The Readying of Applications for Heterogeneous Computing

by

Andy Herdman

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

September 2017

Abstract

High performance computing is approaching a potentially significant change in architectural design. With pressures on the cost and sheer amount of power, additional architectural features are emerging which require a re-think to the programming models deployed over the last two decades.

Today's emerging high performance computing (HPC) systems are maximising performance per unit of power consumed resulting in the constituent parts of the system to be made up of a range of different specialised building blocks, each with their own purpose. This heterogeneity is not just limited to the hardware components but also in the mechanisms that exploit the hardware components. These multiple levels of parallelism, instruction sets and memory hierarchies, result in truly heterogeneous computing in all aspects of the global system.

These emerging architectural solutions will require the software to exploit tremendous amounts of on-node parallelism and indeed programming models to address this are emerging. In theory, the application developer can design new software using these models to exploit emerging low power architectures. However, in practice, real industrial scale applications last the lifetimes of many architectural generations and therefore require a migration path to these next generation supercomputing platforms.

Identifying that migration path is non-trivial: With applications spanning many decades, consisting of many millions of lines of code and multiple scientific algorithms, any changes to the programming model will be extensive and invasive and may turn out to be the incorrect model for the application in question.

This makes exploration of these emerging architectures and programming models using the applications themselves problematic. Additionally, the source code of many industrial applications is not available either due to commercial or security sensitivity constraints.

This thesis highlights this problem by assessing current and emerging hardware with an industrial strength code, and demonstrating those issues described. In turn it looks at the methodology of using proxy applications in place of real industry applications, to assess their suitability on the next generation of low power HPC offerings. It shows there are significant benefits to be realised in using proxy applications, in that fundamental issues inhibiting exploration of a particular architecture are easier to identify and hence address.

Evaluations of the maturity and performance portability are explored for a number of alternative programming methodologies, on a number of architectures and highlighting the broader adoption of these proxy applications, both within the authors own organisation, and across the industry as a whole.

Acknowledgements

This thesis, and the supporting research, was made possible by the support of a number of people; their academic, professional and personal support throughout the years spent on this work have been invaluable.

Firstly, I'd like to thank my employer AWE who has invested not just financially but also allowed dedicated time for me to carry out this research and subsequent write up. In particular thanks go to Paul Tomlinson, Paul Ellicott and Andrew Randewich. A special thank you also goes to Wayne Gaudin for his technical input and general willingness as a sounding board.

Colleagues and peers across the HPC industry have also been of assistance: LANL, LLNL, SNL, PGI, Cray, Intel, IBM have either provided access to hardware or given technical input, without which this work would not have been possible. I would like to thank Sriram Swaminarayan, Todd Gamblin, Scott Futral, Si Hammond, Jim Ang, Mike Glass, Doug Miles, Michael Wolfe, Craig Toepfer, Dave Norton, Alistair Hart, John Levesque, Fiona Burgess, Andy Mason, Andy Mallinson, Victor Gamayunov, Stephen Blair-Chappell and Peter Mayes.

Also I'd like to express my thanks to the PhD students in the High Performance and Scientific Systems Group throughout my time at Warwick: Dr Si Hammond, Dr Andy Mallinson, Dr Olly Perks, Dr David Beckingsale, Dr James Davis, Dr Steve Wright, Dr John Pennycook and Dr Bob Bird for their advice and technical steer. I'd also like to express a big thank you to Prof. Stephen Jarvis for acceptance onto the PhD programme and his supervision.

Finally, I dedicate this PhD to my family, Sarah, Katie, Jack, Mam and sister; and to my Dad - hope you would have been proud - I love you and I miss you.

