THE UNIVERSITY OF WARWICK

University of Warwick institutional repository: http://go.warwick.ac.uk/wrap

A Thesis Submitted for the Degree of PhD at the University of Warwick

http://go.warwick.ac.uk/wrap/57050

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.

THE UNIVERSITY OF WARWICK

Library Declaration and Deposit Agreement

1. **STUDENT DETAILS**

Please complete the following:

Full name: ……………………………………………………………………………………………….

University ID number: ………………………………………………………………………………

2. **THESIS DEPOSIT**

2.1 I understand that under my registration at the University, I am required to deposit my thesis with the University in BOTH hard copy and in digital format. The digital version should normally be saved as a single pdf file.

2.2 The hard copy will be housed in the University Library. The digital version will be deposited in the University's Institutional Repository (WRAP). Unless otherwise indicated (see 2.3 below) this will be made openly accessible on the Internet and will be supplied to the British Library to be made available online via its Electronic Theses Online Service (EThOS) service.
[At present, theses submitted for a Master's degree by Research (MA, MSc, LLM, MS or MMedSci) are not being deposited in WRAP and not being made available via EthOS. This may change in future.]

2.3 In exceptional circumstances, the Chair of the Board of Graduate Studies may grant permission for an embargo to be placed on public access to the hard copy thesis for a limited period. It is also possible to apply separately for an embargo on the digital version. (Further information is available in the *Guide to Examinations for Higher Degrees by Research*.)

2.4 *If you are depositing a thesis for a Master's degree by Research, please complete section (a) below. For all other research degrees, please complete both sections (a) and (b) below:*

(a) Hard Copy

I hereby deposit a hard copy of my thesis in the University Library to be made publicly available to readers (please delete as appropriate) EITHER immediately OR after an embargo period of ……….................... months/years as agreed by the Chair of the Board of Graduate Studies.

I agree that my thesis may be photocopied. YES / NO (*Please delete as appropriate*)

(b) Digital Copy

I hereby deposit a digital copy of my thesis to be held in WRAP and made available via EThOS.

Please choose one of the following options:

EITHER My thesis can be made publicly available online. YES / NO (*Please delete as appropriate*)

OR My thesis can be made publicly available only after…..[date] (Please give date)
YES / NO (*Please delete as appropriate*)

OR My full thesis cannot be made publicly available online but I am submitting a separately identified additional, abridged version that can be made available online.
YES / NO (*Please delete as appropriate*)

OR My thesis cannot be made publicly available online. YES / NO (*Please delete as appropriate*)

3. GRANTING OF NON-EXCLUSIVE RIGHTS

Whether I deposit my Work personally or through an assistant or other agent, I agree to the following:

Rights granted to the University of Warwick and the British Library and the user of the thesis through this agreement are non-exclusive. I retain all rights in the thesis in its present version or future versions. I agree that the institutional repository administrators and the British Library or their agents may, without changing content, digitise and migrate the thesis to any medium or format for the purpose of future preservation and accessibility.

4. DECLARATIONS

(a) I DECLARE THAT:

- I am the author and owner of the copyright in the thesis and/or I have the authority of the authors and owners of the copyright in the thesis to make this agreement. Reproduction of any part of this thesis for teaching or in academic or other forms of publication is subject to the normal limitations on the use of copyrighted materials and to the proper and full acknowledgement of its source.

- The digital version of the thesis I am supplying is the same version as the final, hard-bound copy submitted in completion of my degree, once any minor corrections have been completed.

- I have exercised reasonable care to ensure that the thesis is original, and does not to the best of my knowledge break any UK law or other Intellectual Property Right, or contain any confidential material.

- I understand that, through the medium of the Internet, files will be available to automated agents, and may be searched and copied by, for example, text mining and plagiarism detection software.

(b) IF I HAVE AGREED (in Section 2 above) TO MAKE MY THESIS PUBLICLY AVAILABLE DIGITALLY, I ALSO DECLARE THAT:

- I grant the University of Warwick and the British Library a licence to make available on the Internet the thesis in digitised format through the Institutional Repository and through the British Library via the EThOS service.

- If my thesis does include any substantial subsidiary material owned by third-party copyright holders, I have sought and obtained permission to include it in any version of my thesis available in digital format and that this permission encompasses the rights that I have granted to the University of Warwick and to the British Library.

5. LEGAL INFRINGEMENTS

I understand that neither the University of Warwick nor the British Library have any obligation to take legal action on behalf of myself, or other rights holders, in the event of infringement of intellectual property rights, breach of contract or of any other right, in the thesis.

Please sign this agreement and return it to the Graduate School Office when you submit your thesis.

Student's signature: ....…… Date: ....

Evaluating the Performance of Legacy Applications on Emerging Parallel Architectures

by

Simon John Pennycook

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

December 2012

Abstract

The gap between a supercomputer's theoretical maximum ("peak") floating-point performance and that actually achieved by applications has grown wider over time. Today, a typical scientific application achieves only 5–20% of any given machine's peak processing capability, and this gap leaves room for significant improvements in execution times.

This problem is most pronounced for modern "accelerator" architectures – collections of hundreds of simple, low-clocked cores capable of executing the same instruction on dozens of pieces of data simultaneously. This is a significant change from the low number of high-clocked cores found in traditional CPUs, and effective utilisation of accelerators typically requires extensive code and algorithmic changes. In many cases, the best way in which to map a parallel workload to these new architectures is unclear.

The principle focus of the work presented in this thesis is the evaluation of emerging parallel architectures (specifically, modern CPUs, GPUs and Intel MIC) for two benchmark codes – the LU benchmark from the NAS Parallel Benchmark Suite and Sandia's miniMD benchmark – which exhibit complex parallel behaviours that are representative of many scientific applications. Using combinations of low-level intrinsic functions, OpenMP, CUDA and MPI, we demonstrate performance improvements of up to 7x for these workloads.

We also detail a code development methodology that permits application developers to target multiple architecture types without maintaining completely separate implementations for each platform. Using OpenCL, we develop performance portable implementations of the LU and miniMD benchmarks that are faster than the original codes, and at most 2x slower than versions highly-tuned for particular hardware.

Finally, we demonstrate the importance of evaluating architectures at scale (as opposed to on single nodes) through performance modelling techniques, highlighting the problems associated with strong-scaling on emerging accelerator architectures.

Acknowledgements

I am indebted to many people for the advice, support and friendship that they have provided during my studies at the University of Warwick. It gives me great pleasure to acknowledge them here.

First and foremost I need to thank my supervisor, Prof. Stephen Jarvis, for guiding my research and professional development over the past three years. Thank you for your confidence in me, your enthusiasm, and for giving me the opportunity to undertake a PhD.

Thanks go to my colleagues in the Performance Computing and Visualisation Group – David Beckingsale, Bob Bird, Dr. Adam Chester, James Davis, Henry Franks, Dr. Matthew Leeke, Andrew Mallinson, Oliver Perks and others – for making lunch interesting and a welcome break from work. I reserve special thanks for my office mates, Dr. Simon Hammond, Dr. Gihan Mudalige and Steven Wright; thank you for your encouragement, for proof-reading hundreds of paper drafts, and for your undeniable roles in developing all of the ideas in this thesis.

I also wish to thank the members of the Intel Parallel Computing Lab in Santa Clara, in particular Pradeep Dubey, Chris Hughes, Jason Sewall and Misha Smelyanskiy, for their support and guidance during (and beyond) my internship. My experiences with Intel have given me a far greater understanding of micro-architecture and "ninja" programming, and have shaped the latter half of this thesis – thank you.

Collecting the experimental data for this thesis relied on access to a wide range of supercomputing hardware, and I thank the staff of the Daresbury Laboratory; the Centre for Scientific Computing at the University of Warwick; and the Lawrence Livermore National Laboratory for their maintenance work and for granting me access to their machines.

Outside of work, I would like to thank all of my friends at Warwick and at home in Cornwall for reminding me that it's okay to have fun occasionally; Charlie, Mum, Dad, Grandma, Aunty and the rest of my family for their continued love and support; and finally, my girlfriend Louise – I dedicate this thesis to you.

Declarations

This thesis is submitted to the University of Warwick in support of the author's application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree.

The work presented (including data generated and data analysis) was carried out by the author except in the cases outlined below:

- Analysis of LU's execution time (Chapter 4) was aided by Gihan Mudalige (Oxford e-Research) and Simon Hammond (Sandia National Laboratories).
- Execution times for miniMD on KNC (Chapter 5) were collected by Chris Hughes (Intel Corporation), who also developed the message-passing level of the code.
- Analysis of miniMD's execution time on Intel hardware (Chapters 5 and 6) was aided by Chris Hughes and Misha Smelyanskiy (Intel Corporation).
- Execution times for the GTX 680 (Chapter 6) were collected by Chris Hughes.
- Execution times for the A8-3850 and HD6550D (Chapter 6) were collected by Simon Hammond.
- Execution times for the *DawnDev*, *Hera* and *Sierra* supercomputers (Chapter 7) were collected by Simon Hammond.

Where possible, source code developed by the author is made available online:

- NAS-LU Optimisations:
  `http://www2.warwick.ac.uk/fac/sci/dcs/research/pcav/areas/hpc/download/nas-lu-ports/`
- miniMD Optimisations:
  `http://www2.warwick.ac.uk/fac/sci/dcs/research/pcav/areas/hpc/download/minimd_opencl/`

Parts of this thesis have been previously published by the author:

- S. J. Pennycook, G. R. Mudalige, S. D. Hammond and S. A. Jarvis, Parallelising Wavefront Applications on General-Purpose GPU Devices, In *Proceedings of the UK Performance Engineering Workshop* (UKPEW), University of Warwick, UK, July, 2010 [130]
- S. J. Pennycook, S. D. Hammond, G. R. Mudalige and S. A. Jarvis, Experiences with Modelling Wavefront Algorithms on Many-Core Architectures, In *Proceedings of the Daresbury GPU Workshop*, Daresbury, UK, September, 2010 [123]
- S. J. Pennycook, S. D. Hammond, G. R. Mudalige and S. A. Jarvis, Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark, In *Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems* (PMBS), New Orleans, LA, November, 2010 [124]
- S. J. Pennycook, S. D. Hammond, S. A. Jarvis and G. R. Mudalige, Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark, ACM SIGMETRICS Performance Evaluation Review, 38 (4), ISSN 0163-5999 [125]
- S. J. Pennycook, S. D. Hammond, G. R. Mudalige and S. A. Jarvis, On the Acceleration of Wavefront Applications using Distributed Many-Core Architectures, The Computer Journal, 55 (2), pp. 138–153, ISSN 0010-4620 [126]
- S. J. Pennycook, S. D. Hammond, S. A. Wright, J. A. Herdman, I. Miller and S. A. Jarvis, An Investigation of the Performance Portability of OpenCL, Journal of Parallel and Distributed Computing (JPDC), 2012 (to appear) [127]
- S. J. Pennycook, C. J. Hughes, M. Smelyanskiy and S. A. Jarvis, Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors, In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium* (IPDPS), Boston, MA, May, 2013 [128]
- S. J. Pennycook, S. A. Jarvis, Developing Performance-Portable Molecular Dynamics Kernels in OpenCL, In *Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems* (PMBS), Salt Lake City, UT, November, 2012 [129]

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- The University of Warwick, United Kingdom:
  Warwick Postgraduate Research Scholarship (2009–2012)
- Intel Corporation, Santa Clara:
  Summer Internship (2011)
- Royal Society:
  Industry Fellowship Scheme (IF090020/AM)
- UK Atomic Weapons Establishment:
  "The Production of Predictive Models for Future Computing Requirements" (CDK0660)
  "AWE Technical Outreach Programme" (CDK0724)
  "TSB Knowledge Transfer Partnership" (KTP006740)

Abbreviations

AoS	Array of Structs
AVX	Advanced Vector eXtensions
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FLOP/s	Floating-Point Operations per Second
GFLOP/s	10^9 FLOP/s
GPU	Graphics Processing Unit
HPC	High-Performance Computing
MIC	Many-Integrated Core
MPI	Message Passing Interface
PFLOP/s	10^{15} FLOP/s
SIMD	Single Instruction Multiple Data
SoA	Struct of Arrays
SDK	Software Development Kit
SSE	Streaming SIMD Extensions
TFLOP/s	10^{12} FLOP/s

Contents

Abstract **ii**

Acknowledgements **iii**

Declarations **iv**

Sponsorship and Grants **vi**

Abbreviations **vii**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**

1.1 Motivation 3

1.2 Thesis Contributions 5

1.3 Thesis Overview 7

2 Performance Analysis and Engineering **9**

2.1 Benchmarking 10

2.2 Profiling 11

2.3 Code Optimisation 12

2.4 Performance Modelling 14

2.4.1 Analytical Modelling 15

2.4.2 Simulation 16

2.5 Summary 18

3 Parallel Hardware and Programming Models **19**

3.1 Instruction-Level Parallelism 20

3.2 SIMD / Vectorisation 20
3.3 Multi-threading 23
3.4 Message Passing 25
3.5 Benchmark Platforms 26
3.5.1 Single Nodes 26
3.5.2 Supercomputers 28
3.6 Summary 29

4 Optimisation of Pipelined Wavefront Applications 30

4.1 Benchmark Description 31
4.2 Related Work 33
4.3 Optimisation Challenges 34
4.4 Experimental Setup 34
4.5 Optimisations 36
4.5.1 SIMD and Multi-threading 36
4.5.2 Message Passing 41
4.6 Performance Results 42
4.6.1 Performance Breakdown 42
4.6.2 Architecture Comparison 44
4.7 Summary 47

5 Optimisation of Molecular Dynamics Applications 48

5.1 Benchmark Description 50
5.2 Related Work 51
5.3 Optimisation Challenges 52
5.4 Experimental Setup 53
5.5 Optimisations 55
5.5.1 SIMD 55
5.5.2 Multi-threading and Message Passing 63
5.6 Performance Results 65
5.6.1 Performance Breakdown 65

5.6.2 Thread Scaling . . . 67
5.6.3 Architecture Comparison . . . 70
5.7 Summary . . . 73

6 Developing “Performance-Portable” Applications 74
6.1 “Single Source” Methodology . . . 75
6.1.1 Work-Item and Work-Group Distribution . . . 76
6.1.2 Memory Layout . . . 78
6.1.3 Implicit vs. Explicit Vectorisation . . . 79
6.1.4 Device Fission . . . 79
6.1.5 Communication . . . 81
6.2 Benchmark Parameterisation . . . 81
6.2.1 Memory Layout . . . 82
6.2.2 Implicit vs Explicit Vectorisation . . . 83
6.2.3 Device Fission . . . 85
6.3 Performance Results . . . 86
6.4 Comparison with “Native” Implementations . . . 87
6.4.1 Pipelined Wavefront . . . 87
6.4.2 Molecular Dynamics . . . 89
6.5 Summary . . . 92

7 Predicting Application Performance on Future Architectures 93
7.1 Adapting Models for Future Architectures . . . 94
7.1.1 Single Node . . . 94
7.1.2 Multiple Node . . . 96
7.2 Communication Breakdown . . . 101
7.3 Machine Comparison . . . 101
7.3.1 Scalability . . . 101
7.3.2 Power Consumption . . . 105
7.4 Summary . . . 107

8 Conclusions and Future Work 108

8.1 Limitations 110

8.2 Future Work 112

8.2.1 k-blocking in Wavefront Applications 112

8.2.2 Conflict Resolution in Molecular Dynamics 115

8.2.3 Optimisation of Production Applications 119

Bibliography 120

Appendices 138

A Assembly Generated for OpenCL Gather Operations 138

List of Figures

1.1 A hybrid supercomputing architecture viewed from the perspective of (a) hardware and (b) software. 2

3.1 Parallel software/hardware stack. 19
3.2 Comparison of scalar and intrinsics code for a simple loop. 21
3.3 SPMD code for a simple loop. 22
3.4 Fork-join model used by OpenMP. 23

4.1 Simple loop-nest exhibiting a wavefront dependency (ignoring boundary conditions). 31
4.2 First three steps of a wavefront through a three-dimensional data grid. 31
4.3 Comparison of three different k-blocking depths. 37
4.4 A mapping from a two-dimensional grid of work-items onto three-dimensional data. 40
4.5 Comparison of speed-up for our optimised implementation of LU running on different architectures. 45

5.1 An atom's neighbourhood. 49
5.2 Combining a dot-product and AoS-to-SoA transpose in 128-bit SIMD. 58
5.3 Using a packed store to append to a neighbour list. 61
5.4 The hardware/class hierarchy. 63
5.5 A subdomain split into dependent and independent volumes. . . 64
5.6 Weak-scaling results for miniMD with a cut-off of 2.5. 69
5.7 Strong-scaling results for miniMD with a cut-off of 2.5. 69
5.8 Absolute performance of miniMD in atom-steps/s (higher is better). 72

6.1 OpenCL code-snippet for a kernel supporting all possible combinations of work-item and work-group size. 77
6.2 A dual-socket, hex-core node fissioned into (a) two sub-devices of six compute units; and (b) four sub-devices of three compute units. 80
6.3 Effect of neighbour list transposition on memory access pattern. Each arrow represents a single load instruction. 82
6.4 OpenCL code-snippet for a force compute kernel storing atom data in `float4`s. 83
6.5 Effect of using vector types on memory access pattern. 84
6.6 OpenCL code-snippet for a force compute kernel using vector arithmetic. 85
6.7 Comparison of speed-ups for parameterised force calculation kernel on various architectures. 86
6.8 OpenCL code-snippet for the gather of x, y and z positions in the force compute kernel with 8-way unrolling. 90

7.1 Observed and predicted times for a wavefront kernel on a Tesla C1060. 95
7.2 Observed times for a wavefront kernel on a Tesla C2050. 96
7.3 Breakdown of execution times for LU from the GPU model. . . . 101
7.4 Weak-scaling projections for LU. 102
7.5 Strong-scaling projections for LU. 104

8.1 First three wavefront steps for the new k-blocking policy. 114
8.2 Resolving conflicts using 128-bit SIMD. 116

A.1 Assembly from the vector gather kernel. 139
A.2 Assembly from the kernel with 8-way unrolling. 140

List of Tables

3.1 Hardware specifications of the CPUs used in this thesis. 27
3.2 Hardware specifications of the accelerators used in this thesis. . . 27
3.3 Hardware specifications of the integrated GPUs used in this thesis. 27
3.4 Hardware specifications of the *Sierra* and *Minerva* clusters. . . . 28
3.5 Hardware specifications of the *DawnDev* and *Hera* clusters. . . . 28

4.1 Compiler configurations for the wavefront optimisation study. . . 35
4.2 Performance breakdown for LU (Class C). 43

5.1 System configuration for the molecular dynamics optimisation study. 54
5.2 Comparison of theoretical worst-case instruction counts for four different force compute gather-scatter approaches. 57
5.3 Clock-cycles per neighbour and speed-up versus scalar for force compute gather-scatter approaches. 59
5.4 Static instructions for force compute. 59
5.5 Static instructions for neighbour list build. 62
5.6 Performance breakdown for miniMD (2.048M atoms). 66

6.1 Comparison of execution times (in seconds) for Intel CPU implementations of LU. 87
6.2 Speed-up of LU for two device fission configurations, and three k_B values. 88
6.3 Comparison of execution times (in seconds) for NVIDIA GPU implementations of LU. 89
6.4 Comparison of execution times (in seconds) for Intel CPU implementations of miniMD. 90

6.5 Comparison of execution times (in seconds) for NVIDIA GPU implementations of miniMD. 91

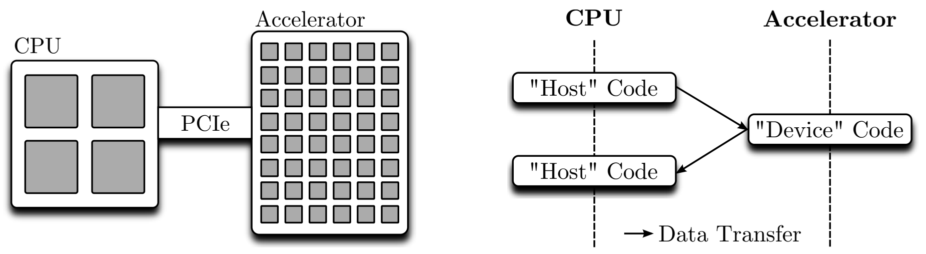
7.1 Model and simulation validations for LU. Execution times are given in seconds. 100
7.2 Compiler configurations for the LU model validation. 100
7.3 Cluster comparison for executing LU (Class E) within a fixed execution time. 105

8.1 Comparison of execution times (in seconds) for LU (Class C) using the old and new *k*-blocking policies. 113
8.2 Slow-down of conflict resolution approach. 117
8.3 Inefficiency of cross-neighbour and cross-atom SIMD. 118

CHAPTER 1
Introduction

Computational modelling and simulation form a key part of today's scientific and engineering research, permitting the rapid validation of theories in place of (or in addition to) physical experimentation. Where such experimentation is costly, impractical or dangerous (*e.g.* aircraft design, climate research, nuclear power), computational methods are essential, and there is an understandable desire for simulations to be run as quickly as possible. To this end, many researchers employ *clusters* and/or *supercomputers*, large machines typically thousands of times more powerful than a single desktop computer. Computer scientists in the field of high performance computing (HPC) seek to understand and maximise the performance of these machines, through the development of new hardware, better suited to arithmetic-intensive workloads; the design of more efficient algorithms; and the optimisation of scientific and engineering applications, to ensure that existing hardware is utilised effectively.

Over the past twenty years, the performance of supercomputers has improved significantly. Measured in terms of arithmetic throughput, floating-point operations per second (FLOP/s), the fastest supercomputer today is almost 300,000 times more powerful than the fastest supercomputer in 1993 [103]. Many modern supercomputers are comprised of several thousand commodity processors connected by some networking interface, and the key to effective utilisation of these machines is dividing problems into sub-tasks that can be solved independently and in *parallel*. Communication between sub-tasks (*e.g.* to satisfy data dependencies) must be handled explicitly by the programmer, and a generation of computer scientists has developed considerable expertise in the development and optimisation of these so-called "message passing" codes. This is a drastically



(a) A CPU and an accelerator connected via a PCI-Express interface.

(b) Offloading work to an accelerator.

Figure 1.1: A hybrid supercomputing architecture viewed from the perspective of (a) hardware and (b) software.

different programming methodology from that required for the vector-processor-based supercomputers of the 1970s and 1980s.

In 2008, HPC underwent another significant shift in technology, with the introduction of a supercomputer named Roadrunner. This machine, built by IBM for the Los Alamos National Laboratory, was noteworthy not only for being the first supercomputer ever to achieve a sustained performance of one petaflop per second (10^{15} FLOP/s), but also for its *hybrid* architecture – a coupling of 7,000 traditional processors with 13,000 computational *accelerators* (*i.e.* massively parallel co-processors[1], to which the CPU can "offload" computation as shown in Figure 1.1). Four years later, the use of such accelerators has become commonplace, powering some of the world's fastest supercomputers. These architectures promise high levels of raw performance, and a better performance-to-power cost ratio (*i.e.* more FLOP/s per Watt), reflecting some of the same motivations for the move away from vector machines in the 1990s. Many believe that the continued use of accelerators is necessary to reach the next big performance milestone – one exaflop per second (10^{18} FLOP/s) – within an acceptable power budget.

[1]The terms "accelerator" and "co-processor" are used interchangeably throughout this thesis.

1.1 Motivation

The speed of scientific and engineering applications has not followed the same trend as the speed of supercomputers. Rather, the gap between a supercomputer's maximum (*peak*) floating-point performance and that achieved by applications has grown wider over time – today, a typical scientific application achieves only 5–20% of any given machine's peak FLOP/s rate. This gap leaves room for significant performance improvements, and it is not unheard of for codes to run an order-of-magnitude faster following optimisation for modern architectures [12, 29, 78, 117, 145, 150, 162]. Such improvements in execution time have the potential to impact the cost of science significantly, with direct and measurable results: codes can be run in less time, or run in the same time on fewer computational resources. The former accelerates scientific delivery and reduces power costs, while the latter enables HPC sites to (*i*) purchase a smaller machine, decreasing procurement and maintenance costs; or (*ii*) make more efficient use of existing machines, by executing several small applications simultaneously.

The widening of this performance gap is largely the result of the so-called clock-speed "free lunch" [153] enjoyed by programmers during the early 1990s. During this time, applications were able to benefit from the steadily increasing clock-speed of CPUs afforded by Moore's Law [104] (*i.e.* the doubling of the number of transistors on a chip every two years). Due to issues concerning power consumption and heat dissipation [112], hardware manufacturers eventually abandoned increasing clock-speeds in favour of alternative methods of improving performance, placing multiple CPU cores on one die, and/or increasing the number of operations per cycle using Single-Instruction-Multiple-Data (SIMD) execution units. A large number of HPC codes have not yet adapted to these changes: they often do not make any distinction between cores on the same die and cores on a separate machine; and many do not make effective use of SIMD. This problem is most pronounced for accelerators, many of which are essentially collections of hundreds of simple, low-clocked cores capable of executing the same instruction on dozens of pieces of data simultaneously.

Complicating the matter further, the vast majority of supercomputers to date have supported the same programming model across successive hardware generations (*i.e.* a programming language like Fortran or C++, coupled with a message passing library), providing the ability to run existing codes on new machines without making any changes. Utilising accelerators, on the other hand, typically requires the adoption of new development tools and programming languages. Prior to the introduction of the Open Computing Language (OpenCL) [76] and OpenACC [2] standards, each accelerator required the use of its own proprietary programming model and, although this is largely no longer the case, applications are still likely to require significant algorithmic and code changes in order to make effective use of new architectures.

The HPC community is thus faced with the daunting task of updating two decades-worth of "legacy" applications, typically hundreds of thousands of lines of source code each, to utilise accelerators and new parallel programming models. The principle focus of the work presented in this thesis is the evaluation of these emerging parallel architectures for representative scientific and engineering codes, and the demonstration of a code development methodology that offers a middle-ground between: (*i*) focusing on a single architecture today, hoping that the resulting code will execute effectively on the hardware of the future; and (*ii*) potentially wasting considerable effort writing efficient code for each new hardware offering.

1.2 Thesis Contributions

The research presented in this thesis makes the following contributions:

- We develop one of the first reported CUDA implementations of a three-dimensional pipelined wavefront application, specifically the LU benchmark from the NAS Parallel Benchmark (NPB) Suite, detailing a number of general optimisations for this class of algorithm. We demonstrate the effect of k-blocking on the amount of exploitable parallelism, and the importance of choosing a k-blocking depth that is appropriate for the target architecture. Furthermore, and in contrast to previous work on two-dimensional wavefronts, we show that satisfying data dependencies via implicit CPU synchronisation (*i.e.* launching one CUDA kernel per hyperplane) can be the best parallelisation approach for some applications.

- We develop the first reported MIC implementation of a molecular dynamics application, specifically the miniMD benchmark from Sandia's Mantevo benchmark suite, and propose several novel improvements to its SIMD and threading behaviour. We examine the impact of instruction overhead on gather/scatter memory accesses, and show how storing data as an array-of-structs (AoS) can improve execution times across CPU and MIC hardware. Our results also highlight that redundant computation, the conventional approach for avoiding scatter write-conflicts on accelerator hardware, is not necessarily the best approach for molecular dynamics codes. Gather/scatter accesses are common to other classes of application (*e.g.* unstructured mesh) which we believe could benefit from optimisations similar to those presented here.

- We demonstrate a methodology for writing "performance-portable" codes using OpenCL, highlighting a number of important hardware and software parameters that should be considered during application development. For both wavefront and molecular dynamics applications, we show that it is possible to maintain an application that is optimised for multiple micro-architecture designs (if source code is sufficiently parameterised) without a significant performance penalty on any one architecture. Our OpenCL implementations of the LU and miniMD benchmarks are at most 2x slower than versions individually optimised for a single platform – and faster than the original Fortran and C codes.

- Finally, we utilise a combination of analytical performance modelling and discrete event simulation to examine the performance of accelerator-based supercomputers at scale. We extend two existing CPU modelling approaches by adding support for the prediction of PCI-Express transfer times, and compare the performance of commodity CPU and GPU clusters to that of an IBM Blue Gene/P. Our results highlight issues associated with the strong-scaling of applications on GPU-based clusters, and therefore suggest that accelerators may not be a suitable architectural choice for capability machines.

1.3 Thesis Overview

The remainder of the thesis is structured as follows:

Chapter 2 presents an account of the concepts, principles and terminology related to the field of high-performance computing, and more specifically that of performance engineering. This account includes a detailed survey of related literature, and describes many of the techniques at the core of this work.

Chapter 3 details the history of, and current state-of-the-art in, parallel hardware and software technologies. The programming challenges addressed by this research arise due to the many alternative forms of parallelism available across a wide variety of different platforms, and the contents of this chapter highlight the key differences between the micro-architectures compared and contrasted in this work.

Chapter 4 and **Chapter 5** present optimisation studies for the LU and miniMD benchmarks, respectively. The optimisation challenges and performance bottlenecks of both codes are identified, and we describe a number of novel optimisations designed to improve: (*i*) utilisation of SIMD execution units; (*ii*) multi-threading behaviour; and (*iii*) communication between devices (and nodes). For both benchmarks, we demonstrate that the same set of optimisations can benefit multiple architecture types.

Chapter 6 proposes an incremental programming methodology that allows HPC sites to develop "performance-portable" applications that adapt to multiple architectures at run- or compile-time. This methodology is evaluated for both benchmark applications, and the performance of the resulting codes is compared to that of the optimised implementations developed in previous chapters.

Chapter 7 demonstrates the application of traditional performance modelling techniques to emerging parallel architectures. We use a combination of analytical modelling and simulation to facilitate the prediction of execution times for accelerator-based supercomputers based on benchmark results from a single node, ultimately permitting a comparison of application performance on future machines.

Chapter 8 concludes the thesis, and discusses the implications of our research for the designers of HPC applications. We identify the limitations of the work presented here, and provide an outline of ongoing and future research.

CHAPTER 2
Performance Analysis and Engineering

As micro-architectures continue to evolve, and manufacturers introduce new hardware features, programming languages must also adapt to support them. The rapid development of supercomputing hardware thus poses a real challenge for the developers of scientific and engineering codes – for best performance, legacy applications representing several decades worth of development should be updated to reflect current architectural trends. However, due to a combination of application size (thousands to millions of lines of code) and the costs of employing domain and computer scientists with sufficient expertise to re-write code and re-validate results, significant code modification (in the worst case, starting from scratch) is not an option for many HPC sites.

The result is an iterative development cycle trading off code maintainability against performance, allowing codes to develop both in terms of functionality and performance optimality over the course of several hardware generations – domain scientists add new features to an existing code base as scientific knowledge and code requirements change, while computer scientists (in particular, performance engineers) work on identifying slow regions of code and accelerating them through the use of new languages and/or hardware.

The work in this thesis focuses on the latter process (*i.e.* performance analysis and engineering), which we consider here as consisting of four steps: benchmarking, profiling, code optimisation and performance modelling.

2.1 Benchmarking

The peak FLOP/s rate quoted by hardware manufacturers represents an architecture's *theoretical* maximum arithmetic throughput. It is becoming increasingly difficult for real-world applications to achieve this level of performance – at a minimum, most current generation hardware requires the evaluation of a single precision addition and multiplication on each of its SIMD execution units in every clock cycle; some hardware also requires that these operations are performed on the same values (*i.e.* a fused multiply-add), or the evaluation of transcendental functions (*e.g.* sine, cosine, reciprocal, square root) in hardware using special function units [92]. Many HPC codes make use of double precision, typically with some imbalance in the number of addition and multiplication operations, and there is thus a clear motivation for the production of alternative metrics which are more representative of the performance achieved by real applications. *Benchmarks* are pieces of code written specifically to collect such performance data, such that architectures can be compared in a more meaningful way.

