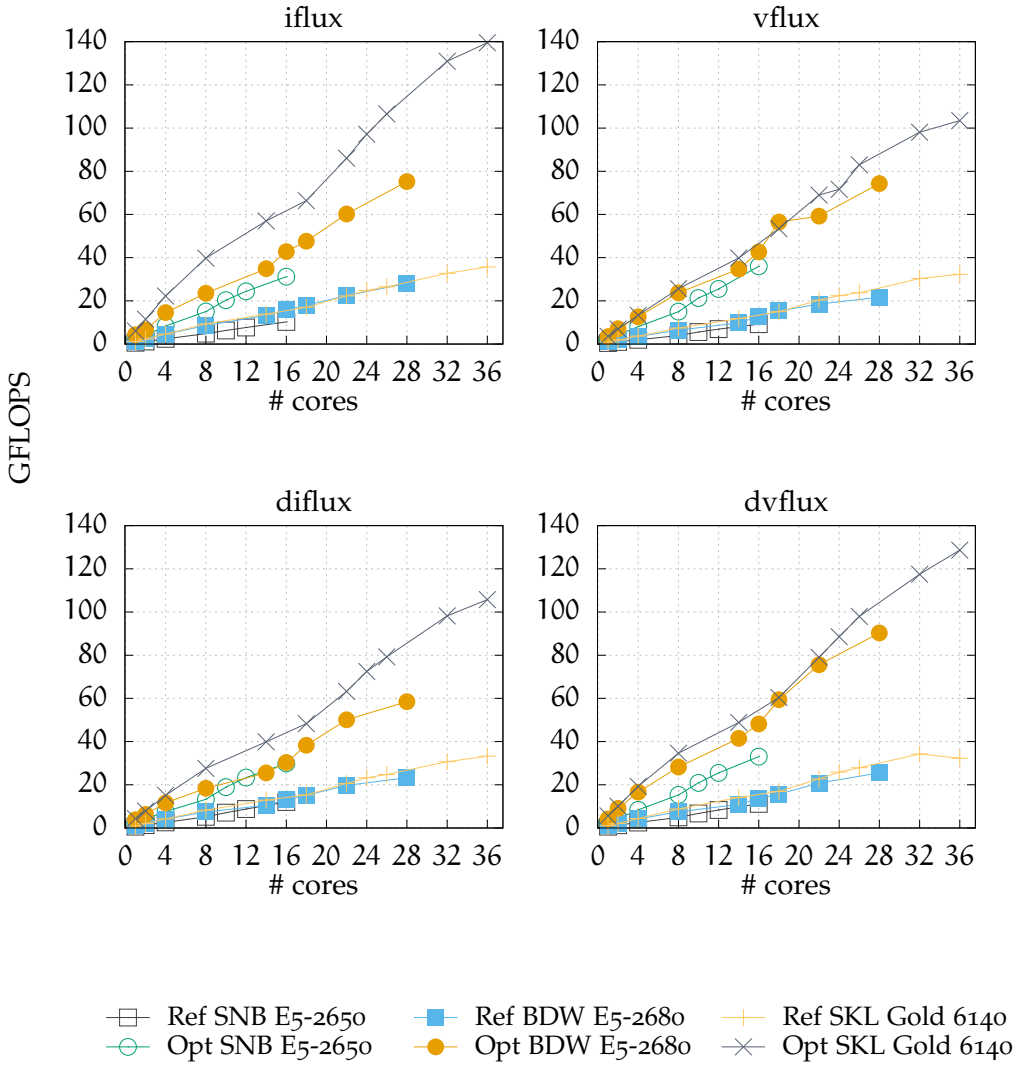


Figure 33: Performance strong scaling of face-based loops (flux computations) on the two-socket multicore nodes.

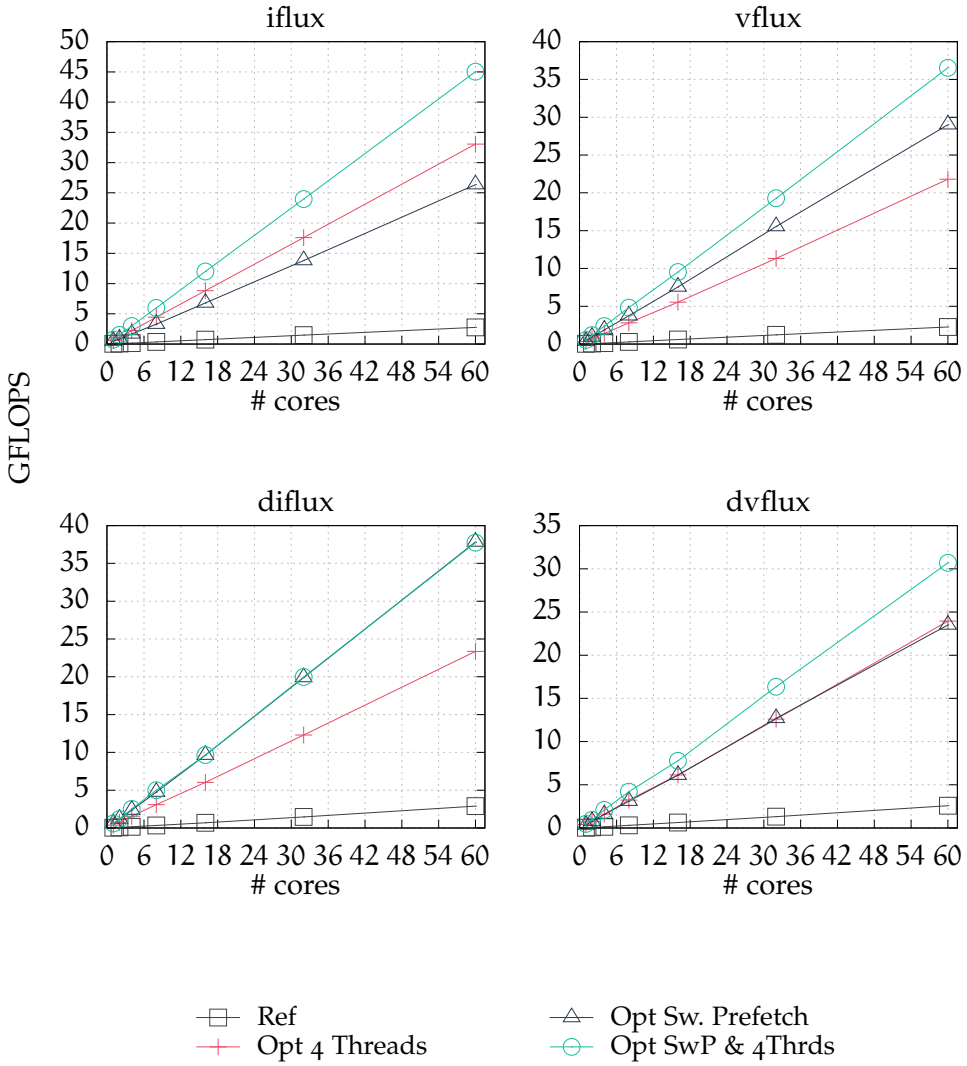


Figure 34: Performance strong scaling of face-based loops (flux computations) on the Intel Xeon Phi Knights Corner 7120P co-processor.

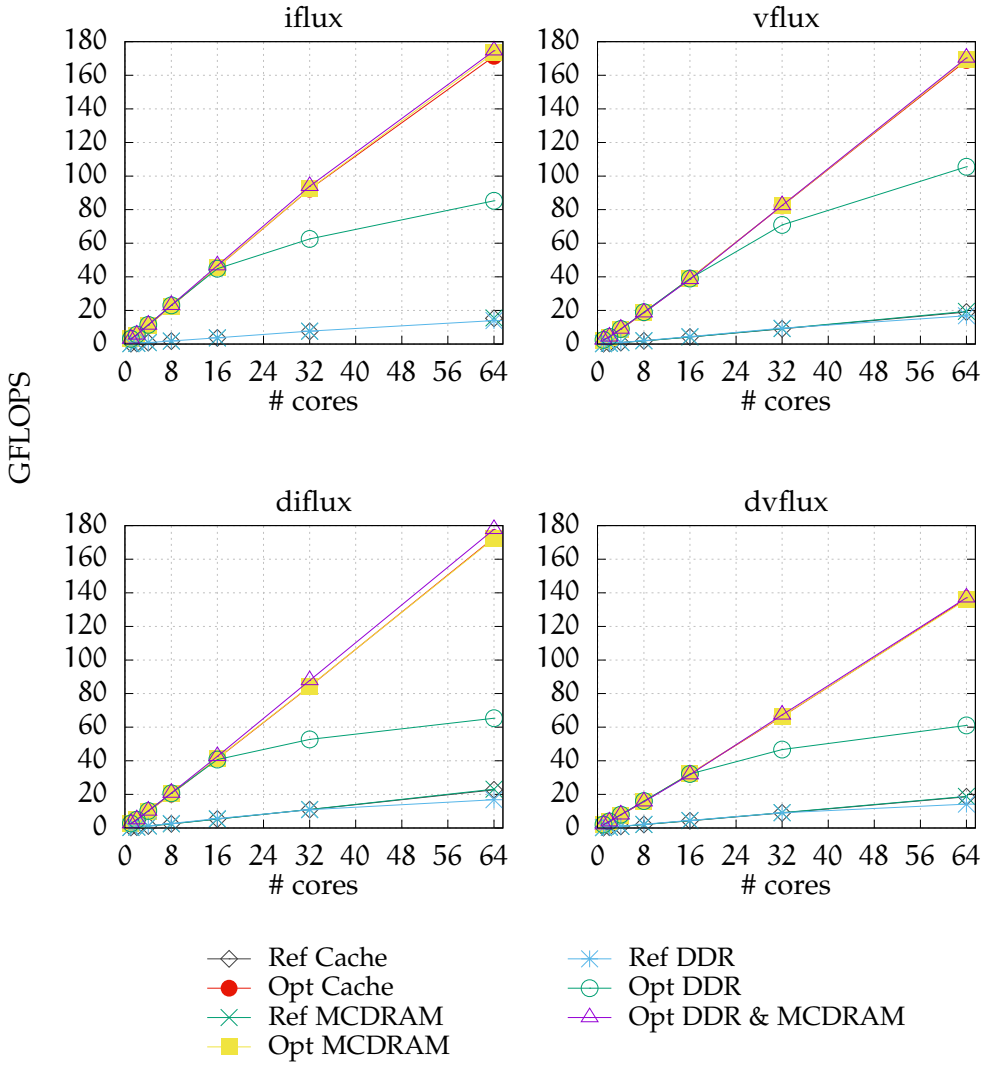


Figure 35: Performance strong scaling of face-based loops (flux computations) on the Intel Xeon Phi Knights Landing 7210 processor.