Andy Herdman - September 2017

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. Parts of this thesis have been previously published by the author in peer reviewed conferences and technical papers:

- Herdman, J. A., et al. "Benchmarking and modelling of POWER7, Westmere, BG/P, and GPUs: an industry case study." In: 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 10), New Orleans, LA, USA. ACM SIGMETRICS Performance Evaluation Review 38.4 (2011): 16-22 [103].
- Herdman, J. A., et al. "Accelerating hydrocodes with OpenACC, OpenCL and CUDA." SC Companion IEEE, 2012 High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, 10-16 Nov 2012 [101].
- Herdman, J. A., et al. "Achieving portability and performance through OpenACC." Proceedings of the First Workshop on Accelerator Programming using Directives (WACCPD '14), held as part of SC14 The International Confernece for HPC, Networking, Storage and Analysis. New Orleans, LA, USA. IEEE Press, 2014 [102].

In addition developments as part of this thesis's contributions, were among the integrated collection of mini-applications included in the 2013 R&D 100 "Oscars of Innovation" [172] award winning Mantevo test suite [104]. Also, Chapter 5 has been selected for inclusion in the forthcoming book: "Parallel Programming with OpenACC" [88].
Other related publications by the author, although not reproduced in this thesis, include:

- Davis, J. A., Mudalige, G. R., Hammond, S. D., Herdman, J. A., Miller, I., and Jarvis, S. A. "Predictive analysis of a hydrodynamics application on large-scale CMP clusters." Computer Science-Research and Development 26, no. 3-4 (2011): 175-185. [79]
- Mallinson, A. C., Beckingsale, D. A., Gaudin, W. P., Herdman, J. A., Levesque, J. M., and Jarvis, S. A. "CloverLeaf: Preparing hydrodynamics

codes for Exascale." In: A New Vintage of Computing : CUG2013, Napa, CA, 6 - 9 May 2013. Published in: A New Vintage of Computing : Preliminary Proceedings (2013) [137]

- Pennycook, S. J., Hammond, S. D., Wright, S. A., Herdman, J. A., Miller, I., and Jarvis, S. A. "An investigation of the performance portability of OpenCL." Journal of Parallel and Distributed Computing 73, no. 11 (2013): 1439-1450. [162]
- Mallinson, A. C., Beckingsale, D. A., Gaudin, W. P., Herdman, J. A., and Jarvis, S. A. "Towards portable performance for explicit hydrodynamics codes." (2013). [138]
- Gaudin, W. P, Mallinson, A. C., Perks, O., Herdman, J. A., Beckingsale, D. A., Levesque, J. M., and Jarvis, S. A. "Optimising Hydrodynamics applications for the Cray XC30 with the application tool suite." The Cray User Group (2014): 4-8. [94]
- Mudalige, G. R., Reguly, I. Z., Giles, M. B., Mallinson, A. C., Gaudin, W. P., and Herdman, J. A. "Performance Analysis of a High-Level Abstraction Based Hydrocode on Future Computing Systems." In High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation, pp. 85-104. Springer International Publishing, 2014. [149]
- Mallinson, A. C., Jarvis, S. A., Gaudin, W. P., and Herdman, J. A. "Experiences at scale with PGAS versions of a Hydrodynamics application." In Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, p. 9. ACM, 2014. [139]

In addition, research conducted during the period of registration has also led to the following publications, which are not directly linked to the contents of this thesis:

- Hammond, S. D., Mudalige, G. R., Smith, J. A., Davis, J. A., Jarvis, S. A., Holt, J., Miller, I., Herdman, J. A., and Vadgama, A. "To upgrade or not to upgrade? Catamount vs. Cray Linux Environment." In Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, pp. 1-8. IEEE, 2010. [99]
- Wright, S. A., Hammond, S. D., Pennycook, S. J., Miller, I., Herdman, J. A., and Jarvis, S. A. "LDPLFS: improving I/O performance without application modification." In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, pp. 1352-1359. IEEE, 2012 [200]

- Perks, O., Beckingsale, D. A., Hammond, S. D., Miller, I., Herdman, J. A., Vadgama, A., Bhalerao, A. H., He, L., and Jarvis, S. A. Towards Automated Memory Model Generation via Event Tracing, The Computer Journal 56(2), June 2012 [165]