So-called "micro" or "kernel" benchmarks are small, simple, pieces of code designed to extract low-level hardware information. The most famous of these benchmarks is LINPACK [40], a linear algebra benchmark that is used to determine a machine's *sustained* (*i.e.* achievable) FLOP/s rate and also its placement on the Top500 [103] list of the world's fastest supercomputers. Similar benchmarks exist for the evaluation of sustained memory bandwidth (*e.g.* STREAM [97]) and network communication performance (*e.g.* MPPTest [53] and SKaMPI [142]). Benchmarks of this kind are useful for drawing conclusions about general trends in hardware [82], and can also help to discover any discrepancies between the hardware specification published by vendors and what is experienced by users [33]. However, it is difficult to combine the simple metrics produced by micro-benchmarks to draw conclusions about the performance of more complex applications. The interaction of different hardware components and subsystems is likely to result in lower performance than that measured by micro-benchmarks, and the behaviour of some components (*e.g.* data cache) is too complex to be captured by these micro-benchmarks in a way that is representative of all applications.

"Macro" or "application" benchmarks are more complex codes designed to exhibit analogous computational behaviours to production applications. The use of such a benchmark may be preferable to the use of the application itself: scientific applications may take days or weeks to complete a simulation; and many applications are of a commercially sensitive (or classified) nature, thus preventing them from being distributed. Popular examples of macro-benchmarks are: the ASC benchmark suite [152] developed by the Los Alamos and Lawrence Livermore National Laboratories; the NAS Parallel Benchmark Suite developed by NASA Ames Research Centre [15, 16]; and the Mantevo benchmark suite [61, 62] developed by Sandia National Laboratories. These benchmarks typically output a breakdown of their execution time, such that the performance bottlenecks of different application and architecture combinations can be identified.

2.2 Profiling

Not all codes and benchmarks break down their execution time into component parts and, even where they do, the breakdown may not be at a low enough level to identify the root cause of poor application performance. In these situations, it is usual to make use of a *profiler* – a tool that monitors an application as it runs, and generates a *profile* of its execution. During performance analysis, the use of profilers allows programmers to examine both high level performance metrics (*e.g.* execution time [52, 116], memory consumption [131, 132, 157], network communication costs [68] and time spent reading/writing from/to disk [163]) and low-level metrics (*e.g.* instructions retired, cache hits, cache misses) recorded from performance counters [106].

One method of profiling (used extensively in this thesis) is to instrument source code directly, manually inserting calls to profiling functions and libraries into particular sections of code. The use of such instrumentation is well-suited to situations where performance engineers are only interested in the performance of a small section of an application, or want to record only a few high-level metrics (such as total execution time). For more general analysis, certain compilers can insert calls to profiling functions at compile time (*e.g.* gprof [52]); other profiling tools can be enabled at link time, wrapping system and library function calls with code that records information about certain events [116, 132, 163], thus allowing applications to be profiled without source code modification or re-compilation.

2.3 Code Optimisation

Once the reasons for an application performing poorly have been identified, performance engineers can begin the process of code optimisation. This can take many forms: simple code transformations (*e.g.* loop unrolling, pipelining and tiling) designed to reduce instruction count, increase throughput or improve cache behaviour [39, 98]; code re-writes that make use of new hardware features (*e.g.* loop vectorisation, SIMD intrinsics) [44]; and completely new algorithms, with lower computational complexity or better suited to modern architectures (*e.g.* being more amenable to parallelisation) [145].

When performing code optimisation studies on a single platform, there is an obvious baseline against which to measure performance improvements – the original code, run in the same configuration. Following the introduction of accelerators, many code optimisations took the form of porting (or “offloading”) arithmetic-intensive portions of an application to an accelerator: some studies featured comparisons between parallel codes utilising accelerators and unoptimised CPU codes [23]; some compared codes using different floating-point precision, or running at different scales [69, 149]; and others ignored important data transfer overheads [51]. Taken out of context, the speed-up figures reported in such studies can be misleading, and many papers have since disputed claims that accelerators are several orders of magnitude faster than traditional architectures [21, 22, 92, 158]. In the optimisation studies presented in this thesis, we strive to keep architectural comparisons as fair as possible – specifically, we: (*i*) spend a significant amount of time optimising baseline codes before considering accelerators; (*ii*) only draw comparisons between architectures running codes of the same floating-point precision; and (*iii*) report full application times, rather than focusing on sections of codes that may be more amenable to acceleration.

The introduction of hybrid supercomputer architectures has had another significant effect on code optimisation – with a wide variety of micro-architectures available even within a single machine, developers are now keen for their applications to exhibit good levels of performance across different hardware, without maintaining separate implementations for each. Recent work has proposed several alternative methodologies for the development of such "performance portable" applications, aiming to achieve high levels of performance on multiple architectures using a single source code.

"Directive"-based programming allows developers to mark (using pragmas) the regions of code that they wish to be cross-compiled for an accelerator, and is quickly becoming the programming model of choice for legacy application developers. The HMPP [37] and OpenACC [2] standards are currently supported by compilers from CAPS, Cray and PGI, and the inclusion of similar directives within OpenMP has also been proposed [4, 5, 89, 90]. Although this approach allows applications developed on one architecture to be compiled for alternative targets with relative ease, it does not necessarily make any guarantees of performance portability – directives do not improve the performance of the original CPU code, and recent work by Lee *et al.* [91] suggests that current generation directives are not expressive enough to facilitate accelerator-specific performance tuning.

Another alternative is the use of domain specific languages (DSLs) and/or "active libraries", which permit application developers to express their problem in a high-level and domain specific manner, leaving the actual code implementation to a library or smart compiler. For example, the OP2 [46, 47] and Liszt [35] projects both provide abstractions for codes that operate on unstructured grids; programmers write applications in terms of operations over constructs such as nodes or edges, and the compiler transforms this representation into a binary optimised for the target platform.

Other research (including that presented in this thesis) has investigated the utility of using the recent OpenCL [76] standard to develop codes in a platform-agnostic manner [42, 83, 159, 161], parameterising codes in a fashion that allows them to adapt to hardware changes – this is discussed in more detail in Chapter 6.

2.4 Performance Modelling

The term "performance modelling" describes a collection of techniques that allow computer scientists to reason about and predict the performance of an application. Models of sufficient accuracy can be used to augment performance analysis and engineering activities, and their use has been demonstrated in identifying performance bottlenecks [34]; evaluating the impact of code optimisations ahead of implementation [109]; predicting the performance of applications when ported to new architectures [102]; and comparing the communication behaviour of codes at scale on different machines [63, 72, 74].

This last point in particular forms an important part of this thesis. HPC codes are rarely run on a single processor, instead running on hundreds or thousands of compute "nodes" in parallel. There is a clear need to reflect this in both benchmarking and code optimisation; benchmark results must be representative of performance when codes are run at scale, and code optimisations that only work at the level of a single node are unlikely to impact scientific delivery significantly. However, there are several reasons that it is desirable to benchmark at much smaller scale (at least initially). Firstly, it allows for new hardware and software to be evaluated without significant investment – an HPC site can buy one or two small computers built on a new architecture, and use these to carry out performance analysis before committing to purchasing a much larger machine [57]. Secondly, and arguably more importantly from a software development perspective, it allows programmers to avoid the problems associated with debugging parallel programs running at scale [18].

Performance modelling in this context can be broadly divided into two alternative techniques: (i) *analytical* modelling, where an application's execution time is represented mathematically as a series of equations; and (ii) *simulation*, where a code (or a representation thereof) is run on a model of a machine that is simulated in software. Several works have looked to use analytical models and simulations together, to lend further credence to their predictions [55, 99, 107].

2.4.1 Analytical Modelling

Generally, an application's execution time (T_{total}) can be represented by the following equation [6]:

$$\begin{aligned} T_{total} = {} & (T_{computation} + T_{communication} - T_{overlap}) \\ & + T_{synchronisation} + T_{overhead} \end{aligned} \tag{2.1}$$

That is, the sum of the time that the application spends performing computation and communication, minus the amount of time during which computation and communication are performed concurrently. $T_{synchronisation}$ and $T_{overhead}$ account for communication costs that cannot be overlapped, such as processor synchronisation and message overheads.

Each of these components is constructed from a series of sub-models, representing the contribution of an application's functions to overall execution time. Computation costs are typically measured empirically, via benchmarks, whereas per-message communication costs are usually predicted based on a simple set of network parameters [10, 32, 105]. Analytical models of this form have been demonstrated to achieve high levels of accuracy on a wide range of applications from various scientific domains [7, 30, 34, 73].

The main advantage of this technique is that, once constructed and appropriately parameterised, an analytical model can be evaluated very quickly – predicting application performance in a new configuration is as simple as substituting different parameter values into the model equations. However, the construction and parameterisation of a model is a difficult task, requiring significant understanding of the code's behaviour in order to represent it mathematically. Recent work, by Mudalige *et al.* [107, 110, 111], suggests that this process could be greatly simplified by building upon pre-existing and pre-validated models that capture the key performance behaviours of whole application classes (as opposed to individual applications).

The analytical models contained in this thesis use LogGP [10] to model communication behaviour, which characterises a network in terms of: latency (L); overhead (o); bandwidth for small messages (g); bandwidth for large messages (G); and the number of processors (P).

2.4.2 Simulation

Simulators aim to address the shortcomings of analytical modelling, shifting the complexity away from individual application models and into a re-usable package capable of representing machine and/or application state in software. The models executed by simulators are much simpler than analytical models, typically taking the form of either: (i) a trace, recorded from an application run by some accompanying tool; or (ii) a representation of the application's source code.

"Micro" simulators replicate the behaviour of individual hardware components at a very low level, tracking system state on a clock-by-clock basis. Such simulators are often used to evaluate alternative hardware designs prior to the fabrication of silicon chips (both in research and industry); to evaluate the performance impact of novel hardware features [8, 85]; or to identify the performance bottlenecks of an application [99].

"Macro" simulators instead aim to capture the behaviour of a machine at a much higher level, with changes to system state occurring in response to particular events (*e.g.* network communications). Like analytical models, macro-simulators typically require that computation times be collected empirically or from a micro-simulator – however, they are able to couple these times with additional system state information (*e.g.* processor/network load from other users and applications) to produce more accurate results.

The increased accuracy of simulation comes at a cost, however, requiring significant computational resources. At the micro-level, each executed instruction must be simulated in software, while at the macro-level, events from thousands of simulated processors must be handled. In both cases, a simulation will take significantly longer to produce performance predictions than an equivalent analytical model.

The research in this thesis makes use of the Warwick Performance Prediction (WARPP) toolkit [55, 56], a macro-simulator similar to Sandia's Structural Simulation Toolkit (SST) [70] and successor to the University of Warwick's Performance Analysis and Characterization Environment (PACE) [26, 119].

2.5 Summary

Due to the size and complexity of scientific and engineering applications, it is becoming common for the performance analysis and engineering process to take place at the level of macro-benchmarks, with optimisations only later applied to production codes [62]. We adopt this methodology in this thesis, using two macro-benchmarks to investigate the utility of alternative parallel hardware and programming models.

This process allows for drastic changes to be made much more rapidly than could ever be possible in the context of a legacy application. Although an analytical (or "paper and pencil") exploration of the design space can typically label a potential optimisation as either fruitless or promising, only micro-architectural simulation or a real implementation (both of which require the algorithm to be represented in code of some form) can provide concrete performance numbers.

CHAPTER 3
Parallel Hardware and Programming Models

Modern computer architectures are highly parallel, featuring hardware support for many different types of parallelism. Figure 3.1 represents this parallelism diagrammatically, mapped to a "stack" of hardware features: superscalar architectures provide instruction-level parallelism (ILP), executing multiple independent instructions in a single clock cycle; SIMD architectures provide data-level parallelism, with each instruction capable of executing on a *vector* of data elements; and multi-core and cluster architectures provide task-level parallelism, supporting several *threads* or *nodes* running independently of one another. As suggested by the stack representation, each of these levels of parallelism can make use of the level beneath it – each node in a supercomputer can run multiple threads, each thread can make use of SIMD instructions, and each SIMD instruction can be scheduled alongside others.

Understanding these levels of parallelism, and how best to utilise them, is key to achieving high levels of performance for scientific and engineering applications. In the remainder of this section, we detail the programming models used for the research in this thesis, and how they map to the parallelism available in current generation hardware.

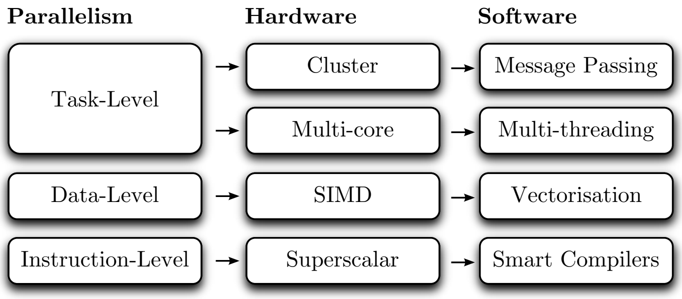


Figure 3.1: Parallel software/hardware stack.

3.1 Instruction-Level Parallelism

Modern architectures support ILP in several forms. "Out-of-order" processors are able to execute instructions as soon as their inputs are available, rather than in the order in which they appear in a program; this is often coupled with "speculative" execution, which allows a processor to execute instructions that may or may not be required (based on the result of some branch condition). Together, these two features allow a processor to utilise clock-cycles that would otherwise be wasted if a program's instructions were to be executed sequentially.

Writing code to maximise ILP is a difficult task, requiring a detailed understanding of both the application and the target architecture. Data dependencies between instructions, instruction throughputs/latencies and the way in which instructions are mapped to an architecture's execution units must all be considered [141]. Furthermore, many of the techniques that a programmer could employ in code to improve ILP (such as loop unrolling) [71] are now supported by optimising compilers; we therefore focus on exploiting data and task level parallelism in this research.

3.2 SIMD / Vectorisation

The SIMD execution units of modern processors are very similar conceptually to the vector processors of the 1970s and 1980s, providing the capability to perform the same operation on a number of data items simultaneously. The number of values manipulated by each instruction (*i.e.* the SIMD "width") differs by architecture, and is typically considerably less than on old vector processors, but has been following an upward trend: the 128-bit Streaming SIMD Extensions (SSE) of x86 architectures have been augmented by 256-bit Advanced Vector Extensions (AVX); the new Intel Many-Integrated Core (MIC) architecture supports 512-bit SIMD; and GPUs typically support 1024- or 2048-bit SIMD. We list SIMD widths in bits because a SIMD register can be used to store data of multiple types – for example, a 128-bit SIMD register could store

```
for (int i = 0; i < 128; i++) {
  a[i] = b[i] + c[i];
}
```

(a) Scalar

```
for (int i = 0; i < 128; i += 4) {
  __m128 _a;
  __m128 _b = _mm_load_ps(&b[i]);
  __m128 _c = _mm_load_ps(&c[i]);
  _a = _mm_add_ps(_b, _c);
  _mm_store_ps(&a[i], _a);
}
```

(b) SSE

Figure 3.2: Comparison of scalar and intrinsics code for a simple loop.

four 32-bit single precision floating-point values, or two 64-bit double precision floating-point values.

Traditional programming languages used for HPC codes (such as C, C++ or Fortran) are scalar in nature, and thus do not make use of an architecture's SIMD execution units when compiled. In order to utilise these SIMD instructions, programmers must either: (*i*) use compilers supporting *auto-vectorisation*, which transform scalar code into an equivalent vector representation; or (*ii*) identify vectorisation opportunities explicitly, through extensions to the original language or *intrinsic* functions.

Coding with intrinsic functions is often the best way to achieve maximum performance, since each intrinsic maps directly to a hardware instruction (or series of instructions). It also allows programmers to reason about potential bottlenecks much more readily during development, since operations that do not make efficient use of SIMD will have been written by hand. However, it is for exactly these reasons that programmers often prefer *not* to use intrinsics – they are less readable than scalar code and require an in-depth understanding of an architecture's SIMD execution capabilities. Further, intrinsic functions differ by instruction set and may not be supported in the same way by all compilers. Figure 3.2 compares scalar code for a simple loop written in C++ to an implementation using SSE intrinsics.

```
__kernel simple_loop(...) {
  int i = get_global_id(0);
  if (i < 128) {
      a[i] = b[i] + c[i];
  }
}
```

Figure 3.3: SPMD code for a simple loop.

Many programmers thus prefer to rely on auto-vectorisation. Most auto-vectorisation compilers (*e.g.* Cray, Intel, PGI, Sun) will attempt to apply vectorisation to a function's inner-most loop, effectively unrolling the loop a number of times to match the hardware's SIMD width. This operation relies on the compiler being able to identify that the operations in the loop are independent, and may require some programmer intervention (such as insertion of pragmas) [81] to assert that auto-vectorisation is possible. This approach works well for simple loops, but programmers with sufficient domain and hardware knowledge are likely to be able to construct a more efficient instruction sequence for complex loops – the compiler has no knowledge of the problem domain or how the code will be run, and therefore cannot make certain assumptions (*e.g.* that all inputs will be valid and not `NaN`).

An alternative paradigm that targets SIMD execution units is that of Single-Program-Multiple-Data (SPMD) programming, where programmers write code from the perspective of an individual parallel task. The resulting code is much more readable than when intrinsics are used, and also eases the process of auto-vectorisation for the compiler, since each task is guaranteed to be independent. This SPMD model has gained significant traction in HPC for programming GPU architectures, in the form of NVIDIA's CUDA [118] (which grew from Buck's earlier Brook [25] language), but recent developments have shown that it is equally applicable to CPUs; the open-source ispc (Intel SPMD Program Compiler) [135] and the Intel OpenCL compiler both support this form of auto-vectorisation. Figure 3.3 shows how the loop from Figure 3.2 could be implemented in OpenCL; each iteration of the loop is carried out by a separate task, identified by a global index.

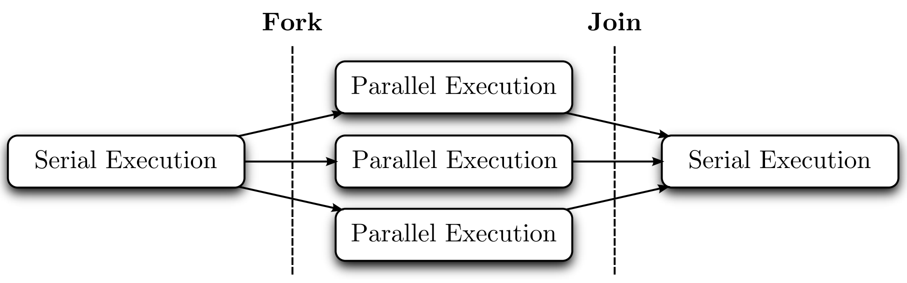


Figure 3.4: Fork-join model used by OpenMP.

The research in this thesis utilises all three of these vectorisation approaches, comparing the level of performance achieved for our benchmarks when using auto-vectorisation, explicit vectorisation with intrinsics and SPMD programming. For SPMD, we use OpenCL, which launches functions known as *kernels* across a number of parallel *work-items*.

3.3 Multi-threading

The power (and cooling) required by an architecture increases more rapidly with clock-speed than with number of cores [45]. Modern hardware designs are thus typified by a high number of cores with low clock-speeds, and efficient parallel execution relies on running parts of an application on all cores concurrently. This is accomplished via *multi-threading*, which sees threads execute some sub-problem or task independently using the resources of a separate core. Many modern architectures also support simultaneous multi-threading (SMT) or "hyper-threading", allowing multiple threads to execute on and share the resources of a single core: x86 CPUs typically support 2-way SMT; MIC supports 4-way SMT; and GPUs support a large number of threads per core (the exact number differs by vendor and architecture revision).

A common model of parallel execution adopted by multi-threaded applications is called *fork-join*. As represented in Figure 3.4, a master thread executing serially creates ("forks") a number of additional threads during parallel code sections, and waits for each thread to complete ("join") before continuing with serial execution. This is the approach taken by OpenMP [4], which allows programmers to mark (via pragmas) the loops in their program that can be executed in parallel.

An alternative programming model is adopted by Cilk [20] and its successor, Intel Cilk Plus [67], which allow programmers to identify (via extensions to the C language) any dependencies that exist between functions. The result is a dependency graph, which the Cilk scheduler uses to determine which functions should be run by which threads. This approach is better suited to exposing "nested parallelism" than the OpenMP approach of using threads to execute independent loop iterations, since a function and the functions it calls can run in parallel; it also potentially exposes more parallelism, since computation from multiple functions (and hence multiple loops) can be performed simultaneously.

Using a low-level library such as POSIX threads (Pthreads) [114] gives a programmer greater control over the threading behaviour of an application, but at the expense of significantly more complicated code. OpenMP/Cilk are therefore typically favoured because of their ease of use.

The SPMD model is also applicable to multi-threading. All tasks are known to be independent at compile time, and compilers are thus free to distribute work across threads in addition to (or instead of) SIMD execution units without affecting correctness. Both CUDA and OpenCL allow for threads/work-items to be grouped together into blocks/*work-groups*, providing certain guarantees regarding synchronisation and memory access behaviours within a work-group, but do not guarantee that all work-items will be executed by the same thread. Neither programming model permits synchronisation between work-items in different work-groups.

3.4 Message Passing

Whereas communication between SIMD execution units and threads is possible through shared memory, separate nodes in a supercomputer have only one method of communication – the passing of messages back and forth across some dedicated network interface. In HPC, such communication is most commonly performed through the use of communication libraries built upon the standardised Message Passing Interface (MPI) [3], although recent work has investigated the use of Partitioned Global Address Space (PGAS) models [147] and OpenCL [79] for programming clusters of machines.

Due to the ubiquity of MPI programs, and its familiarity to programmers, many HPC codes use MPI not only for inter-node communication but also for intra-node communication (*i.e.* in place of threading). This has been shown to cause problems, such as increased memory overhead (due to redundant storage and unnecessary communication buffers) [19, 28, 84] and increased communication times (due to multiple threads contending for use of a single network interface) [54]. As such, much research has investigated the utility of so-called "hybrid" MPI/OpenMP programming approaches [140, 151], providing a better mapping from software to the underlying hardware.

Where accelerators are employed, message passing appears in two forms: firstly, any communication between the *host* CPU and an attached accelerator *device* must take place via explicit message passing over the PCI Express (PCIe) interface that connects them; and secondly, any communication between devices in separate nodes must travel over both PCIe and the network. Recent advances in accelerator technology aim to alleviate these problems, by giving CPU and accelerator cores direct access to the same memory (*e.g.* AMD Fusion, Intel HD Graphics, NVIDIA Project Denver), or by giving the accelerator direct access to the network interface [100].

3.5 Benchmark Platforms

The research in this thesis makes use of a wide variety of different architectures, from a number of hardware vendors. Not all machines are used in every study, due to: (*i*) limited access to many of the large-scale machines, which are shared resources external to the University of Warwick; (*ii*) incompatibilities between programming techniques and hardware; and (*iii*) the rapid development of hardware during the course of our work – we report results from the most recent hardware iteration available at the time each study was performed.

3.5.1 Single Nodes

We divide the architectures used into three types: CPUs (Table 3.1); accelerators and co-processors, including both discrete GPUs and Intel MIC (Table 3.2); and "fused" architectures, featuring some combination of CPU and GPU cores on the same chip (Table 3.3).

In all cases, performance is reported as peak GFLOP/s in single precision, and bandwidth as peak transfer rate from main memory in gigabytes per second (GB/s). As noted previously, these figures are not necessarily representative of the level of sustained performance that can be achieved by an application. Power is reported as thermal design power (TDP) in Watts, and for fused architectures includes the power drawn by both the CPU and GPU cores. We additionally list the instruction set supported by the CPUs.

As this thesis makes use of OpenCL, we also list the number of *compute units* and *processing elements* for each device. On CPUs, each core is a compute unit consisting of one processing element; with SMT, each hyper-thread appears as a distinct compute unit (*e.g.* the Intel Xeon E3-1240 has 4 cores with 2-way SMT and therefore has 8 compute units). GPUs from different vendors are divided into compute units and processing elements in different ways: on the Intel GPUs we use, each execution unit is a compute unit of 8 processing elements; on the NVIDIA GPUs, each stream multiprocessor (compute unit) consists of 8, 32 or 192 “CUDA cores” (processing elements) on the Tesla, Fermi and Kepler architectures respectively; and on the discrete and integrated AMD GPUs, both of which belong to the Evergreen series, a compute unit contains 16 stream cores (compute units) of 5 simple processing elements each.

	Intel			**AMD**
	X5550	**E3-1240**	**E5-2660**	**A8-3850**
Cores	4	4	8	4
Compute Units	4	8	16	4
Proc. Elements	4	8	16	4
Peak GFLOP/s	85	211	281	93
Bandwidth (GB/s)	32	21	51	30
TDP (Watts)	95	80	95	100
Instruction Set	SSE 4.2	AVX	AVX	SSE 4a
Micro-architecture	Nehalem	Sandy Bridge	Ivy Bridge	Llano

Table 3.1: Hardware specifications of the CPUs used in this thesis.

	NVIDIA					**AMD**	**Intel**
	8400GS	**9800GT**	**C1060**	**C2050**	**GTX680**	**V7800**	**5110P**[a]
Cores	1	14	30	14	8	18	60
Compute Units	1	14	30	14	8	18	240
Proc. Elements	8	112	240	448	1536	1440	240
Peak GFLOP/s	33	504	933	1288	3090	2016	2022
Bandwidth (GB/s)	6	58	102	144	192	128	320
TDP (Watts)	25	105	189	238	195	150	225
Micro-architecture	G98	G92	Tesla	Fermi	Kepler	Cypress	Knights Corner

Table 3.2: Hardware specifications of the accelerators used in this thesis.

	Intel	**AMD**
	HD4000	**HD6550D**
Compute Units	16	5
Proc. Elements	128	400
Peak GFLOP/s	147	480
Bandwidth (GB/s)	26	30
TDP (Watts)	77	100
Micro-architecture	HD Graphics 4000	BeaverCreek

Table 3.3: Hardware specifications of the integrated GPUs used in this thesis.

[a]Experimental results were recorded from evaluation silicon, with slight differences from the listed specification.

3.5.2 Supercomputers

The studies contained in this thesis were performed using four supercomputers: a cluster of dual-socket, hex-core Intel Xeon nodes (*Sierra*); an IBM Blue Gene/P (*DawnDev*); a cluster of quad-socket, quad-core AMD Opteron nodes (*Hera*); and the GPU partition of a cluster of dual-socket, hex-core Intel Xeon nodes (*Minerva*). *Sierra*, *DawnDev* and *Hera* are housed in the Open Computing Facility at the Lawrence Livermore National Laboratory; *Minerva* is located at the Centre for Scientific Computing at the University of Warwick.

The machines in Table 3.4 are used in Chapter 6 and those in Table 3.5 are used in Chapter 7. All details were correct at the time of writing.

	Sierra	***Minerva*** (GPU Partition)
Nodes	1,944	6
CPUs/Node	2 × Intel X5660	2 × Intel X5650
Cores/Node	12	12
Core Frequency	2.8 GHz	2.66 GHz
Peak TFLOP/s	261.3	16.9
Memory per Node	24 GB	24 GB
OS	CHAOS 4.4	SUSE Enterprise Linux 11
Interconnect	InfiniBand QDR (QLogic)	InfiniBand TrueScale 4X-QDR
Accelerators/Node	None	2 × NVIDIA M2050

Table 3.4: Hardware specifications of the *Sierra* and *Minerva* clusters.

	DawnDev	***Hera***
Nodes	1,024	864
CPUs/Node	4 × PowerPC 450d	4 × AMD Opteron
Cores/Node	4	16
Core Frequency	0.85 GHz	2.3 GHz
Peak TFLOP/s	13.9	127.2
Memory per Node	4 GB	32 GB
OS	Compute Node Kernel	CHAOS 5.0
Interconnect	Blue Gene Torus + Tree	InfiniBand DDR (Mellanox)
Accelerators/Node	None	None

Table 3.5: Hardware specifications of the *DawnDev* and *Hera* clusters.

3.6 Summary

Modern micro-architectures support many forms of parallelism, each of which is paired with a corresponding programming model or language feature. However, many scientific and engineering codes (in particular, legacy applications) do not utilise all levels of this parallel stack effectively. Most use MPI, since it is a base requirement for the utilisation of large-scale supercomputing systems, but the lower levels of the stack are often overlooked.

It is easy to map "embarrassingly parallel" codes to these lower levels efficiently, but realistic applications (such as the LU and miniMD benchmarks) pose a greater challenge due to the presence of complex data dependencies and irregular memory access patterns. The advent of accelerator-based supercomputers is a perfect opportunity to analyse the performance behaviours of such codes and to explore the capabilities of modern architectures – an application will typically require some degree of algorithmic change due to the limitations of current accelerator hardware (*e.g.* no global synchronisation across threads, fewer registers) and any optimisations identified as being beneficial to accelerators are also likely to be applicable to modern CPU architectures.

CHAPTER 4
Optimisation of Pipelined Wavefront Applications

Pipelined wavefront applications are common to many areas of HPC, including computational fluid dynamics [15, 16] and particle physics [1, 111]. These applications are characterised by a particular data dependency pattern, which we refer to henceforth as a "wavefront dependency". The simple loop-nest in Figure 4.1 demonstrates this dependency, with the computation of the value for a grid point (i, j, k) depending upon the values of three of its neighbours, computed by previous loop iterations: $(i, j, k-1)$, $(i, j-1, k)$ and $(i-1, j, k)$.

The *hyperplane* algorithm described by Lamport [86] allows for these loops to be solved in parallel, in spite of the dependency. It is based on the key observation that all of the data-points lying on a particular hyperplane can be computed independently; for three-dimensional problems, this hyperplane is defined as $h = i + j + k$, but the algorithm is applicable to problems of any dimensionality. Figure 4.2 demonstrates the first three steps of a three-dimensional wavefront, for $h = 0$, $h = 1$ and $h = 2$; the current step is coloured light grey and previous steps in dark grey.

The performance of wavefront applications is well understood on clusters of conventional multi-core architectures [65, 107, 109, 110, 111], but at the time that the research presented in this thesis was performed, GPU-based implementations of three-dimensional wavefront applications (both on a single device and at cluster scale) were scarce. To our knowledge, the work described in this chapter constitutes the first port of the LU benchmark to a GPU. The principle use of this benchmark is comparing the suitability of different architectures for production computational fluid dynamics applications [144], and thus the results presented here have implications for large-scale production codes.

```
for (int k = 0; k < max_k; k++) {
  for (int j = 0; j < max_j; j++) {
    for (int i = 0; i < max_i; i++) {
      a[k][j][i] = a[k-1][j][i] + a[k][j-1][i] + a[k][j][i-1];
    }
  }
}
```

Figure 4.1: Simple loop-nest exhibiting a wavefront dependency (ignoring boundary conditions).

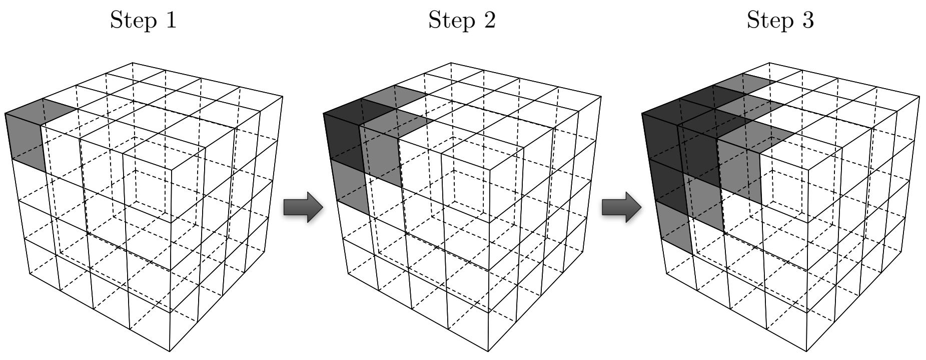


Figure 4.2: First three steps of a wavefront through a three-dimensional data grid.

4.1 Benchmark Description

The LU benchmark belongs to the NPB Suite, a set of parallel aerodynamic simulation benchmarks. The code implements a simplified compressible Navier-Stokes equation solver, which employs a Gauss-Seidel relaxation scheme with symmetric successive over-relaxation (SSOR) for solving linear and discretised equations. The reader is referred to [15] for a thorough discussion of the mathematics.

LU uses a three-dimensional data grid of size N^3 (*i.e.* the problem is always a cube). As of release 3.3.1, NASA provide seven different application "classes" for which the benchmark is capable of performing verification: Class S (12^3), Class W (33^3), Class A (64^3), Class B (102^3), Class C (162^3), Class D (408^3) and Class E (1020^3). The use of these standard problem classes in this work ensures that our results are directly comparable to those reported elsewhere in the literature.