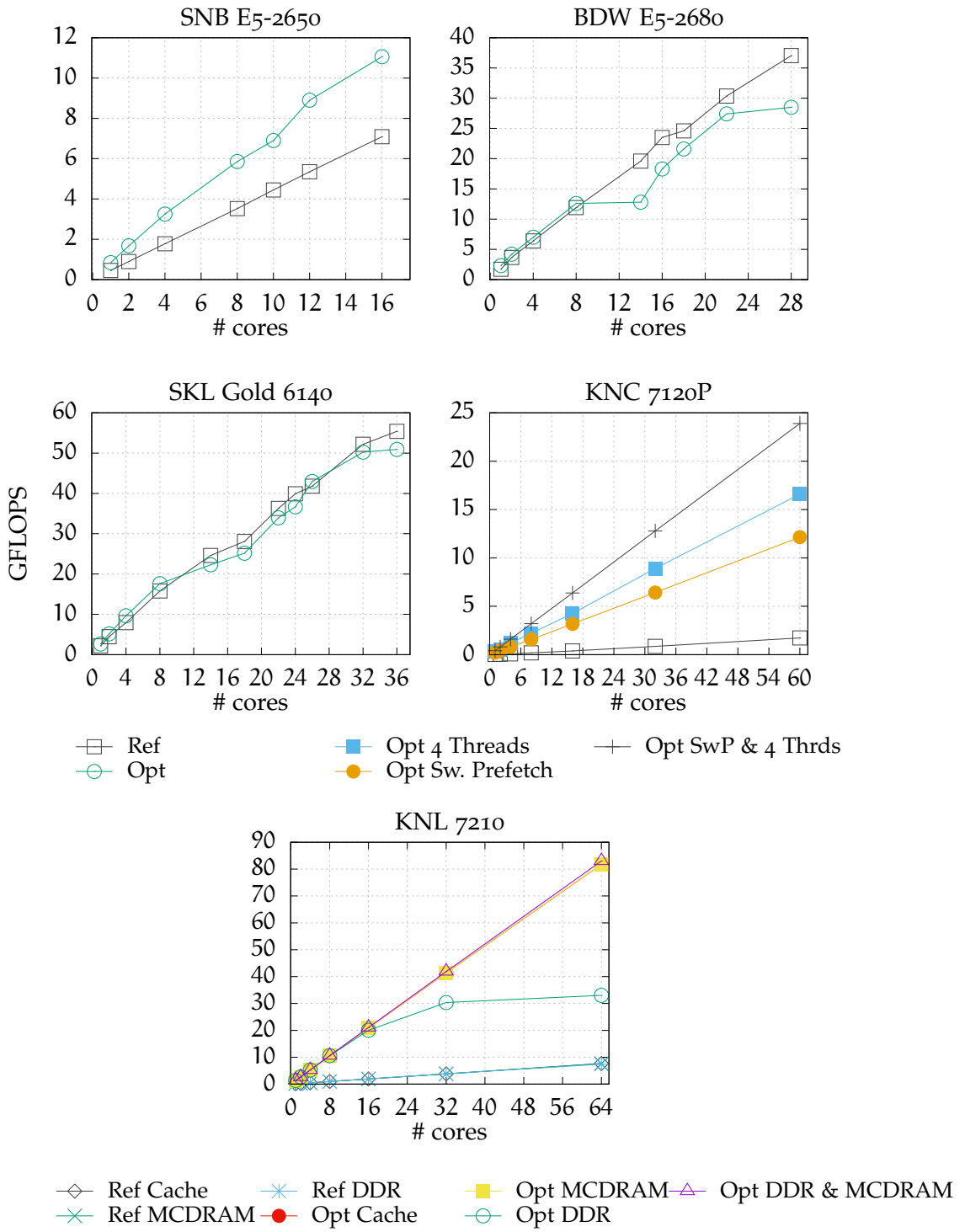


Figure 36: Peformance strong scaling of cell-based loop kernel (updates to primitive variables) on all processors.

between 2.6 to 2.8X faster on the multicore CPUs, 8.6X on Knights Corner and 5.6X on the best performing configuration of Knights Landing in Quadrant and Flat mode with MCDRAM and DDR4 accesses interleaved. As expected, running only with DDR4 is 2.5X slower than the other modes that involve the utilisation of MCDRAM. The approach of utilising both memories explicitly is the fastest version although by a very small percentage. On Knights Corner, running 4 concurrent threads per MPI rank combined with software prefetching or performing software prefetching only exhibit the same performance as the application scales across the whole coprocessor and saturates the available memory bandwidth.

5.5.3 *Performance modelling*

A visualisation of the Roofline model based on applying the optimisations presented in Section 5.4 can be seen in Figure 37 for the two-socket multicore CPU nodes and Figure 38 for the manycore Intel Xeon Phi nodes. For brevity, face-based loops are represented only by the kernel with the highest arithmetic intensity (i.e. `iflux`).

SINGLE CORE Analysis of the single core rooflines on the multicore systems demonstrates that the performance of cell-based loops is limited by the memory bandwidth while the performance of face-based loops is limited by the imbalance in floating point operations since the FMA units are not exploited on Broadwell and Skylake. The same considerations also apply to the manycore systems at this granularity.

FULL NODE At full node concurrency, the presence of irregular access patterns in face-based loops is the main impediment to performance for these classes of kernels. In contrast, the performance of cell-based loops is very close or slightly surpasses that of the memory bandwidth. This is possible due to a combination of prefetching and the fact that some of the data is available in the cache hierarchy and does not come from main DRAM as assumed by the performance model. This is a limitation that is worth considering and therefore allows for certain optimisation that involve the exploitation of the cache hierarchy or memory parallelism to pierce through the peak memory bandwidth roofline.

# cores	mesh 1			mesh 2		
	reference (s)	optimised (s)	speedup (x)	reference (s)	optimised (s)	speedup (x)
SNB E5-2650						
1	83.0	29.0	2.8	184.0	61.0	3.0
2	42.0	15.3	2.7	86.8	29.7	2.9
4	23.0	8.9	2.5	44.7	15.1	2.9
8	13.6	5.8	2.3	24.7	9.0	2.7
10	10.7	4.5	2.3	19.5	7.5	2.6
12	9.4s	3.7	2.5	18.3	7.2	2.6
16	7.4s	2.8	2.6	14.4	5.5	2.6
BDW E5-2680						
1	46.5	16.8	2.7	91.0	35.7	2.5
2	24.9	9.1	2.7	47.4	20.2	2.3
4	14.6	5.5	2.6	26.5	11.2	2.3
8	8.4	3.8	2.2	16.1	7.4	2.1
14	6.4	3.0	2.1	10.4	5.9	1.7
16	5.8	2.5	2.3	9.1	5.2	1.7
18	5.3	2.4	2.2	8.8	4.6	1.9
22	4.6	1.9	2.4	7.9	3.8	2.0
28	3.8	1.4	2.7	7.6	2.8	2.7
SKL Gold 6140						
1	47.8	15.5	3.0	91.3	32.1	2.8
2	25.5	8.0	3.1	49.9	16.4	3.0
4	13.4	4.5	2.9	28.1	8.8	3.1
8	7.4	2.9	2.5	15.5	5.5	2.8
14	5.1	2.2	2.3	10.7	4.2	2.5
18	4.7	2.1	2.2	9.0	3.7	2.4
22	3.9	1.6	2.4	7.8	3.1	2.4
24	3.8	1.5	2.5	7.2	2.8	2.5
26	3.5	1.4	2.5	6.6	2.6	2.5
32	2.9	1.2	2.4	6.0	2.2	2.7
36	2.8	1.0	2.8	5.3	1.9	2.7

Table 6: Comparison between time to solution update measured in seconds of baseline and best optimised implementation on the multicore CPUs.