- Perks, O., Beckingsale, D. A., Dawes, A. S., Herdman, J. A., Mazauric, C., and Jarvis, S. A. "Analysing the influence of InfiniBand choice on OpenMPI memory consumption." In High Performance Computing and Simulation (HPCS), 2013 International Conference on, pp. 186-193. IEEE, 2013. [164]

- Wright, S. A., Hammond, S. D., Pennycook, S. J., Bird, R. F., Herdman, J. A., Miller, I., Vadgama, A., Abhir Bhalerao, A., and Stephen A. Jarvis, S. A. "Parallel file system analysis through application I/O tracing." The Computer Journal 56, no. 2 (2013): 141-155. [199]

- Beckingsale, D. A., Perks, O., Gaudin, W. P., Herdman, J. A., and Jarvis, S. A. "Optimisation of patch distribution strategies for AMR applications." In Computer Performance Engineering, pp. 210-223. Springer Berlin Heidelberg, 2013. [61]

- Bird, R. F., Gillies, P., Bareford, M. R., Herdman, J. A., and Jarvis, S. A. "Mini-app driven optimisation of inertial confinement fusion codes." In Cluster Computing (CLUSTER), 2015 IEEE International Conference on, pp. 768-776. IEEE, 2015. [64]

- Beckingsale, D. A., Gaudin, W. P., Herdman, J. A., and Jarvis, S. A. "Resident block-structured adaptive mesh refinement on thousands of graphics processing units." In Parallel Processing (ICPP), 2015 44th International Conference on, pp. 61-70. IEEE, 2015. [58]

- Dickson, J., Wright, S. A., Maheswaran, S., Herdman, J. A., Miller, M. C., and Jarvis, S. A. "Replicating HPC I/O workloads with proxy applications." (2016). [83]

- Dickson, J., Wright, S. A., Maheswaran, S., Herdman, J. A., Harris, D., Miller, M. C., and Jarvis, S. A. "Enabling portable I/O analysis of commercially sensitive HPC applications through workload replication." Cray User Group 2017 Proceedings (2017). [82]

Sponsorship and Grants

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

- UK Atomic Weapons Establishment, Further Education Funding.

Abbreviations

AIX	Advanced Interactive eXecutive
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices®
ANL	Argonne National Laboratory
APEX	Alliance for application Performance at EXtreme scale
API	Application Programming Interface
APU	Accelerated Processing Unit
ARC	Advanced Research Computing
ARM	Advanced RISC Machines
ASCI	Accelerated Strategic Computing Initiative
ASC	Advanced Simulation and Computing Program
AVX	Advanced Vector eXtensions
AWE	Atomic Weapons Establishment
BGK	Bhatnagar-Gross-Krook
CAAR	Center for Application Accelerator Readiness
CAF	Co-Array Fortran
CAPS	Compiler and Architecture for Embedded and Superscalar Processors
CCE	Cray Compiling Environment
CEA	Commissariat á l'énergie atomique et aux énergies alternatives
CFD	Computational Fluid Dynamics
CORAL	Collaboration of Oak Ridge, Argonne, and Lawrence Livermore
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CSCS	Centro Svizzero di Calcolo Scientifico
CTBT	Comprehensive Test Ban Treaty
CUDA	Compute Unified Device Architecture
DARPA	Defence Advanced Research Projects Agency
DoE	Department of Energy
DDR	Double Data Rate
DP	Double Precision
DRAM	Dynamic Random Access Memory
DSL	Domain Specific Language
ECC	Error Correcting Code
EDR	Enhanced Data Rate (InfiniBand)
ECP	Exascale Computing Plan
EFLOP/s	10^{18} FLOP/s
FLC	Federal Laboratory Consortium