In the MPI implementation of the benchmark, this data grid is decomposed over a two-dimensional processor array of size $P_x \times P_y$, assigning each of the processors a stack of N_z data "tiles" of size $N_x/P_x \times N_y/P_y \times 1$. Initially, the algorithm selects a processor at a given vertex of the processor array which solves the first tile in its stack. Once complete, the edge data (which has been updated during this solve step) is communicated to two of its neighbouring processors. These adjacent processors – previously held in an idle state via the use of MPI-blocking primitives – then proceed to compute the first tile in their stacks, while the original processor solves its second tile. Once the neighbouring processors have completed their tiles, their edge data is sent downstream. This process continues until the processor at the opposite vertex to the starting processor solves its last tile, resulting in a "sweep" of computation through the data array.

The pseudo-code in Algorithm 4.1 details the SSOR loop that accounts for the majority of LU's execution time. Each of the subroutines in the loop exhibit different parallel behaviours: `jacld` and `jacu` carry out a number of independent computations per grid-point, to pre-compute the values of arrays used in the forward and backward wavefront sweeps; `blts` and `buts` are responsible for the forward and backward sweeps respectively; `l2norm` computes a parallel reduction (on user-specified iterations); and `rhs` carries out three parallel stencil update operations, which have no data dependencies between grid-points. The number of loop iterations is configurable by the user at both compile- and run-time, but is typically 250–300 in Classes A through E. The reader's attention is drawn to the location of the calls to the `exchange_1` function – in the real code, these calls occur as the first and last lines of code inside `blts` and `buts`; we show them outside of `blts` and `buts` to facilitate later discussion.

Algorithm 4.1 Pseudo-code for the SSOR loop.

```
for $iter = 1$ to $max\_iter$ do

    for $k = 1$ to $N_z$ do
        call jacld($k$)                 ▷ form lower triangular part of Jacobian matrix.
        call exchange_1($k$, recv)      ▷ receive data from north/west neighbours.
        call blts($k$)                  ▷ perform lower triangular solution.
        call exchange_1($k$, send)      ▷ send data to south/east neighbours.
    end for

    for $k = N_z$ to 1 do
        call jacu($k$)                  ▷ form upper triangular part of Jacobian matrix.
        call exchange_1($k$, recv)      ▷ receive data from south/east neighbours.
        call buts($k$)                  ▷ perform upper triangular solution.
        call exchange_1($k$, send)      ▷ send data to north/west neighbours.
    end for

    call l2norm()
    call rhs()                          ▷ compute steady-state residuals.
    call l2norm()

end for
```

4.2 Related Work

A number of studies have investigated the use of accelerator architectures for the Smith-Waterman string matching algorithm (a two-dimensional wavefront algorithm) [9, 93, 113], and two previous studies [48, 134] detail the implementation of a different three-dimensional wavefront application (Sweep3D [1]).

The first of these Sweep3D studies [134] utilises the Cell Broadband Engine (B.E.), exploiting five levels of parallelism in the implementation. The performance benefits of each are shown in order, demonstrating a clear path for the porting of similar codes to the Cell B.E. architecture. In the second [48], the Sweep3D benchmark is ported to CUDA and executed on a single Tesla T10 processor. Four stages of optimisation are presented: the introduction of GPU threads, using more threads with repeated computation, using shared memory and using a number of other methods that contribute only marginally to performance. The authors conclude that the performance of their GPU solution is good, extrapolating from speed-up figures that it is almost as fast as the Cell B.E. implementation described in [134].

4.3 Optimisation Challenges

The amount of parallelism available in a wavefront application depends upon the problem size (*i.e.* the maximum values of i, j and k) and is variable throughout an application run – as the value of h increases, more valid combinations of i, j and k exist (*i.e.* the size of the hyperplane increases). How best to map this limited parallelism to modern architectures is unclear, and the data dependency must be satisfied at all levels of the parallel stack described in Chapter 3.

The memory access pattern exhibited by wavefront applications is very predictable, but is not well suited to SIMD execution. If the data grid remains stored in its default memory layout (*i.e.* row-major form) then the values of all of the grid-points satisfying $h = i + j + k$ for a given value of h will not be contiguous in memory, requiring expensive gather and scatter operations.

The memory requirements of LU ($\approx$ 160 GB for a 1020^3 Class E problem) are also considerably larger than the amount of RAM available per node in commodity clusters, and the amount of memory available to many accelerators is limited by their use of GDDR, thus necessitating the use of large distributed machines. The MPI implementation of the original benchmark requires frequent network communication, making the bandwidth and latency of the PCIe bus a potential bottleneck.

4.4 Experimental Setup

Version 3.2 of the LU benchmark, on which our work is based, is written in Fortran 77 and utilises MPI for communication between processing elements. The GPU implementation makes use of NVIDIA's CUDA. The standard language choice for developing CUDA programs is C/C++ and, although the Portland Group offer a commercial alternative (CUDA Fortran), the first stage in our porting of LU was to convert the entire application to C. The resulting C code is 1.4x *slower* than the original Fortran; therefore, the performance improvements discussed in the remainder of this chapter arise from utilisation of the GPU, rather than from any optimisations introduced during the process of changing language.

Although NASA explicitly requests the use of double precision in the benchmark suite, the ported version of the benchmark was instrumented to allow the selection of floating-point precision at compile time. The accuracy of the single precision implementation is lower (*i.e.* the error exceeds the default epsilon of 10^{-8}), but the mathematics is otherwise identical, and can be executed significantly faster than the double precision implementation on some GPUs.

We use four NVIDIA GPUs in these experiments. The GeForce 8400GS and 9800GT are consumer cards that are not designed for HPC – we include them mainly out of interest, as a means of evaluating the use of typical workstation GPUs for scientific workloads. The Tesla C1060 and C2050 are NVIDIA's flagship HPC cards, based on the "Tesla" and "Fermi" architectures respectively.

The compiler configuration for the experiments in this chapter are given in Table 4.1. We note that, although it is more usual to use the flag `-arch="sm_20"` for the Tesla C2050, we found that `-arch="sm_13"` resulted in better performance for our code. We have since learned that this is due to several side-effects of compiling with `-arch="sm_13"`, specifically that the compiled code uses 32-bit pointers (and hence fewer registers) and uses faster mathematical functions with lower precision.

Device	**Compiler**	**Options**
Intel X5550 (Fortran)	Sun Studio 12 (Update 1)	`-O5 -native` `-xprefetch -xunroll=8` `-xipo -xvector`
Intel X5550 (GPU Host)	GNU 4.3	`-O2 -msse3 -funroll-loops`
GeForce 8400GS/9800GT	NVCC	`-O2 -arch="sm_11"`
Tesla C1060/C2050	NVCC	`-O2 -arch="sm_13"`

Table 4.1: Compiler configurations for the wavefront optimisation study.

4.5 Optimisations

The focus of this study is the optimisation of wavefront applications for GPU architectures. Although some of the optimisations we propose are also beneficial to CPU architectures, we discuss them mainly in the context of NVIDIA's GPUs. We therefore divide the optimisations into two types: (*i*) those that affect the SIMD and multi-threading behaviour of the application (since CUDA does not make any distinction between SIMD and threading); and (*ii*) those that affect the message-passing behaviour of the application, between nodes and over PCIe.

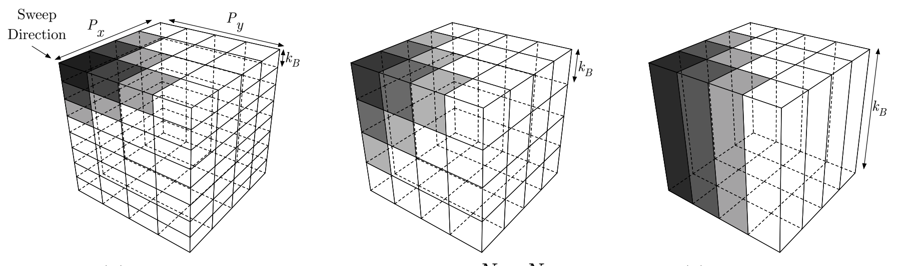
4.5.1 SIMD and Multi-threading

k-blocking

In the default version of LU's SSOR loop (Algorithm 4.1), each processor solves a "tile" of size $N_x \times N_y \times 1$ at each time step prior to communication. Our implementation (Algorithm 4.2) employs an optimisation commonly known as k-blocking – a name that arises from previous optimisation studies featuring the Sweep3D code [64] – which instead partitions the z axis into tiles of height k_B. k_B can be set to any value between 1 and N_z, but to simplify the discussion and explanation of k-blocking we assume that it divides N_z.

Algorithm 4.2 Pseudo-code for the SSOR loop with k-blocking.

```
 1: for $iter = 1$ to $max\_iter$ do
 2:
 3:     for $b = 1$ to $\frac{N_z}{k_B}$ do
 4:         for $k = (b-1) \times k_B$ to $b \times k_B$ do
 5:             call jacld($k$)            ▷ form lower triangular part of Jacobian matrix.
 6:         end for
 7:         call exchange_1($b$, recv)     ▷ receive data from north/west neighbours.
 8:         for $k = (b-1) \times k_B$ to $b \times k_B$ do
 9:             call blts($k$)             ▷ perform lower triangular solution.
10:         end for
11:         call exchange_1($b$, send)     ▷ send data to south/east neighbours.
12:     end for
13:
14:     ⋮                                  ▷ repeat for jacu and buts.
15:
16: end for
```



(a) $k_B = 1$ (b) $k_B = \min\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)$ (c) $k_B = N_z$

Figure 4.3: Comparison of three different k-blocking depths.

This optimisation was implemented in Sweep3D to make more effective use of network bandwidth (at the expense of delays to downstream processors) by aggregating k_B small messages into one larger message. We use k-blocking for a fundamentally different reason: maximising the amount of parallelism available in each hyperplane. The reader is reminded that a hyperplane is defined as $h = i + j + k$; for a tile with a fixed value of k, exploitable parallelism is restricted to the other two dimensions.

Figure 4.3 compares three potential k-blocking depths: (a) a k-block depth of 1, which minimises the amount of time any processor spends waiting on its first message and represents the behaviour found in the original benchmark; (b) a k-block depth of $\min(N_x/P_x, N_y/P_y)$, which provides an approximately cubic unit of computation and balances the need for a large k-block for compute efficiency with a small k-block for MPI efficiency; and (c) a k-block of depth N_z, which maximises the surface of the hyperplanes on each processor for as much of the run as possible. The current sweep-step is shown in light grey, downstream processors that are waiting for data are shown in white, and previous sweep steps are shown in progressively darker shades.

Which of these k_B values will be optimal on a given architecture is dependent upon the level of parallelism available (*i.e.* SIMD width, number of cores/threads) and, for multi-node runs, the behaviour of the network (*i.e.* its latency and bandwidth). For current-generation CPUs, we believe that the best configuration is $k_B = 1$, since the x and y dimensions are likely to provide sufficient parallelism for the relatively small SIMD widths of SSE and AVX. For GPUs, setting k_B as large as possible maximises SIMD efficiency, but causes too large a delay to downstream processors; we therefore choose to set $k_B = \min(N_x/P_x, N_y/P_y)$. It is important to note that this performance trade-off is only a concern for multi-node runs – since a single node run requires no MPI communication, we can set $k_B = N_z$.

Loop-Unrolling and Fusion

The pseudo-code in Algorithm 4.3 describes the original loop structure of the `blts` and `buts` methods. We also include the k loop and communication steps from Algorithm 4.1, to highlight the reasoning behind the original design of these loops – the first set of loops apply updates based on the results of the previous tile (and could in theory be run in parallel), while the second set of loops use a two-dimensional wavefront to compute the current tile.

Algorithm 4.3 Pseudo-code for the original `blts`.

```
for all $k$ do

    call exchange_1($k$, recv)

    for all $j$ do
        for all $i$ do
            for $m = 0$ to $4$ do
                call update ($k$, $j$, $i$, $m$) using ($k-1$, $j$, $i$, $m$)
            end for
        end for
    end for

    for all $j$ do
        for all $i$ do
            for $m = 0$ to $4$ do
                call update ($k$, $j$, $i$, $m$) using ($k$, $j-1$, $i$, $m$) and ($k$, $j$, $i-1$, $m$)
            end for
        end for
    end for

    call exchange_1($k$, send)

end for
```

We fuse these two sets of loops over j and i into a single set of loops, thus replicating the structure of the loop in Figure 4.1, which has several benefits: it enables a single kernel to carry out the updates in all three dimensions; it enables the GPU to hide memory latency more effectively, as fewer loads need to be completed before the solution can be updated in the j and i directions; and the number of registers required to hold intermediate values is decreased, which may increase occupancy (*i.e.* the ratio of active work-items to the maximum number of work-items supported on a single compute unit).

Thread Synchronisation

The lack of a method of global synchronisation across work-items makes it difficult to implement the hyperplane algorithm in a naïve fashion. We consider two alternative forms of work-item synchronisation, both making use of the "implicit CPU synchronisation" [143] that occurs between kernel invocations:

1. **Blocked Wavefront**
   In this first method, we decompose the total three-dimensional data grid into a number of smaller sub-grids of size $n_x \times n_y \times n_z$. Each sub-grid can then be solved (without violating the data dependency) by following a coarse wavefront sweep over sub-grids. Each of these coarse wavefront steps corresponds to a kernel invocation, where each sub-grid is solved by a work-group. The dependency is preserved within a work-group through the use of local synchronisation (*i.e.* `__syncthreads`).

2. **SIMD Hyperplane**
   The second method is more in keeping with Lamport's original hyperplane method, in that all of the grid-points lying on a particular hyperplane are solved in parallel in a SIMD fashion. We launch a separate kernel for each hyperplane: the first kernel solves a hyperplane consisting of only a single grid point, the next solves 3 grid points, followed by 6 grid points and so on. There is no need for local synchronisation within a kernel.

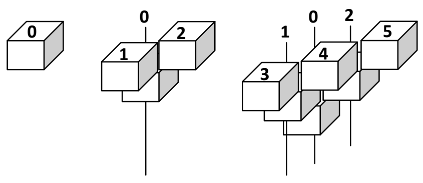


Figure 4.4: A mapping from a two-dimensional grid of work-items onto three-dimensional data.

Previous optimisation efforts for the Smith-Waterman algorithm by Aji *et al.* [9] (featuring a similar "tiled wavefront" optimisation, in two-dimensions) suggest that the blocked wavefront algorithm will be most performant, citing the cost of implicit CPU synchronisation. The authors introduce a novel method of global synchronisation within their kernel to alleviate this cost, and see a three-fold increase in performance. However, these results do not match our own – during our benchmarking (results not shown) the SIMD hyperplane method out-performed the blocked wavefront approach in all configurations tested.

Memory Access Pattern

For each grid-point, the `jacld` and `jacu` methods (the preconditioning steps of the wavefront sweep, shown in Algorithm 4.1) read in five solution values and five values from three neighbours (20 values in total). These data are used to populate four 5×5 matrices (100 values in total) per grid-point, which are later used by the `blts` and `buts` methods. In our optimised code, we move this calculation into the wavefront section; instead of loading 100 values per grid-point, we load 20 values and perform the `jacld` and `jacu` calculations inline. In addition to reducing the amount of memory required for each problem size, this optimisation decreases the number of memory accesses made by the `blts` and `buts` kernels, while also increasing their computational intensity.

We also ensure that each of our memory accesses is *coalesced*, with all work-items accessing contiguous memory locations. For the wavefront sections, we use a memory layout based on the mapping depicted in Figure 4.4, where each work-item is responsible for a column of grid-points (*i.e.* i and j are fixed, but

k is variable); for the other sections of the code, we use the original row-major form. We switch between layouts using a simple memory streaming kernel, which reads from one memory layout and writes to the other – this is more efficient than using a sub-optimal memory layout within the compute kernels, which access each grid-point multiple times. The lack of global synchronisation within kernels prevents this rearrangement from being performed in place, and we therefore make use of a separate rearrangement buffer on the GPU. The amount of memory required for this buffer is significantly less than the amount required to store temporary results prior to fusing the `jacld`/`jacu` and `blts`/`buts` kernels.

4.5.2 Message Passing

Problem Decomposition

Under the two-dimensional domain decomposition used in the original CPU implementation, if N_z increases then N_x/P_x and N_y/P_y must decrease in accordance with the memory limit of a node. This is significant because the size of a three-dimensional grid's largest hyperplane is bounded by the product of its two smallest dimensions – as N_x/P_x and N_y/P_y decrease, so too does the amount of available parallelism. A three-dimensional domain decomposition would enable us to deal with an increase in N_z by adding more processors, thus preventing a decrease in parallelism.

To investigate this possibility, we model a $960 \times 960 \times 960$ grid decomposed over 64 processors. Firstly, we determine the number of grid-points per processor: in a two-dimensional decomposition (*i.e.* an $8 \times 8 \times 1$ processor array), each processor is assigned a block of $120 \times 120 \times 960$ grid-points; in a three-dimensional decomposition (*i.e.* a $4 \times 4 \times 4$ processor array), each processor has a block of size $240 \times 240 \times 240$. Secondly, we note that the solution of a block of size $N_x \times N_y \times N_z$ requires $N_x + N_y + N_z - 2$ wavefront steps.

As a corollary, a processor array of size $P_x \times P_y \times P_z$ will have a compute time of:

$$\left(P_x + P_y + \left\lceil \frac{\lceil N_z/P_z \rceil}{k_B} \right\rceil \times P_z - 2\right) W \tag{4.1}$$

where W represents the time that a processor takes to compute a block of size $N_x/P_x \times N_y/P_y \times k_B$.

Thus an $8 \times 8 \times 1$ processor array, with $k_B = 120$, has a compute time of $22W_{120}$, whereas a $4 \times 4 \times 4$ processor array, with $k_B = 240$, has a compute time of $10W_{240}$. In order for the performance of the three-dimensional decomposition to match that of the two-dimensional decomposition, even when assuming zero communication cost, the value of W_{240} cannot be more than 2.2x greater than W_{120}.

This will obviously not be the case for a serial processor, since a 240^3 block contains 8 times as many grid-points as a 120^3 block – $W_{240} \approx 8 \times W_{120}$. The result is less obvious for a parallel processor, depending upon the amount of parallelism available; we find that, for our GPU implementation, the cost of processing a 240^3 block is approximately 6x greater than the cost of processing a 120^3 block, demonstrating that a 3D decomposition would not result in a performance gain.

4.6 Performance Results

4.6.1 Performance Breakdown

Table 4.2 presents a breakdown of LU's execution time, for a Class C problem in double precision, into seven components: the four methods constituting the wavefront sweeps (`jacld`, `blts`, `jacu`, and `buts`); the stencil operation (`rhs`); data rearrangement between sections (Rearrangement); and all other components of the simulation (Other). The reader is reminded that the CPU codes do not rearrange memory, since they are serial and do not use the hyperplane

Component	**X5550**		**C1060**	**C2050**
	Orig.	**Opt.**		
`jacld` %	18.02	26.68	18.57	19.54
`blts` %	17.40			
`jacu` %	16.82	26.68	18.57	19.54
`buts` %	17.36			
`rhs` %	27.84	43.92	43.37	43.17
Rearrangement %	—	—	12.18	11.16
Other %	2.57	3.48	6.77	8.48

Table 4.2: Performance breakdown for LU (Class C).

algorithm, and the optimised code combines the Jacobian pre-conditioning and triangular solver steps into one function (*i.e.* the cost of `jacld` is absorbed into `blts`, and that of `jacu` into `buts`).

In the original code, the wavefront section accounts for 70% of execution time, and the stencil section the majority of remaining time. In our optimised code, the wavefront section remains the most expensive component but accounts for significantly less time (50%) due to the removal of the separate Jacobian pre-conditioning steps. The `rhs` function benefits less from our optimisations, and therefore its contribution to execution time increases.

On the GPUs, the wavefront section and stencil operation account for a very similar fraction of execution time. Again, the wavefront kernels benefit most from our optimisations – since they are the focus of this study – whereas the stencil operation and other components are accelerated to a lesser degree, and are thus relatively more expensive. The acceleration of stencil kernels on GPUs has been addressed in other work [66, 117].

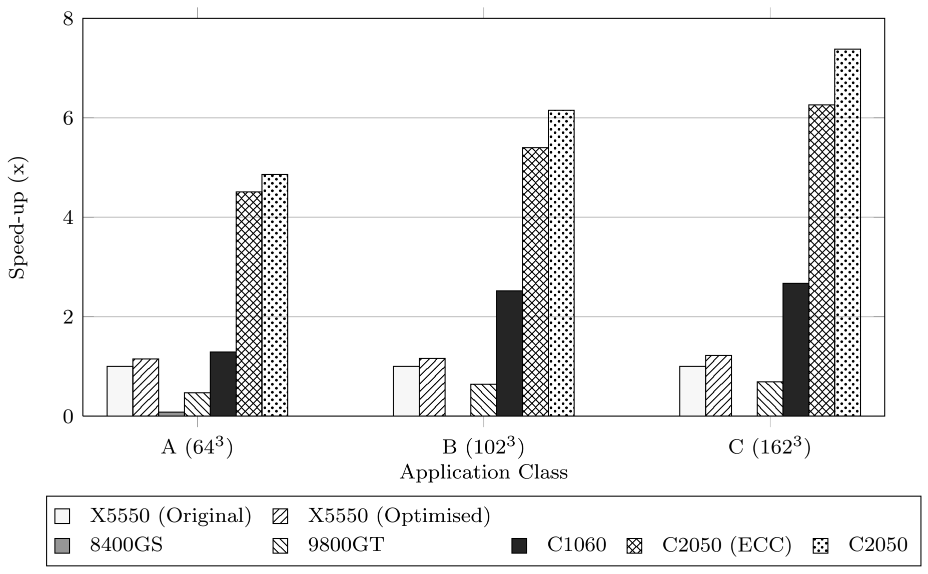
4.6.2 Architecture Comparison

The graphs in Figure 4.5 show the speed-up of our optimised implementations of LU, executing on an Intel X5550 processor and on a range of NVIDIA GPUs, compared to the original code executing on an Intel X5550 processor. We show all results on one graph to highlight: (*i*) the impact of our optimisations on CPU hardware; (*ii*) the difference between CPU and GPU performance for this application; and (*iii*) the performance impact of architectural changes between NVIDIA hardware generations.

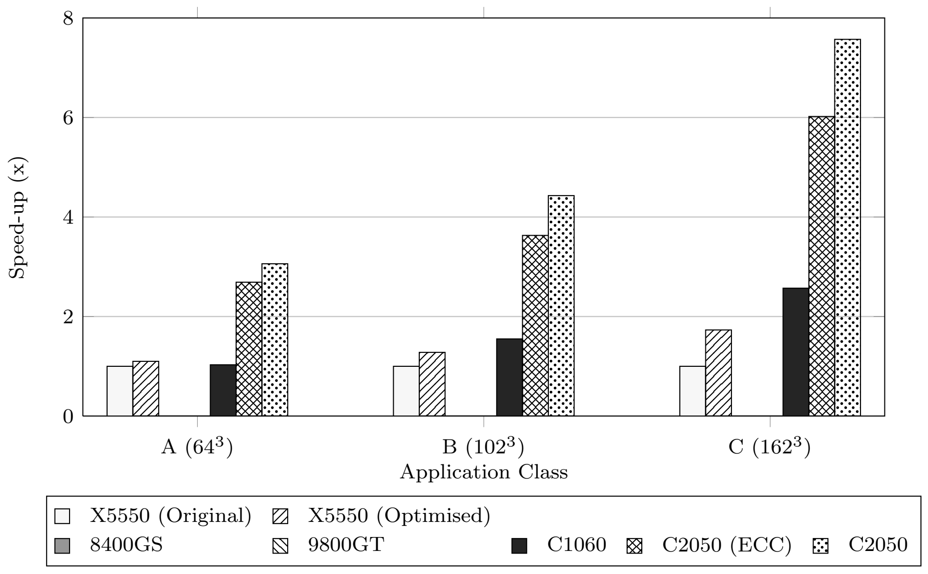
Where possible, we present results in single and double precision, for three of the seven application classes supported by LU: A (64^3), B (102^3) and C (162^3); some of the GPUs do not appear in all comparisons due to limited memory and/or hardware constraints. Specifically, the 8400GS only has enough memory to executing the Class A problem, and only the C1060 and C2050 support double precision floating-point arithmetic.

We see that the GPU solution comfortably outperforms both the original and optimised Fortran benchmarks, for all three problem classes, when run on HPC hardware. Unexpectedly, the GPU solution on such hardware appears to be memory bound: the performance hit suffered when moving from single to double precision is consistently around 2x, despite a 12x difference in theoretical peak for single and double precision on the C1060; and disabling Error Correcting Codes (ECC) on the C2050 increases performance for a Class C problem run in double precision by almost 15% (which is roughly in line with the expected change in bandwidth) [120].

We also see that the performance gap between the two architecture types increases with problem size. This is a direct result of the increase in the number of grid-points per hyperplane and thus the amount of exploitable parallelism. Due to its lower-clocked cores, the GPU takes longer to solve a small number of grid-points (in the worst case, the single grid-point at the beginning and end of each wavefront sweep) than the CPU – the speed-up we see is due to the GPU being faster at processing hyperplanes near the problem's centre.



(a) Single precision.



(b) Double precision.

Figure 4.5: Comparison of speed-up for our optimised implementation of LU running on different architectures.

Finally, we see that the performance gap between consecutive generations of NVIDIA hardware is greater than the increase in either peak GFLOP/s or peak global memory bandwidth (Table 3.2). Increased parallelism, relaxed coalescence criteria and the introduction of an L2 cache all increase *effective* bandwidth, and we believe that these hardware changes are responsible for the performance improvements that we see; the increased number of cores and threads enables more grid-points on the hyperplane to be worked on simultaneously, and the cache decreases access times for memory locations used by more than one work-item per hyperplane – the reader is reminded that, ignoring boundary conditions, the memory location for a grid-point (i, j, k) is accessed by the work-items assigned to $(i + 1, j, k)$, $(i, j + 1, k)$ and $(i, j, k + 1)$.

4.7 Summary

In this chapter, we present optimised implementations of the NAS LU benchmark for CPUs and GPUs. Benchmark results are provided for a wide range of GPU hardware, including consumer cards and NVIDIA's flagship HPC Tesla and Fermi processors. We show that the same set of optimisations can benefit wavefront applications running on both types of architecture, improving the performance of the original Fortran benchmark by up to 1.7x, and demonstrate the utility of combining (or "fusing" [164]) kernels to reduce an application's memory footprint and remove the cost of accessing temporary global arrays.

Our GPU implementation executing on an NVIDIA Tesla C2050 is up to 7x faster than an optimised Fortran implementation executing on an Intel X5550, demonstrating (in contrast to previous work on two-dimensional wavefronts) that Lamport's hyperplane algorithm *can* be ported effectively to new accelerator-based architectures – a compelling argument in favour of the use of GPUs for scientific three-dimensional wavefront codes in single workstation environments.

As expected, the performance improvement afforded by the use of accelerators for this class of application is highly dependent upon the amount of exploitable parallelism available. Many MPI-based three-dimensional wavefront codes treat the problem grid as a stack of "tiles" of size $N_x/P_x \times N_y/P_y \times 1$, artificially limiting the amount of such parallelism to two dimensions; the results presented here demonstrate the importance of using k-blocking to regain this lost parallelism. For a Class C problem, the performance difference between application runs using $k_B = 1$ and $k_B = 162$ is $\approx$ 40x (results not shown), and we expect that the importance of k-blocking will grow as the amount of parallelism supported by hardware increases. It remains to be seen how the trade-off of k-blocking depth against delay to downstream processors will impact upon the code's performance at scale – we examine this further in Chapter 7.

CHAPTER 5
Optimisation of Molecular Dynamics Applications

Molecular dynamics simulations can involve millions of atoms and, for a given timestep, the forces acting between all pairs of atoms must be calculated. In practice, the calculation of forces is split into two parts: *short-range* forces, which tend to zero within a finite distance; and *long-range* forces, which do not. The short-range force calculation accounts for the majority of execution time, and is the focus of the research in this chapter.

During the calculation of the short-range forces, it is safe to assume that the force between atoms separated by more than some "cut-off" distance (R_c) is negligible. Therefore, it is not necessary to consider all atom-pairs at each timestep – an approximate but sufficient answer can be reached by evaluating only the forces between an atom and its near neighbours. For a simulation with an average of k neighbours per atom, this reduces the complexity of force calculation from $O(N^2)$ to $O(Nk)$.

"Cell-based" simulations determine the set of neighbouring atoms by dividing the problem domain into "cells", evaluating the forces between an atom and the contents of some set of surrounding cells (*e.g.* for cells of size R_c, there are 27 cells that could potentially contain atoms closer than R_c). In such simulations, with a fixed atom density (ρ), $k = \rho \times 27{R_c}^3$. An alternative method is to make use of a pre-computed Verlet list ("neighbour list") [156] for each atom, which contains the indices of all atoms separated by less than $R_c + R_s$ (Figure 5.1). R_s is a "skin distance" that allows a neighbour list to be re-used for several iterations; it must be carefully chosen, based on other simulation parameters, such that no atom can move more than R_s between neighbour list rebuilds. This is the method we make use of in this work, since a neighbour list allows

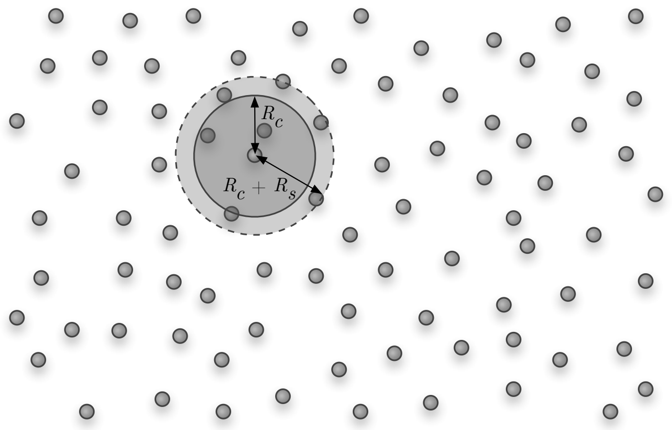


Figure 5.1: An atom's neighbourhood.

for fewer atom-pairs to be evaluated each timestep: $k = \rho \times \frac{4}{3}\pi(R_c + R_s)^3$.

The number of distance calculations can be reduced further by utilising Newton's third law (N3) – the force that atom i exerts on atom j ($F_{i,j}$) is equal in magnitude, but opposite in direction, to the force that atom j exerts on atom i, and therefore only needs to be calculated once. A given atom pair (i, j) thus appears only in the neighbour list for i or j, and the computed force is applied to both atoms.

Molecular dynamics is an area of HPC that has seen significant application speed-ups reported from the use of accelerators [11, 24, 27, 59, 122, 136, 154, 155], owing to the large amount of exploitable parallelism present in force calculation. The work described in this chapter investigates the optimisation of this class of application for CPUs, and also presents the first port (to our knowledge) of a molecular dynamics code to the Intel MIC architecture. The benchmark that we use (miniMD [61, 62]) is a simplified version of Sandia's LAMMPS [137, 139] package, intended for use in optimisation studies such as this one – despite supporting only the Lennard-Jones (LJ) inter-atomic potential, miniMD therefore has performance and scaling behaviours that are representative of a much larger and more complex code.

5.1 Benchmark Description

Unlike LAMMPS, miniMD supports only one simulation, in which atoms are spaced uniformly across a three-dimensional lattice, with random initial velocities that are scaled in line with the desired temperature. The problem is *spatially decomposed* across processors (*i.e.* each MPI task is responsible for all atoms falling within some subvolume of space) and the main simulation loop is repeated for a user-specified number of timesteps.

Each iteration of the loop (which is depicted as pseudo-code in Algorithm 5.1) begins by updating atom positions based upon their current velocities. If the neighbour list is deemed to be out of date (based on the number of iterations since the last rebuild), all processors: check to see if any of the atoms in their subvolume should be moved to another processor (atom *exchange*); construct lists of atoms that need to be ghosted on neighbouring processors (*i.e.* atoms near the *borders* of the subvolume); import/export atom positions based upon this list; and then rebuild the neighbour list. If the neighbour list is not out of date, then all processors import/export atom positions based upon the most up-to-date list of border atoms. Following these communication steps, each processor calculates the short-range forces exerted upon other atoms by those in its subvolume; since such forces may affect atoms on other processors, a second communication step (*reverse* communication, or "rcomm") is necessary. Finally, the current atom velocities are stored (to compute temperature, in user-specified iterations) before being updated based upon their acceleration. The 48 on Line 22 is a hard-coded multiplier used by miniMD to convert force to acceleration.