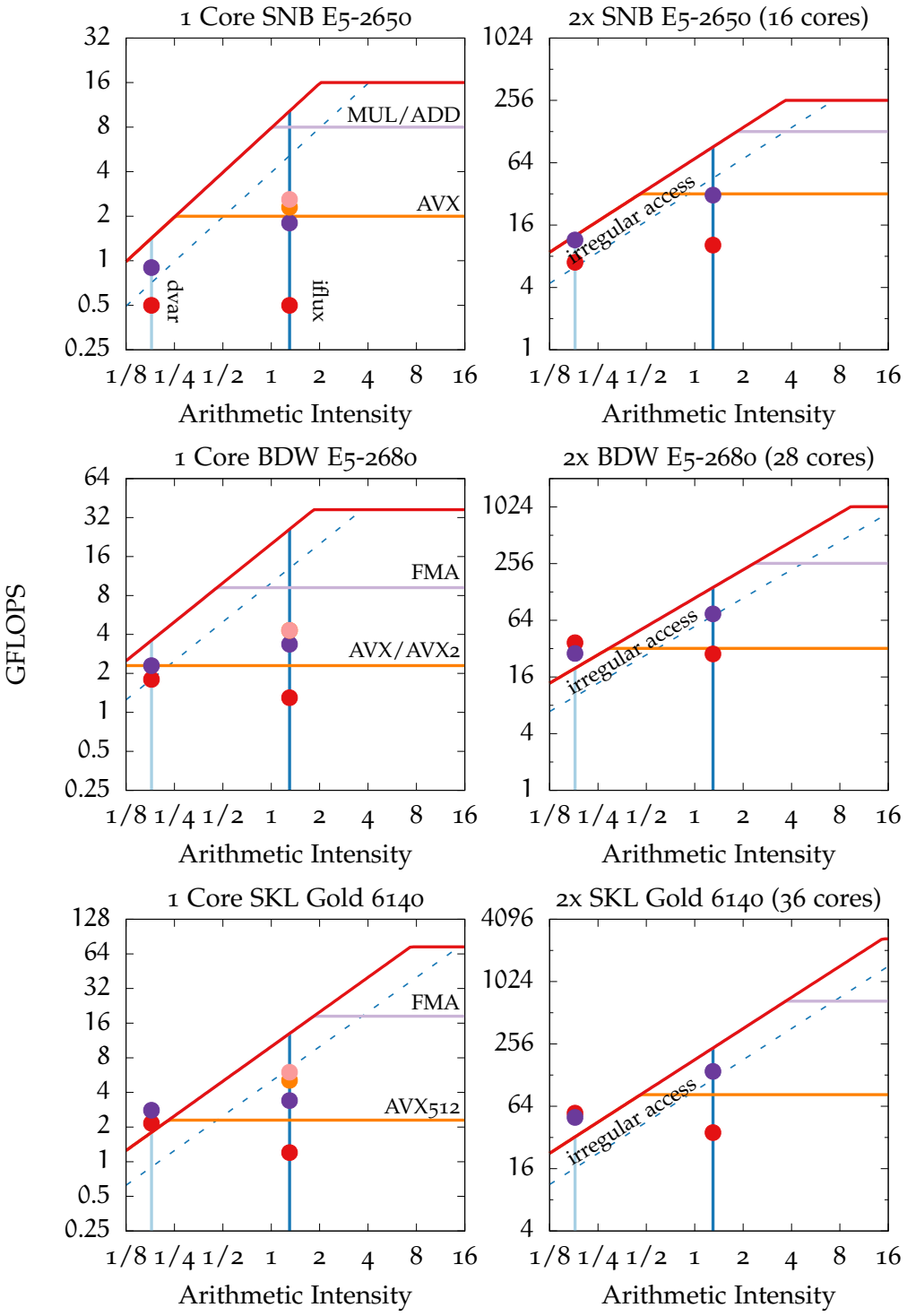


Figure 37: Multicore rooflines of single core and full node configurations for both classes of computational kernels. Key: ● Reference ● Vectorization ● Gather-/Scatter ● Sw. Prefetch

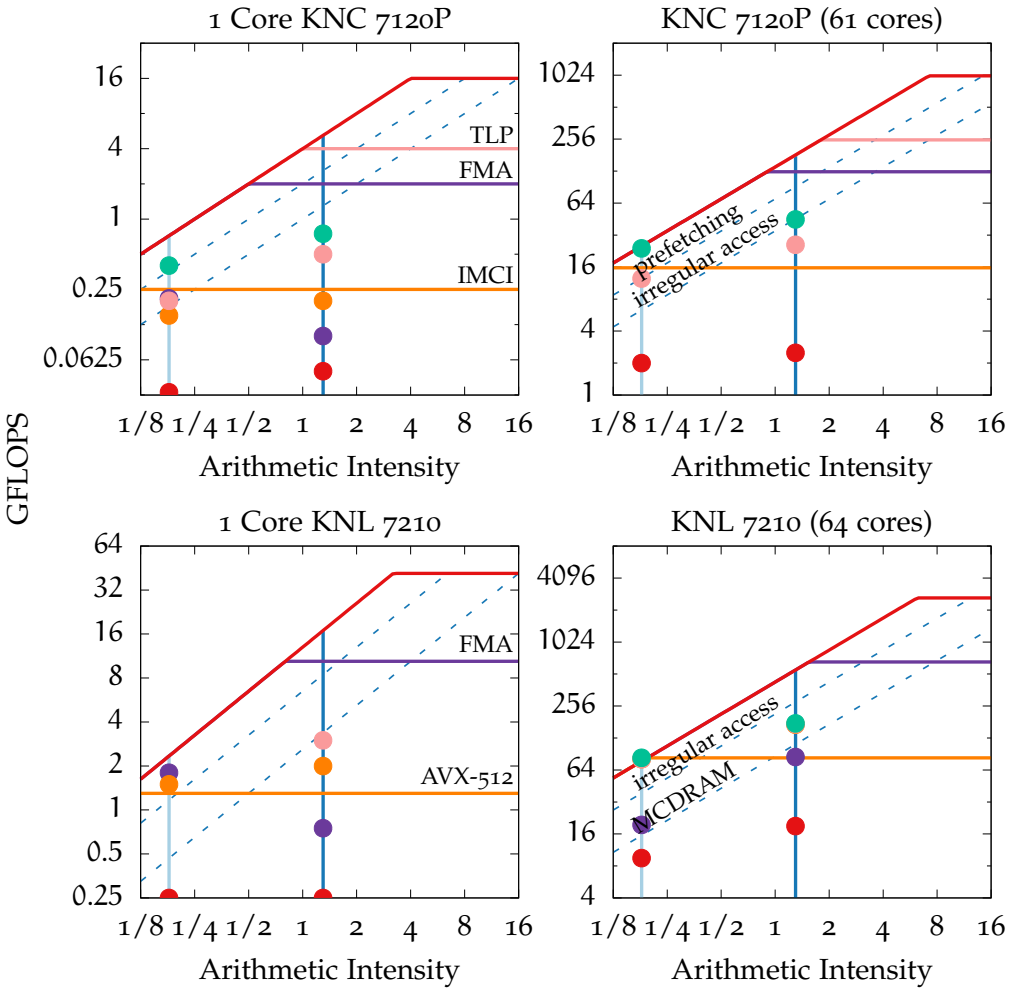


Figure 38: Manycore rooflines of single core and full node configurations for both classes of computational kernels.
Key for KNC 7120P system and 1 Core KNL 7210: ● Reference ● Vectorization ● Gather/Scatter ● Sw. Prefetch ● Multi-threading. Key for KNL 7210 (64 cores): ● Reference Cache ● Opt DDR ● Opt Cache ● Opt MCDRAM ● Opt MCDRAM & DDR

# cores	mesh 1				
	reference (s)	optimised x 4 threads (s)	optimised prefetch (s)	optimised prefetch x 4 threads (s)	speedup (x)
KNC 7120P					
1	1349.0	193.6	183.4	149.1	9.0
2	703.0	111.9	89.7	78.9	8.9
4	385.8	56.7	48.2	40.8	9.4
8	192.9	34.4	28.5	25.1	7.6
16	110.1	17.5	14.3	13.6	8.0
32	62.5	10.0	8.4	7.9	7.9
60	43.8	6.0	5.0	5.0	8.7

Table 7: Comparison between time to solution update measured in seconds of baseline and optimised versions with 4 hyperthreads and no software prefetching, software prefetching and no hyperthreading and both software prefetching and 4 hyperthreads on the Intel Xeon Phi Knights Corner coprocessor. The speed-up is calculated relative to baseline results and the timings of the best optimisation (`prefetch x 4 threads`). On this architecture, we could only perform experimental runs on `mesh 1` at full concurrency due to the 16GB memory limit.

On the Knights Landing architecture, the difference between using MCDRAM as cache or explicit memory buffer as well as exploiting both MCDRAM and DDR4 is very small. It is however believed that better performance of the interleaved version can be obtained by auto-tuning and testing a variety of data structure allocations in order to find the right balance.

5.5.4 *Architectural Comparison*

FACE-BASED LOOPS In face-based loops (Figure 39), the high arithmetic intensity means that architectures with wider SIMD units and larger number of cores will perform best since these kernels tend to scale well across vector lanes and across cores and sockets. Consequently, best performance at single core granularity for kernels computing fluxes at cell interfaces was obtained on the Skylake processor followed by Broadwell. The performance of a single Knights Landing core compared to Sandy Bridge was on par while the Knights Corner core performed the worst which is to be expected due to the in-order

# cores	mesh 1			mesh 2		
	reference (s)	optimised (s)	speedup (x)	reference (s)	optimised (s)	speedup (x)
KNL 7210 Cache						
1	205.0	33.7	6.0	411.0	71.1	5.7
2	116.0	17.8	6.5	238.4	37.8	6.3
4	57.1	9.4	6.0	120.4	19.3	6.2
8	29.0	5.3	5.4	63.3	10.0	6.3
16	16.2	2.8	5.7	32.1	5.3	6.0
32	9.1	1.6	5.6	18.23	3.1	5.8
64	5.9	1.0	5.9	10.5	2.0	5.2
KNL 7210 MCDRAM						
1	204.5	33.2	6.1	405.1	69.1	5.8
2	115.0	17.8	6.4	230.9	37.4	6.1
4	57.3	9.4	6.0	119.4	18.9	6.3
8	28.9	5.3	5.4	61.6	9.8	6.2
16	16.0	2.8	5.7	31.2	5.3	5.8
32	8.9	1.6	5.5	17.9	3.0	5.9
64	5.7	1.0	5.7	10.3	1.9	5.4
KNL 7210 DDR						
1	190.7	31.9	5.9	377.1	66.2	5.6
2	106.8	17.1	6.2	213.6	35.9	5.9
4	53.1	9.1	5.8	111.2	18.3	6.0
8	27.1	5.2	5.2	57.3	9.7	5.9
16	15.4	3.0	5.1	29.6	5.7	5.1
32	9.2	2.6	3.5	18.1	4.9	3.6
64	7.6	2.6	2.9	13.6	4.9	2.7
KNL 7210 DDR+MCDRAM						
1	-	30.2	-	-	62.5	-
2	-	17.5	-	-	36.4	-
4	-	9.1	-	-	18.7	-
8	-	5.3	-	-	9.6	-
16	-	2.8	-	-	5.1	-
32	-	1.6	-	-	3.0	-
64	-	0.9	-	-	1.8	-

Table 8: table

Comparison between time to solution update measured in seconds of baseline and best optimised implementation on the Intel Xeon Phi Knights Landing 7210 processor and across different memory modes.

execution unit. However, it is interesting to observe the improvements in single core performance between successive Intel Xeon Phi generations.