FDR	Fourteen Data Rate (InfiniBand)
FLOP/s	Floating-Point Operations per Second
FMA	Fused Multiply Add
FPU	Floating-Point-Unit
FSB	Front Side Bus
FYP	Five-Year Plan
GFLOP/s	10^9 FLOP/s
GPC	Graphics Processing Cluster
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
GTC	GPU Technology Conference
GT/s	gigatransfers per second
HAAPs	Heterogeneous Advanced Architecture Platforms
HBM	High Bandwidth Memory
HMC	High Memory Cube
HMW	High Memory Watermark
HPC	High-Performance Computing
HSA	Heterogeneous System Architecture
HT	Hyper Threading
IB	InfiniBand
IBM	International Business Machines®
ISC	International Supercomputing Conference
IDE	Integrated Development Environment
ILP	Instruction-Level Parallelism
ISA	Instruction Set Architecture
ISV	Independent Software Vendor
IWOCL	International Workshop on OpenCL
KNC	Knights Corner: 1st generation MIC architecture
KNF	Knights Ferry: prototype MIC architecture
KNH	Knights Hill: 3rd generation MIC architecture
KNL	Knights Landing: 2nd generation MIC architecture
LANL	Los Alamos National Laboratory
LEO	Language Extensions for Offload
LLNL	Lawrence Livermore National Laboratory
LOC	Lines of Code
MCDRAM	Multi-Channel DRAM
MEXT	Ministry of Education, Culture, Sports, Science and Technology
MIC	Many-Integrated Core
MMX	Multimedia Extension
MPI	Message Passing Interface

MPICH	MPI CHameleon
MPP	Massively Parallel Programming
NAS	NASA Advanced Supercomputing
NASA	National Aeronautics and Space Administration
NCSA	National Centre for Supercomputing Applications
NERSC	National Energy Research Scientific Computing Center
NNSA	National Nuclear Security Administration
NPB	NAS Parallel Benchmarks
NRE	Non-Recurring Engineering
NSCI	National Strategic Computing Initiative
NUMA	Non-Uniform Memory Access
NVRAM	Non-Volatile Random Access Memory
OO	Object Oriented
ORNL	Oak Ridge National Laboratory
OS	Operating System
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PFLOP/s	10^{15} FLOP/s
PGAS	Partitioned Global Address Space
PGI	The Portland Group®
PGO	Profile Guided Optimisation
POWER	Performance Optimized With Enhanced RISC
PRACE	Partnership for Advanced Computing in Europe
PtAC	Path to Agile Coding
PtMC	Path to Many Core
PTX	Parallel Thread eXecution
QDR	Quad Data Rate (InfiniBand)
QCD	Quantum Chromodynamics
QCM	Quad Chip Module
QPI	QuickPath Interconnect
QPX	Quad Processing eXtension
QUDA	QCD on CUDA
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SAMRAI	Structured Adaptive Mesh Refinement Application
SCC	Single-chip Cloud Computer
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction, Multiple Data
SPEC	Standard Performance Evaluation Corporation

SPH	Smoothed Particle Hydrodynamics
SKU	Stock Keeping Unit
SM	Streaming Multiprocessor
SMP	Symmetric Multi Processor
SMT	Simultaneous Multi-Threading
SNL	Sandia National Laboratories
SoC	System-on-a-Chip
SP	Single Precision
SRAM	Static Random Access Memory
SSE	Streaming SIMD Extensions
STL	Standard Template Library
TBB	Thread Building Blocks
TDP	Thermal Design Power
TFLOP/s	10^{12} FLOP/S
TLP	Thread-Level Parallelism
UDel	University of Delaware
UPC	Unified Parallel C
VMX	Vector Multimedia Extension
VPU	Vector Processing Unit
WOC	Words of Code

Definitions

Amdahl's Law

Amdahl's Law [47] states the theoretical limit for the speedup of a fixed, strong scaling problem, as the number of executing processors is increased.

Shown in Equation 1 where S is the speedup of the whole problem, s is the speedup of the parallel section of the task, p is the proportion of the parallel section (or none serial section)of the problem.

$$S = \frac{1}{(1-p) + \frac{p}{s}} \tag{1}$$

It shows that the theoretical speedup of the whole problem increases with processors and that regardless of the magnitude of the improvement, the theoretical speedup is always limited by the serial part of the task.