For the default values of R_c and R_s provided by miniMD (2.5 and 0.3, respectively), execution can be broken down as follows: force calculation is the most expensive component, responsible for more than 80% of execution time; the neighbour list build accounts for 10%; and the remaining time is split between inter-node communication and time integration (to update atom velocities and positions).

Algorithm 5.1 Pseudo-code for miniMD's main simulation loop.

```
for t = 1 to timesteps do

    for all atoms do
        position ← position + dt × velocity
    end for

    if neighbour list is out of date then
        call exchange()                    ▷ move atoms between processors
        call borders()                     ▷ update import/export lists
        call comm()                        ▷ import/export atom positions
        call neighbour_build()             ▷ rebuild neighbour lists
    else
        call comm()                        ▷ import/export atom positions
    end if

    call force_update()                    ▷ evaluate short-range forces

    call rcomm()                           ▷ import/export atom forces

    for all atoms do
        old velocity ← velocity
        velocity ← velocity + 48dt × force
    end for

end for
```

5.2 Related Work

The most similar research to that presented in this thesis are two recent attempts to optimise LAMMPS for execution on NVIDIA GPUs: LAMMPS$_{\text{GPU}}$ [24] and LAMMPS$_{\text{CUDA}}$ [154, 155]. Both of these studies ultimately use single precision floating-point when quoting their best speed-up figures, but it is not clear whether the CPU code used as a baseline was run in single or double precision. In [62], the authors allude to a performance study of miniMD in single precision and note that there was "no appreciable performance enhancement". We verify that there is no significant performance benefit from using single precision in scalar code, since the application is not memory bound. However, the results in this chapter show that the use of single precision in SIMD can lead to significant performance improvements.

Other research has focused on improving the algorithmic complexity of molecular dynamics. One common aim is to reduce the number of distance comparisons made during the computation of short-range forces; extensions to both Verlet's original method [156] of maintaining a list of interacting atom pairs [50, 60, 96] and the so-called "link-cell" method [49, 95, 165] have been proposed, along with new approaches [13, 43, 94, 133] that make use of domain-specific knowledge to improve the search for near-neighbours. Improvements to communication complexity have also been investigated [148].

More hardware-focused optimisations have been considered, including: the potential of scheduling molecular dynamics across heterogeneous systems featuring some mix of CPU and GPU cores [24, 58]; sorting atoms according to their position in space, to improve cache behaviour [11, 101]; using single and "mixed" floating-point precision, to avoid expensive double precision operations [24, 88]; and the use of hardware purpose-built for molecular dynamics simulations [41, 115].

5.3 Optimisation Challenges

The use of a neighbour list per atom makes it difficult to utilise SIMD execution units effectively. Consider the case of W SIMD execution units, computing the force between W atoms and their neighbours: the positions of the W neighbours are unlikely to be stored contiguously in memory, thus requiring gather and scatter operations. Such operations are costly on modern SIMD architectures, both in terms of instruction overheads and memory accesses. These problems affect both the short-range force calculation (which reads from the neighbour list) and the building of the neighbour list itself.

Although each of the steps in a molecular dynamics simulation exhibit significant amounts of parallelism, there is a strict dependency between them: atom positions must be updated before forces can be calculated, and atom positions are dependent upon the forces calculated in the previous timestep. In a multi-node simulation, both positions and forces are required from neighbouring processors; this combination of dependency between timesteps and frequent network communication make the efficient use of accelerator architectures challenging, due to the latency and bandwidth of the PCIe bus.

5.4 Experimental Setup

For maximum performance on any modern architecture, it is important to make use of single precision floating-point wherever possible. The SIMD units on Intel hardware are 2x wider for single precision than double precision, and performance is typically ≈2x higher as a result; we use single precision here to maximise performance, and to demonstrate that our algorithms can scale to wider SIMD than is currently available for double precision. Early results from our double and mixed precision implementations suggest that the performance impact is what one would expect and is similar to that reported for GPU codes (*i.e.* that the double precision code is twice as slow, and the mixed precision code somewhere between).

Although the use of "fast math" flags and instructions could have provided an additional boost to performance, their cumulative effect on long-running simulations should ideally be examined by domain experts. Therefore, any performance numbers that we report are from using exact, albeit single precision, floating-point math and are on average 10–15% worse than when approximate reciprocals are employed.

To ensure that we started from a strong baseline implementation, we applied some previously proposed optimisations to miniMD before beginning our SIMD analysis. A number of these optimisations are already present in some form within LAMMPS: the aggregation of an atom's forces takes place in registers; atoms are sorted according to their position in space; and alignment is ensured via the insertion of padding into the AoS layout used to store atom data. We also improve the neighbour list build algorithm using optimisations proposed in [60], which are not currently present in LAMMPS.

	Xeon E5-2660	**Xeon Phi 5110P**[a]
Sockets×Cores×Threads	$2 \times 8 \times 2$	$1 \times 60 \times 4$
Clock (GHz)	2.2	1.053
Single Precision GFLOP/s	563	2022
L1 / L2 / L3 Cache (KB)	32 / 256 / 20,480	32 / 512 / -
DRAM	128 GB	8 GB GDDR
Bandwidth from STREAM [97]	76 GB/s	170 GB/s

PCIe Bandwidth	10 GB/s
Compiler Version	Intel v13.0.030
Compiler Flags	`-O3 -xHost -restrict -ipo -fno-alias`
MPI Version	Intel v4.0.3

Table 5.1: System configuration for the molecular dynamics optimisation study.

[a]Experimental results were recorded from evaluation silicon, with slight differences from the listed specification.

The system configuration for the server used in our experiments is given in Table 5.1. We use a Knights Corner (KNC) Intel Xeon Phi co-processor, which has 60 x86 cores and hyper-threading support for four hardware threads per core. The CPU and KNC binaries were compiled with the same compiler, and in all experiments (except where noted) we use all of the available cores on both architectures, running the maximum number of hyper-threads supported (two per CPU core and four per KNC core). On the CPU, we use AVX for both our 128- and 256-bit SIMD experiments, to better isolate the effects of SIMD width – any performance difference between 256-bit AVX and SSE will arise from a combination of increased SIMD width and reduced register pressure due to the three-operand instructions introduced by AVX.

To demonstrate the performance and scalability of our optimised code, we present results for simulations with multiple atom counts and two different cut-off distances (2.5 and 5.0). The first of these is the standard cut-off distance used in miniMD's LJ benchmark, whereas the second is used to investigate the effects of inter-atomic potentials with larger cut-off distances. This approach matches that of [24], and the number of neighbours per atom under these conditions is similar to that of the LAMMPS Rhodopsin protein benchmark. All experiments use cross-neighbour SIMD, since both cut-off distances provide sufficient parallelism.

All other simulation parameters are the defaults provided by miniMD: $\rho = 0.8442$, $T = 1.44$, $N_{rebuild} = 20$, $R_s = 0.3$, timesteps = 100. We report performance in atom-steps per second (*i.e.* # atoms×timesteps/execution time), to enable direct comparison between our results and those of prior work [154, 155], and report execution times in seconds.

5.5 Optimisations

The focus of this study is the optimisation of molecular dynamics applications for x86 architectures. Although some of the optimisations we propose are also beneficial to GPU architectures, we discuss them mainly in the context of Intel's CPU and MIC hardware. For this reason, we divide the optimisations into two types: (*i*) those that affect the SIMD behaviour of the application; and (*ii*) those that affect the threading and message-passing behaviour of the application. This differs from the division found in the previous chapter, since x86 architectures handle SIMD and threading very differently from GPUs.

5.5.1 SIMD

Short-Range Force Calculation

miniMD's short-range force compute function is a simple loop-nest, iterating over atoms and their neighbours (Algorithm 5.2). The loop computes the interatomic distance between an atom i and each of its neighbours j, and updates the forces of both i and j if the distance is less than R_c. The calculation of this force is not arithmetic-intensive, requiring only 23 floating-point operations. The reader's attention is drawn to the fact that the force between two atoms is computed in such a way as to avoid expensive square root calculations.

Kim *et al.* demonstrate that the force compute loop from GROMACS can be auto-vectorised [80]; similarly, we find that Intel's C++ compiler is able to auto-vectorise miniMD's loop with a little assistance. In particular, we must add a compiler directive (`#pragma ivdep`) to the loop over neighbours to resolve the

Algorithm 5.2 Pseudo-code for short-range force calculation.

```
 1: for all atoms $i$ do
 2:     for all neighbours $k$ do
 3:         j = neighbour_list[k]
 4:         delx = xi - pos[j+0]
 5:         dely = yi - pos[j+1]
 6:         delz = zi - pos[j+2]
 7:         rsq = (delx × delx) + (dely × dely) + (delz × delz)
 8:         if (rsq $\leq R_c$) then
 9:             sr2 = 1.0 / rsq
10:             sr6 = sr2 × sr2 × sr2
11:             f = sr6 × (sr6 - 0.5) × sr2
12:             fxi += f × delx
13:             fyi += f × dely
14:             fzi += f × delz
15:             force[j+0] -= f × delx
16:             force[j+1] -= f × dely
17:             force[j+2] -= f × delz
18:         end if
19:     end for
20: end for
```

possible dependence in force array updates, since the compiler does not know that each of an atom's neighbours is unique. The auto-vectorised code can be made more efficient by moving the force updates outside of the branch, so that the compiler knows that the memory accesses involved are safe for iterations that fail the if-check. We also pad the number of neighbours to the nearest multiple of the SIMD width (W) using "dummy" neighbours – atoms placed at infinity that always fail the cut-off check – to handle situations where the number of neighbours is not divisible by W.

After auto-vectorisation, each of the arithmetic operations on Lines 4–17 operates at 100% SIMD efficiency. However, the branch on Line 8 and the memory accesses on Lines 4–6 and 15–17 may introduce significant inefficiency. The branch is handled via blending/masking, so Lines 9–11 are executed even for neighbours that fail the cut-off check. The amount of inefficiency this causes depends upon the skin distance R_s. More fundamental to this loop, the memory accesses on Lines 4–6 (neighbour positions) and 15–17 (neighbour forces) are to potentially non-contiguous memory locations and thus require gather and scatter operations.

	Scalar L/S	**Scalar G/S**	**Vector L/S + Shuffles**	**Vector L/S + Dot Product**
Load Neighbour IDs	W	W	W	W
Gather j Positions	$3W$	$48W$	$7 + W$	W
Compute delx, dely, delz	3	3	3	4
Compute rsq	5	5	5	7
Compare rsq to R_c	2	2	2	2
Compute f × del*	9	9	9	14
Update i Forces	3	3	3	4
Gather/Scatter j Forces	$6W$	$96W$	$16 + 2W$	$2W$
Update j Forces	3	3	3	4
Total Instructions	$25 + 10W$	$25 + 145W$	$25 + 4W$	$35 + 4W$

Table 5.2: Comparison of theoretical worst-case instruction counts for four different force compute gather-scatter approaches.

The auto-vectorised code is 1.7x faster than scalar code on the CPU, and 1.3x faster on KNC. As discussed, we expect to see smaller speed-ups than W due to SIMD inefficiencies, but these results show that the inefficiency is quite high. This is primarily due to the gather and scatter operations, which implicitly transpose between AoS and SoA memory layouts. The overhead of these gather and scatter operations lies not in the cost of memory access (as might be expected) but in the number of instructions required by the transpose. If the gathers and scatters were as cheap as vector loads and stores, then we would see a significant speed-up – 7.04x (out of a maximum 8x) on the CPU using 256-bit AVX. The remaining inefficiency is relatively small and, since it comes from the branch, is a trade-off with neighbour list build cost.

To highlight the instruction overhead of gathers and scatters, we now consider four alternative hand-vectorised gather-scatter implementations: using scalar loads and stores (Scalar L/S) to populate SIMD registers (*i.e.* mimicking the compiler's auto-vectorisation); using the dedicated gather and scatter instructions (Scalar G/S) on KNC; and two approaches that take advantage of atom data being stored in AoS format, by loading/storing entire atoms using 128-bit instructions (Vector L/S). Table 5.2 lists the theoretical worst-case number of instructions required by each approach (for SIMD width W, and discounting instructions introduced due to hardware constraints).

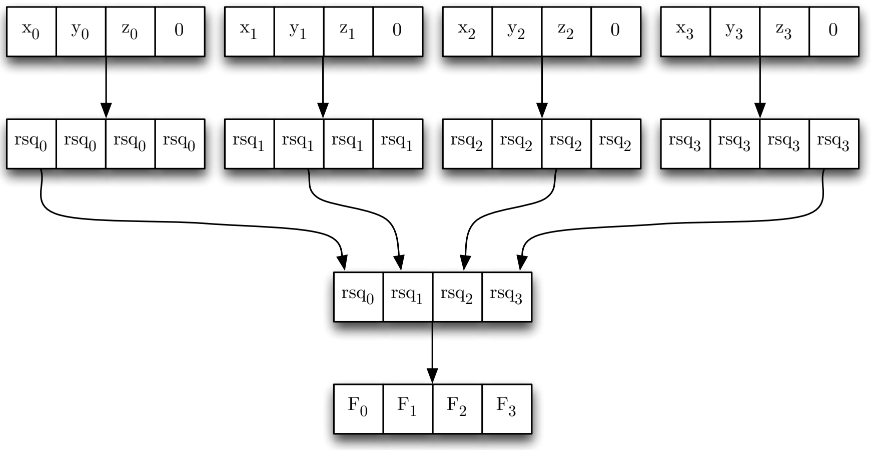

x_0 y_0 z_0 0
x_1 y_1 z_1 0
x_2 y_2 z_2 0
x_3 y_3 z_3 0
rsq_0 rsq_0 rsq_0 rsq_0
rsq_1 rsq_1 rsq_1 rsq_1
rsq_2 rsq_2 rsq_2 rsq_2
rsq_3 rsq_3 rsq_3 rsq_3
rsq_0 rsq_1 rsq_2 rsq_3
F_0 F_1 F_2 F_3

Figure 5.2: Combining a dot-product and AoS-to-SoA transpose in 128-bit SIMD.

For Scalar L/S, each scalar memory access requires an instruction, as does each insertion/extraction to/from a SIMD register. KNC's dedicated gather and scatter instructions do not help us in the worst case – we must execute the gather instruction once for each cache line touched, thus incurring some loop overhead (up to 16 iterations, for each gather of x, y or z). The Vector L/S approach only requires W 128-bit loads/stores, and we can replace the scalar insertion/extraction code with an efficient in-register transpose that uses shuffle instructions. Alternatively, we can combine this transpose with the calculation of rsq, as shown in Figure 5.2 (Vector L/S + Dot Product), decreasing the number of instructions required for gathers/scatters at the expense of extra compute. Which of these two transpose approaches will be faster depends upon the target architecture; some architectures may lack support for dot-products or fast horizontal adds entirely, or feature shuffle instructions that are significantly cheaper than floating-point arithmetic.

Table 5.3 compares the number of clock cycles per neighbour (and speed-up over a scalar implementation on the same hardware) for the four alternative gather-scatter approaches, and Table 5.4 presents a breakdown of the number of static instructions in the inner-most loop for our best approach. We now

Approach	**CPU**				**KNC**	
	128-bit SIMD		**256-bit SIMD**		**512-bit SIMD**	
Scalar L/S	12.97	(2.02x)	11.48	(2.28x)	23.75	(2.59x)
Scalar G/S	—		—		15.82	(3.89x)
Vector L/S + Shuffles	10.89	(2.40x)	8.02	(3.26x)	12.94	(4.75x)
Vector L/S + Dot Product	10.34	(2.53x)	7.64	(3.43x)	11.78	(5.22x)

Table 5.3: Clock-cycles per neighbour and speed-up versus scalar for force compute gather-scatter approaches.

	Scalar	**128-bit**	**256-bit**	**512-bit**
# Neighbours/Iteration	1	4	8	16
Load Neighbour IDs	1	4	8	16
Gather j Positions	0	4	8	16
Compute delx, dely, delz	3	4	4	4
Compute rsq	5	7	7	16
Compare rsq to R_c	2	2	2	1
Compute f $\times$ del*	9	14	14	7
Update i Forces	3	4	4	4
Gather/Scatter j Forces	6	8	16	32
Update j Forces	3	4	4	4
Other Instructions	6	3	3	52
Total Instructions/Neighbour	38.0	13.5	8.75	9.5

Table 5.4: Static instructions for force compute.

list instructions introduced by the compiler due to hardware constraints (*e.g.* a finite number of registers); these instructions are listed as *Other Instructions*. The reader's attention is also drawn to the fact that KNC's gather/scatter instructions do improve performance in practice – due to our sorting of atom positions, we are more likely to load several atoms from one cache line than to touch 16 distinct cache lines.

In general, arithmetic-dominated operations scale well, and the number of instructions is comparable across SIMD widths. Even following our optimisations, the number of instructions for gathers and scatters scales poorly with SIMD width; ignoring "Other Instructions", gathers and scatters account for 31%, 48% and 64% of the remaining instructions for 128-, 256- and 512-bit SIMD respectively. KNC has a high number of "Other Instructions" due primarily to register pressure on the general-purpose and mask registers; this may not manifest for another instruction set with the same or higher SIMD width.

To estimate the performance loss (in cycles) due to the high instruction overhead for gathers and scatters, we use 256-bit AVX to evaluate the performance of two "ideal" cases, where all of an atom's neighbours are contiguous in memory. With data still stored in AoS format, and thus still needing transposition, performance improves by 1.4x; with data stored in SoA, performance improves by 2x.

Neighbour List Build

miniMD uses a "link-cell" approach to reduce the size of the set of potential neighbours examined during the neighbour list build. First, atoms are placed into subvolumes of space called "bins", using a spatial hash. Then, the set of potential neighbours for each atom is defined as those atoms that fall into a "stencil" of surrounding bins pre-computed at the start of the simulation. The majority of the neighbour list build's execution time is spent in a loop (Algorithm 5.3) that runs after this binning process. For each atom, this loop iterates through the set of potential neighbours, storing in the neighbour list those which are closer than $R_c + R_s$.

Algorithm 5.3 Pseudo-code for the neighbour list build.

```
1: for all atoms $i$ do
2:     numneigh = 0
3:     for all potential neighbours $k$ do
4:         j = potential_neighbour[k]
5:         delx = xi - pos[j+0]
6:         dely = yi - pos[j+1]
7:         delz = zi - pos[j+2]
8:         rsq = (delx $\times$ delx) + (dely $\times$ dely) + (delz $\times$ delz)
9:         if (rsq $\leq R_c + R_s$) then
10:            neighbour[numneigh] = j
11:            numneigh++
12:        end if
13:    end for
14: end for
```

As before, our vectorisation targets the inner-most loop over neighbours. The core behaviour of this loop is very similar to that of the force compute – it computes the distance between two atoms, and compares that distance to some

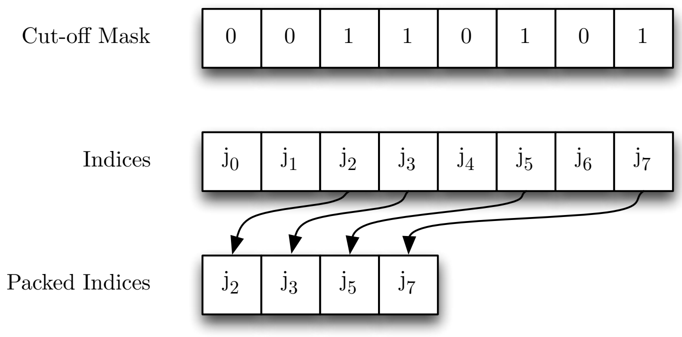


Figure 5.3: Using a packed store to append to a neighbour list.

cut-off. However, the loop does not auto-vectorise due to a loop dependence on Lines 10 and 11; the memory location to which each neighbour index should be written depends upon the number of previous neighbours that pass the cut-off check. An efficient way to vectorise appending to a list in this manner is to use a *packed store*, the basic operation of which is demonstrated in Figure 5.3. For a SIMD register packed with rsq values, the result of a comparison with $R_c + R_s$ is a W-bit mask, and a packed store writes a subset of indices (from another SIMD register) to contiguous memory based upon this mask. KNC's instruction set includes a packed store instruction, which we can emulate on other hardware; for both 128-bit and 256-bit AVX, we achieve it with a mask look-up, a single shuffle and a vector store. We determine the number of neighbours appended to the list by counting the number of bits set in the comparison mask.

For the force compute, each atom gathers a distinct set of neighbours, and thus there is no opportunity to re-use any data transposed during the gather. This is not true of the neighbour list build; surrounding the outer loop over atoms with a new loop over bins enables us to gather (and transpose) the set of potential neighbours once and then re-use it for several atoms. For the architectures considered here, this is beneficial for two reasons: first, the cost of the AoS-to-SoA transpose is amortised over several atoms; and second, the transposed set of neighbours exhibits better cache behaviour. We believe our approach to be applicable to GPU architectures also, since the transposed set

	Scalar	**128-bit**	**256-bit**	**512-bit**
# Neighbours/Iteration	1	4	8	16
Load Positions & Compute rsq	8	8	8	6
Compare rsq to $R_c + R_s$	2	1	1	1
Load Neighbour IDs	1	1	2	1
Append to Neighbour List	2	8	17	5
Other Instructions	3	5	5	15
Total Instructions/Neighbour	16.00	5.75	4.13	1.75

Table 5.5: Static instructions for neighbour list build.

could be stored in shared local memory.

The choice of bin size is an important trade-off: with large bins, the gathered SoA stencil receives more re-use but will contain more atoms; with small bins, the SoA stencil receives less re-use but also contains fewer atoms. The best choice depends on whether the cost of gathering atoms is more than that of extra distance calculations – the CPU favours smaller bins, whereas KNC favours larger. Besides this simple parameter change, the algorithm is the same across both architectures.

Table 5.5 presents a breakdown of the number of static instructions in the inner-most loop for our optimised approach. As before, "Other Instructions" accounts for those introduced by the compiler due to hardware constraints. Since the data transpose happens outside of the key loop, the number of instructions to load positions and compute rsq remains constant across SIMD widths, except for 512-bit SIMD on KNC; KNC has fewer instructions here because it has fused multiply-add instructions, which eliminates two arithmetic instructions. As before, KNC has a higher number of "Other Instructions" per neighbour, but these are mostly software prefetches and mask manipulations (to handle iterations with fewer than 16 neighbours).

The number of instructions required to append to the neighbour list is the least consistent across architectures. Due to the lack of 256-bit integer support in AVX, our implementation uses 128-bit stores, and thus this operation does not scale with SIMD width. In contrast, KNC's cost for this operation is very low, due to its packed store instruction.

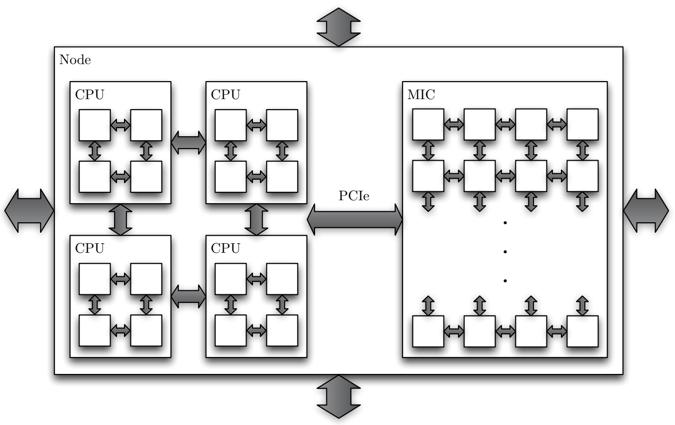


Figure 5.4: The hardware/class hierarchy.

5.5.2 Multi-threading and Message Passing

Problem Decomposition

We augment the MPI decomposition found in the original miniMD with a hierarchy of subdomains, as shown in Figure 5.4. At the first level, we divide the problem domain amongst nodes, and each node runs a single MPI task. We then further subdivide a node's subdomain amongst sockets (where a socket is either a CPU socket or a KNC socket/card), and finally we split each socket's subdomain amongst some number of threads. We specify the fraction of a node's subvolume assigned to the KNC hardware at run-time.

The use of such a hierarchy allows for communication between subdomains to be specialised: threads running on the same socket can communicate directly through shared memory; threads running on different sockets can communicate either through shared memory or over PCIe; and all threads can pack their off-node communication into a single MPI message rather than competing for the network interface. Using a spatial decomposition at each level allows us to use ghost atoms to handle the update-conflicts between threads, and helps to reduce the size of messages sent between the CPU and KNC.

This arrangement of subdomains is represented in code as a hierarchy of abstract C++ classes: an `MPI_Domain`, a `Socket_Domain` and a `Thread_Domain`.

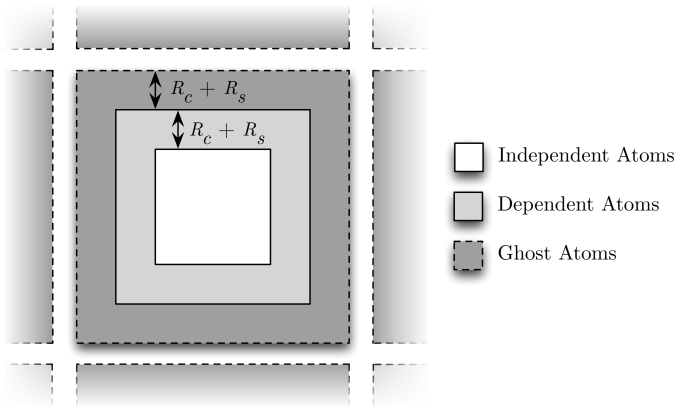


Figure 5.5: A subdomain split into dependent and independent volumes.

These abstract classes contain all of the common functionality across CPU and KNC hardware, and make up the bulk of the code. Where different code is required (*e.g.* for performance reasons, or because of differing communication mechanisms), this is implemented in a subclass. This minimises the code that must be re-written for optimisation on a particular architecture.

Asynchronous Communication

A possible bottleneck for KNC performance is the latency and bandwidth of the PCIe bus. To minimise the amount of PCIe communication, we adopt the same communication mechanism as [24] and opt not to use N3 between different sockets. Although this results in a small amount of redundant computation (for those atom-pairs that cross socket boundaries), it reduces the amount of PCIe communication by a factor of two since we can skip sending force contributions back to the "owner" socket of each atom.

We further optimise PCIe communication by overlapping it with useful work. Our decision to not use N3 across sockets means that we need only hide a single exchange of messages between the CPU and KNC for each iteration of the simulation loop (sending position updates for atoms used by neighbour sockets) unless we need to rebuild the neighbour lists. To facilitate this, we divide atoms into the three types shown in Figure 5.5: those that interact only with other atoms in the same subdomain (*independent* atoms); those that potentially interact with atoms in another subdomain (*dependent* atoms); and those that are copies of atoms in another subdomain (*ghost* atoms). We can compute the forces for all atom-pairs not featuring ghost atoms without any cross-domain communication; therefore, we overlap PCIe communication with this computation, and hide it almost completely.

This optimisation could also be applied at the MPI level, but would require a significant re-write of miniMD's communication routines. The current communication scheme places an ordering on communication between processors, such that only one communication step is required in each direction – for example, a processor receiving an atom's position from a neighbour in the x direction may forward the information to a neighbour in the y direction. Although this reduces the number of messages that are sent, it also complicates the use of MPI's asynchronous communication routines.

5.6 Performance Results

5.6.1 Performance Breakdown

Table 5.6 gives a breakdown of a 2.048M atom simulation into four components: the calculation of short-range forces (Force); building the neighbour lists (Neigh); communication (Comm), which includes force and position communication, as well as the exchanging of atoms across subvolume boundaries; and any remaining time (Other), which comprises the integration steps for computing updated velocities and positions. The CPU+KNC breakdown is given from the perspective of the CPU and KNC in separate columns.

For our versions of miniMD, the force compute remains the largest component of execution time for both cut-offs. However, as the component that is most accelerated by our use of SIMD, it takes a smaller fraction of time. For the AVX implementation, we see speed-ups of 4.0x and 4.9x for the cut-offs of 2.5 and 5.0 respectively, and KNC provides an additional speed-up of 1.4x in both cases.

Component	**Orig.**	**CPU (AVX)**	**KNC**	**CPU+KNC (CPU)**	**CPU+KNC (KNC)**
Cut-off of 2.5					
Force %	82.0	63.9	63.2	55.8	62.4
Neigh %	9.0	13.5	12.6	10.7	12.1
Comm %	4.9	9.5	15.4	26.5	17.7
Other %	4.0	13.1	8.7	7.1	7.8
Cut-off of 5.0					
Force %	90.3	86.0	85.4	78.9	80.5
Neigh %	6.7	6.9	6.4	6.2	6.7
Comm %	1.7	4.1	6.3	13.3	11.0
Other %	0.4	3.0	2.0	1.7	1.9

Table 5.6: Performance breakdown for miniMD (2.048M atoms).

We see a greater speed-up for the larger cut-off for two reasons: firstly, the time spent in force compute is dependent upon the number of inter-atomic distances that must be evaluated, which grows with the cube of the cut-off; and secondly, due to our use of SIMD across neighbours, SIMD efficiency is improved.

Our SIMD acceleration of the neighbour list construction improves its performance considerably; thus, its contribution to execution time remains relatively constant. For the AVX implementation, we see speed-ups of 2.1x and 4.6x for the cut-offs of 2.5 and 5.0 respectively, and KNC provides an additional speed-up of 1.5x in both cases. One might expect this component of the simulation to become relatively more expensive for larger cut-offs (as with the force compute), since it also depends upon the number of atom-pairs. However, although the distance computation costs scale similarly, a larger cut-off results in more atoms per bin and therefore significantly lowers looping overheads.

KNC uses significantly more threads than the CPU, and thus spends a larger fraction of time in inter-thread communication. Further, it spends time in PCIe communication. Although the cost of exchanging updated atom positions every iteration is mostly hidden, it remains exposed when moving atoms between nodes and for the first exchange of position information after a neighbour list build. We note that since our experiments here are on a single node, we could have avoided all PCIe communication for KNC-only runs. However, our goal was to represent multi-node execution faithfully, and thus all data that would be sent over MPI in a multi-node run is sent to the CPU over PCIe and handled appropriately. These factors lead to KNC spending 19% and 7% more time in communication than the AVX implementation on the CPU. For CPU+KNC, we see that the CPU spends a much larger fraction of its time in communication than when running without KNC; this increase is primarily due to time spent handling PCIe communication not present in CPU-only runs.

The fraction of time spent in Other is larger in our versions of miniMD than in the original, since it benefits least from the use of SIMD and threading. The position/velocity updates (Lines 3–5 and 18–21, in Algorithm 5.1) scale very poorly due to limited memory bandwidth – these operations involve very little computation per atom, and require streaming through multiple large arrays that do not fit in the on-die caches (*e.g.* for 2.048M atoms, the velocity update touches 48 bytes per atom – a total of $\approx$ 98MB). As noted in Table 5.1, KNC's effective memory bandwidth is twice that of the CPU, and this is reflected in its performance for this operation – KNC is 2.1x and 2.2x faster than the CPU for the two cut-offs.

5.6.2 Thread Scaling

Figure 5.6 shows the execution times for the original miniMD and our implementation when weak-scaled, with a cut-off of 2.5. We made every effort to ensure that the number of atoms per core remained as close to 32K as possible (in line with the LAMMPS benchmark [138]) and that the total problem volume remained a cube. For AVX, the execution time grows by 24% from 1 to 16 cores; and for KNC, it grows by 24% from 1 to 60 cores. Scaling is better for a cut-off of 5.0, due to the larger fraction of time spent in force compute and the neighbour list build; for AVX, the execution time grows by 6% from 1 to 16 cores; and for KNC, it grows by 12% from 1 to 60 cores. The original miniMD scales slightly better in both cases due to its much worse overall performance. Its execution time grows by 15% and 5% going from 1 to 16 cores, for the cut-offs of 2.5 and 5.0 respectively.