At full node concurrency, the MCDRAM in the Knights Landing architecture as well as the higher core numbers and wide SIMD units contribute to the best overall performance. Unsurprisingly, the second best performance was exhibited by the Skylake node followed by Broadwell, Sandy Bridge and Knights Corner. As previously mentioned, the difference in performance between Skylake and Broadwell is a factor of two for kernels with high arithmetic intensities (`iflux`) due to AVX-512, however, in kernels that are memory bound such as the viscous fluxes and linearised inviscid fluxes (`vflux` and `diflux`), the difference between the two is 50%. This correlates well with their respective memory bandwidth performance.

CELL-BASED LOOPS As cell-based loops are heavily bound by memory performance, the best performance at single core is obtained by the Skylake system due to the highest per core bandwidth. At full node concurrency, best performance is obtained on the Knights Landing processor due to the superior bandwidth of MCDRAM. As mentioned earlier, the difference in performance between architectures for memory-bound kernels is significantly lower even though some of the processors such as the Sandy Bridge system are up to five years older than the two-socket Skylake node. This further epitomises the issue of processor speeds increasing at a significant higher rate than the performance of the memory system.

FULL APPLICATION In terms of the whole solver represented as average time per Newton-Jacobi iteration, the overall winner is the Knights Landing processor at full node concurrency followed closely by the multicore Skylake node and the remaining multicore systems. The difference between the Knights Landing and Skylake systems and the Broadwell node is approximately 50%, 3X compared to Sandy Bridge and more than 5X compared to the Knights Corner coprocessor.

5.6 CONCLUSIONS

This chapter presented a number of optimisations for improving the performance of unstructured CFD applications on multicore and manycore processors

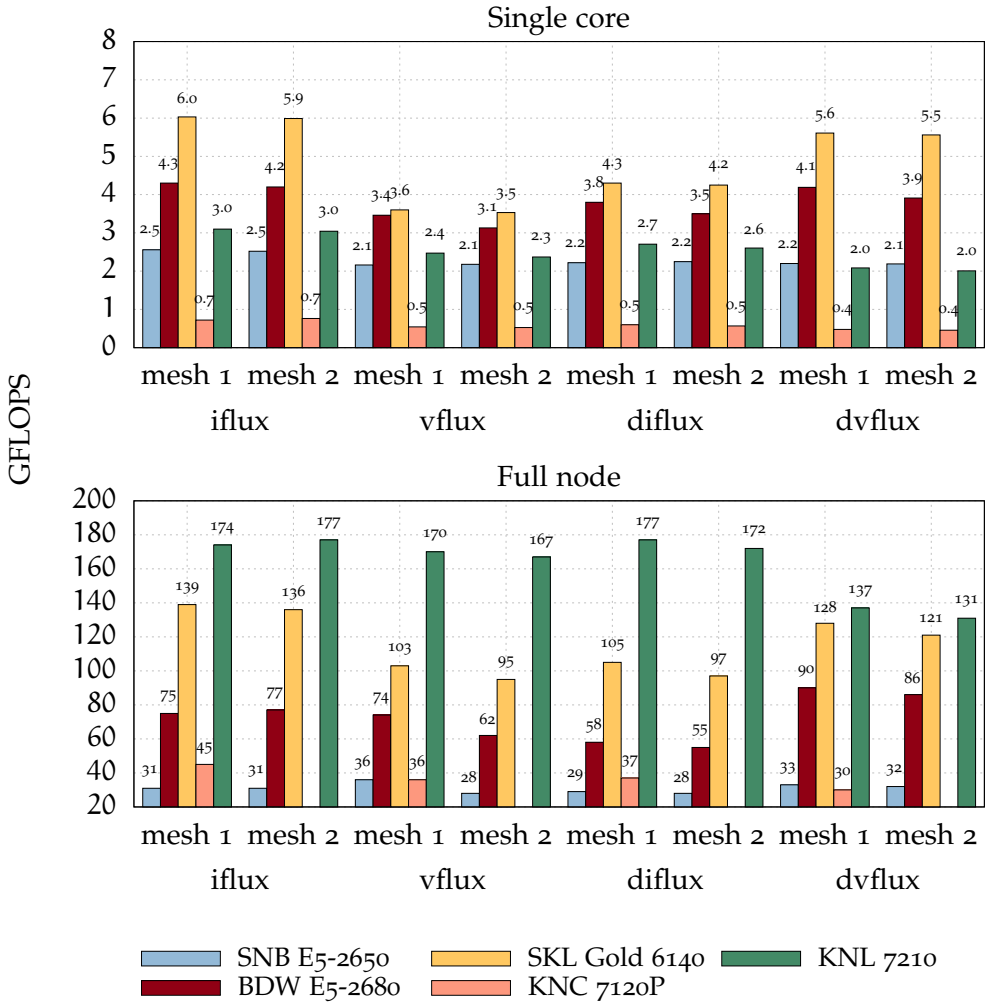


Figure 39: Architectural comparison of the performance of face-based loops (flux computations) represented as GFLOPS at single core and full node concurrency and across all architectures (higher is better). The results presented for Knights Landing were obtained by at full concurrency by interleaving the memory access between DDR and MCDRAM (i.e. DDR+MCDRAM)

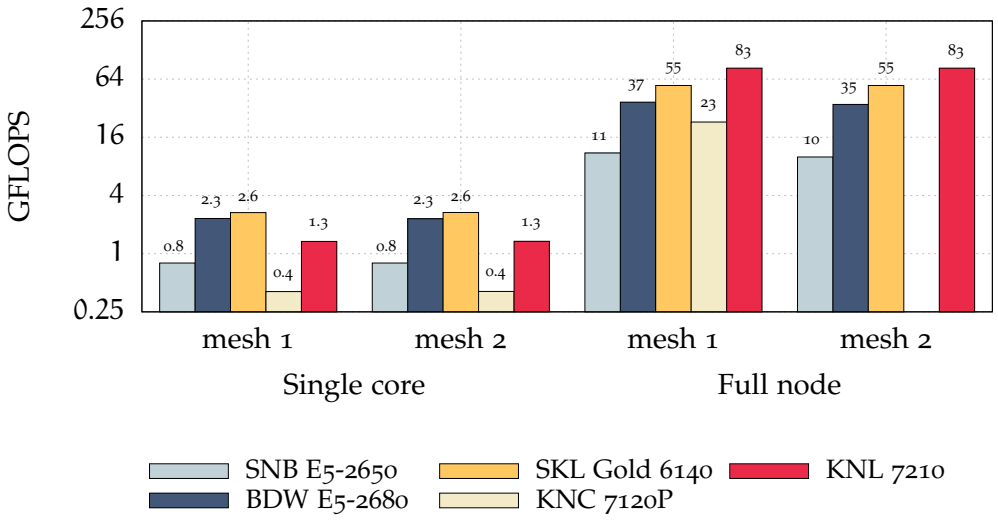


Figure 40: Architectural comparison of the performance of cell-based loops (update to primitive variables) represented as GFLOPS at single core and full node concurrency and across all architectures (higher is better). The results presented for Knights Landing were obtained by at full concurrency by interleaving the memory access between DDR and MCDRAM (i.e. DDR+MCDRAM).

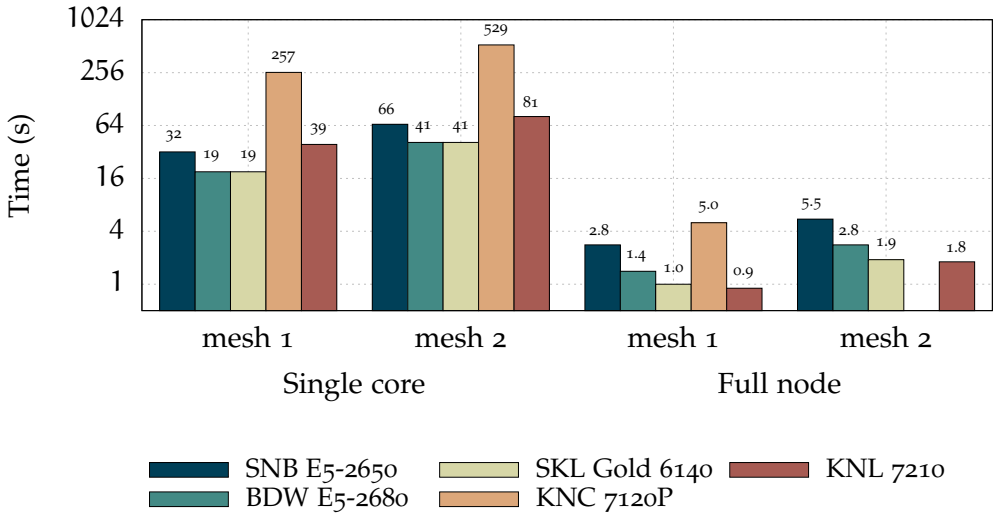


Figure 41: Architectural comparison of the performance of the whole application represented as time per Newton-Jacobi iteration at single core and full node concurrency and across all architectures (lower is better). The results presented for Knights Landing were obtained by at full concurrency by interleaving the memory access between DDR and MCDRAM (i.e. DDR+MCDRAM).

and demonstrated their implementation and impact in a code of representative size and complexity to an industrial application. Examples of optimisations included grid renumbering, vectorization, colouring, data layout transformations, in-register transpositions, software prefetching, loop tiling and multithreading.