Bandwidth Bound

A bandwidth bound algorithm is one that has reached the physical limits of the underlying hardware in terms of access to global memory.

Computational Kernel

A collection of application program code, such as multiple loop-block structures, which has been logically co-located within the same program function or subroutine, and collectively performs a particular well-defined task or operation.

Concurrency

A system consisting of multiple streams of independent operations active at one time.

Compute Bound

A computational operation whose time is primarily decided by the time taken to operate on the data, rather than the time to load the data into memory. In such a scenario the use of a faster processor will afford a proportional gain in overall performance.

Data Parallel

The distribution of data across multiple hardware processing elements, enabling the same task to be carried out in unison on multiple data points.

Deep Copy

A copy of a data structure duplicating not only the structure itself, but all associated sub-structures.

ECC RAM

Error-Correcting Code (ECC) RAM is a type of memory with built in error detection and correction, through the use of parity bits. This makes the memory immune to single bit errors, increasing reliability.

Front Side Bus

A Front Side Bus (FSB) is a legacy communication interface that was used in Intel processors, carrying data between the central processing unit (CPU) and the memory controller.

Ghost/Halo Cells

With parallel grid based computations it is frequently necessary to access data which resides in another processor's memory space. Such a situation usually occurs at the boundary of a processor's computational region. To improve the performance of fetching the data, a buffer is used to replicate the whole boundary region on the local processor. This data is rarely computed, but used as input to the computation of other cells.

Hackathon

An event, in which a large number of people meet to engage in collaborative computer programming, focusing on a particular problem, platform, application or language.

Heterogeneous Computing

The description a computer system consisting of one or more of the following: multiple instruction set architectures (ISA), multiple processor types, multiple mechanisms to exploit system parallelism, multiple memory or data hierarchies.

Latency Bound

A latency bound algorithm is one whose performance is inhibited due to the time to carry out memory fetches on dependent data.

Memory Bound

A computational operation whose time is primarily governed by the rate at which data can be moved from memory to the processor, rather than the actual computational operation. There are two distinct classes of memory bound algorithms: latency bound and bandwidth bound. In both scenarios the use of a faster processor will not afford a gain in overall performance. Improvements in performance will only be afforded be improvements to the memory system, such as faster RAM, more memory bandwidth, or enhancements in cache within the processor.

Memory Wall

The much faster improvement of processor speed as compared with dynamic random access memory (DRAM) speed, resulting in processor speed improvements being masked for particular classes of algorithms by the relatively slow improvements to DRAM speed [202].

Moore's Law

A prediction, rather than a fundamental law; Gordon Moore of Intel predicted a doubling in the number of a microchips components every two years. As this increase in components was directly proportional to the speed of a chip, this was frequently expressed as a doubling in the performance of a chip/processor. Although the original prediction by Moore is still realised, the increase in processor frequency, or clock speed is not.

Power Wall

The significant increasing loss of efficiency due to overheating as the clock frequency of a CPU increases.

Ragged Array

An array with rows of non-uniform length.

Stencil Framework

A framework that performs cyclic updates to data according to a predefined pattern. Usually a particular stencil is associated with a particular algorithmic domain.

Stride One

Also known as "unit stride", is an array with a stride of exactly the same size as the size of each of its elements.

Strong Scaling

The act of increasing the processing resources used to solve a problem of the same size, with the resultant performance continuing to scale.

Task Parallel

The enablement of each hardware processing element to execute a different execution thread on the same, or different data.

Thread

An independent process, with associated data and instructions.

Thread Safe

A property of a library or programme component, which can have multiple threads executing simultaneously in a manner which still produces correct results.

Turnover

The point, in Strong Scaling, where increasing the number of processors to solve a problem results in a slower solution time; that is the problem no long demonstrates Strong Scaling.

Weak Scaling

The act of increasing the processing resources used to solve a problem of increasing size, where the processing resources remains the same with the time to solution remaining the same, or decreasing.