The reader is reminded that if our code weak-scaled perfectly, execution time would not be affected by a change in the number of threads. That execution time increases here is due to increased communication costs between threads – although the amount of computation per thread remains fixed, threads will potentially need to exchange position and force data for more neighbouring subdomains, and the cost of thread synchronisation will also increase.

Figure 5.7 shows the execution times for the original miniMD and our implementation when strong-scaled on a 1.372M atom problem, with a cut-off of 2.5. The original code achieves a 14x speed-up on 16 cores, whilst our AVX implementation achieves only 12x. This is due to the significant speed-up we see for the force compute and neighbour list build; the other components do not scale as well and are relatively more expensive. KNC achieves only a 40x speed-up on 60 cores for the same reason – the force compute, neighbour list build, communication and other components see speed-ups of 52x, 41x, 7x and 14x respectively. A cut-off of 5.0 leads to much better parallel efficiency, and our implementation achieves a 14x speed-up on 16 CPU cores. We see a 50x speed-up on 60 KNC cores – the force compute, neighbour list build, communication and other components see speed-ups of 55x, 45x, 6x and 14x respectively.

With perfect strong-scaling, we would hope to see a 60x speed-up overall in both cases. As discussed previously, memory bandwidth does not scale linearly with the number of threads; memory-bound operations, such as integration, inter-thread communication (through shared memory) and writing to the neighbour list thus have much poorer strong-scalability than the force compute. All components of the simulation are also affected by potential workload imbalance (since the number of atoms in each subvolume of space depends on runtime behaviour and simulation parameters) which will also contribute to poor scalability.

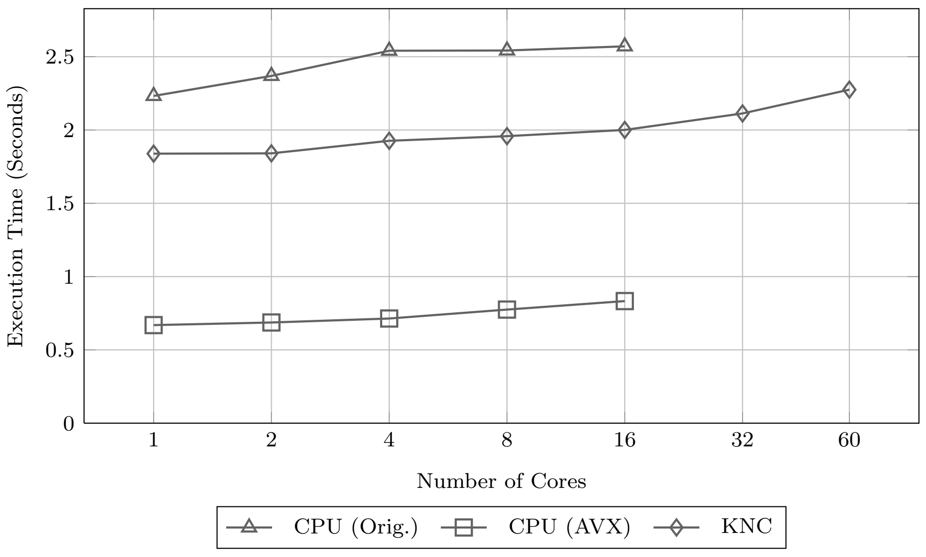


Figure 5.6: Weak-scaling results for miniMD with a cut-off of 2.5.

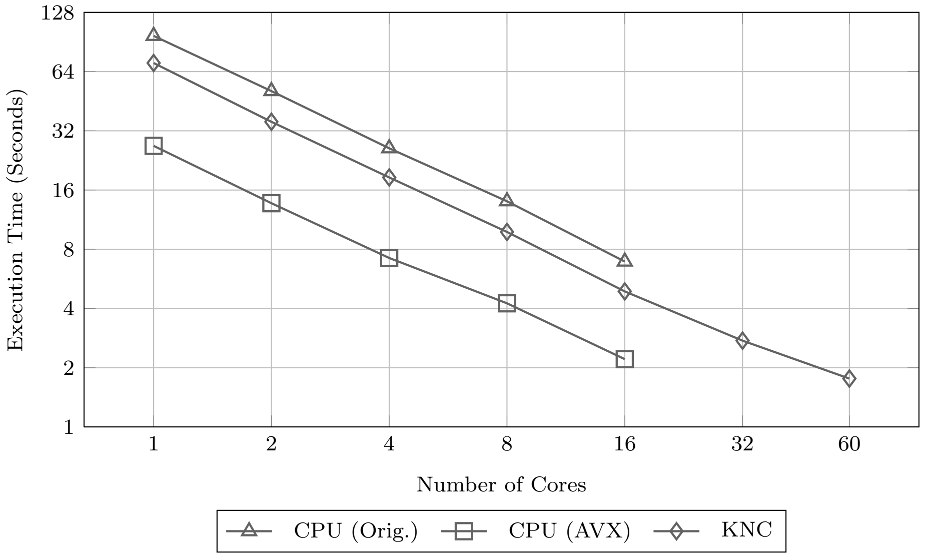


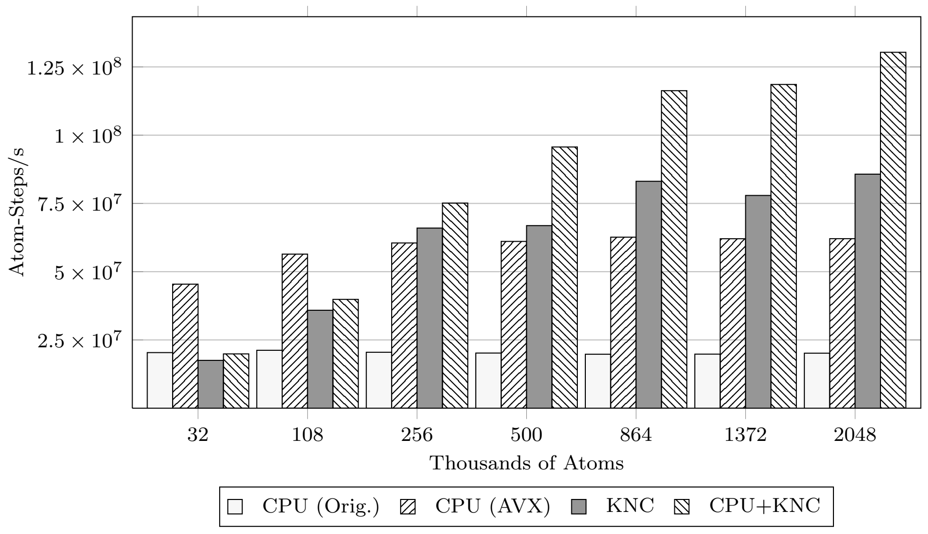
Figure 5.7: Strong-scaling results for miniMD with a cut-off of 2.5.

5.6.3 Architecture Comparison

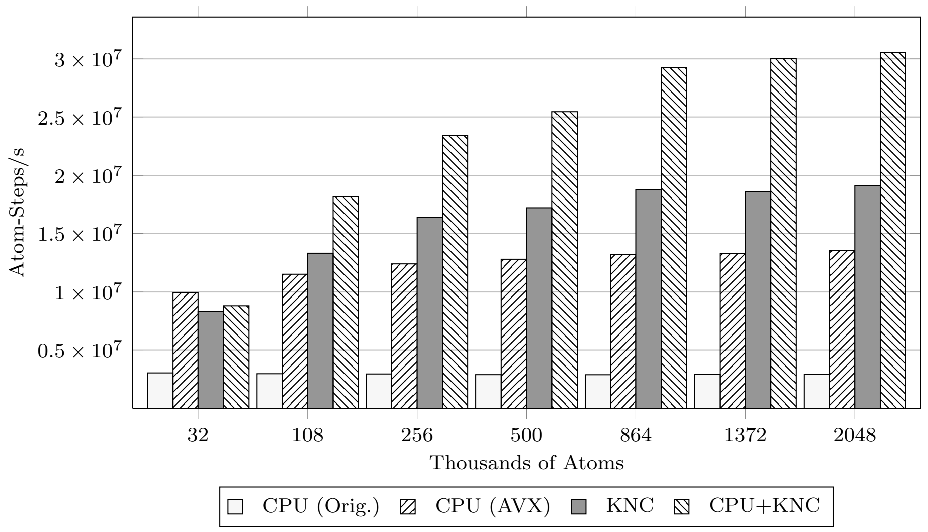
The graphs in Figure 5.8 compare the absolute performance (in atom-steps/s) of our implementation with that of the original miniMD. For the problem sizes shown, the performance of miniMD for a given cut-off distance is almost constant – atom density is fixed, and thus the computational cost *per atom* remains the same across problem sizes. Our implementations of miniMD, on the other hand, improve as problem size increases. This is primarily because smaller problems are dominated by inter-thread communication. For very small atom counts ($\approx$ 4K) the original miniMD exhibits the same behaviour; for some simulations, it is quicker to run on less than the maximum number of cores (all numbers in the graphs are for the maximum number of cores). For the AVX implementation, performance starts to level off at 256K atoms, while KNC and especially CPU+KNC see better performance from even larger simulations. For a cut-off of 2.5, performance improves with the co-processor starting at 256K atoms, and at 108K atoms for a cut-off of 5.0. Real-world implementations should thus take problem size into account when choosing the number of threads, cores and accelerators to use in order to avoid degrading performance on very small problems.

Our optimised AVX code is consistently faster than the original scalar implementation, although the gains grow with problem size, as already described. For a cut-off of 2.5, it is up to 4x faster, and for a cut-off of 5.0 up to 5x faster – a difference that can be attributed to the increased amount of parallelism in problems with higher cut-off distances. One takeaway from this result, besides the effectiveness of our particular optimisations, is the need to revisit and re-tune CPU code when investigating the potential utility of accelerators.

KNC has a peak floating-point rate over four times that of the dual-socket Intel Xeon used for these experiments (Table 5.1), but it achieves only 1.4x higher performance. For force compute, the CPU has a significant per-thread advantage over KNC; it requires fewer cycles per neighbour than the KNC implementation (Table 5.3) and runs at a higher clock frequency. Many operations are either not implemented in SIMD, or do not achieve 100% SIMD efficiency and, although these issues exist on the CPU, their effects are more prominent on KNC due to its wider SIMD. Further, molecular dynamics (particularly the LJ potential) is not dominated by fused multiply-adds, which leads to reduced utilisation of the SIMD arithmetic hardware on KNC – every regular addition, subtraction or multiplication wastes 50% of the compute capability. The CPUs do not have fused multiply-add hardware, but where the number of additions/subtractions and multiplications is not balanced, we also waste compute capability. KNC threads (like those of GPUs and other accelerators) are also more sensitive to exposed cache or memory latency due to the simplicity of KNC cores – cache misses that cannot be hidden by other threads on the same core are more expensive. This is problematic, since the access patterns of the force compute and neighbour list build are too unpredictable to be captured by a hardware prefetcher, and the overhead of software prefetching is too high.



(a) Cut-off of 2.5



(b) Cut-off of 5.0

Figure 5.8: Absolute performance of miniMD in atom-steps/s (higher is better).

5.7 Summary

In this chapter, we present an analysis of the vectorisation of molecular dynamics codes, and demonstrate that gathers and scatters are one of the key bottlenecks. We detail efficient implementations of the neighbour list build and short-range force calculation functions that scale with both SIMD width and number of threads, and also demonstrate a mechanism by which code can be shared effectively across Xeon and Xeon Phi processors. The ability of Intel MIC hardware to run existing codes with few changes, and for programmers to explore the potential performance benefits of Intel's SIMD instruction sets on existing CPU hardware, should make it an attractive prospect for many HPC centres.

We compare the performance of our optimised implementation to that of the original miniMD benchmark, and show it to be consistently faster (by up to 5x on the same hardware and up to 10x with the addition of an Intel Xeon Phi co-processor) for a range of problem sizes and cut-off distances. This considerable performance increase highlights the need to optimise x86 codes and ensure that SIMD is being used effectively on modern CPU architectures. For problems with a large amount of exploitable parallelism, we show that KNC is up to 1.4x faster than a dual-socket, oct-core Intel Xeon E5-2660 server. Although specialised for molecular dynamics, we believe that the techniques that we describe are applicable to other classes of scientific and engineering applications. Other codes featuring gather-scatter memory access patterns (*e.g.* unstructured mesh) could benefit from similar SIMD optimisations, while our methodology of sharing computational work between the CPU and KNC could be utilised by codes solving other spatially decomposed problems (*e.g.* computational fluid dynamics).

CHAPTER 6
Developing "Performance-Portable" Applications

As demonstrated in the previous two chapters, the optimisation of codes for modern SIMD architectures (including computational accelerators) presents several challenges beyond constructing the initial message-passing structure of an application. These challenges include:

(*i*) how and where to locate program data, since many accelerators have localised storage (*data locality*);

(*ii*) how to structure data to improve performance (*memory layout*);

(*iii*) how to transfer data between the host processor and any attached accelerator devices efficiently (*data transfer cost*); and

(*iv*) how to develop an application such that it can run across different architectures (*portability*).

We have shown, using benchmark applications from two different problem domains, that optimisations designed to address the first three of these challenges can be applied in a platform-agnostic manner, benefiting multiple architectures. However, the performance-critical sections of our optimised codes have thus far been written using platform-specific programming languages – CUDA for NVIDIA GPUs, and SSE/AVX/KNC intrinsics for Intel CPUs and MIC. In this chapter, we address the issue of *performance portability*, that is, developing applications that achieve a high level of performance on a wide range of architectures from a single source code.

There are several reasons to assess the practicality of a single source approach to application design: it is easier to maintain a single code that targets all platforms, as opposed to separate hand-tuned versions of the same code for each alternative platform; it reduces the risk of being locked into a single vendor solution; it simplifies the process of benchmarking, since code does not need to be ported before benchmarking can take place; and it represents a "safer" investment for HPC sites, since new codes (and ported legacy codes) will run on both existing and future architectures.

In the remainder of this chapter, we use OpenCL to develop performance-portable implementations of both LU and miniMD, replicating the optimisations discussed in previous chapters. A performance-portable code is clearly more desirable if its performance is competitive with that of "native" implementations, developed in a platform-specific language – we therefore compare the performance of our OpenCL implementations to: (*i*) the original benchmarks (prior to our optimisation efforts), allowing us to reason about the utility of our single-source methodology for HPC sites starting from legacy scalar baselines; and (*ii*) our optimised versions of the benchmarks, allowing a fairer comparison between the levels of performance achievable through the use of OpenCL and "native" programming methodologies.

6.1 "Single Source" Methodology

One of the issues associated with even simple CUDA and OpenCL programs is that optimisation can be very difficult. The specifications of accelerators vary in several respects (*e.g.* number of registers per work-item, amount of shared memory, coalescence criteria) and each kernel has a number of adjustable parameters (*e.g.* the number of work-items and work-groups). The optimal values for these parameters on one architecture may not be optimal on others, and several papers have suggested that this issue is best handled through "auto-tuning" – a process that sees a code automatically searching a given parameter space as it runs, tuning itself to maximise platform performance. Such a process is already common in BLAS functions for new architectures [160], and has been applied to other CUDA and OpenCL codes [42, 47, 83]. We adopt this parameterisation approach in our OpenCL implementations of LU and miniMD, focusing on three high-level criteria we believe to be important when targeting multiple platforms: *work-item/work-group distribution*, *memory layout*, and effective *SIMD width*.

The choice of floating-point precision is also an important parameter. Across CPUs and GPUs, double precision compute is approximately twice as slow as single precision, and the cost of data movement (*i.e.* memory copies or MPI/PCIe communication) is similarly affected. All of the LU results presented use double precision, in keeping with the precision of the original benchmark; we use single precision for the miniMD benchmark, for similar reasons. However, supporting multiple precisions within a single-source application could very easily be achieved through the use of macros and the C pre-processor.

We also support one application-specific parameter for each benchmark: for LU, the k-blocking depth, which we expect to have different optimal values for different hardware; and for miniMD, whether or not N3 is used by the force compute kernel. Similar parameters will exist for other classes of application, and will need to be considered during algorithm design, but the identification of such parameters is beyond the scope of this thesis.

6.1.1 Work-Item and Work-Group Distribution

The number of work-items that can execute in parallel differs by architecture. On accelerator devices, it is typically very high, and a large number of work-items are required in order to hide memory latency effectively; on CPU devices, there is significantly less parallelism available, since each core can execute the instructions of only one thread (or two, with hyper-threading) in parallel. We consider two alternative methods of work-item and work-group distribution for our implementation: *fine-grained* distribution, where one work-item is launched for each grid-point that must be computed (regardless of how many compute units and processing elements the device has); and *coarse-grained* distribution, where one work-group is launched per compute unit, containing a number of work-items equal to the compute unit's SIMD width.

```
for (k = get_global_id(2); k < kmax; k += get_global_size(2) {
  for (j = get_global_id(1); j < jmax; j += get_global_size(1) {
    for (i = get_global_id(0); i < imax; i += get_global_size(0) {
      // Kernel body
    }
  }
}
```

Figure 6.1: OpenCL code-snippet for a kernel supporting all possible combinations of work-item and work-group size.

For codes that set work-group size manually, it is necessary to choose between these two different distribution methods based on architecture type; CPUs prefer coarse-grained distribution, whereas GPUs prefer fine-grained. Alternatively, OpenCL allows for a `null` parameter to be supplied instead of a work-group size, permitting the runtime to select the "best" distribution of work-items based on kernel and hardware parameters – in our experiments, we saw no difference between fine- and coarse-grained distribution when a `null` parameter was supplied. However, it instead becomes necessary to round up the total number of work-items such that it is a multiple of the runtime's "preferred work-group size multiple" (a value that can be queried from the device).

Supporting different values for this parameter at runtime is crucial for performance-portability; hard-coding a "good" work-group size (or multiple) based on the design of any single architecture or SDK is likely to have a significant impact upon the scheduling of work-items on another. For example, we found during our benchmarking (results not shown) that neglecting to ensure that the total number of work-items is divisible by 128 when using the Intel SDK can lead to an order-of-magnitude slow-down for some kernels. Such a slow-down is easily avoided, but only if the code is sufficiently parameterised.

Regardless of how work-group size is decided, kernels must be written in a way that ensures correct results for any combination of work-item and work-group size. Each of our OpenCL kernels is thus enclosed in a set of three nested loops, as shown in Figure 6.1. *imax*, *jmax* and *kmax* refer to the maximum grid-point co-ordinates considered by a given kernel in each dimension, and `get_global_id` and `get_global_size` are two built-in OpenCL functions that return a work-item's ID and the total number of work-items in each dimension respectively. These loops are structured in such a way that the kernel will execute for every grid point from $(0, 0, 0)$ to $(imax, jmax, kmax)$, irrespective of the number or configuration of the work-items and work-groups launched. For example, we consider two extreme cases: a single work-group containing a single work-item will loop from 0 to the maximum in steps of 1; and a set of $imax \times jmax \times kmax$ work-groups containing a single work-item will execute the kernel for exactly one grid-point each.

6.1.2 Memory Layout

If contiguous work-items are mapped to SIMD execution units (as is the case on GPUs, and for auto-vectorised code on CPUs) then accesses to non-contiguous memory locations require a gather operation (*i.e.* an uncoalesced memory access, in CUDA parlance). If work-items are not mapped to SIMD execution units (as is the case for scalar code on CPUs), then a gather is not required; for scalar code, it is more important to choose a memory layout that exhibits good spatial and temporal cache locality.

Supporting multiple memory layouts within a single application is much simpler than one might expect. In our benchmark codes, we replace all accesses to arrays via particular indices with inline functions calls (in the C host code) and macros (in the OpenCL C device code), and thus permit the selection of memory layout at runtime. For example, the original code used to load the k^{th} neighbour index from miniMD's neighbour list (`neighbour_list[k]`) is replaced by a call to a macro (`NEIGHBOUR_INDEX(k)`); the behaviour of this macro can be selected at run-time, on a per-device basis.

Complicating the matter, however, is the fact that the optimal memory layout for one kernel is not necessarily the optimal memory layout for the next – even when both kernels are running on the same hardware, different layouts will be more efficient for different memory access patterns. It is therefore desirable to support a different memory layout for each kernel, as discussed in the context of

LU in Section 4 (where we supported different layouts for the code's wavefront and stencil update sections). We currently detect such situations manually, and hard-code kernels that switch between memory layouts, but this could be improved in future work.

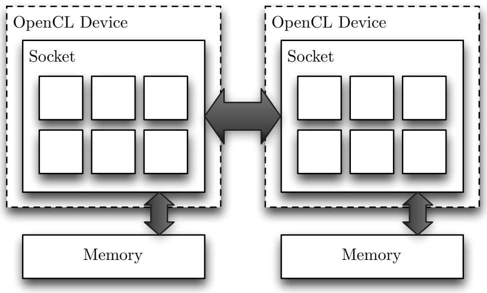
6.1.3 Implicit vs. Explicit Vectorisation

There are two forms of vectorisation supported by OpenCL compilers: *implicit* vectorisation (or auto-vectorisation), which sees the compiler pack the work of contiguous work-items into SIMD units; and *explicit* vectorisation, where a kernel makes use of vector types (*e.g.* `float4` or `double2`). On CPUs, the AMD SDK generates SSE/AVX code only in the latter case, whereas the Intel SDK attempts to auto-vectorise kernels. On GPUs all three SDKs auto-vectorise the kernels; if a vector operation is encountered in a kernel, it is replaced by a series of equivalent scalar operations and re-vectorised.

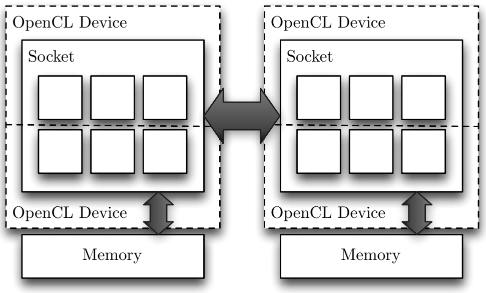
Although not all compilers benefit from the use of vector arithmetic, the use of vector types within kernels is still encouraged for load and store operations. Using vector types in this way serves as a hint to the compiler that certain values are actually stored contiguously in memory, allowing for more efficient gather/scatter instruction sequences to be generated.

6.1.4 Device Fission

Although a multi-socket CPU node *can* be treated as a shared memory system, it is arguably unwise to do so in every case; platform-agnostic implementations such as those considered here do not necessarily consider the system's memory hierarchy and are therefore unlikely to exhibit good memory behaviour. More specifically, due to the fact that our OpenCL implementations have no control over the order in which work-groups and work-items are executed, nor the compute units to which they are allocated, we cannot guarantee that the set of work-groups processed by any given compute unit will exploit either temporal or spatial cache locality.



(a) Configuration 1



(b) Configuration 2

Figure 6.2: A dual-socket, hex-core node fissioned into (a) two sub-devices of six compute units; and (b) four sub-devices of three compute units.

A recent addition to the OpenCL standard known as *device fission* may go some way towards solving this problem, by allowing the runtime to "fission" (*i.e.* split) a single device into multiple sub-devices. This grants an OpenCL application the ability to assign work to specific CPU cores, but is not currently supported by GPUs. However, Kepler's ability to support multiple MPI tasks feeding work to the same GPU could be used to the same effect.

In keeping with our policy of making only minimal changes to source, our implementation couples device fission with existing MPI parallelism; instead of running one MPI task per node as before, we run one MPI task per created sub-device (Figure 6.2). The advantage of this approach is that the only new code required is a simple check of whether fission is to be used in a given run, detection of the number of sub-devices to create, and the assignment of sub-devices to MPI tasks based on rank. As such, we argue that it is a logical way to add support for device fission into an existing MPI-based code.

We acknowledge that there are likely to be some overheads introduced by device fission, and that these may grow as the number of sub-devices created increases. Each MPI task will consume additional system resources (compared to a "pure" MPI implementation) to manage the task queue for its sub-device, and there may be other scheduling conflicts between threads created and managed by the OpenCL runtime. Specifically, it is unknown whether device fission guarantees that the affinity of a created sub-device will remain fixed, or whether it is possible for multiple MPI ranks to be assigned the same subset of cores.

6.1.5 Communication

Our performance-portable implementations do not make any distinction between integrated and discrete devices. As such, performance on the CPU and integrated GPUs is likely to be decreased by the presence of unnecessary `memcpy` operations – wherever a code would transfer data to a discrete device over PCIe, it transfers a copy of the data to *itself* via shared memory. Although this is inefficient, and will need to be addressed in future work, its effects on the execution times reported in this section are minimal, since our optimisations seek to reduce the amount of communication between host and device.

6.2 Benchmark Parameterisation

In this section, we demonstrate the process of parameterising a scalar baseline implementation of a code, detailing how each of the SIMD and threading optimisations described for miniMD in Chapter 5 map to our single-source development methodology. We do not detail this process for wavefront applications, due to the similarities between CUDA and OpenCL – the implementation of our OpenCL port is almost identical to the CUDA port already discussed in Chapter 4. The optimisation process for our molecular dynamics code is more complicated, owing to the significant differences between OpenCL and hand-vectorisation with intrinsics.

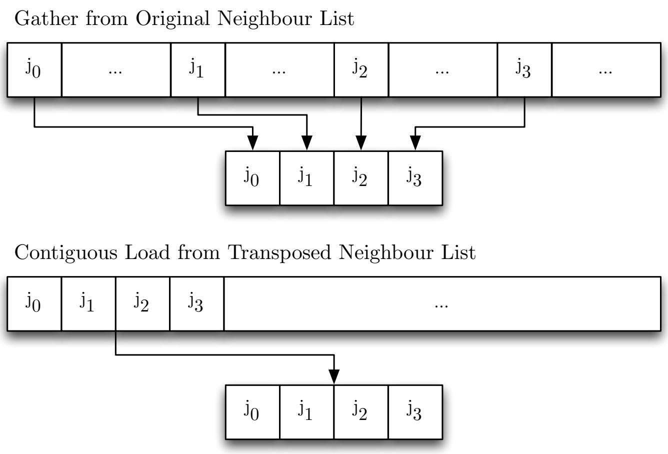


Figure 6.3: Effect of neighbour list transposition on memory access pattern. Each arrow represents a single load instruction.

6.2.1 Memory Layout

As shown in Chapter 5, storing atom positions in AoS is more efficient than storing them in SoA, even when using SIMD execution, since any kernel that accesses one component of the struct (*i.e.* x, y or z) will also access the other two. We therefore do not make any changes to the layouts of the position, force, velocity or old velocity arrays.

However, since the simplest method of mapping computation to work-items is via cross-atom (as opposed to cross-neighbour) parallelism, we support the selection of two alternative neighbour list layouts: storing each of the neighbours for a given atom contiguously, as in the original benchmark; and storing the list in transposed form (*i.e.* storing the 0^{th} neighbour of W atoms, followed by the 1^{st} neighbour, and so on). For the transposed form, we insert “dummy” neighbours as padding when W atoms have a different number of neighbours. Supporting both neighbour lists is very simple, since the first storage format is actually a special case of the second – the original neighbour list is equivalent to a transposed neighbour list with $W = 1$.

Figure 6.3 demonstrates the benefit of this memory layout change: prior to transposition, W contiguous work-items must read from memory with a stride

```
__kernel force_compute(float4* pos, float4* force...) {
    int i = get_global_id(0);
    float4 posi = pos[i];
    float4 fi = (float4) (0.0, 0.0, 0.0, 0.0);
    for (k = 0; k < number_of_neighbours[i]; k++) {
        int j = NEIGHBOUR_INDEX(k);
        float delx = posi.x - pos[j].x;
        float dely = posi.y - pos[j].y;
        float delz = posi.z - pos[j].z;
        ...
    }
    force[i] = fi;
}
```

Figure 6.4: OpenCL code-snippet for a force compute kernel storing atom data in `float4`s.

of k (where k is the number of neighbours); and following transposition, all neighbour list accesses become a single load from contiguous memory locations (when W is set to the hardware's SIMD width).

Surprisingly, the choice of memory layout can affect more than memory behaviour. The Intel SDK attempts auto-vectorisation only when the compiler expects it to improve performance (based on some heuristic), and we believe the presence of a large number of gather operations in the original kernel caused it to fail this check – storing the neighbour list in transposed form removes a significant number of these gathers, and enables the compiler to auto-vectorise our short-range force kernel.

6.2.2 Implicit vs Explicit Vectorisation

Defining the position and force arrays using vector types (Figure 6.4) serves as a hint to the compiler that the x, y and z components of an atom's position are stored contiguously; for all of the hardware considered in this study, the full position of a given atom can be loaded in a single memory access, and this hint permits the compiler to employ a more efficient gather (Figure 6.5). On NVIDIA GPUs, the accesses to x, y and z become coalesced; and the Intel compiler is able to emit a more efficient sequence of instructions – an AoS-to-SoA transpose, in place of a series of scalar loads – on both CPU and integrated GPU hardware.

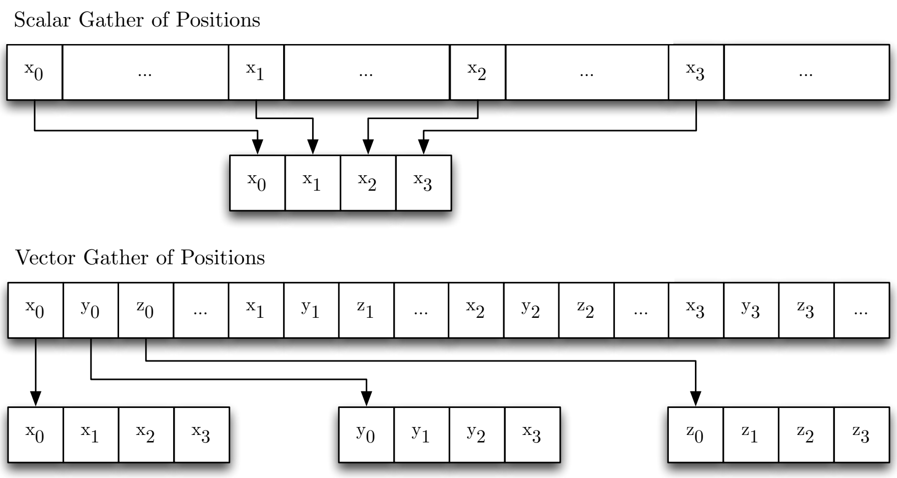

Scalar Gather of Positions
x_0 ... x_1 ... x_2 ... x_3 ...
x_0 x_1 x_2 x_3
Vector Gather of Positions
x_0 y_0 z_0 ... x_1 y_1 z_1 ... x_2 y_2 z_2 ... x_3 y_3 z_3 ...
x_0 x_1 x_2 x_3
y_0 y_1 y_2 x_3
z_0 z_1 z_2 z_3

Figure 6.5: Effect of using vector types on memory access pattern.

The reader is reminded that we were required to hand-code this same AoS-to-SoA transpose with intrinsics when using standard C (Chapter 5), highlighting that OpenCL is better-suited to expressing vectorisation opportunities than traditional serial languages.

It may seem like a sensible next step would be to convert the kernel's arithmetic to vector form (Figure 6.6), potentially exposing intra-loop parallelism to the compiler and resulting in a more succinct (and arguably more readable) version of the code. However, it degrades performance on most of the hardware considered here. On devices employing auto-vectorisation, our use of the dot-product function introduces two additional instructions per loop iteration – the compiler does not know that the last element of the `float4` is 0, and so includes it in the dot-product. On devices relying on explicit vectorisation, there are SIMD inefficiencies: for CPUs supporting AVX, our explicit vectorisation uses only four of the eight SIMD execution units available; and the majority of the arithmetic (the calculation of the force, based on rsq) remains scalar.

Unrolling the loop over neighbours in our explicit vectorisation kernel, such that it computes the force between i and several neighbours simultaneously, allows us to regain this lost SIMD efficiency. Theoretically, compilers could employ such an optimisation themselves, but we found that this is not yet the

```
__kernel force_compute(float4* pos, float4* force...) {
    int i = get_global_id(0);
    float4 posi = pos[i];
    float4 fi = (float4) (0.0, 0.0, 0.0, 0.0);
    for (k = 0; k < number_of_neighbours[i]; k++) {
        int j = NEIGHBOUR_INDEX(k);
        float4 del = posi - pos[j];
        float rsq = dot(del, del);
        float sr2 = 1.0 / rsq;
        float sr6 = sr2 * sr2 * sr2;
        float f = sr6 * (sr6 - 0.5) * sr2;
        f = (rsq < cutoff) ? f : 0.0;
        fi += del * f;
    }
    force[i] = fi;
}
```

Figure 6.6: OpenCL code-snippet for a force compute kernel using vector arithmetic.

case. Our parameterised version of the kernel thus supports unrolling by two different factors: 4-way unrolling, using `float4` types, to improve utilisation of SSE instructions and the very long instruction word (VLIW) architectures of AMD GPUs; and 8-way unrolling, using `float8` types, to make use of AVX instructions.