Renumbering the mesh using the Reverse Cuthill-Mckee algorithm led to a reduction in the distance between memory references in face-based loops by up to 53X although this only resulted in actual speed-ups for such kernels of up to 10%.

Vectorization contributed to the largest speed-ups ranging between 2-5X and 2-3X on the multicore CPUs and the Intel Xeon Phi manycore processors for both face-based and cell-based loops and 2-3X for the whole application across all architectures. Although vectorizing cell-based loops was relatively straight forward using the features of the OpenMP 4.0 API, the vectorization of face-based loops required that these kernels are re-written and re-structured due to their more complex access patterns.

Colouring the faces enabled the vectorization of scatter operations in face-based loops. This resulted in relatively worse performance for these kernels on the multicore CPUs mainly due to the increase in cache misses as the distances between face-end points increased as well. Despite this, the usefulness of colouring was demonstrated further down the line when evaluated in conjunction with other optimisations.

The advantage of the AoS memory layout compared to SoA has been demonstrated in kernels that contain indirect and irregular access patterns such as the computation of fluxes due to their more efficient utilisation of the cache hierarchy. However, this was shown to have a detrimental effect on the performance of cell-based loops. To alleviate this, hand-tuned conversion primitives were presented based on compiler intrinsics which performed fast transpositions to and from the AoS and SoA layouts. These improved performance in face-based and cell-based kernels and were found to be particularly beneficial on the manycore processors and on architectures with wide SIMD lengths.

The utility of auto-tuning for finding parameters such as prefetch distances or tile size across all processors has been demonstrated and the large impact that this can have on performance. Moreover, it has been shown that software prefetching is highly recommended on architectures with large L2 caches and on the Knights Corner co-processor due to the in-order core design and lack

of hardware prefetchers. Another optimisation that made use of auto-tuning is loop tiling which improved performance of the linear solver by blocking data in the L2 cache.

On the Intel Xeon Phi Knights Landing processor, the performance of all available memory modes was studied together with the introduction of a third option in which both MCDRAM and DDR are utilised. The best approach was found to be the latter although by a small margin when compared to the other options that make use of the MCDRAM. The utilisation of only the DDR interface came last by a significant margin which demonstrates that the exploitation of the high-bandwidth memory package is imperative for best performance.

The utilisation of multiple threads per MPI rank led to very good performance on the Knights Corner architecture, especially when combined with software prefetching. In contrast, this was not beneficial on the Knights Landing processor where it led to worse performance since the KNL core is out-of-order and multiple threads were competing for the available core resources.

Finally, all of the presented techniques were implemented in a single code base with abstractions for architectural specific optimisations being based on traditional language constructs available in the C++ programming language and on compiler pre-processing machinery. As for their overall performance benefit, their implementation led to full application speed-ups ranging between 2.6-2.8X on the multicore CPUs and 5-8X on the manycore Xeon Phi processors at double precision and across two different mesh sizes of a realistic industrial test case.

6

CONCLUSIONS

This chapter reviews the achievements of this thesis, discusses its limitations and outlines the most important aspects worth taking into account when mapping existing or new CFD applications onto the architectural features of modern processors. This is presented in the form of advice to CFD practitioners and is applicable to future processors following similar architectural paradigms. Finally, future research directions are discussed that could improve the work presented herein.

6.1 SUMMARY OF FINDINGS

Chapter 2 presented the processor trends that have led to the advent of multicore and manycore processors. This was followed by a discussion of their architectural features and the importance of exploiting parallelism across multiple granularities in order to obtain high performance on them. The implications that these have on the performance of CFD codes have also been reviewed with the conclusion that current legacy applications used in industry and in research, although able to exploit coarse grained parallelism across processor nodes via message passing, are not amenable to parallelism at finer granularities and therefore exhibit poor performance on modern processors unless their implementation is revisited. This state of affairs has led to the main motivation behind this work based on evaluating and presenting the optimisations required for mapping CFD algorithms within the finite volume framework onto the architectural features of modern multicore and manycore processors.

To this end, chapter 4 presented a number of optimisations for structured mesh applications in a block-structured Euler code used for turbomachinery computations. The optimisations were evaluated across three distinct architectures such as: the Intel Xeon Sandy Bridge and Haswell multicore CPUs and the Intel Xeon Phi Knights Corner coprocessor and in two computational kernels with distinct access patterns: stencil operations arising from the computation of fluxes at cell interfaces and the evaluation of cell attributes in ker-

nels that perform updates to state vectors. The practicalities of enabling efficient vectorization across both types of computational kernels was discussed along with the trade-off between performance and portability by evaluating a number of approaches such as compiler directives, compiler intrinsics and Agner Fog's vector library class. For stencil operators, it was demonstrated that efficient vectorization relies upon the use of aligned vector load and store operations which can only be made possible through techniques such as inter-register shuffle operations or transpositions. In kernels with regular and streaming access patterns such as cell-based kernels, efficient vectorization can be achieved by relying solely on compiler directives found in OpenMP 4.0 and the addition of padding and aligned memory allocations. The importance of the underlying data layout was also discussed and it was demonstrated that the hybrid AoSSoA format leads to superior performance compared to SoA in structured mesh applications as it allows for more efficient vector load and store operations and utilises fewer memory resources. The utility of cache-blocking the separate sweeps when computing fluxes at the interfaces was also demonstrated along with a number of domain decomposition approaches for extracting thread-level parallelism. The best version of the latter was based on decomposing each block into two dimensional tiles and mimicking the same behaviour as the message passing layer where halo swaps were performed explicitly via shared memory. Finally, the impact of each optimisation across the three architectures was appraised using the Roofline performance model. The difference in performance between the optimised version of the kernel based on stencil computations at full node concurrency was 3X on the multicore architectures and 24X on the manycore Knights Corner coprocessor when running on 180 threads. For the cell-based state vector update kernel that features a regular memory access pattern and is bound by memory bandwidth, the difference in performance between the optimised and reference implementation was approximately 2X on the multicore CPUs and 13X on the manycore Knights Corner platform.

Chapter 5 presented optimisations and techniques useful for improving the performance of unstructured CFD applications on modern multicore and manycore processors. The optimisations were implemented in a code of representative size and complexity to an industrial application and demonstrated in two distinct classes of computational patterns that form the backbone of unstructured CFD codes: gather and scatter operations in face-based loops for the

evaluation of fluxes and regular and streaming access patterns that are present in cell-based loops for updating state vectors. It was demonstrated that vectorization of face-based loops is possible using compiler directives based on the OpenMP 4.0 API as long as such kernels are re-written in order to isolate the scatter operations which prevent vectorization thus allowing for both the gather and the computation stage to be vectorized. Full vectorization of all face-based loops was achieved via the implementation of reordering algorithms which not only allowed for SIMD scatter operations by removing the dependencies at the face-end points but also minimized the jump in the second indices. Prior to that, a renumbering of the grid was performed via the Reverse Cuthill Mckee algorithm which reduced the distance between references in face-based loops and therefore allowed for a better exploitation of the memory system. The advantage of the AoS data layout over SoA was also demonstrated in kernels with irregular access patterns due to their more efficient utilisation of the cache hierarchy. Conversion between AoS and SoA data layouts for vector operations was performed using hand tuned compiler intrinsics along with ways by which architectural specific implementations of such primitives can be abstracted away in the application code. Another important contribution in this chapter was the implementation of an auto-tuning strategy for finding optimal parameters such as prefetch distances or tile sizes across all of the evaluated platforms. This resulted in palpable speed-ups on the back of software prefetching as well as loop tiling especially on newer architectures that integrate large L2 caches and on the in-order Knights Corner coprocessor. Other optimisations included the evaluation of the AoSSoA layout for face attributes in order to reduce the number of memory streams and improve memory performance and the implementation of multithreading as a separate layer of parallelism by extending the functionality of the reodering algorithms.

On the Knights Landing architecture, the performance of the available memory modes has been assessed together with a hybrid approach that exploited both DDR4 and the MCDRAM interface. Finally, all optimisations were also correlated with the Roofline model and led to full application speed-up ranging between 2.6X and 2.8X on the multicore CPUs and 5-8X on the manycore Xeon Phi processors at double precision and across two different computational domain sizes.

6.2 ADVICE FOR CFD PRACTITIONERS

The findings in this thesis can be used to construct a list of recommendations and procedures aimed at CFD practitioners and developers who wish to map their new or existing application onto the architectural features of current and future multicore and manycore processors. These are described in more detail below.

VECTORIZATION As seen from the results for both structured and unstructured applications, vectorization is the most important optimisation on modern processors. This is likely to be the case for future architectures as well which are expected to increase the underlying vector register width further and continue to improve the SIMD ISA.

As a result, developers should make sure that their application is flexible in such manner that data parallelism can be exploited across different vector lengths. This is important for ensuring best performance across a variety of multicore and manycore processors. Another important consideration for making good use of the vector units is how to efficiently load data into them. As seen in Chapters 4 and 5, this differs depending on the underlying access pattern i.e. stencil operators or gather and scatter operations. Consequently, although best performance is achieved by using architectural specific optimisations that fully harness the capability of each platform i.e. AVX-512 or IMCI, abstractions for these implementations are necessary in order to ensure portability. Furthermore, applications such as unstructured mesh solvers also require colouring and reordering algorithms so that efficient vector accumulate and store operations can be utilised.

DATA LAYOUT Another important optimisation with respect to CFD applications is the format used for storing cell-centred and face variables. For structured mesh codes, the SoA or the hybrid AoSSoA are the best choice, as demonstrated in Chapter 4. This is due to the fact that these translate to efficient SIMD load and store operations due to the regular and streaming access patterns of structured solvers. However, for unstructured mesh solvers, the AoS format is in fact more efficient for cell-centred data structures as it leads to a reduction in cache misses in kernels with gather and scatter operations, as seen in

Chapter 5. With regard to face attributes, SoA is still the preferred choice in unstructured grid codes since faces are evaluated in consecutive order.