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

The drivers for ever increasing levels of compute performance are many. From the scientific view point, pushing the boundaries of discovery in the fields of health, finance, science, industry and military, to a purely commercial view point with the attraction of the worlds best students and technical minds to the industrial infrastructure of a nation.

Internationally, strategic drives from China, USA, Japan and the European Union are striving to be the first to design and deploy exascale class machines, namely systems capable of performing one billion billion floating point operations (FLOP) per second (EFLOP/s).

China first laid claim to the world's fastest supercomputer: the NVIDIA® Tesla®GPU accelerated Intel®Xeon®, Tianhe-1A [144], in November 2010 and re-laid the claim in November of 2015 with the Intel Xeon / Intel®Xeon Phi™ based, 54.9 petaFLOP (PFLOP) Tianhe-2 [133]. However, with embargoes from the US restricting future imports of these processor technologies to the home of Tianhe-2, the National Supercomputer Center and also China's Jiangnan Institute of Computer Technology, the country is actively pursuing it's own micro-processor development via its ShenWei, FeiTeng and Loongson micro-processor technology [112]. Announced in June 2016, eclipsing Tianhe-2 at the top of the Top500 list, was Sunway TaihuLight. The building blocks of the system are the ShenWei SW26010 processor. With 40,960 nodes, comprising of 10,649,600 computing cores, it is twice the speed and three times as efficient as Tianhe-2 [85]. China's social, economic and political strategy is described in successive "Five-Year Plans" (FYP), the 12th of which (2011-2015) [72] instigated the programme resulting in Sunway TaihuLight. The subsequent 13th FYP (2016-2020) sets out the delivery of an exascale machine by the end of the decade.

Japan's SPARC64™based Fujitsu manufactured K-computer [203], still one of the top ten fastest machines in the world, debuting as the fastest in June of 2011 [148], with a peak performance of 11.3 PFLOPs. The country's Ministry of Education, Culture, Sports, Science and Technology (MEXT) announced the Flagship2020 project in April 2014. Flagship2020's goal is to deploy a machine with "100 times more application performance" than the K-computer at the RIKEN Advanced Institute for Computational Science by 2020. As announced at ISC 2016 [116] this platform will utilise ARM®based, rather than SPARC64, from Fujitsu [170].

A high level policy from the Office of Science and Technology Policy in the United States, resulted in the US National Strategic Computing Initiative (NSCI) [108] to "*create a cohesive, multi-agency strategic vision and investment strategy that assures the United States sustains or extends its historical lead and strategic advantage in High Performance Computing (HPC) technology for national security, economic prosperity and scientific discovery*" that has the goal of deploying an exascale class system by 2023 within the Department of Energy (DoE). The Exascale Computing Plan (ECP) is the delivery mechanism for this strategy setting out the PathForward program to oversee hardware development, funding a range of vendors to carry out research and development towards a deployable exascale system. A number of pre-exascale systems have been announced from the DoE under a phased Non-Recurring Engineering (NRE) route, which includes the Advanced Technology Systems (ATS). The first phase consists of the Trinity (ATS-1) [128] and Cori [151] systems at Los Alamos National Laboratory (LANL) and the National Energy Research Scientific Computing Center (NERSC) respectively. The second phase is under the auspices of the CORAL (Collaboration of Oak Ridge, Argonne, and Lawrence Livermore) project, consisting of two IBM®POWER9®with NVIDIA GPUs based systems to be deployed at Lawrence Livermore National Laboratory (LLNL), Sierra (ATS-2) [136] and Oak Ridge National Laboratory (ORNL), Summit [156] and a Cray®, Intel Xeon Phi based system, Aurora [50], to be located at Argonne national Laboratory (ANL). A third phase, APEX (Alliance for application Performance at EXtreme scale) will consist of systems at LANL (ATS-3, Crossroads [127]) and at NERSC (NERSC-9).