6.2.3 Device Fission

In addition to improving memory behaviour, we can leverage fission to permit the use of N3 on CPUs. By starting a single work-item per fissioned compute unit, we replicate the behaviour of the original miniMD code – each MPI task (and hence each work-item) operates in its own memory space, on its own atoms, removing the potential for write conflicts.

We consider this optimisation only because it does not impact performance portability – although we re-introduce the neighbour force updates, we guard them with a pragma (*i.e.* `#ifdef N3`). Also, as discussed previously, the code changes necessary to support fission in this way are minimal, affecting only the initial OpenCL setup code.

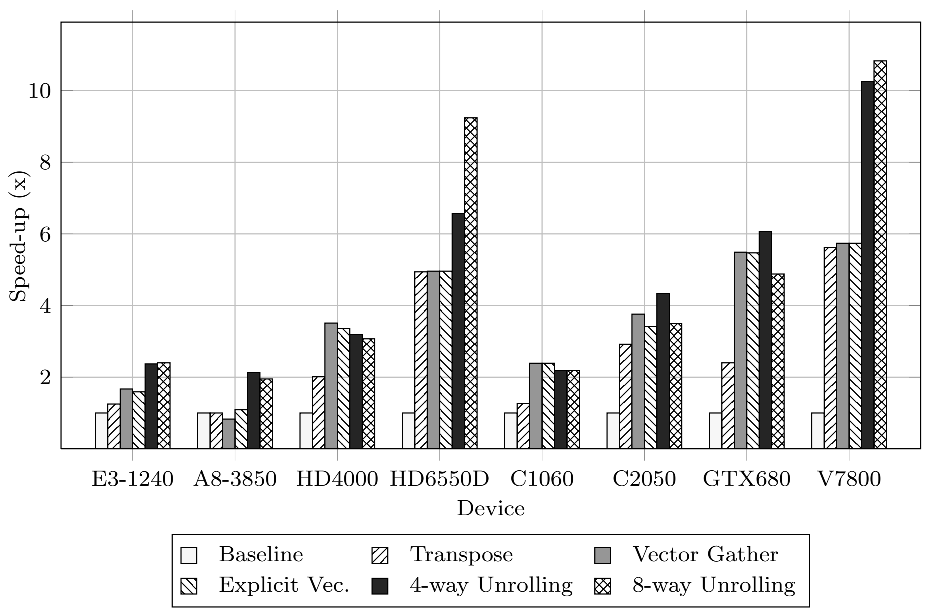


Figure 6.7: Comparison of speed-ups for parameterised force calculation kernel on various architectures.

6.3 Performance Results

The graph in Figure 6.7 presents the total performance improvement of our optimisations, relative to the original scalar baseline code. This performance improvement is cumulative, since each optimisation builds upon the previous – the only exception to this is 8-way unrolling, which replaces 4-way unrolling. The speed-up of our best kernel (*i.e.* explicit vectorisation with unrolling) is consistently greater than 2x across all architectures; these results show that it is possible to accelerate an OpenCL code significantly, on a wide range of hardware, without focusing on development for any one particular micro-architecture.

Generally speaking, the GPUs see more benefit from the memory optimisations while the CPUs see more benefit from explicit vectorisation. This reflects both the simpler cores of GPUs (which are more sensitive to memory latencies, and are thus more impacted by the original memory access pattern) and the immaturity of OpenCL compilers for CPUs. The AMD GPUs are the only architecture to benefit significantly from both types of optimisation (with a

Implementation	**Sweeps**		`rhs`	**Other**	**Total**	**Speed-up**	
	`blts`	`buts`				**(Orig.)**	**(Opt.)**
Original Fortran	138.21	133.37	108.66	10.04	390.28	1.00x	0.64x
Optimised Fortran	66.38	64.52	109.28	8.65	248.83	1.57x	1.00x
OpenCL (Intel)	78.19	77.09	131.49	55.83	342.60	1.14x	0.73x
OpenCL (AMD)	76.23	74.15	129.13	49.57	329.08	1.19x	0.76x

Table 6.1: Comparison of execution times (in seconds) for Intel CPU implementations of LU.

total speed-up of $\approx$ 11x). However, where a code change does not improve performance, it does not significantly degrade it, demonstrating the performance portability of our optimisations.

6.4 Comparison with "Native" Implementations

6.4.1 Pipelined Wavefront

CPU Comparison

Table 6.1 compares the performance of our OpenCL implementation of LU with the original and optimised Fortran implementations, executing on an Intel X5550. On a single node, our OpenCL implementation is up to 1.2x faster than the original Fortran, but 1.4x slower than our optimised Fortran implementation.

There is not likely to be one single reason for this performance gap, but rather several contributing factors. Firstly, we note that we see almost a 2x difference in performance between the GNU Fortran compiler and the Sun Studio compiler used to collect our results; we attribute this difference to the latter's ability to apply intensive inter-procedural optimisations (IPO) across all source files, increasing compile time significantly but often providing better performance. OpenCL compilers currently do not support such functionality. Secondly, there are differences in memory behaviour: the OpenCL runtime creates threads which may operate on memory located in remote memory banks, whereas each of the MPI tasks in the Fortran implementation operates only on its own, local, mem-

Configuration 1: 2-way Device Fission				
Processor Cores	**MPI Ranks**	$k_B = 1$	$k_B = \min\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)$	$k_B = N_z$
24	4	2.04x	1.06x	0.95x
48	8	1.91x	1.03x	0.89x
96	16	1.82x	0.97x	0.96x

Configuration 2: 4-way Device Fission				
Processor Cores	**MPI Ranks**	$k_B = 1$	$k_B = \min\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)$	$k_B = N_z$
12	4	3.03x	0.91x	0.78x
24	8	2.94x	1.05x	0.71x
48	16	3.03x	1.03x	0.77x

Table 6.2: Speed-up of LU for two device fission configurations, and three k_B values.

ory space. Finally, the implementations of `blts` and `buts` in the native code are serial in nature, whereas their implementations in the OpenCL code use Lamport's hyperplane algorithm. The algorithm is much better suited to parallelisation (and thus to portability across architecture types), but may suffer from increased synchronisation overhead (between kernels) [146] and other inefficiencies.

The latter two issues could be solved through the use of fission, and a coarse wavefront across compute units. Unfortunately, our investigation into this possibility was limited by the restriction that LU must be run on a number of MPI tasks that is a power of two. For the fission configurations we *were* able to test, on the *Sierra* supercomputer, we see that the effect of fission depends on the k_B parameter: for $k_B = 1$, 2-way and 4-way fission lead to speed-ups of 2x and 2.9x respectively; for $k_B = \min(N_x/P_x, N_y/P_y)$, we see worse performance in some configurations, with a maximum improvement of 1.2x; and for $k_B = N_z$, performance is always worse, due to an increase in pipeline fill time. These results are listed in Table 6.2.

Device	Implementation	Sweeps		`rhs`	Other	Total	Speed-up
		`blts`	`buts`				
C1060	CUDA	29.09	29.94	67.95	29.69	156.67	1.00x
	OpenCL	31.86	32.32	70.23	35.26	169.67	0.92x
C2050	CUDA	10.09	9.11	22.29	10.14	51.63	1.00x
	OpenCL	8.83	8.6	28.44	8.81	54.68	0.94x

Table 6.3: Comparison of execution times (in seconds) for NVIDIA GPU implementations of LU.

GPU Comparison

Table 6.3 compares the performance of our OpenCL implementation of LU with the optimised CUDA implementation, executing on an NVIDIA Tesla C1060 and C2050. The CUDA implementation is marginally faster in both cases (1.08x and 1.06x). Neither of these gaps is as large as those reported elsewhere for initial ports of CUDA codes [42, 83], but these have been shown to be due to differences between the optimisations carried out by NVIDIA's CUDA and OpenCL compilers – we believe that the small difference in performance shown here is a reflection of the similarity between our CUDA and OpenCL implementations, in particular the hand-unrolled and hand-inlined nature of our kernels.

6.4.2 Molecular Dynamics

CPU Comparison

Table 6.4 compares the performance of our OpenCL implementation of miniMD with the two native C++ implementations, executing on an Intel E3-1240. The AMD SDK provides the fastest runtime for our OpenCL application, beating the Intel SDK by approximately 10%. This is also 1.5x faster than the original C++ version of miniMD, demonstrating that OpenCL is a suitable development tool for utilising the SIMD architectures of modern CPUs. However, our OpenCL implementation is 2x slower than our heavily-optimised version of miniMD; its force compute and neighbour list build are 2x and 6x faster respectively. The biggest causes of these performance gaps are: firstly, that the AVX code generated by current OpenCL compilers is inefficient (compared to our hand-

Implementation	**Force**	**Neigh**	**Comm**	**Other**	**Total**	**Speed-up (Orig.)**	**Speed-up (AVX)**
Original C++	2.77	0.26	0.19	0.21	3.42	1.00x	0.35x
C++ with AVX	0.76	0.11	0.12	0.20	1.19	2.87x	1.00x
OpenCL (Intel)	1.41	0.60	0.25	0.17	2.43	1.41x	0.49x
OpenCL (AMD)	1.33	0.58	0.16	0.17	2.25	1.52x	0.53x

Table 6.4: Comparison of execution times (in seconds) for Intel CPU implementations of miniMD.

```
float8 fxjtmp = (f[j1].x, f[j2].x, f[j3].x, f[j4].x,
                 f[j5].x, f[j6].x, f[j7].x, f[j8].x);
float8 fyjtmp = (f[j1].y, f[j2].y, f[j3].y, f[j4].y,
                 f[j5].y, f[j6].y, f[j7].y, f[j8].y);
float8 fzjtmp = (f[j1].z, f[j2].z, f[j3].z, f[j4].z,
                 f[j5].z, f[j6].z, f[j7].z, f[j8].z);
```

Figure 6.8: OpenCL code-snippet for the gather of x, y and z positions in the force compute kernel with 8-way unrolling.

vectorised intrinsics); and secondly, that both the original miniMD and our optimised AVX code make use of N3, and thus do half as much computational work during the force compute and neighbour list build.

We believe that the first of these problems will be solved by future compiler releases. OpenCL lacks gather/scatter and transpose constructs for its vector types, and so we implement the gather of positions as shown in Figure 6.8 – neither the AMD or Intel compiler (currently) recognises that this can be performed as an in-register AoS-to-SoA transpose, and we provide an in-depth analysis of the assembly generated by the compiler for this code in Appendix A. The second issue can be solved using fission, as discussed in Section 6.2.3: for the Intel SDK, this improves force compute performance by 1.1x, and neighbour list build performance by 2x, increasing the speed-up over the original miniMD to 1.7x; for the AMD SDK, using fission results in worse performance than the original miniMD, increasing communication costs by 9x – possibly due to scheduling conflicts between the threads spawned by MPI and AMD's OpenCL runtime.

Device	Implementation	Force	Neigh	Comm	Other	Total	Speed-up
C1060	CUDA	0.67	0.35	0.06	0.09	1.18	1.00x
	OpenCL	1.69	0.31	0.34	0.07	2.40	0.49x
C2050	CUDA	0.36	0.22	0.05	0.06	0.70	1.00x
	OpenCL	0.58	0.12	0.28	0.05	1.03	0.68x

Table 6.5: Comparison of execution times (in seconds) for NVIDIA GPU implementations of miniMD.

GPU Comparison

Table 6.5 compares the performance of our OpenCL implementation of miniMD and LAMMPS$_{\text{CUDA}}$ [154, 155]. As before, we use the kernel with explicit vectorisation and 4-way unrolling. For the complete application, our implementation is 2x slower than CUDA on the C1060, and 1.5x slower than CUDA on the C2050. A large portion of this difference can be attributed to the poor communication scheme used by our OpenCL code, which is almost 6x slower than that used by LAMMPS$_{\text{CUDA}}$. The CUDA code executes all communication routines on the device and, since we are using a single node, is able to avoid all PCIe communication; our OpenCL code, on the other hand, executes all communication routines on the host.

For the force compute kernel alone, our OpenCL implementation is 2.5x and 1.6x slower than CUDA on the C1060 and C2050 respectively. Again, this is partly due to documented differences between NVIDIA's CUDA and OpenCL compilers [42, 83]. LAMMPS$_{\text{CUDA}}$ also makes use of read-only texture memory, which is cached, to store atom positions. Support for texture memory could be added to our kernel and guarded with pragmas, enabled for GPUs in a similar way to how fission is enabled for CPUs – we intend to explore this option in future work.

6.5 Summary

In this chapter, we discuss the feasibility of using OpenCL to create platform-agnostic, performance-portable, applications. We discuss a development methodology that allows for kernels to be parameterised based on key differences between architectures, highlighting the importance of designing codes that allow for work-item/work-group distribution, memory layout and SIMD width to be altered at runtime.

We report on the development of performance-portable versions of the two benchmarks featured in Chapters 4 and 5, and demonstrate that they perform well on a wide range of hardware from different vendors (including CPUs and integrated GPUs from AMD and Intel, and discrete GPUs from AMD and NVIDIA). We thus show that it is possible to develop an application that is optimised for multiple micro-architecture designs without sacrificing portability or maintainability – an unexpected result that will be welcomed by the maintainers of large, legacy, code-bases.

Our OpenCL implementations of the LU and miniMD benchmarks are at most 2x slower than “native” implementations highly optimised for particular platforms (on a single node), and whether this performance compromise is an acceptable penalty for increased code portability is open to debate. We acknowledge that there will be particular areas of large codes for which it is necessary (or preferable) to maintain separate platform-specific source code for algorithmic reasons, or common functions (*e.g.* matrix operations) for which highly optimised and platform-specific libraries are made available by vendors. However, we believe that the methodology demonstrated in this chapter is a simple way to achieve acceptable levels of performance across different architectures – and that such an approach is well suited to many parallel workloads.

CHAPTER 7
Predicting Application Performance on Future Architectures

The procurement of a new supercomputer or cluster is an expensive and time-consuming process. In addition to the price of hardware, HPC sites must budget for power, cooling and maintenance throughout a machine's lifetime, costing millions of dollars each year. As we have demonstrated in previous chapters, an increase in theoretical peak performance does not necessarily result in an equivalent increase in application performance; there is an understandable desire to evaluate the performance of applications on new hardware ahead of time.

For this reason, HPC sites are often given advance access to engineering samples of new architectures, or remote access to existing machines, permitting them to benchmark a small machine before committing to a larger order. The utility of analytical modelling and simulation techniques for estimating the performance of traditional machines at scale, based on these single node benchmarks, has been demonstrated in previous work [57, 72]. The contents of this chapter address the adaptation of these techniques for accelerator-based architectures. Specifically, we consider using models to predict the effects of: (*i*) iterative performance improvements to a given architecture type (*e.g.* Tesla to Fermi); and (*ii*) network communication and scaling behaviour. Our results demonstrate that the significant speed-ups shown on single nodes do not necessarily persist at scale, highlighting the importance of considering multi-node runs when comparing architectures.

7.1 Adapting Models for Future Architectures

7.1.1 Single Node

Several other works have demonstrated that it is possible to accurately model the performance of select application kernels based on source code analysis, or through low-level hardware simulation of GPUs [14, 17, 31, 36, 75, 166]. We seek to produce a performance model at a higher level of abstraction, adopting the analytical modelling approach of Mudalige *et al.* [107] – specifically, we seek to derive a "grind-time" (W_g), so-called because it represents the time spent working (or "grinding"), for each element of computation that must be performed. The lowest level at which we can acquire times for many current-generation accelerator devices is that of a complete kernel, and we thus cannot benchmark a value of W_g directly. Instead, we time the value for the complete kernel (W).

Essentially, our model states that the execution time of a kernel will stay constant so long as the number of work-groups assigned to each compute unit remains the same. An increase in the number of work-groups scheduled to each compute unit will increase the execution time by some factor (based on the ability to hide memory latency via time-slicing); any further blocks scheduled to a compute unit after they have been saturated will require additional processing steps, and we model these as occurring serially. We thus predict a stepping behaviour in execution times, with steps occurring every (# compute units $\times$ work-group size) work-items. These steps will differ in size, based on whether the addition of an extra work-group fully saturates a compute unit; this depends upon register usage, work-group size and the amount of shared memory required.

We apply the model to the wavefront kernel from our GPU implementation of LU, executing on a Tesla C1060. The C1060 has 30 compute units and each of our work-groups contains 64 work-items, thus our model predicts an increase in execution time every $30 \times 64 = 1920$ work-items. The kernel uses 107 registers, limiting the number of concurrent work-groups per block to 2, and so we expect

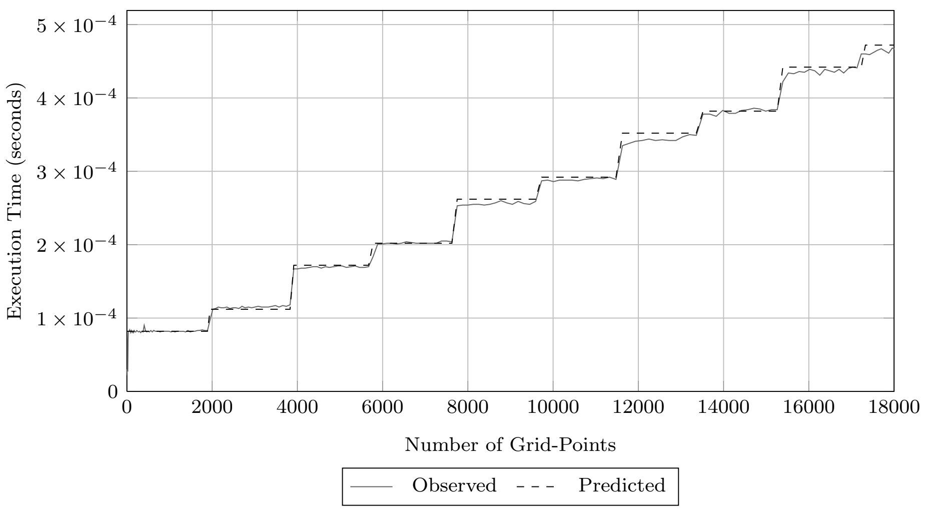


Figure 7.1: Observed and predicted times for a wavefront kernel on a Tesla C1060.

two different step sizes – we derive their values empirically, benchmarking the kernel for 0–4000 grid-points.

The graph in Figure 7.1 compares the observed and predicted times for each hyperplane step, from 1 to 18000 grid-points. Our model accurately predicts the stepping points, and matches the observed times very closely. The model error increases slightly for large numbers of grid-points, but we believe this to be the result of memory contention issues not covered by this simple model (*e.g.* partition camping). That such a simple analytical model has any predictive accuracy at all is due only to the relative simplicity of early accelerator architectures. As the architectures have grown more complex (and arguably more CPU-like), introducing features such as data caches, out-of-order execution, support for multiple concurrent kernels and, most recently, "dynamic parallelism" (*i.e.* the ability for kernels to launch other kernels) [121], the complexity required to accurately capture the performance of an individual work-item analytically has increased.

To demonstrate this, we next apply the model to the same kernel executing on a Tesla C2050. This GPU has 14 compute units, supporting a maximum

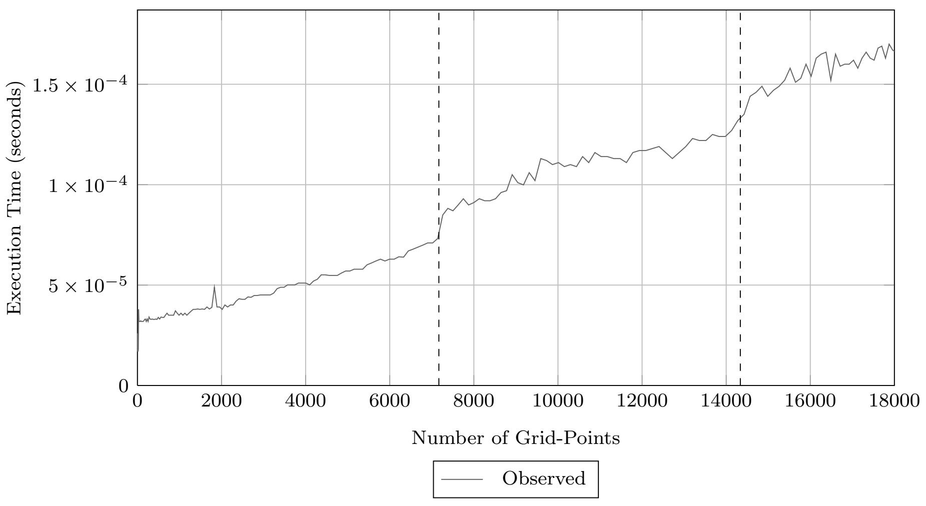


Figure 7.2: Observed times for a wavefront kernel on a Tesla C2050.

of 8 work-groups, and is built on NVIDIA's "Fermi" micro-architecture. As before, and as shown in Figure 7.2, the increases in execution time correspond to increases in the number of work-groups scheduled to each compute unit – every $14 \times 64 \times 8 = 7168$ threads (marked in the graph by dashed lines). However, the execution time between these stepping points is not fixed, instead increasing linearly, thus violating the assumption underpinning our model.

7.1.2 Multiple Node

At the multi-node level, the use of such a high level of model abstraction remains applicable. In theory, the MPI costs of an application should remain the same (assuming that the same algorithm is used across architectures) and we can thus model the performance of an accelerator at scale by substituting execution times benchmarked on an accelerator into an existing MPI-level model. We first demonstrated the utility of this approach in [126], and it has since been adopted by others to investigate the performance of computational fluid dynamics applications running on GPUs [108].

The use of models allows us to investigate the performance of larger test cases (Class D and Class E) than we were able to consider in previous chapters.

These problem sizes are too large to be executed on a single GPU device, and necessitate the use of multiple GPUs. All of the models in this section assume one GPU per node; although other configurations are clearly possible, this assumption simplifies the issue of PCIe contention (four GPUs per node would commonly share two PCIe buses).

An Analytical Performance Model

We employ a simplified form of the reusable wavefront model of Mudalige *et al.* [107], to model the coarse wavefronts over processors:

$$T_{\text{comm}} = \left[\left(P_x + P_y + \frac{N_z}{k_B} - 2\right) - 1\right] \times T_c \tag{7.1}$$

$$T_{\text{comp}} = \left(P_x + P_y + \frac{N_z}{k_B} - 2\right) \times W \tag{7.2}$$

where T_{comm} represents the total communication time, and T_c the time per communication phase; and where T_{comp} represents the total computation time, and W the time taken to compute a single block of depth k_B. Essentially, the model states that there are $(P_x + P_y + N_z/k_B - 2)$ coarse wavefront steps, and that communication occurs after all such steps except for the last.

T_c is computed based upon a time-per-byte, calculated from network latency and bandwidth in conjunction with message sizes derived from the problem size and number of processors. We also augment the original network model to include PCIe transfer times, representing the costs associated with reading/writing data from/to a GPU:

$$T_{\text{PCIe}}(\text{bytes}) = \text{PCIe latency} + \frac{1}{\text{PCIe bandwidth}} \times \text{bytes} \tag{7.3}$$

$$T_{\text{network}}(\text{bytes}) = \text{network latency} + \frac{1}{\text{network bandwidth}} \times \text{bytes} \tag{7.4}$$

$$\begin{aligned} T_c = {} & T_{\text{network}}(MessageSize_{NS}) + T_{\text{PCIe}}(MessageSize_{NS}) \\ & + T_{\text{network}}(MessageSize_{EW}) + T_{\text{PCIe}}(MessageSize_{EW}) \end{aligned} \tag{7.5}$$

where $MessageSize_{NS}$ and $MessageSize_{EW}$ represent the message sizes for north-south and east-west sends/receives respectively. Network latency and bandwidth are calculated based on results from a modified version of the `p2p.ski` test included in the SKaMPI [142] benchmark, executed for a number of message sizes on a varying number of core/node counts in order to account for contention. PCIe latency and bandwidth are obtained using the `bandwidthTest` benchmark in the NVIDIA CUDA SDK.

W is computed based on a "grind-time" per grid-point (W_g):

$$W = W_g \times \left(\frac{N_x}{P_x} \times \frac{N_y}{P_y} \times k_B \right) \tag{7.6}$$

which we collect empirically using benchmark runs.

A Simulation-based Performance Model

In order to verify our analytical model results, we also employ a performance model based on discrete event simulation. We use the WARPP simulator [56], which utilises coarse-grained compute models as well as high-fidelity network modelling to enable the accurate assessment of parallel application behaviour at large scale.

A key feature of WARPP is that it also permits the modelling of compute and network noise through the application of a random distribution of noise to compute or networking events. In this study, we present two sets of runtime predictions from the simulator: a standard, noiseless simulation; and a simulation employing noise in data transmission times.

In the simulations including noise, the network events have a Gaussian distribution (with a standard deviation consistent with benchmarked data) applied to MPI communications. The simulator is therefore able to create a range in communication costs, which reflect the delays caused by other jobs and background networking events present in the machine. As with the analytical model, we augment WARPP's network model with a sub-model capturing the behaviour of the PCIe bus. Due to WARPP's modular design, this is a simple change; we increase network communication times for a given number of bytes based upon a linear piece-wise regression of observed PCIe message times.

Model Validation

Validations of both performance models are presented Table 7.1. We compare the execution times for LU on three clusters (a cluster of C1060 GPUs at the Daresbury Laboratory, and the *Hera* and *DawnDev* machines at the Lawrence Livermore National Laboratory) for two application classes (C and D) to the predictions from the analytical model and discrete event simulator. The reader is reminded that the GPU cluster has one C1060 per node; that *Hera* has 16 AMD Opteron cores per node; and that *DawnDev* is a Blue Gene/P with four PowerPC cores per node. The simulation without noise is deterministic, but for the simulation with network noise we also list the 95% confidence interval (C.I). The compiler configurations for *DawnDev* and *Hera* are given in Table 7.2, and the compiler configuration for the GPU implementation remains the same as in previous chapters.

Model accuracy varies between the machines, but is between 80% and 90% for almost all runs. When reading these accuracy figures it is important to understand that the code is executed on shared machines, and validating the models on contended machines tends to increase the error due to network contention and machine load (the model error tends to be lower when the machine is quieter). For this reason, the introduction of noise to the simulator for *Hera* is important – this resource is much more heavily used (and contended), as demonstrated by the inclusion of minimum and maximum values in Table 7.1.

A degree of inaccuracy in the models is to be expected on all machines, as we do not capture issues such as process placement on the host machine; a poor allocation by the scheduler will impact on runtime, and thus model error. However, the high levels of correlation between the analytical and simulation-based models – in spite of the presence of other jobs and background noise – provide a significant degree of confidence in their predictive accuracy.

Machine	**Nodes**	**Actual**			**Anal.**	**Simulation**		
							With Noise	
		Min.	**Mean**	**Max.**		**No Noise**	**Mean**	**95% C.I.**
C1060 (Class C)	1	153.26	153.30	153.37	153.26	147.15	147.17	(147.17, 147.17)
	4	67.06	67.25	67.58	70.45	66.43	69.00	(68.82, 69.19)
	8	52.72	52.92	53.08	52.72	50.50	53.92	(53.74, 54.12)
	16	44.29	44.46	44.51	44.47	42.85	46.09	(45.98, 46.21)
C1060 (Class D)	4	1359.93	1367.65	1372.85	1417.57	1375.28	1393.32	(1390.88, 1395.75)
	8	735.53	736.60	737.47	744.24	723.83	745.47	(744.88, 747.36)
	16	414.31	414.97	415.45	424.65	413.13	432.88	(431.75, 434.00)
Hera (Class C)	2	81.97	86.74	96.02	87.11	84.55	98.70	(98.60, 98.80)
	4	58.37	60.22	62.14	47.13	45.05	59.34	(59.29, 59.39)
	8	32.18	32.90	33.70	27.26	14.66	40.27	(40.20, 40.34)
Hera (Class D)	8	472.67	539.25	561.55	428.58	417.49	461.43	(461.26, 461.59)
	16	281.01	283.41	285.73	227.06	218.73	262.50	(262.44, 262.55)
	32	192.40	195.52	197.35	122.42	115.51	160.46	(160.36, 160.57)
	64	114.59	122.11	131.30	67.60	64.19	107.80	(107.68, 107.91)
DawnDev (Class C)	32	49.76	49.81	49.91	55.50	55.87	60.46	(60.43, 60.48)
	64	29.11	29.12	29.14	31.20	31.34	34.99	(34.97, 35.00)
	128	19.55	19.56	19.56	19.26	19.24	22.7	(22.85, 22.88)
	256	14.39	14.50	14.58	12.12	11.97	15.13	(15.11, 15.14)
DawnDev (Class D)	32	736.84	736.84	736.85	720.84	723.66	745.10	(745.06, 745.13)
	64	386.34	386.40	386.47	379.11	381.37	398.58	(398.54, 398.63)
	128	217.43	217.63	217.93	200.86	201.87	217.64	(217.61, 217.67)
	256	123.60	123.97	124.20	107.33	107.76	119.72	(119.69, 119.75)

Table 7.1: Model and simulation validations for LU. Execution times are given in seconds.

Device	**Compiler**	**Options**

BlueGene/P	IBM XLF	`-O5 -qhot -Q`
		`-qipa=inline=auto`
		`-qipa=inline=limit=32768`
		`-qipa=level=2`
		`-qunroll=yes`

AMD Opteron	PGI 8.0.1	`-O4 -tp barcelona-64`
		`-Mvect=sse -Mscalarsse`
		`-Munroll=c:4 -Munroll=n:4`
		`-Munroll=m:4 -Mpre=all`
		`-Msmart -Msmartalloc`
		`-Mipa=fast,inline,safe`

Table 7.2: Compiler configurations for the LU model validation.

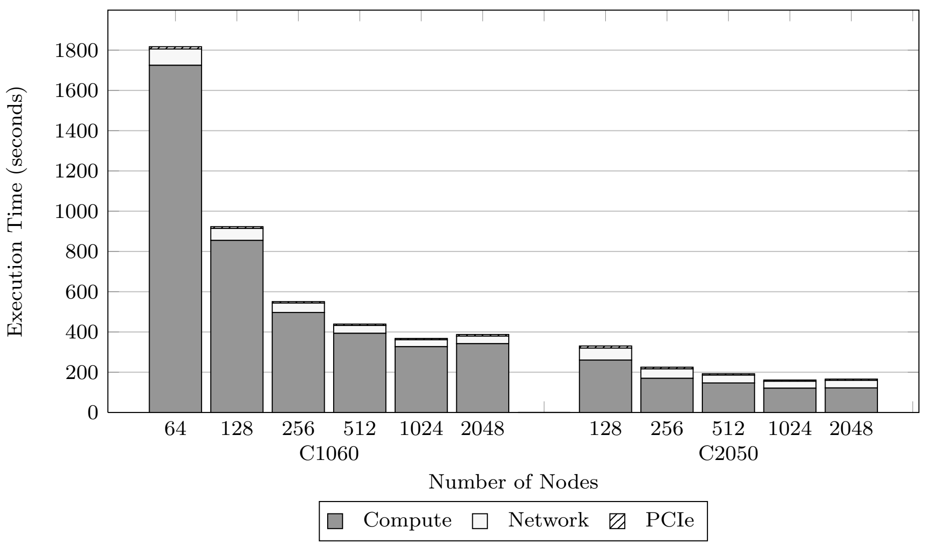


Figure 7.3: Breakdown of execution times for LU from the GPU model.

7.2 Communication Breakdown

The graph in Figure 7.3 shows a breakdown of the execution times for a Class E problem on different numbers of GPUs in terms of compute, network communication and PCIe transfer times. Even at scale, the biggest contributor to execution time is computation, followed by network communication and finally PCIe transfers. These results are surprising, but reflect the optimised nature of our GPU implementation of LU, which transfers the minimum amount of data across the PCIe bus at each communication step.