Changing the underlying data layout of an application is an arduous and error prone task, especially in a code of representative size. Consequently, a key advice to CFD developers is to provide an abstraction layer in the application for the storage format of such data structures. This would allow for any layout to be changed at compile time depending on both the underlying architecture and the computational patterns.

AUTO-TUNING Although the majority of processor architectures in high-performance computing adhere to an abstract processor model as seen in Chapter 2, they all exhibit certain particularities which are important to consider for best performance. Examples of these are: size and latency of caches, length of vector registers, core design i.e. in-order or out-of-order, single threaded or n-way multithreaded, memory or cluster modes etc. Consequently, for optimisations that rely on these attributes in their implementation, performance portability can only be achieved by means of auto-tuning. This has been demonstrated in this thesis in Chapter 5 where an auto-tuning phase was used to select the best distance parameter for software prefetching across a wide range of architectures as well as the appropriate tile size for the implementation of loop tiling.

As a result, applications should be modified or developed to allow for "magic" values such as tile size or prefetch distance to be automatically chosen at an initial auto-tuning stage every time the application is executed on a new architecture.

MULTITHREADING Running more than one thread per physical core is not always useful for multicore processors since one thread can fully utilise the entire execution pipeline and available resources. However, for in-order architectures such as the Intel Xeon Phi Knights Corner or GPGPUs, running more than one thread per core can help hide memory latencies and avoid pipeline stalls as seen in both Chapters 4 and 5.

Consequently, the advice for CFD developers is to consider enabling another level of parallelism in their application for multithreading that can be enabled or disabled depending on the underlying architecture. This was demonstrated in Chapter 5 with the help of colouring algorithms where faces with depend-

ency free end-points were processed by each thread in groups equal in size to the width of the underlying vector register and in a round robin fashion.

6.3 LIMITATIONS

In chapter 4, one existing limitation in the presented approach is the lack of loop tiling. In structured codes, temporal blocking or skewed tiling is significantly easier to achieve than in their unstructured counterpart and could have had a sizeable impact on performance. This could have been implemented by hoisting the entire solver under the same iteration space which would proceed in steps equal or a multiple of the underlying SIMD size thereby exploiting data parallelism as well as minimizing traffic to main memory. Another limitation in chapter 4 is the implementation of Henretty's [37] dimension lifted transposition for aligned vector load and stores in stencil operations. In this work, the transposition operations were only performed in one dimension as the fluxes were computed in two distinct passes in the i and j direction. This required that data is re-transposed again after the i and before the j sweep. However, in Henretty's original work, the transposition of the data can be performed more efficiently and only once by fusing both sweeps.

The work in chapter 5 could have been improved with respect to the hybrid MPI/OpenMP implementation. In such an implementation, the domain owned by each MPI rank is further decomposed across existing threads thereby mimicking the same execution model as the message passing layer while exploiting the superior latency of shared memory. Such approaches were first presented by Aubry et al [10] and it is believed that as the number of cores per node continue to increase, a hybrid approach of one rank per socket with subsequent threads pinned to all other physical cores and communicating via shared memory would lead to even better performance and scalability.

6.4 FUTURE WORK

The most natural extension of this work would be to validate the claims that the optimisations presented herein for both structured and unstructured solvers are also applicable to more specialised manycore processors such as NVIDIA GPUs. Thus, a good avenue to test this claim would be through a compiler

directive approach such as OpenACC or OpenMP 4.0/4.5 in the first instance. This would make a fair comparison between the performance that can be extracted across all architectures whilst maintaining the same code structure. Following from this, hand-tuned kernels can be implemented in OpenCL or CUDA with results being correlated via performance models.

BIBLIOGRAPHY

[1] *ACM Gordon Bell Prize*. http://awards.acm.org/bell. Accessed: 31-08-2017.

[2] *ARCHER. Knights Landing Testing & Development Platform.* http://www.archer.ac.uk/documentation/knl-guide/. Accessed: 01-08-2017.

[3] ARK. *Intel Xeon Platinum 8180 Processor*. https://ark.intel.com/products/120496/Intel-Xeon-Platinum-8180-Processor-38_5M-Cache-2_50-GHz. Accessed: 08-01-2018.

[4] ARM. *ARM Developer*. https://developer.arm.com/. Accessed: 09-01-2018.

[5] Sam Ainsworth and Timothy M. Jones. "Software Prefetching for Indirect Memory Accesses". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO '17. Austin, USA: IEEE Press, 2017, pp. 305–317. ISBN: 978-1-5090-4931-8.

[6] G.D. Albada, B. Leer and Jr. Roberts W.W. "A Comparative Study of Computational Methods in Cosmic Gas Dynamics". English. In: *Upwind and High-Resolution Schemes*. Ed. by M.Yousuff Hussaini, Bram Leer and John Rosendale. Springer Berlin Heidelberg, 1997, pp. 95–103. ISBN: 978-3-642-64452-8.

[7] 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.

[8] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes and B. F. Smith. "Achieving High Sustained Performance in an Unstructured Mesh CFD Application". In: *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. SC '99. Portland, Oregon, USA: ACM, 1999. ISBN: 1-58113-091-0. DOI: 10.1145/331532.331600.

[9] K. Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. Electrical Engineering and Computer Sciences, University of California at Berkeley.

[10] R. Aubry, G. Houzeaux, M. Vázquez and J. M. Cela. "Some useful strategies for unstructured edge-based solvers on shared memory machines". In: *International Journal for Numerical Methods in Engineering* 85.5 (2011), pp. 537–561. ISSN: 1097-0207. DOI: 10.1002/nme.2973.

[11] H. David Bailey, F. Robert Lucas and W. Samuel Williams. *Performance Tuning of Scientific Applications*. New York, United States: CRC Press, 2011. ISBN: 9781439815694.

[12] Shekhar Borkar. "Thousand Core Chips: A Technology Perspective". In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. San Diego, California: ACM, 2007, pp. 746–749. ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278667.

[13] Shekhar Borkar and Andrew A Chien. "The future of microprocessors". In: *Communications of the ACM* 54.5 (2011), pp. 67–77.

[14] T. Brandvik and G. Pullan. "SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms". In: *2010 10th IEEE International Conference on Computer and Information Technology*. 2010, pp. 1181–1188. DOI: 10.1109/CIT.2010.214.

[15] D. A. Burgess and M. B. Giles. "Renumbering Unstructured Grids to Improve the Performance of Codes on Hierarchical Memory Machines". In: *Adv. Eng. Softw.* 28.3 (Apr. 1997), pp. 189–201. ISSN: 0965-9978. DOI: 10.1016/S0965-9978(96)00039-7.

[16] *C++ Vector Library Class*. http://www.agner.org/optimize/#vectorclass. Accessed: 10-05-2015.

[17] Mauro Carnevale, Jeffrey S Green and Luca Di Mare. "Numerical studies into intake flow for fan forcing assessment". In: *Proceedings of ASME Turbo Expo 2014: Turbine Technical Conference and Exposition*. 2014, pp. 16–20.

[18] Mauro Carnevale, Feng Wang and Luca di Mare. "Low Frequency Distortion in Civil Aero-engine Intake". In: *Journal of Engineering for Gas Turbines and Power* 139.4 (2017), p. 041203.

[19] Dan Curran, Christian B Allen, Simon McIntosh-Smith and David Beckingsale. "Towards Portability For A Compressible Finite-Volume CFD Code". In: *54th AIAA Aerospace Sciences Meeting*. 2016, p. 1813.

[20] 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.

[21] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf and Katherine Yelick. "Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008, 4:1–4:12. ISBN: 978-1-4244-2835-9.

[22] Zachary DeVito et al. "Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 9:1–9:12. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063396.

[23] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous and A. R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. ISSN: 0018-9200. DOI: 10.1109/JSSC.1974.1050511.

[24] Luca Di Mare, Davendu Y Kulkarni, Feng Wang, Artyom Romanov, Pandia R Ramar and Zacharias I Zachariadis. "Virtual gas turbines: Geometry and conceptual description". In: *Proceedings of ASME TurboExpo, Vancouver, Canada* (2011).

[25] Austen C. Duffy, Dana P. Hammond and Eric J. Nielsen. *Production Level CFD Code Acceleration for Hybrid Many-Core Architectures*. Tech. rep. NASA/TM-2012-217770. National Aeronautics and Space Administration.

[26] Thomas D. Economon, Dheevatsa Mudigere, Gaurav Bansal, Alexander Heinecke, Francisco Palacios, Jongsoo Park, Mikhail Smelyanskiy, Juan J. Alonso and Pradeep Dubey. "Performance optimizations for scalable implicit RANS calculations with SU2". In: *Computers & Fluids* 129 (2016),

pp. 146 –158. ISSN: 0045-7930. DOI: http://dx.doi.org/10.1016/j.compfluid.2016.02.003.

[27] L. Eeckhout. "Is Moore's Law Slowing Down? What's Next?" In: *IEEE Micro* 37.4 (2017), pp. 4–5. ISSN: 0272-1732. DOI: 10.1109/MM.2017.3211123.

[28] *FUN3D*. https://fun3d.larc.nasa.gov/. Accessed: 31-08-2017.

[29] Mohammed A. Al Farhan, Dinesh K. Kaushik and David E. Keyes. "Unstructured computational aerodynamics on many integrated core architecture". In: *Parallel Computing* 59 (2016). Theory and Practice of Irregular Applications, pp. 97 –118. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2016.06.001.

[30] Pawel Gepner, Victor Gamayunov and David L. Fraser. "Early performance evaluation of AVX for HPC". In: *Procedia Computer Science* 4.0 (2011). Proceedings of the International Conference on Computational Science, ICCS 2011, pp. 452 –460. ISSN: 1877-0509.

[31] M. B. Giles and I. Reguly. "Trends in high-performance computing for engineering calculations". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 372.2022 (2014). ISSN: 1364-503X. DOI: 10.1098/rsta.2013.0319.

[32] 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.