Under PRACE (Partnership for Advanced Computing in Europe) there are a large range of exascale research and development activities under way [52], with a number of national Tier-0 Class systems in production [171]. However, the only committed plans specifically targeting an exascale machine are from Atos®, through its technology brand Bull, based on its Sequana supercomputer [69]. An exascale machine is targeted for the Commissariat á l'énergie atomique et aux énergies alternatives (CEA) by the end of the decade, with a pre-exascale Intel Xeon Phi based machine, the TERA-1000, in construction at present.

Irrespective of whichever nation succeeds in the race to become the first to reach the exascle landmark; there is a significant potential for the architectures of the first exascale platforms from ShenWei, ARM, Intel®, IBM®to be considerably different to those deployed in today's supercomputers.

1.1 Architectural Changes

The increasing number of transistors on a micro-processor, as predicted by Moore's law [184], has provided a continuous, dependable improvement in processor performance for several decades. As a prediction of the increase in the number of chip's components as a function of time, it initially predicted a doubling of a chip's components every year, and later revised to every two years. As this increase in components was directly proportional to the speed of a chip, this was frequently expressed as a doubling of a chip's performance. Although the original prediction by Moore is still realised, the increase in a processors frequency, or clock speed is not; this stopped for two main reasons.

Firstly the "memory wall": the relative increase in processor frequency and that of the memory to "feed" the processing unit began to diverge. Irrespective of the speed a CPU can process data, if no data is present to be processed, then striving for ever faster processor speeds is a redundant exercise.

Secondly the "power wall": The thermal power, namely heat, that is generated in an active CPU needs to be dissipated. The power (P) generated by a CPU is directly proportional to the capacitance (C), clock frequency (f), and the square of the supply voltage (V), as expressed in Equation 1.1.

$$P \propto fCV^2 \tag{1.1}$$

As any increase in clock frequency also requires an increase in the supply voltage, there is essentially a resultant cubic increase in power from increasing the clock frequency; this is evidently unsustainable.

1.1.1 Emergence of Multi-Core

Because of these two contributing factors, from around 2003, instead of increasing the clock frequency of a single uni-processor, an increase in the number of processing cores became a dominant trend, resulting in over 20 cores in a current CPU micro-architecture. With the individual clock frequencies of these multiple cores plateauing, the only increase in the power envelope in the system is that of capacitance, which as Equation 1.1 shows, results in a linear increase in power.

Although this strategy of increasing core counts looks to continue, in isolation it will not meet the performance demands of future HPC whilst still operating within an acceptable power envelope. By design a modern core of a CPU is a general purpose execution engine; that is it is designed to do a wide range of tasks in a performant manner. However, there are a number of processors that have very specific, or specialised, purpose at which they excel.

One such example is the Graphics Processing Unit (GPU), whose original design specification was that of manipulating the pixels that go to make up display screen images. Analogous to the methodology of making an individual CPU's core less complex, by adding complexity to the cores of a GPU the ability to enable general purpose execution becomes a possibility. This was achieved through the addition of error correction codes (ECC), adding increasing floating point execution units and the ability through new languages to program these floating point unit's capabilities.

It is predicted that as core count increases, the cores themselves will not only reduce in individual clock frequency but also in complexity, plus the possibility of heterogeneous systems with not just multiple cores, but with multiple different cores. Ultimately this will result in systems comprised of a much larger number of lower-power, lower-performance cores than seen today.

1.1.2 Instruction Level Parallelism

Heterogeneity is also present in the parallelism models required to maximise the utilisation of these multi-core systems.

Instruction level parallelism (ILP) describes the numerous mechanisms deployed in a modern microprocessor to implicitly execute machine instructions simultaneously, hence achieving a level of instruction parallelisation. By considering the order and repeatability of the instructions usually associated with a particular program's flow, a "production line" of instructions can be created. Referred to as *pipelining*, staggered instructions can be overlapped to achieve parallelism. Coupled with pipelining is *branch prediction*, where the processor makes an educated guess as to the outcome of a branch and begins to carry out subsequent instructions in advance. The trade-offs between pipeline depth, complexity of branch prediction algorithms and penalties for miss-predictions improve with each generation of micro architecture [90] increasing the throughput of generated instructions. However, with a greater number of simpler processing cores per CPU, the trend is for individual core's piplines to become shallower and for branch prediction to become less speculative, hence moving the exploitation of parallelism from an implicit compiler-based approach, or hardware design, to one of explicit implementation from the application developer.