7.3 Machine Comparison

7.3.1 Scalability

Weak Scaling

We investigate how the time to solution varies with the number of processors for a fixed problem size *per processor*. This allows us to assess the suitability of processors for *capacity* clusters, by exposing the cost of adding extra nodes to solve increasingly large problems.

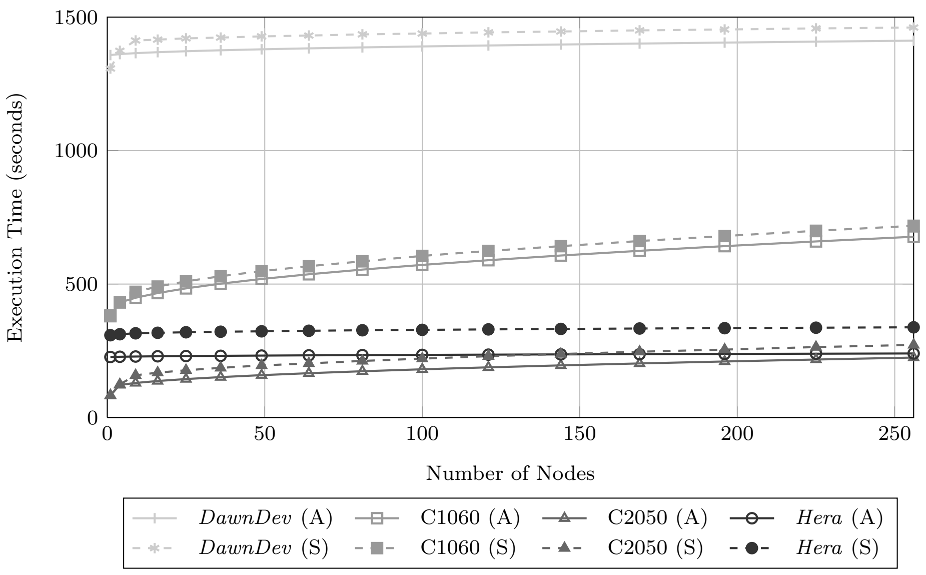


Figure 7.4: Weak-scaling projections for LU.

The reader is reminded that LU operates on grids of size N^3 and uses a 2D domain decomposition across processors. This renders the seven verifiable problem classes unusable in a weak-scaling study; it is impossible to fix the subdomain of each processor at a given number of grid-points ($N_x/P_x \times N_y/P_y \times N_z$) whilst increasing N_x, N_y and N_z. To compensate for this, we fix the height of the problem grid to 1024. Although this prevents us from verifying the solution reported against a known solution, we are still able to verify that both implementations produce the *same* solution. The number of grid-points per node is set to $64 \times 64 \times 1024$, as this provides each GPU with a suitable level of parallelism.

Figure 7.4 shows model projections for execution times up to a maximum problem size of $1024 \times 1024 \times 1024$ (corresponding to 256 nodes in each cluster). Both the analytical model (A) and the simulation with noise applied (S) provide similar runtime projections, as demonstrated by the close performance curves. Typically, if a code exhibits good weak scalability, then the execution time will remain largely constant as additional nodes are added. Our results show that, across all the architectures studied, the execution time of LU increases with the number of nodes – a side-effect of the wavefront dependency. As the number of nodes increases, so too does the pipeline fill time of each wavefront sweep.

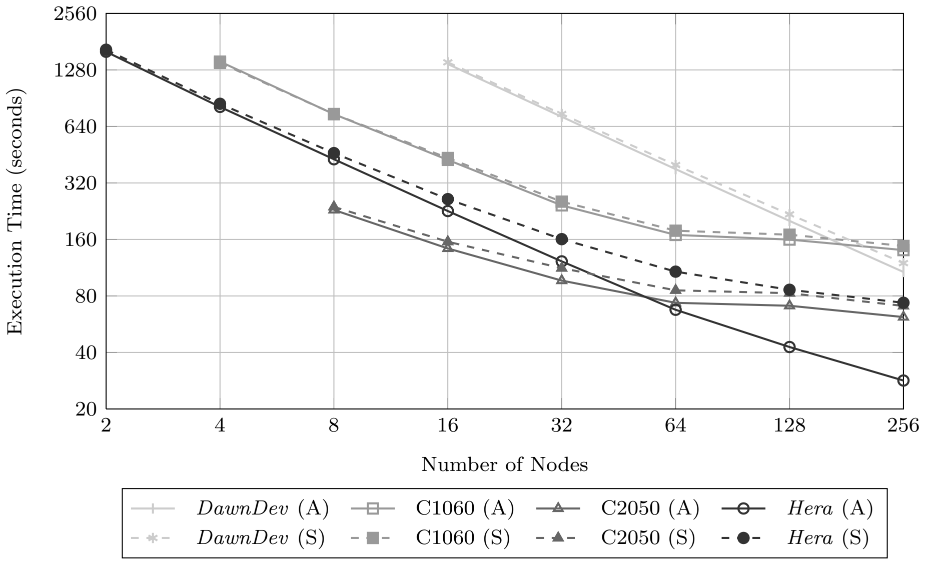
It is apparent from the graph that the weak scalability of our GPU implementation is worse than that of its CPU counterparts. This is due to the selection of a relatively large k_B for the GPU implementation, which we have shown in previous chapters is necessary for good parallel efficiency. Since each GPU must process more grid-points than a CPU prior to communication with its neighbours, the addition of an extra node has a larger effect on pipeline fill time; the same situation arises if a large k_B value is chosen for the CPU implementation. Increased communication times (due to PCIe transfers) will also increase pipeline fill time.

The results in this section suggest that accelerators are a suitable architectural choice for capacity clusters. Other scientific and engineering applications may not weak scale as poorly as wavefront applications, and the increase in per node performance afforded by accelerators persists during weak scaling.

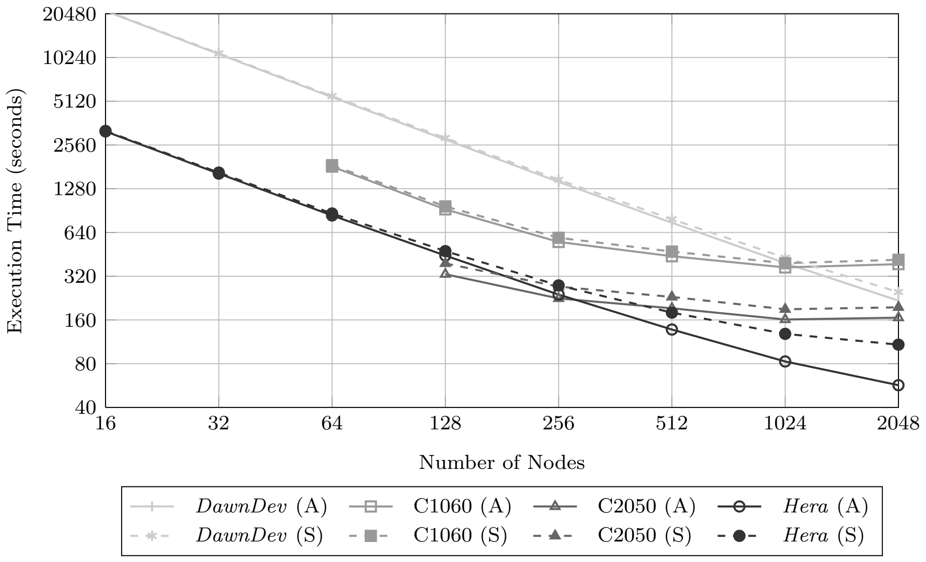
Strong Scaling

We investigate how the time to solution varies with the number of processors for a fixed *total* problem size, demonstrating the utility of adding an extra processor for the acceleration of a given problem. This will be of interest when employing *capability* clusters.

As the total problem size is fixed, we do not encounter the same problems as we did with the weak-scaling study (*i.e.* we are able to use the standard problem classes). Figure 7.5 therefore shows analytical model and simulation projections for the execution times of (a) Class D and (b) Class E problems, for increasing node counts. The modelled cluster of C2050 GPUs provides the best performance at small scale for both problem sizes. However, *DawnDev* and *Hera* demonstrate higher levels of scalability; the execution times for the CPU-based clusters continue to decrease as the number of nodes is increased, whereas those for the GPU-based clusters tend to plateau at a relatively low



(a) Class D



(b) Class E

Figure 7.5: Strong-scaling projections for LU.

	Nodes	**Time (s)**	**Power Consumption**		**Theoretical Peak (TFLOP/s)**
			Compute (kW)	**Total (kW)**	
C1060	1024	367.84	192.31	286.72	79.87
C2050	256	224.98	60.93	81.92	131.84
DawnDev	2048	217.32	32.77	69.96	27.85
Hera	256	239.12	97.28	137.00	38.44

Table 7.3: Cluster comparison for executing LU (Class E) within a fixed execution time.

number of nodes.

The results in this section suggest that accelerators are not necessarily a suitable architectural choice for capability clusters, and demonstrates the importance of considering multi-node performance during the evaluation of emerging architectures. Despite the high single-node performance of our GPU-accelerated implementation, its strong scalability is limited; we see very minor speed-ups from the introduction of additional nodes. We acknowledge that there are inherent issues with the strong scalability of pipelined wavefront applications (*e.g.* the pipeline fill), which limit the ability of these codes to achieve high levels of performance at scale even on traditional CPU architectures (other studies report a percentage of peak around 5–10% on CPUs [74]). However, accelerators are additionally affected by the decrease in problem size per node, which results in a decreasing amount of exploitable parallelism – as shown in Chapters 4 and 5, accelerators lose their advantage over traditional CPU architectures for small problem sizes. This issue is not specific to the codes studied in this thesis.

7.3.2 Power Consumption

In addition to performance, another important metric to consider when comparing machines of this scale is power consumption, since this has a significant impact on their total cost. Therefore, Table 7.3 lists the power consumption and theoretical peak for each of the four machines modelled, for a fixed execution time of a Class E problem. We present two different power figures for each machine: firstly, the TDP of the compute devices used, to represent the maximum amount of power each architecture *could* draw to perform computation on the devices employed (and hence the amount of power the system must be provided with in the worst case); and secondly, benchmarked power consumption, to represent the amount of power each architecture is likely to draw in reality and also account for the power consumption of other hardware in the machine.

In the case of the Tesla C1060 and C2050 machines, the power consumption of an HP Z800-series compute blade utilising a single Intel X5550 Nehalem processor and one GPU was recorded during runs of a Class C problem. The figure listed therefore represents the power usage of the entire blade: one Tesla C1060 or C2050 GPU, a single quad-core processor, 12GB of system memory and a single local disk. It does not include the power consumption for a high-performance network interconnect. For the supercomputers at Lawrence Livermore National Laboratory (to which we do not have physical access), the benchmarked figures are the mean recorded power consumption during typical application and user workloads [87].

Of the four machines, *DawnDev* has the lowest power consumption and lowest theoretical peak, yet also achieves the lowest execution time. The C2050 cluster, on the other hand, has the second lowest power consumption and achieves the second lowest execution time, yet has the highest theoretical peak. This demonstrates that although GPU-based solutions are considerably more space- and power-efficient than commodity CPU clusters, integrated solutions (such as IBM's Blue Gene) afford even higher levels of efficiency and scalability. Furthermore, the level of sustained application performance offered by GPU clusters is closer than expected to that of existing cluster technologies – and lower as a percentage of peak.

7.4 Summary

This chapter presents the most comprehensive evaluation of LU's performance on emerging architectures to be published to date. We use two recently developed application performance models to project benchmark results from a single node to larger systems and to deconstruct the execution times of our solution, allowing us to examine the proportions of runtime accounted for by communication between nodes and the extent to which PCIe communication is a bottleneck.

While distributed accelerator clusters can deliver substantial levels of theoretical peak performance, achieving sustained application performance at scale is still a challenge. Like-for-like comparisons to existing technologies such as IBM's Blue Gene platform also help to show that the power-efficiency of GPU solutions – a much cited reason for their adoption – is in fact comparable for this class of application. Our results also show that, for wavefront applications at least, CPU and GPU-accelerated clusters currently offer comparable levels of performance in spite of large differences in theoretical peak. The techniques employed in this work demonstrate a low-cost and accurate method of assessing application performance on contemporary and future HPC systems, and our results emphasise the importance of considering scale in application and machine design, in contrast to benchmarking on single nodes or small clusters.

CHAPTER 8

Conclusions and Future Work

The gap between the level of performance achieved by legacy applications on current-generation hardware and the *potential* performance of the same applications following optimisation is significant, and demonstrated by other research to be as large as an order-of-magnitude for some workloads [12, 29, 78, 117, 145, 150, 162]. This gap will grow if left unchecked – trends suggest that SIMD widths will continue to increase, along with the number of cores, and exascale machines are predicted to have several orders of magnitude more parallelism than current generation petascale machines [38].

The results in this thesis (specifically, those in Chapters 4 and 5) further highlight the existence of this performance gap, but also demonstrate that a number of optimisations are in fact common to multiple architecture types. They also show that two applications which may initially appear poorly suited to modern architectures, due to complex memory access patterns and data dependencies, can in fact benefit from the increasing levels of parallelism available in recent micro-architecture designs. Even if the programming paradigm changes (which we expect that it will), developers that explore optimisation opportunities today are likely to have an easier task when exascale computing arrives; those that do not, will likely see very poor performance on tomorrow's machines.

However, many HPC sites have traditionally been reluctant to tune their codes, lest they become tied to a particular vendor; the ability to maintain code is commonly seen as more important than achieving the best possible performance. The results in Chapter 6 suggest that these goals are not mutually exclusive, demonstrating that it is possible to develop and optimise an application (with a single source code) for multiple platforms simultaneously. In our experience, OpenCL is also better suited to expressing parallelism and vectorisation opportunities than traditional serial languages – for example, we show that the Intel compiler is able to generate efficient AoS-to-SoA transpose routines that have to be hand-coded in intrinsics in C – and our OpenCL implementations of the LU and miniMD benchmarks are consistently faster than the original scalar codes. Although we acknowledge that it is unlikely that such platform-agnostic codes will exceed the performance of heavily optimised "native" implementations, the performance overhead we see (relative to our best efforts) is much smaller than one might expect (2x). The ability of OpenCL compilers to map work to hardware is likely to increase as compilers mature, and we therefore expect this overhead to shrink over time.

In Chapter 7 we demonstrate that, although single-node benchmarking of emerging architectures can deliver meaningful initial results, it is important to consider the performance of applications running at scale. In particular, we highlight the problems associated with strong-scaling on accelerator architectures, due to the decreasing amount of exploitable parallelism per node. This result suggests that there remains a place in HPC for serial, high-clocked cores, and further motivates the need to write codes that adapt to multiple architectures. Most accelerators are (currently) paired with traditional CPU cores, and some CPUs even feature accelerator cores on the same die – codes should be written in a way that enables them to fall back to traditional architectures for serial tasks (and parallel tasks run at extreme scales) in order to make the best use of these new heterogeneous architectures.

Taken together, the work presented in this thesis details a methodology for the evaluation of scientific and engineering application performance on emerging parallel architectures. This three-step process, consisting of: (*i*) code optimisation for a new architecture; (*ii*) incorporation of new optimisations into a platform-agnostic implementation; and (*iii*) projection of performance at scale; demonstrates a clear development path for HPC sites seeking to maximise the performance of their applications on current- and future-generation hardware, without wasting considerable effort maintaining multiple code-paths. Assuming that hardware vendors continue to support the OpenCL standard (or an equivalent open standard) and that the standard continues to incorporate new hardware features, we believe that the use of single-source applications of the kind we describe will bridge the gap between architectures, giving HPC sites much greater vendor mobility, improving code maintainability across hardware generations and significantly easing the process of benchmarking new architectures.

8.1 Limitations

The primary limitation of this thesis is its focus upon two particular scientific applications, specifically the LU and miniMD benchmarks. Although this may limit the generality of the optimisations and programming techniques presented, these codes were chosen because they are both representative of much larger and more complex codes; LU of applications like Sweep3D [1] and Chimaera [111], and miniMD of the popular LAMMPS package [137, 139]. Further, the parallel behaviours of these codes are common to a wider range of scientific applications, and our results thus have implications for scientific and engineering codes from other problem domains (*e.g.* unstructured mesh, computational fluid dynamics).

A secondary limitation is that much of the work in this thesis evaluates proposed optimisations and programming methodologies on a specific range of hardware. Hardware and software both develop at an alarming rate, and as such many of the systems used in this thesis have since been decommissioned, or upgraded in ways that may lessen the impact of our optimisations. Nevertheless, our results provide an insight into the state of various hardware and software packages at the time that the experiments were performed – and insight into the huge upheaval in parallel architectures at the time this research was undertaken. Further, we believe that our experiences are typical of those that will be experienced for any new architecture, and that the methodology we describe for porting legacy codes (or benchmarks) to new systems will be a useful case study for many HPC sites.

Another potential limitation is our use of the OpenCL programming model for our performance portability study. At the time of writing, OpenCL host code must be written in C and C++, with device kernels written in OpenCL C (an extension of C99); since the majority of legacy HPC codes are written in Fortran, developers may be reluctant to adopt this model. However, the most recent version of the OpenCL standard introduces a Standard Portable Intermediate Representation (SPIR) [77] designed as a possible target for compilers of languages other than OpenCL C – we feel it is likely that other researchers (or compiler developers) will leverage this SPIR to allow OpenCL kernels to be written in Fortran.

The final limitation is that our optimisations focus on reducing time-to-solution (by increasing arithmetic throughput). These metrics are arguably the most important for the users of an HPC resource, as they have the most direct impact on their ability to run simulations. However, there are a number of additional metrics that we do not consider which are likely to be of interest to the maintainers of such systems, such as machine size, cost and reliability. Many of these are linked to machine performance (some indirectly), and could thus be extrapolated from execution times – we demonstrate this for power consumption in Chapter 7. The degree to which a code base is considered maintainable (and readable) is another important concern, particularly when dealing with legacy applications, but this is subjective and difficult to measure quantitatively (*e.g.* counting lines of code does not account for code complexity, or developer experience).

8.2 Future Work

There are a number of potential avenues for future research building on the work presented in this thesis. Specifically, we describe two optimisations (one for each of the benchmarks studied) that require further validation on future architectures; and discuss the need to re-evaluate our proposed optimisations in the context of production applications.

8.2.1 k-blocking in Wavefront Applications

The size of the hyperplane in a wavefront application starts at one grid-point, increases until some maximum (dependent on problem decomposition and k_B), and then decreases until it has passed through the last grid-point. Following communication, the next time step executes another "mini-sweep" for the next k-block, again starting from a single grid-point. Conducting computation in this manner wastes parallel efficiency whenever the size of the hyperplane is less than the number of work-items an architecture can execute in parallel.

To address this issue, we propose a new k-blocking algorithm that permits a processor to operate on grid-points beyond the current k-block boundary, thus maintaining the angle established during the start of the "mini-sweep". The parallel and SIMD efficiency achieved per node under this new k-blocking policy is the same as that seen when using a k_B value of N_z under the old policy, but permits processors to communicate prior to the completion of their entire $N_x/P_x \times N_y/P_y \times N_z$ volume.

Early Results

Table 8.1 compares the execution times of the old and new k-blocking policies, when this optimisation is applied to LU for a Class C problem on 4, 8 and 16 GPUs. We see that the performance of the new policy is significantly better than the old policy for small values of k_B, but slightly worse for large values. This performance gap is the result of a combination of factors: increased pipeline fill

Nodes	k_B	Old Policy	New Policy
4	1	759.99	80.81
	81	67.25	77.13
8	1	554.34	74.94
	41	52.92	61.34
16	1	381.06	77.50
	41	44.46	55.60

Table 8.1: Comparison of execution times (in seconds) for LU (Class C) using the old and new k-blocking policies.

time (due to increased communication costs); the cubic nature of LU problems (which limits the number of wavefront steps operating at 100% efficiency); and potentially unforeseen issues in our implementation of LU.

Future Architectures

To investigate the potential of our new k-blocking algorithm on future hardware with wider SIMD, and to study its performance for other wavefront applications besides LU, we again make use of the reusable wavefront model of Mudalige *et al.* [107].

Old Policy:

$$T_{\text{comm}} = \left[\left(P_x + P_y + \frac{N_z}{k_B} - 2\right) - 1\right] \times T_c \tag{8.1}$$

$$T_{\text{comp}} = \left(P_x + P_y + \frac{N_z}{k_B} - 2\right) \times \left[\left(\frac{N_x}{P_x} + \frac{N_y}{P_y} + k_B - 2\right) \times T_h\right] \tag{8.2}$$

New Policy:

$$M = \frac{\min(N_z, \frac{N_x}{P_x} + \frac{N_y}{P_y} + k_B - 2)}{k_B} \tag{8.3}$$

$$T_{\text{comm}} = \left[M \times (P_x + P_y - 3) + \left(\frac{N_z}{k_B}\right)\right] \times T_c \tag{8.4}$$

$$\begin{aligned} T_{\text{comp}} = &\left[(P_x + P_y - 1) \times \left(\frac{N_x}{P_x} + \frac{N_y}{P_y} + k_B - 2\right) \times T_h\right] \\ &+ \left[\left(\frac{N_z}{k_B} - 1\right) \times k_B \times T_h\right] \end{aligned} \tag{8.5}$$

where T_c represents some communication cost (as before), and where W is pre-

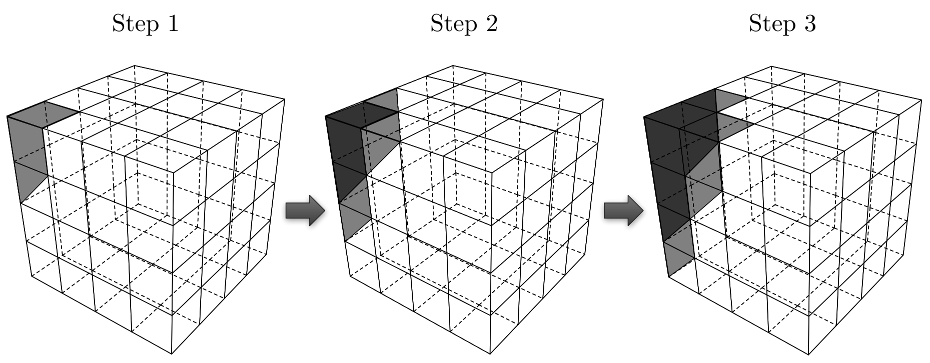


Figure 8.1: First three wavefront steps for the new k-blocking policy.

dicted analytically based on the computational cost per hyperplane (T_h). Modelling each hyperplane step as requiring the same time is arguably unrealistic, but is representative of two different situations: (i) a large problem, decomposed over many processors (where there is little exploitable parallelism); and (ii) using processors with infinite SIMD width (the best case for both policies).

The difference between these two models is quite simple. On the compute side, the old policy executes a coarse wavefront (with $P_x + P_y + (N_z/k_B) - 2$ steps) over blocks of size $N_x/P_x \times N_y/P_y \times k_B$; the new policy executes a similar wavefront but pays the ramp-up cost ($N_x/P_x + N_y/P_y + k_B - 2$ hyperplanes) only once on each processor – the subsequent $N_z/k_B - 1$ blocks require only k_B hyperplanes. On the communication side, the old policy sees each processor receive only one message before it is able to complete its first k-block; the new policy, as shown in Figure 8.1, requires each processor to instead receive M messages. This change in communication behaviour is due to the fact that a processor receiving k_B tiles can only execute k_B hyperplanes; a processor must thus receive $N_x/P_x + N_y/P_y + k_B - 2$ tiles from upstream before it is able to complete its first k-block.

These models predict that there will be some situations in which we should expect our new algorithm to out-perform the original. Specifically, there is much greater potential for improved performance in problems where N_z is significantly larger than N_x or N_y, such that the cost of the pipeline fill is amortised; and in situations where computational cost far outweighs that of communication. We leave the application of this optimisation to other codes, and the validation of these models, to future work.

8.2.2 Conflict Resolution in Molecular Dynamics

The SIMD analysis in Chapter 5 assumes that the number of neighbours per atom is sufficiently large (compared to the SIMD width) that the amount of padding required to make the number of neighbours a multiple of W is small. This holds true for miniMD's default simulation (≈ 28 neighbours per atom) on the hardware we use, but architectures with wider SIMD, or simulations with very small cut-off distances, require a different approach to achieve high efficiency.

There are typically thousands of atoms in a molecular dynamics simulation, and thus moving to "cross-atom" SIMD (*i.e.* vectorising the loop over atoms) exposes significantly more parallelism than a "cross-neighbour" approach – and sufficient parallelism for many hardware generations to come. For the force compute loop, using SIMD in this fashion results in two changes compared to our previous code: (i) gathers and scatters are potentially required at the level of the outer loop, to accumulate forces into atoms; and (ii) there is a potential for update conflicts, since several atoms may share a neighbour – if we attempt to update the same neighbour multiple times simultaneously, only one of the updates will take effect.

In the absence of fast atomic gather-scatter operations [85], we propose that update-conflicts be handled by arranging the neighbour lists such that there are no duplicate indices in any set of W neighbours. Detecting duplicates within a group of W neighbours requires $O(W^2)$ comparisons but, since comparison hardware typically scales with SIMD width, we expect the number of needed SIMD comparison instructions to be a more tractable $O(W)$.

Our algorithm for conflict resolution is as follows. For every set of W indices, check for a conflict (*i.e.* a duplicate index). If there are no conflicts, then this

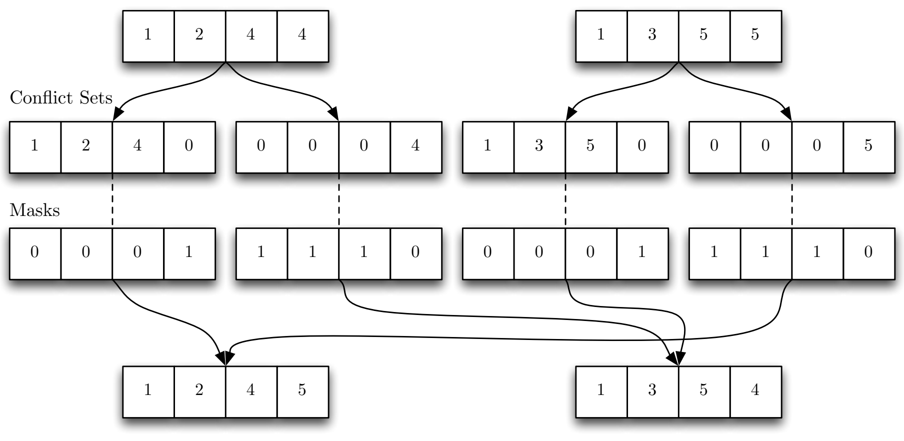


Figure 8.2: Resolving conflicts using 128-bit SIMD.

set of indices can be written to the neighbour list. If there are conflicts, we split the set into at most W parts: one containing all of the indices that do not conflict, and up to $W - 1$ sets containing one index each. As before, we insert "dummy" neighbours (index 0) located at infinity as necessary, for padding. These sets are then stored as conflict sets, along with a bit-mask denoting the location of 0 indices. When all neighbour sets have been considered, conflict sets are matched based upon their masks, combined if possible, and written to the neighbour list. Figure 8.2 demonstrates this process for 128-bit SIMD, for two sets of W indices with a single conflict each.

The operation of this conflict resolution algorithm is orthogonal to the neighbour list build itself, and so we implement it as a post-processing step that removes conflicts from an existing neighbour list. Since miniMD's neighbour list introduces an imbalance in the number of neighbours per atom, and the amount of computation required for a group of W atoms is dependent upon the maximum number of neighbours within the group, we also sort "windows" of atoms according to their number of neighbours.

Window Size	**256k**			**2048k**		
	1.5	**2.5**	**5.0**	**1.5**	**2.5**	**5.0**
Force Compute						
1	0.89x	1.21x	1.25x	0.89x	1.20x	1.24x
64	0.73x	0.98x	1.05x	0.77x	0.98x	1.05x
infinite	0.84x	1.17x	1.16x	1.91x	3.03x	2.93x
Total Simulation						
1	1.09x	1.30x	1.35x	1.04x	1.24x	1.37x
64	0.96x	1.06x	1.15x	0.96x	1.04x	1.15x
infinite	0.99x	1.17x	1.21x	1.50x	2.32x	2.72x

Table 8.2: Slow-down of conflict resolution approach.

Early Results

The results in Table 8.2 give the slow-down (relative to our best cross-neighbour SIMD approach) of this cross-atom approach with conflict resolution on a CPU with 256-bit AVX. We include results for three different window sizes, and six different simulations: 256k and 2048k atoms, with R_c set to 1.5, 2.5 and 5.0 (giving an average of 6, 28 and 221 neighbours respectively). The window size clearly has a significant impact upon the performance of the force compute: a window size of 1 (*i.e.* not sorting atoms) exposes the cost of the imbalance in neighbour list lengths, which increases with the size of the cut-off; an infinite window size (*i.e.* completely sorting atoms based on their number of neighbours) results in the least imbalance but significantly worse performance due to poor cache locality, particularly for simulations with a large number of atoms; and a window size of 64 strikes a balance between the two, consistently giving the best performance.

Even with a window size of 64, our conflict resolution approach is 4–15% slower overall for the larger two cut-off distances. This is not surprising; cut-off distances of 2.5 and 5.0 have sufficient cross-neighbour parallelism that cross-atom SIMD does not improve SIMD efficiency, and so the only significant change to execution time comes from the overhead of conflict resolution. For a cut-off of 1.5, however, where atoms have fewer neighbours than the SIMD width, we see a significant speed-up for the force compute (1.37x). The speed-up is less overall

	128-bit	**256-bit**	**512-bit**	**1024-bit**	**2048-bit**
Cross-Neighbour	3.19%	9.36%	18.21%	45.04%	71.60%
Cross-Atom (No CR)	1.44%	2.45%	4.51%	10.63%	26.28%
Cross-Atom (CR)	1.69%	3.50%	6.55%	13.74%	33.21%

Table 8.3: Inefficiency of cross-neighbour and cross-atom SIMD.

(1.04x), but this is because the contribution of force compute to execution time is lower for this simulation.

Table 8.3 shows the percentage increase in the number of neighbours (due to padding) for cross-neighbour SIMD and cross-atom SIMD, using a cut-off of 2.5 and a window of 64. For cross-atom SIMD, we give the amount of padding before and after conflict resolution (no CR and CR). The cross-atom padding is much lower than the cross-neighbour padding, but the inefficiency of both approaches grows with SIMD width: for cross-neighbour, more padding is required to reach a multiple of W; for cross-atom, (i) the conflict rate increases with SIMD width, requiring more padding to resolve conflicts, and (ii) the imbalance in the number of neighbours per atom increases. For (i), if we compare the pre- and post-conflict resolution padding amounts, we see that resolving conflicts introduces little padding. For (ii), the amount of padding is modest except for 2048-bit SIMD, but this is because our sorting window size is the same as W for 2048-bit SIMD – a larger window is needed for wider SIMD.

Future Architectures

Implementations that use threading must also guarantee the independence of tasks allowed to execute simultaneously. Our conflict resolution approach could be used to guarantee independence beyond a single thread, but this is not practical for a many-core architecture; for KNC, we would need to resolve conflicts for 3840 parallel tasks (60 cores $\times$ 4 threads $\times$ 16 SIMD units). It is thus unclear how best to map this conflict resolution algorithm to a programming language like OpenCL, since the notions of SIMD and threading are very similar. Most probably, we would use a number of work-groups of W work-items each, where each work-group is treated as a thread and the W work-items are assumed to be synchronous. The approach could apply to larger work-group sizes, but this would require additional local synchronisation between threads, which may prove too expensive.

That a post-processing step such as this one can be performed efficiently on current hardware, and results in a speed-up where expected, demonstrates that it is a suitable method for handling SIMD update-conflicts in molecular dynamics simulations. However, we leave the implementation of this algorithm within OpenCL (and hence its evaluation on GPU architectures) to future work.

8.2.3 Optimisation of Production Applications

The optimisations that we propose in this work, along with our single-source development and performance modelling techniques, are evaluated exclusively in the context of macro-benchmarks. There is a clear need to demonstrate that the same process can be applied to a production application – however, due to the complexity and size of these applications, such an undertaking is beyond the scope of this thesis.