[33] M. B. Giles, G. R. Mudalige, C. Bertolli, P. H. J. Kelly, E. László and I. Reguly. "An Analytical Study of Loop Tiling for a Large-Scale Unstructured Mesh Application". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 2012, pp. 477–482. DOI: 10.1109/SC.Companion.2012.68.

[34] F. Grasso and C. Meola. *Handbook of Computational Fluid Mechanics*. London: Academic Press, 1996, pp. 159–282.

[35] William D Gropp, Dinesh K Kaushik, David E Keyes and Barry F Smith. "High-performance parallel implicit CFD". In: *Parallel Computing* 27.4 (2001). Parallel computing in aerospace, pp. 337 –362. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/S0167-8191(00)00075-2.

[36] Amiram Harten, Peter D. Lax and Bram van Leer. "On Upstream Differencing and Godunov-Type Schemes for Hyperbolic Conservation Laws". In: *SIAM Review* 25.1 (1983), pp. 35–61.

[37] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam and P. Sadayappan. "Data Layout Transformation for Stencil Computations on Short-vector SIMD Architectures". In: *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*. CC'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 225–245. ISBN: 978-3-642-19860-1.

[38] M. D. Hill and M. R. Marty. "Amdahl's Law in the Multicore Era". In: *Computer* 41.7 (2008), pp. 33–38. ISSN: 0018-9162. DOI: 10.1109/MC.2008.209.

[39] Charles Hirsch. *Numerical Computation of Internal and External Flows*. Chichester, West Sussex, UK: John Wiley and Sons, 1990.

[40] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-037. 2017.

[41] *Intel Xeon Scalable Processors*. https://newsroom.intel.com/press-kits/next-generation-xeon-processor-family/. Accessed: 18-07-2017.

[42] A. Jameson. "Time dependent calculations using multigrid, with applications to unsteady flows past airfoils and wings". In: *AIAA paper* 1596 (1991).

[43] Jim Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Boston, United States: Morgan Kaufmann, 2013. ISBN: 978-0-12-410414-3.

[44] Jim Jeffers, James Reinders and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming*. Morgan Kaufmann, 2016. ISBN: 978-0-12-809194-4.

[45] Jaekyu Lee, Hyesoon Kim and Richard Vuduc. "When Prefetching Works, When It Doesn't, and Why". In: *ACM Trans. Archit. Code Optim.* 9.1 (Mar. 2012), 2:1–2:29. ISSN: 1544-3566. DOI: 10.1145/2133382.2133384.

[46] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten and Krste Asanović. "Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators". In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), pp. 129–140. ISSN: 0163-5964. DOI: 10.1145/2024723.2000080.

[47] Bram van Leer. "Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method". In: *Journal of Computational Physics* 32.1 (1979), pp. 101 –136. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(79)90145-1.

[48] Anders Logg, Kent-Andre Mardal and Garth Wells. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642230989.

[49] Rainald Löhner. "Cache-efficient renumbering for vectorization". In: *International Journal for Numerical Methods in Biomedical Engineering* 26.5 (2010), pp. 628–636. ISSN: 2040-7947. DOI: 10.1002/cnm.1160.

[50] MIT. *MISES*. http://web.mit.edu/drela/Public/web/mises/. Accessed: 13-05-2015. 2009.

[51] D.J. Mavriplis. *Revisiting the Least-squares Procedure for Gradient Reconstruction on Unstructured Meshes*. Tech. rep. NASA/CR-2003-212683. National Aeronautics and Space Administration.

[52] Dimitri J Mavriplis. "Unstructured-mesh discretizations and solvers for computational aerodynamics". In: *AIAA journal* 46.6 (2008), pp. 1281–1298.

[53] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. A continually updated technical report. http://www.cs.virginia.edu/stream/. Charlottesville, Virginia: University of Virginia, 1991-2007.

[54] 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.

[55] Simon McIntosh-Smith, Michael Boulton, Dan Curran and James Price. "On the performance portability of structured grid codes on many-core computer architectures". In: *International Supercomputing Conference*. Springer. 2014, pp. 53–75.

[56] Gordon E. Moore. "Readings in Computer Architecture". In: ed. by Mark D. Hill, Norman P. Jouppi and Gurindar S. Sohi. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. Chap. Cramming More Components Onto Integrated Circuits, pp. 56–59. ISBN: 1-55860-539-8.

[57] S. K. Moore. "Multicore is bad news for supercomputers". In: *IEEE Spectrum* 45.11 (2008), pp. 15–15. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2008.4659375.

[58] Todd C. Mowry, Monica S. Lam and Anoop Gupta. "Design and Evaluation of a Compiler Algorithm for Prefetching". In: *SIGPLAN Not.* 27.9 (Sept. 1992), pp. 62–73. ISSN: 0362-1340. DOI: 10.1145/143371.143488.

[59] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli and P. H. J. Kelly. "OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures". In: *2012 Innovative Parallel Computing (InPar)*. 2012, pp. 1–12. DOI: 10.1109/InPar.2012.6339594.

[60] D. Mudigere, S. Sridharan, A. Deshpande, J. Park, A. Heinecke, M. Smelyanskiy, B. Kaul, P. Dubey, D. Kaushik and D. Keyes. "Exploring Shared-Memory Optimizations for an Unstructured Mesh CFD Application on Modern Parallel Systems". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 723–732. DOI: 10.1109/IPDPS.2015.114.

[61] J. P. Murphy and D. G. MacManus. "Ground vortex aerodynamics under crosswind conditions". In: *Experiments in Fluids* 50.1 (2011), pp. 109–124. ISSN: 1432-1114. DOI: 10.1007/s00348-010-0902-4.

[62] G. Ofenbeck, R. Steinmann, V. Caparros, D.G. Spampinato and M. Puschel. "Applying the roofline model". In: *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. 2014, pp. 76–85. DOI: 10.1109/ISPASS.2014.6844463.

[63] OpenCL. *OpenCL Overview*. https://www.khronos.org/opencl/. Accessed: 10-01-2018.

[64] *OpenMP 4.0 Specifications*. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf. Accessed: 15-05-2016.

[65] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[66] David Patterson. "The Trouble With Multi-core". In: *IEEE Spectr.* 47.7 (July 2010), pp. 28–32. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2010.5491011.

[67] Simon J. Pennycook, Chris J. Hughes, M. Smelyanskiy and S. A. Jarvis. "Exploring SIMD for Molecular Dynamics, Using Intel®Xeon®Processors and Intel®Xeon Phi Coprocessors". In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1085–1097. ISBN: 978-0-7695-4971-2. DOI: 10.1109/IPDPS.2013.44.

[68] T. Piazza, H. Jiang, P. Hammarlund and R. Singhal. *Technology Insight: Intel(R) Next Generation Microarchitecture Code Name Haswell*. Tech. rep. Intel Corporation, 2012.

[69] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall and Paul H. J. Kelly. "Firedrake: Automating the Finite Element Method by Composing Abstractions". In: *ACM Trans. Math. Softw.* 43.3 (Dec. 2016), 24:1–24:27. ISSN: 0098-3500. DOI: 10.1145/2998441.

[70] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran and S. McIntosh-Smith. "The OPS Domain Specific Abstraction for Multi-block Structured Grid Computations". In: *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. 2014, pp. 58–67. DOI: 10.1109/WOLFHPC.2014.7.

[71] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. J. Kelly and D. Radford. "Acceleration of a Full-Scale Industrial CFD Application with OP2". In: *IEEE Transactions on Parallel and Distributed Systems* 27.5 (2016), pp. 1265–1278. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2453972.

[72] 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* 28.2 (2016), pp. 557–577. ISSN: 1532-0634. DOI: 10.1002/cpe.3621.

[73] Philip L Roe. "Approximate Riemann solvers, parameter vectors, and difference schemes". In: *Journal of computational physics* 43.2 (1981), pp. 357–372.

[74] C. Rosales. "Porting to the Intel Xeon Phi: Opportunities and Challenges". In: *Extreme Scaling Workshop (XSW), 2013*. 2013, pp. 1–7. DOI: 10.1109/XSW.2013.5.

[75] P. E. Ross. "Why CPU Frequency Stalled". In: *IEEE Spectr.* 45.4 (Apr. 2008), pp. 72–72. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2008.4476447.

[76] Scott Rostrup and Hans De Sterck. "Parallel hyperbolic PDE simulation on clusters: Cell versus GPU". In: *Computer Physics Communications* 181.12 (2010), pp. 2164 –2179. ISSN: 0010-4655. DOI: http://dx.doi.org/10.1016/j.cpc.2010.07.049.

[77] Karl Rupp. *CPU, GPU and MIC Hardware Characteristics over Time*. https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/. Accessed: 28-12-2017.

[78] Richard M. Russell. "The CRAY-1 Computer System". In: *Commun. ACM* 21.1 (Jan. 1978), pp. 63–72. ISSN: 0001-0782. DOI: 10.1145/359327.359336.

[79] *SU2, the open-source CFD code*. http://su2.stanford.edu. Accessed: 31-08-2017.

[80] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar and Pradeep Dubey. "Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?" In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Portland, Oregon: IEEE Computer Society, 2012, pp. 440–451. ISBN: 978-1-4503-1642-2.

[81] J. M. Shalf and R. Leland. "Computing beyond Moore's Law". In: *Computer* 48.12 (2015), pp. 14–23. ISSN: 0018-9162. DOI: 10.1109/MC.2015.374.

[82] Anand Lal Shimpi. *Intel's Haswell Architecture Analyzed: Building a new PC and a new Intel*. https://www.anandtech.com/show/6355/intels-haswell-architecture. Accessed: 12-07-2015.

[83] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal and Y. C. Liu. "Knights Landing: Second-Generation Intel Xeon Phi Product". In: *IEEE Micro* 36.2 (2016), pp. 34–46. ISSN: 0272-1732. DOI: 10.1109/MM.2016.25.