The application that LU is based upon is maintained by NASA, so any further work in this space will likely have to be carried out in conjunction with them – or through collaboration with the maintainers of another scientific wavefront application, such as the UK Atomic Weapons Establishment. The exploration of our SIMD and threading optimisations within the context of LAMMPS is more accessible, since it is open-source and modular by design. There are already several user-maintained “packages” targeting different programming paradigms (*e.g.* OpenMP, CUDA and OpenCL), and work that is currently ongoing seeks to add support for our SIMD and MIC implementations to LAMMPS in this way.

Bibliography

[1] The ASCI Sweep3D Benchmark. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html, 1995.

[2] OpenACC 1.0 Specification. http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, November 2011.

[3] Message Passing Interface Forum. http://www.mpi-forum.org/, October 2012.

[4] OpenMP 4.0 Specification. http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf, November 2012.

[5] Technical Report on Directives for Attached Accelerators. Technical Report TR1, OpenMP Architecture Review Board, Champaign, IL, November 2012.

[6] V. S. Adve. *Analyzing the Behavior and Performance of Parallel Programs.* PhD thesis, University of Wisconsin-Madison, 1993.

[7] V. S. Adve and M. K. Vernon. Performance Analysis of Mesh Inteconnection Networks with Deterministic Routing. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):225–246, 1994.

[8] J. H. Ahn, M. Erez, and W. J. Dally. Scatter-Add in Data Parallel Architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 132–142, San Francisco, CA, February 2005. IEEE Computer Society.

[9] A. M. Aji and W. C. Feng. Accelerating Data-Serial Applications on GPGPUs: A Systems Approach. Technical Report TR-08-24, Blacksburg, VA, 2008.

[10] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages Into the LogP Model – One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, Santa Barbara, CA, July 1995. ACM.

[11] J. A. Anderson, C. D. Lorenz, and A. Travesset. General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

[12] N. Arora, A. Shringarpure, and R. W. Vuduc. Direct N-body Kernels for Multicore Platforms. In *Proceedings of the International Conference on Parallel Processing*, ICPP '09, pages 379–387, Vienna, Austria, September 2009. IEEE Computer Society.

[13] S. Artemova, S. Grudinin, and S. Redon. A Comparison of Neighbor Search Algorithms for Large Rigid Molecules. *Journal of Computational Chemistry*, 32(13):2865–2877, 2011.

[14] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '10, pages 105–114, Bangalore, India, January 2010. ACM.

[15] D. Bailey et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[16] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[17] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, pages 163–174, Boston, MA, April 2009. IEEE.

[18] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden. Extending a Traditional Debugger to Debug Massively Parallel Applications. *Journal of Parallel and Distributed Computing*, 64(5):617–628, 2004.

[19] S. Biswas, B. R. de Supinski, M. Schulz, D. Franklin, T. Sherwood, and F. T. Chong. Exploiting Data Similarity to Reduce Memory Footprints. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, pages 152–163, Anchorage, AK, May 2011. IEEE Computer Society.

[20] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, Santa Barbara, CA, 1995. ACM.

[21] R. Bordawekar, U. Bondhugula, and R. Rao. Believe it or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application! Technical Report RC24982, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, 2010.

[22] R. Bordawekar, U. Bondhugula, and R. Rao. Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU. Technical Report RC25033, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, 2010.

[23] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '09, Rome, Italy, May 2009. IEEE Computer Society.

[24] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington. Implementing Molecular Dynamics on Hybrid High Performance Computers – Short Range Forces. *Computer Physics Communications*, 182:898–911, 2011.

[25] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.

[26] J. Cao, D. Kerbyson, E. Papaefstathiou, and G. Nudd. Performance Modeling of Parallel and Distributed Computing Using PACE. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference*, IPCCC '00, pages 485–492, Phoenix, AZ, February 2000. IEEE.

[27] D. Case et al. AMBER 11. `http://www.ambermd.org`, May 2011.

[28] L. Chai, A. Hartono, and D. Panda. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER '06, pages 1–10, Barcelona, Spain, September 2006. IEEE Computer Society.

[29] J. Chhugani, C. Kim, H. Shukla, J. Park, P. Dubey, J. Shalf, and H. D. Simon. Billion-Particle SIMD-Friendly Two-Point Correlation on Large-Scale HPC Cluster Systems. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage*

and Analysis, SC '12, pages 1–11, Salt Lake City, UT, November 2012. IEEE Computer Society.

[30] S.-H. Chiang and M. K. Vernon. Characteristics of a Large Shared Memory Production Workload. In *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP '01, pages 159–187, Cambridge, MA, June 2001. Springer-Verlag.

[31] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A Parallel Functional Simulator for GPGPU. In *Proceedings of the IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, MASCOTS '10, pages 351–360, Miami, FL, August 2010. IEEE Computer Society.

[32] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, San Diego, CA, May 1993. ACM.

[33] A. Danalis, P. Luszczek, J. Dongarra, G. Marin, and J. S. Vetter. BlackjackBench: Portable Hardware Characterization. In *Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, PMBS '11, pages 7–8, Seattle, WA, November 2011. ACM.

[34] J. A. Davis, G. R. Mudalige, S. D. Hammond, J. A. Herdman, I. Miller, and S. A. Jarvis. Predictive Analysis of a Hydrodynamics Application on Large-Scale CMP Clusters. *Computer Science - Research and Development*, 26(3–4):175–185, June 2011.

[35] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-Based PDE Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, Seattle, WA, 2011. ACM.

[36] G. Diamos et al. GPU Ocelot. `http://code.google.com/p/gpuocelot/`, 2012.

[37] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Proceedings of the Workshop on General-Purpose Processing on Graphics Processing Units*, GPGPU '07, Boston, MA, October 2007.

[38] J. Dongarra et al. The International Exascale Software Project Roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[39] J. J. Dongarra and A. R. Hinds. Unrolling Loops in FORTRAN. *Software – Practice and Experience*, 9:219–226, 1979.

[40] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present and Future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[41] R. O. Dror et al. Exploiting 162-Nanosecond End-to-End Communication Latency on Anton. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, New Orleans, LA, November 2010. ACM.

[42] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-platform GPU Programming. Technical Report UT-CS-10-656, Knoxville, TN, 2010.

[43] Q. F. Fang, R. Wang, and C. S. Liu. Movable Hash Algorithm for Search of the Neighbor Atoms in Molecular Dynamics Simulation. *Computational Materials Science*, 24:453–456, 2002.

[44] C. Garcia, R. Lario, M. Prieto, L. Pinuel, and F. Tirado. Vectorization of Multigrid Codes Using SIMD ISA Extensions. In *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS '03, Nice, France, April 2003. IEEE Computer Society.

[45] D. Geer. Chip Makers Turn to Multicore Processors. *Computer*, 38(5):11–13, May 2005.

[46] M. Giles, G. Mudalige, B. Spencer, C. Bertolli, and I. Reguly. Designing OP2 for GPU Architectures. *Journal of Parallel and Distributed Computing*, 2012 (to appear).

[47] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly. Performance Analysis of the OP2 Framework on Many-core Architectures. *SIGMETRICS Performance Evaluation Review*, 38(4):9–15, 2011.

[48] C. Gong, J. Liu, Z. Gong, J. Qin, and J. Xie. Optimizing Sweep3D for Graphic Processor Unit. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '10, pages 416–426, Busan, Korea, May 2010.

[49] P. Gonnet. A Simple Algorithm to Accelerate the Computation of Non-Bonded Interactions in Cell-Based Molecular Dynamics Simulations. *Journal of Computational Chemistry*, 28(2):570–573, 2007.

[50] P. Gonnet. Pairwise Verlet Lists: Combining Cell Lists and Verlet Lists to Improve Memory Locality and Parallelism. *Journal of Computational Chemistry*, 33(1):76–81, 2012.

[51] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '08, Austin, TX, November 2008. IEEE/ACM.

[52] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 120–126, Boston, MA, June 1982. ACM.

[53] W. Gropp and E. L. Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Proceedings of the European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI '99, pages 11–18, Barcelona, Spain, 1999. Springer-Verlag.

[54] D. A. Grove and P. D. Coddington. Communication Benchmarking and Performance Modelling of MPI Programs on Cluster Computers. *The Journal of Supercomputing*, 34:201–217, 2005. 10.1007/s11227-005-2340-2.

[55] S. D. Hammond. *Performance Modelling and Simulation of High Performance Computing Systems*. PhD thesis, University of Warwick, 2011.

[56] S. D. Hammond et al. WARPP: A Toolkit for Simulating High-Performance Parallel Scientific Codes. In *Proceedings of the International Conference on Simulation Tools and Techniques*, SimuTools '09, Rome, Italy, March 2009. ICST.

[57] S. D. Hammond, G. R. Mudalige, J. A. Smith, A. B. Mills, S. A. Jarvis, J. Holt, I. Miller, J. A. Herdman, and A. Vadgama. Performance Prediction and Procurement in Practice: Assessing the Suitability of Commodity Cluster Components for Wavefront Codes. *IET Software*, 3(6):509–521, 2009.

[58] S. S. Hampton, S. R. Alam, P. S. Crozier, and P. K. Agarwal. Optimal Utilization of Heterogeneous Resources for Biomolecular Simulations. In

Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–11, New Orleans, LA, November 2010. IEEE/ACM.

[59] M. J. Harvey, G. Giupponi, and G. De Fabritiis. ACEMD: Accelerating Biomolecular Dynamics in the Microsecond Time Scale. *Journal of Chemical Theory and Computation*, 5(6):1632–1639, 2009.

[60] T. N. Heinz and P. H. Hünenberger. A Fast Pairlist-Construction Algorithm for Molecular Simulations Under Periodic Boundary Conditions. *Journal of Computational Chemistry*, 25(12):1474–1486, 2004.

[61] M. Heroux and R. Barrett. Mantevo Project. `http://software.sandia.gov/mantevo/`, May 2011.

[62] M. A. Heroux et al. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, Albuquerque, NM, 2009.

[63] A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, and S. Pakin. A Performance Comparison Through Benchmarking and Modeling of Three Leading Supercomputers: BlueGene/L, Red Storm, and Purple. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing*, SC '06, Tampa, FL, 2006. ACM Press.

[64] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications. *International Journal of High Performance Computing Applications*, 14(4):330–346, 2000.

[65] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the International Conference on Parallel Processing*, ICPP '00, Toronto, Canada, August 2000. IEEE Computer Society.

[66] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the International Conference on Supercomputing*, ICS '12, pages 311–320, Venice, Italy, June 2012. ACM.

[67] Intel Corporation. Intel Cilk Plus. `http://software.intel.com/en-us/intel-cilk-plus`, 2011.

[68] M. Itzkowitz and Y. Maruyama. HPC Profiling with the Sun Studio Performance Tools. In *Tools for High Performance Computing*, pages 67–93. 2010.

[69] D. A. Jacobsen, J. C. Thibault, and I. Senocak. An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. In *Proceedings of the AIAA Aerospace Sciences Meeting*, Orlando, FL, January 2010. AIAA.

[70] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo. A Simulator for Large-Scale Parallel Computer Architectures. *International Journal of Distributed Systems and Technologies*, 1(2):57–73, 2010.

[71] N. P. Jouppi and D. W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-III, pages 272–282, Boston, MA, 1989. ACM.

[72] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '01, pages 37–37, Denver, CO, 2001. ACM.

[73] D. J. Kerbyson, A. Hoisie, and S. D. Pautz. Performance Analysis and Grid Computing. chapter Performance Modeling of Deterministic Transport Computations, pages 21–39. Kluwer Academic Publishers, Norwell, MA, 2004.

[74] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. A Comparison between the Earth Simulator and AlphaServer Systems Using Predictive Application Performance Models. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, IPDPS '03, Nice, France, April 2003. IEEE Computer Society.

[75] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling GPU-CPU Workloads and Systems. In *Proceedings of the Workshop on General-Purpose Processing on Graphics Processing Units*, GPGPU 10, pages 31–42, Pittsburgh, PA, March 2010. ACM.

[76] Khronos OpenCL Working Group. OpenCL 1.2 Specification. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, November 2011.

[77] Khronos OpenCL Working Group. SPIR 1.0 Specification for OpenCL. `http://www.khronos.org/registry/cl/specs/spir_spec_1.0-provisional.pdf`, November 2012.

[78] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 339–350, Indianapolis,IN, June 2010. ACM.

[79] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the International Conference on Supercomputing*, ICS '12, pages 341–352, Venice, Italy, June 2012. ACM.

[80] S. Kim and H. Han. Efficient SIMD Code Generation for Irregular Kernels. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPOPP '12, pages 55–64, New Orleans, LA, February 2012. ACM.

[81] M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell. Extending OpenMP* with Vector Constructs for Modern Multicore SIMD Architectures. In *Proceedings of the International Workshop on OpenMP*, IWOMP '12, pages 59–72, Rome, Italy, 2012. Springer-Verlag.

[82] P. M. Kogge and T. J. Dysart. Using the TOP500 to Trace and Project Technology and Architecture Trends. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 28:1–28:11, Seattle, WA, November 2011. IEEE/ACM.

[83] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In *Proceedings of the International Workshop on Automatic Performance Tuning*, iWAPT '11, Berkeley, CA, June 2010. Springer.

[84] M. Koop, T. Jones, and D. Panda. Reducing Connection Memory Requirements of MPI for InfiniBand Clusters: A Message Coalescing Approach. In *IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 495–504, Rio de Janeiro, Brazil, May 2007. IEEE Computer Society.

[85] S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen. Atomic Vector Operations on Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '08, pages 441–452, Beijing, China, June 2008. IEEE.

[86] L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17:83–93, February 1974.

[87] Lawrence Livermore National Laboratory. Livermore Computing Systems Summary. `https://computing.llnl.gov/resources/systems_summary.pdf`, 2010.

[88] S. Le Grand, A. W. Gotzx, and R. C. Walker. SPFP: Speed Without Compromise – A Mixed Precision Model for GPU Accelerated Molecular Dynamics Simulations. (to appear), 2012.

[89] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, New Orleans, LA, 2010. IEEE Computer Society.

[90] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '09, pages 101–110, Raleigh, NC, 2009. ACM.

[91] S. Lee and J. S. Vetter. Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 1–11, Salt Lake City, UT, November 2012. IEEE Computer Society.

[92] V. W. Lee et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, ISCA '10, pages 451–460, Saint-Malo, France, June 2010. ACM.

[93] S. Manavski and G. Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[94] D. R. Mason. Faster Neighbor List Generation Using a Novel Lattice Vector Representation. *Computer Physics Communications*, 170:31–41, 2005.

[95] W. Mattson and B. M. Rice. Near-Neighbor Calculations Using a Modified Cell-Linked List Method. *Computer Physics Communications*, 119:135–148, 1999.

[96] T. Maximova and C. Keasar. A Novel Algorithm for Non-Bonded-List Updating in Molecular Simulations. *Journal of Computational Biology*, 13(5):1041–1048, 2006.

[97] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[98] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[99] P. Mehra, C. H. Schulbach, and J. C. Yan. A Comparison of Two Model-based Performance-Prediction Techniques for Message-Passing Parallel Programs. *SIGMETRICS Performance Evaluation Review*, 22(1):181–190, May 1994.

[100] Mellanox Technologies. NVIDIA GPUDirect Technology – Accelerating GPU-based Systems. `http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf`, May 2010.

[101] S. Meloni, M. Rosati, and L. Colombo. Efficient Particle Labeling in Atomistic Simulations. *Journal of Chemical Physics*, 126(12), 2007.

[102] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. GROPHECY: GPU Performance Projection from CPU Code Skeletons. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 14:1–14:11, Seattle, WA, 2011. IEEE/ACM.

[103] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 Supercomputer Sites. `http://top500.org/`, November 2012.

[104] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 19 April 1965.

[105] C. A. Moritz and M. I. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. *SIGMETRICS Performance Evaluation Review*, 26:254–263, June 1998.

[106] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A Portable Interface to Hardware Performance Counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, Monterey, CA, June 1999. IEEE.

[107] G. R. Mudalige. *Predictive Analysis and Optimisation of Pipelined Wavefront Applications Using Reusable Analytic Models.* PhD thesis, University of Warwick, 2009.

[108] G. R. Mudalige, M. B. Giles, C. Bertolli, and P. H. J. Kelly. Predictive Modeling and Analysis of OP2 on Distributed Memory GPU Clusters. *SIGMETRICS Performance Evaluation Review*, 40(2):61–67, October 2012.

[109] G. R. Mudalige, S. D. Hammond, J. A. Smith, and S. A. Jarvis. Predictive Analysis and Optimisation of Pipelined Wavefront Computations. In *Proceedings of the Workshop on Advances in Parallel and Distributed Computational Models*, APDCM '09, pages 1–8, Rome, Italy, May 2009. IEEE.

[110] G. R. Mudalige, S. A. Jarvis, D. P. Spooner, and G. R. Nudd. Predictive Performance Analysis of a Parallel Pipelined Synchronous Wavefront Application for Commodity Processor Cluster Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER '06, pages 1–12, Barcelona, Spain, September 2006. IEEE Computer Society.

[111] G. R. Mudalige, M. K. Vernon, and S. A. Jarvis. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '08, pages 1–14, Miami, FL, April 2008. IEEE.

[112] T. Mudge. Power: A First-Class Architectural Design Constraint. *Computer*, 34(4):52–58, 2001.

[113] Y. Munekawa, F. Ino, and K. Hagihara. Design and Implementation of the Smith-Waterman Algorithm of the CUDA-Compatible GPU. In *Proceedings of the IEEE International Conference on Bioinformatics and Bioengineering*, BIBE '08, Athens, Greece, October 2008. IEEE.

[114] G. J. Narlikar and G. E. Blelloch. Pthreads for Dynamic and Irregular Parallelism. In *Proceedings the ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–16, Orlando, FL, November 1998. IEEE.

[115] T. Narumi, Y. Ohno, N. Okimoto, T. Koishi, A. Suenaga, N. Futatsugi, R. Yanai, R. Himeno, S. Fujikawa, M. Taiji, and M. Ikei. A 55 TFLOPS Simulation of Amyloid-Forming Peptides from Yeast Prion Sup35 with the Special-Purpose Computer System MDGRAPE-3. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '06, pages 1–13, Tampa, FL, November 2006.

[116] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, San Diego, CA, June 2007. ACM.

[117] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, New Orleans, LA, November 2010. IEEE Computer Society.

[118] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.

[119] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. PACE – A Toolset for the Performance Prediction of Parallel and Distributed Systems. *International Journal of High Performance Computing Applications*, 14(3):228–251, 2000.

[120] NVIDIA Corporation. Tesla C2050 / C2070 GPU Computing Processor Datasheet. `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf`, July 2010.

[121] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 – The Fastest, Most Efficient HPC Architecture Ever Built. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, May 2012.

[122] S. Olivier, J. Prins, J. Derby, and K. Vu. Porting the GROMACS Molecular Dynamics Code to the Cell Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '07, Long Beach, CA, March 2007. IEEE.

[123] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, and S. A. Jarvis. Experiences with Porting and Modelling Wavefront Algorithms on Many-Core Architectures. In *Proceedings of the Daresbury GPU Workshop*, Daresbury, UK, September 2010.

[124] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, and S. A. Jarvis. Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark. In *Proceedings of the International Workshop on Performance Modeling, Benchmark and Simulation of HPC Systems*, PMBS '10, New Orleans, LA, November 2010.

[125] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, and S. A. Jarvis. Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark. *ACM SIGMETRICS Performance Evaluation Review*, 38(4), 2011.

[126] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Wright, and S. A. Jarvis. On the Acceleration of Wavefront Applications using Distributed Many-Core Architectures. *The Computer Journal*, 55(2):138–153, February 2012.

[127] S. J. Pennycook, S. D. Hammond, S. A. Wright, J. A. Herdman, I. Miller, and S. A. Jarvis. An Investigation of the Performance Portability of OpenCL. *Journal of Parallel and Distributed Computing*, (to appear), 2012.

[128] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '13, Boston, MA, May 2013.

[129] S. J. Pennycook and S. A. Jarvis. Developing Performance-Portable Molecular Dynamics Kernels in OpenCL. In *Proceedings of the International Workshop on Performance Modeling, Benchmark and Simulation of HPC Systems*, PMBS '12, Salt Lake City, UT, November 2012.

[130] S. J. Pennycook, G. R. Mudalige, S. D. Hammond, and S. A. Jarvis. Parallelising Wavefront Applications on General-Purpose GPU Devices. In *Proceedings of the UK Performance Engineering Workshop*, UKPEW '10, Warwick, UK, July 2010.

[131] O. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. WMTools – Assessing Parallel Application Memory Utilisation at Scale. In *Proceedings of the European Conference on Computer Performance Engineering*, EPEW '11, pages 148–162, Berlin, Heidelberg, 2011. Springer-Verlag.

[132] O. F. J. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. WM-Trace – A Lightweight Memory Allocation Tracker and Analysis Framework. In *Proceedings of the UK Performance Engineering Workshop*, UKPEW '11, Bradford, UK, July 2011.

[133] R. J. Petrella, I. Andricioaei, B. R. Brooks, and M. Karplus. An Improved Method for Nonbonded List Generation: Rapid Determination of

Near-Neighbor Pairs. *Journal of Computational Chemistry*, 24(2):222–231, 2003.

[134] F. Petrini et al. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '07, pages 1–10, Long Beach, CA, July 2007. IEEE.

[135] M. Pharr and W. R. Mark. ispc: A SPMD Compiler for High-Performance CPU Programming. In *Proceedings of Innovative Parallel Computing: Foundations & Applications of GPU, Manycore and Heterogeneous Systems*, InPar '12, San Jose, CA, May 2012.

[136] J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '08, pages 1–9, Austin, TX, November 2008.

[137] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117:1–19, 1995.

[138] S. Plimpton et al. LAMMPS Benchmarks. `http://lammps.sandia.gov/bench.html`, May 2011.

[139] S. Plimpton et al. LAMMPS Molecular Dynamics Simulator. `http://lammps.sandia.gov/`, May 2011.

[140] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '09, pages 427–436, Weimar, Germany, February 2009. IEEE Computer Society.

[141] B. R. Rau and J. A. Fisher. Instruction-level Parallel Processing: History, Overview, and Perspective. *Journal of Supercomputing*, 7(1-2):9–50, May 1993.

[142] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A Detailed, Accurate MPI Benchmark. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 1497:52–59, 1998.

[143] S. Ryoo et al. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the ACM*

SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '08, pages 73–82, Salt Lake City, UT, February 2008. ACM.

[144] S. Saini and D. H. Bailey. NAS Parallel Benchmark (Version 1.0) Results 11-96. Technical Report NAS-96-18, NASA Ames Research Center, November 1996.

[145] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications? In *Proceedings of the International Symposium on Computer Architecture*, ISCA '12, pages 440–451, Portland, OR, June 2012. IEEE.

[146] S. Seo, G. Jo, and J. Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *IEEE International Symposium on Workload Characterization*, IISWC '11, pages 137–148, Austin, TX, November 2011. IEEE.

[147] H. Shan, N. J. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann. A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI. In *Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, PMBS '11, pages 13–14, Seattle, WA, 2011. ACM.

[148] D. E. Shaw. A Fast, Scalable Method for the Parallel Evaluation of Distance-Limited Pairwise Particle Interactions. *Journal of Computational Chemistry*, 26(13):1318–1328, 2005.

[149] T. Shimokawabe et al. An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, New Orleans, LA, November 2010.

[150] M. Smelyanskiy, D. Holmes, J. Chhugani, A. Larson, D. Carmean, D. Hanson, P. Dubey, K. Augustine, D. Kim, A. Kyker, V. Lee, A. Nguyen, L. Seiler, and R. Robb. Mapping High-Fidelity Volume Rendering for Medical Imaging to CPU, GPU and Many-Core Architectures. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1563–1570, November 2009.

[151] L. Smith and M. Bull. Development of Mixed Mode MPI/OpenMP Applications. *Scientific Programming*, 9(2,3):83–98, 2001.

[152] T. Spelce. ASC Sequoia Benchmark Codes. `https://asc.llnl.gov/sequoia/benchmarks/`, September 2012.

[153] H. Sutter. The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[154] C. R. Trott. *LAMMPScuda - A New GPU-Accelerated Molecular Dynamics Simulations Package and its Application to Ion-Conducting Glasses*. PhD thesis, Ilmenau University of Technology, 2011.

[155] C. R. Trott, L. Winterfield, and P. S. Crozier. General-Purpose Molecular Dynamics Simulations on GPU-based Clusters. arXiv:1009.4330v2, 2010.

[156] L. Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159(1):98–103, 1967.

[157] J. S. Vetter and M. O. McCracken. Statistical Scalability Analysis of Communication Operations in Distributed Applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 123–132, Snowbird, UT, 2001. ACM.

[158] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. E. Guney, and A. Shringarpure. On the Limits of GPU Acceleration. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, HotPar '10, Berkeley, CA, June 2010.

[159] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson. Comparing Hardware Accelerators in Scientific Applications: A Case Study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):58–68, January 2011.

[160] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.

[161] S. Wienke, D. Plotnikov, D. an Mey, C. Bischof, A. Hardjosuwito, C. Gorgels, and C. Brecher. Simulation of Bevel Gear Cutting with GPGPUs – Performance and Productivity. *Computer Science - Research and Development*, pages 1–10, 2011.

[162] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Optimization of a Lattice Boltzmann Computation on State-of-the-Art Multicore Platforms. *Journal of Parallel and Distributed Computing*, 69(9):762–777, September 2009.

[163] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. Bhalerao, and S. A. Jarvis. Parallel File System Analysis Through Application I/O Tracing. *The Computer Journal*, 2012.

[164] H. Wu, G. Diamos, A. Lele, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission. In *Proceedings of the Multicore and GPU Programming Models, Languages and Compilers Workshop*, Shanghai, China, May 2012.

[165] Z. Yao, J.-S. Wang, G.-R. Liu, and M. Cheng. Improved Neighbor List Algorithm in Molecular Simulations Using Cell Decomposition and Data Sorting Method. *Computer Physics Communications*, 161:27–35, 2004.

[166] Y. Zhang and J. Owens. A Quantitative Performance Analysis Model for GPU Architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture*, HPCA '11, pages 382–393, San Antonio, TX, February 2011. IEEE Computer Society.

APPENDIX A
Assembly Generated for OpenCL Gather Operations

Figures A.1 and A.2 list the assembly generated by the Intel SDK for two of our miniMD kernels: the auto-vectorised vector gather kernel, and the kernel with explicit vectorisation and 8-way unrolling, respectively. The instructions have been re-ordered and grouped to improve readability, and any instructions not directly related to the gather of positions have been removed.

Both listings begin in the same way, loading the positions of eight neighbours into eight 128-bit (XMM) registers. These positions are stored in AoS format ($\{x, y, z, 0\}$) and both pieces of assembly transpose the gathered positions into SoA format ($\{x_1, x_2, ...\}$, $\{y_1, y_2, ...\}$ and $\{z_1, z_2, ...\}$), storing the result in three 256-bit (YMM) registers. During auto-vectorisation, the compiler recognises that this operation can be performed by an in-register AoS-to-SoA transpose, emitting a sequence of 14 shuffle and permute instructions. The sequence generated for our explicit vector kernel is significantly less efficient, containing 37 instructions.

Each of the `vinsertps` instructions in Figure A.2 extracts the x, y or z component from one XMM register (in AoS), and inserts it into another (in SoA). The result is six XMM registers, two for each of x, y and z, which are then combined into three YMM registers using three `vinsertf128`s. These inserts together account for 21 of the 37 instructions.

The `vinsertps` instruction has the capability to extract *any* 32-bit element from an XMM register, and we were therefore surprised to see that each of its uses here extracts the lowest 32 bits. This requires the generation of the 16 remaining instructions, to rearrange the XMM registers – each `vpshufd` instruction moves an atom's y component to the lowest 32-bits of the register,

```
// Load {x, y, z, 0} for eight atoms.
vmovaps     XMM7, XMMWORD PTR [R11 + R15]
vmovaps     XMM9, XMMWORD PTR [R10]
vmovaps     XMM10, XMMWORD PTR [R9]
vmovaps     XMM11, XMMWORD PTR [R8]
vmovaps     XMM12, XMMWORD PTR [RDI]
vmovaps     XMM13, XMMWORD PTR [RSI]
vmovaps     XMM14, XMMWORD PTR [RDX]
vmovaps     XMM15, XMMWORD PTR [R12]

// Build SoA registers for x, y and z.
vperm2f128  YMM11, YMM11, YMM15, 32
vperm2f128  YMM9, YMM9, YMM13, 32
vshufps     YMM13, YMM9, YMM11, 68
vperm2f128  YMM10, YMM10, YMM14, 32
vperm2f128  YMM7, YMM7, YMM12, 32
vshufps     YMM12, YMM7, YMM10, 68
vshufps     YMM14, YMM12, YMM13, -35
vpermilps   YMM14, YMM14, -40
vshufps     YMM12, YMM12, YMM13, -120
vpermilps   YMM12, YMM12, -40
vshufps     YMM9, YMM9, YMM11, -18
vshufps     YMM7, YMM7, YMM10, -18
vshufps     YMM7, YMM7, YMM9, -120
vpermilps   YMM7, YMM7, -40
```

Figure A.1: Assembly from the vector gather kernel.

and each `vmovhlps` instruction does the same for z.

We believe that these behaviours are caused by a combination of compiler immaturity and the way in which we have expressed the gather operation in OpenCL (Figure 6.8). Future compiler releases or an alternative representation of the gather in code may address these issues, leading to considerably better performance for this kernel on hardware supporting AVX instructions.

```
// Load {x, y, z, 0} for eight atoms.
vmovdqa     XMM3, XMMWORD PTR [R9 + R13]
vmovdqa     XMM4, XMMWORD PTR [R9 + R13]
vmovdqa     XMM6, XMMWORD PTR [R9 + R13]
vmovdqa     XMM7, XMMWORD PTR [R9 + R13]
vmovdqa     XMM8, XMMWORD PTR [R9 + R13]
vmovdqa     XMM9, XMMWORD PTR [R9 + R13]
vmovdqa     XMM11, XMMWORD PTR [R9 + R13]
vmovdqa     XMM12, XMMWORD PTR [R9 + R13]

// Build SoA register for x component.
vinsertps   XMM5, XMM4, XMM3, 16
vinsertps   XMM5, XMM5, XMM6, 32
vinsertps   XMM5, XMM5, XMM7, 48
vinsertps   XMM10, XMM9, XMM8, 16
vinsertps   XMM10, XMM10, XMM11, 32
vinsertps   XMM10, XMM10, XMM12, 48
vinsertf128 YMM5, YMM10, XMM5, 1

// Build SoA register for y component.
vpshufd     XMM13, XMM4, 1
vpshufd     XMM14, XMM3, 1
vinsertps   XMM13, XMM13, XMM14, 16
vpshufd     XMM14, XMM6, 1
vinsertps   XMM13, XMM13, XMM14, 32
vpshufd     XMM14, XMM7, 1
vinsertps   XMM13, XMM13, XMM14, 48
vpshufd     XMM14, XMM9, 1
vpshufd     XMM15, XMM8, 1
vinsertps   XMM14, XMM14, XMM15, 16
vpshufd     XMM15, XMM11, 1
vinsertps   XMM14, XMM14, XMM15, 32
vpshufd     XMM15, XMM12, 1
vinsertps   XMM14, XMM14, XMM15, 48
vinsertf128 YMM13, YMM14, XMM13, 1

// Build SoA register for z component.
vmovhlps    XMM4, XMM4, XMM4
vmovhlps    XMM3, XMM3, XMM3
vinsertps   XMM3, XMM4, XMM3, 16
vmovhlps    XMM4, XMM6, XMM6
vinsertps   XMM3, XMM3, XMM4, 32
vmovhlps    XMM4, XMM7, XMM7
vinsertps   XMM3, XMM3, XMM4, 48
vmovhlps    XMM4, XMM9, XMM9
vmovhlps    XMM6, XMM8, XMM8
vinsertps   XMM4, XMM4, XMM6, 16
vmovhlps    XMM6, XMM11, XMM11
vinsertps   XMM4, XMM4, XMM6, 32
vmovhlps    XMM6, XMM12, XMM12
vinsertps   XMM4, XMM4, XMM6, 48
vinsertf128 YMM3, YMM4, XMM3, 1
```

Figure A.2: Assembly from the kernel with 8-way unrolling.