[84] Milan Stanić. "Design of Energy-Efficient Vector Units for In-order Cores". PhD thesis. University Politècnica de Caralunya, 2016.

[85] Herb Sutter and James Larus. "Software and the Concurrency Revolution". In: *Queue* 3.7 (Sept. 2005), pp. 54–62. ISSN: 1542-7730. DOI: 10.1145/1095408.1095421.

[86] *VCLKNC*. https://bitbucket.org/veclibknc/vclknc. Accessed: 10-08-2015.

[87] Michael Vavra. *Aero-Thermodynamics and Flow in Turbomachines*. Los Alamitos, CA, USA: John Wiley, 1960, pp. 139–145. ISBN: 0-8186-0819-6.

[88] Feng Wang, Mauro Carnevale, Gan Lu, Luca di Mare and Davendu Kulkarni. "Virtual Gas Turbine: Pre-Processing and Numerical Simulations". In: *ASME Turbo Expo 2016: Turbomachinery Technical Conference and Exposition*. American Society of Mechanical Engineers. 2016, V001T01A009–V001T01A009.

[89] D.C. Wilcox. "Reassessment of the scale-determining equation for advanced turbulence models". In: *AIAA Journal* 26 (Nov. 1988), pp. 1299–1310. DOI: 10.2514/3.10041.

[90] Samuel W. Williams. "Auto-tuning performance on multicore computers". PhD thesis. University of California at Berkeley, 2008.

[91] 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.

[92] Samuel Williams, Leonid Oliker, Jonathan Carter and John Shalf. "Extracting Ultra-scale Lattice Boltzmann Performance via Hierarchical and Distributed Auto-tuning". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 55:1–55:12. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063458.

[93] Wm. A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588.

APPENDIX

A

APPENDIX

A.1 RESULTS OF AUTO-TUNING PREFETCH DISTANCES

				Speedup				
L1i	L2i	L1d	L2d	iflux	vflux	diflux	dvflux	newton jacobi
4	-	2	-	1.10	0.99	0.99	0.92	0.99
4	8	2	4	1.04	1.05	0.95	0.97	0.99
4	16	2	8	1.09	1.07	0.93	0.96	0.97
4	32	2	16	1.11	1.08	0.92	0.96	0.97
8	-	4	-	1.10	1.05	0.96	0.94	0.99
8	16	4	8	1.01	1.04	0.93	0.93	0.95
8	32	4	16	1.02	1.04	0.92	0.93	0.95
8	64	4	32	1.02	1.06	0.92	0.93	0.96
16	-	8	-	1.11	1.06	0.96	0.94	0.99
16	32	8	16	1.01	1.04	0.92	0.92	0.94
16	64	8	32	1.01	1.05	0.93	0.99	0.96
16	128	8	64	1.00	1.03	0.91	0.98	0.95
32	-	16	-	1.11	1.08	0.95	0.92	0.98
32	64	16	32	1.00	1.04	0.91	0.98	0.96
32	128	16	64	0.99	1.02	0.91	0.97	0.95
32	256	16	128	0.98	1.00	0.89	0.95	0.94
64	-	32	-	1.11	1.08	0.94	0.94	0.99
64	128	32	64	0.98	1.01	0.90	0.88	0.92
64	256	32	128	0.97	0.99	0.89	0.86	0.91
64	512	32	256	0.94	0.94	0.85	0.84	0.89

Table 9: Speed-up of flux computation kernels and whole solver (newton-jacobi) depending on prefetch distance on Sandy Bridge E5-2650 CPU.

				Speedup				
L1i	L2i	L1d	L2d	iflux	vflux	diflux	dvflux	newton jacobi
4	-	2	-	0.95	0.93	0.92	0.92	0.95
4	8	2	4	0.97	0.91	0.91	0.83	0.92
4	16	2	8	0.96	0.90	0.95	0.82	0.93
4	32	2	16	0.95	0.89	0.92	0.81	0.91
8	-	4	-	0.96	1.00	0.97	0.86	0.95
8	16	4	8	0.94	0.87	0.91	0.80	0.91
8	32	4	16	0.95	0.89	0.92	0.87	0.93
8	64	4	32	0.94	0.88	0.90	0.85	0.91
16	-	8	-	0.94	0.99	0.94	0.84	0.93
16	32	8	16	0.93	0.87	0.90	0.86	0.93
16	64	8	32	0.93	0.88	0.90	0.84	0.91
16	128	8	64	0.93	0.87	0.90	0.83	0.91
32	-	16	-	0.94	0.99	0.93	0.83	0.93
32	64	16	32	0.93	0.87	0.90	0.84	0.91
32	128	16	64	0.92	0.86	0.88	0.83	0.90
32	256	16	128	0.89	0.83	0.86	0.79	0.88
64	-	32	-	0.93	0.99	0.93	0.89	0.94
64	128	32	64	0.92	0.83	0.88	0.75	0.88
64	256	32	128	0.89	0.82	0.86	0.72	0.86
64	512	32	256	0.85	0.76	0.79	0.68	0.82

Table 10: Speed-up of flux computation kernels and whole solver (newton-jacobi) depending on prefetch distance on Broadwell

				Speedup				
L1i	L2i	L1d	L2d	iflux	vflux	diflux	dvflux	newton jacobi
8	-	4	-	1.06	1.04	1.03	1.06	1.03
8	16	4	8	1.13	1.17	1.16	1.28	1.13
8	32	4	16	1.11	1.16	1.17	1.28	1.13
8	64	4	32	1.10	1.14	1.15	1.23	1.12
16	-	8	-	1.12	1.15	1.14	1.30	1.13
16	32	8	16	1.12	1.16	1.16	1.21	1.12
16	64	8	32	1.09	1.13	1.15	1.20	1.11
16	128	8	64	1.08	1.10	1.13	1.18	1.09
32	-	16	-	1.11	1.15	1.14	1.29	1.12
32	64	16	32	1.09	1.13	1.15	1.20	1.11
32	128	16	64	1.07	1.09	1.13	1.17	1.09
32	256	16	128	1.05	1.07	1.12	1.14	1.08
64	-	32	-	1.11	1.14	1.14	1.25	1.11
64	128	32	64	1.08	1.10	1.13	1.15	1.09
64	256	32	128	1.05	1.06	1.11	1.12	1.07
64	512	32	256	1.04	1.04	1.10	1.09	1.06
-	16	-	8	1.15	1.20	1.18	1.36	1.15
-	32	-	16	1.13	1.18	1.18	1.35	1.15
-	64	-	32	1.12	1.17	1.16	1.31	1.14
-	128	-	64	1.10	1.15	1.15	1.29	1.13
-	256	-	128	1.09	1.13	1.15	1.26	1.12

Table 11: Speed-up of flux computation kernels and whole solver (newton-jacobi) depending on prefetch distance on Skylake

				Speedup				
L1i	L2i	L1d	L2d	iflux	vflux	diflux	dvflux	newton jacobi
8	-	4	-	1.65	2.76	2.40	1.99	1.66
8	16	4	8	1.80	4.49	3.93	1.97	1.97
8	32	4	16	1.81	4.52	4.10	2.27	1.92
8	64	4	32	1.79	4.50	4.10	2.25	1.99
16	-	8	-	1.72	3.65	3.60	1.97	1.85
16	32	8	16	1.74	4.09	3.96	2.04	1.90
16	64	8	32	1.71	3.96	3.87	2.00	1.93
16	128	8	64	1.69	3.98	3.85	1.98	1.87
32	-	16	-	1.68	3.40	3.48	1.88	1.68
32	64	16	32	1.67	3.65	3.63	1.90	1.60
32	128	16	64	1.66	3.61	3.62	1.88	1.70
32	256	16	128	1.63	3.53	3.51	1.83	1.76
64	-	32	-	1.65	3.18	3.26	1.76	1.74
64	128	32	64	1.64	3.37	3.38	1.79	1.73
64	256	32	128	1.60	3.25	3.25	1.68	1.65
64	512	32	256	1.56	3.12	3.14	1.65	1.64

Table 12: Speed-up of flux computation kernels and whole solver (newton-jacobi) depending on prefetch distance on KNC with no hyperthreading

				Speedup				
L1i	L2i	L1d	L2d	iflux	vflux	diflux	dvflux	newton jacobi
8	-	4	-	1.09	1.02	1.01	0.81	0.96
8	16	4	8	1.39	1.65	1.48	1.07	1.19
8	32	4	16	1.36	1.60	1.47	1.00	1.18
8	64	4	32	1.33	1.57	1.49	0.96	1.16
16	-	8	-	1.26	1.34	1.46	1.06	1.17
16	32	8	16	1.32	1.52	1.53	0.96	1.16
16	64	8	32	1.30	1.47	1.44	0.92	1.13
16	128	8	64	1.29	1.41	1.45	0.88	1.12
32	-	16	-	1.27	1.37	1.49	1.05	1.11
32	64	16	32	1.29	1.45	1.46	0.91	1.13
32	128	16	64	1.24	1.37	1.43	0.86	1.10
32	256	16	128	1.21	1.30	1.38	0.82	1.07
64	-	32	-	1.26	1.36	1.44	1.00	1.15
64	128	32	64	1.23	1.33	1.42	0.84	1.09
64	256	32	128	1.17	1.25	1.35	0.78	1.05
64	512	32	256	1.11	1.16	1.26	0.74	1.00
-	16	-	8	1.54	1.96	1.80	1.22	1.31
-	32	-	16	1.53	1.93	1.80	1.20	1.33
-	64	-	32	1.47	1.83	1.70	1.11	1.27
-	128	-	64	1.42	1.69	1.63	1.03	1.23
-	256	-	128	1.31	1.54	1.48	0.95	1.17

Table 13: Speed-up of flux computation kernels and whole solver (newton-jacobi) depending on prefetch distance on KNL quadrant cache mode

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede with subsequent modifications by Ioan Hadade for adhering to Imperial College regulations. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*.

Final Version as of 30th April 2018 (`classicthesis`).