Analogous to achieving ILP through pipelineing, today's micro-architectures are also able to partition and duplicate their resources resulting in the ability to issue multiple instructions at the same time. Known as superscalar architectures they also require an aware compiler to generate an instruction mix amenable to the hardware. This type of hardware multithreading is extended in the concept of simultaneous multi-threading (SMT) which enables multiple threads to issue instructions per clock cycle with the desired result being greater utilisation of pipelines. Implemented examples of this technology includes Intel's hyper-threading and IBM's SMT.

1.1.3 Data Parallelism

Exploiting parallelism from data can be achieved in a number of ways and to an extent, all of them need to be utilised to fully exploit hardware which provides a significant amount of its on node parallelism through each mechanism.

Single instruction, multiple data (SIMD) is a technique which describes taking a block, or *vector*, of data and applying the same instruction to all data entries in the vector. Beginning with multimedia extension (MMX™technology) dedicated integer instructions could be executed on such vector registers. This was extended to cover floating point instructions with Streaming SIMD Extentions (SSE) and also a doubling in size of the vector to a 128-bit register. Over a number of SSE generations, additional new vector instructions were added. Advanced Vector Instructions (AVX) again doubled the register size to 256-bit with new instructions, and available today in Intel's Xeon Phi architecture is AVX-512, which provides 512-bit registers.

There is an obvious trend towards increasing the vector sizes and the instructions that can be applied to data residing in these registers. To fully exploit these architectural features, an application's data needs to be stored in appropriate structures to enable the compiler to recognise the potential to apply such vector instructions. The implementation of specialist gather/scatter operations have assisted with this issue, in that vector operations can now be applied to non-contiguous data, but there is still an onus on the application developer to be aware of how their application's data is structured.

Data parallelism can also be applied to exploit the shared memory nature of hardware. Chapter 3 details a range of programming methodologies (OpenCL™(Open Compute Language) software [125], OpenMP™ [21], OpenACC [23], pthreads [153]), that can be used to target such shared memory multi-processors. Through the use of these constructs the application programmer can split the workload within their application across physical cores within an SMT region of the underlying hardware.

Indeed, although Hardware Multi-threading, as described in Section 1.1.2, was conceived to exploit ILP by increasing the number of instructions carried out per clock cycle, it can also be targeted explicitly in the same way with a high-level directive approach. However, as multi-threading's primary design is to cope with lightweight ILP instructions, the overheads of heavyweight threading such as Message Passing Interface (MPI) or OpenMP, plus the additional overhead of the MPI/OpenMP management and runtime, utilising all available hardware threads usually leads to a detrimental impact on performance, so requires careful deployment.

Data parallelism can also be used to reduce overheads associated with distributed parallelism. Although an MPI distributed application can execute on a multi-core CPU, by replacing the on-node parallelism with a data parallel model the overall memory footprint may be reduced by negating the need for memory hungry message passing buffers. Additionally, as detailed in Section 1.1.4, the overheads in moving data will become a significant contributor to the overall energy consumption of a system, hence a shared memory model capitalising on the locality of the data is desirable.

1.1.4 Deeper Memory Hierarchies

Keeping power consumption under control is not only a problem for computation. Although dynamic memory technology has improved, indeed memory density has increased at a faster rate than that of the processors speed; however, the speed of the memory has not. This has resulted in what is termed the “Memory Wall”; that is the limitation in an application’s ability to utilise the benefits available from the increases in the processors speed due to the relative slow speeds to obtain the data from memory to process.

Indeed the relative gap between the energy used in computation of data to that of moving the same data is increasing. Forecasts for exascale predict the cost to move a double-precision data object from memory to the floating-point unit (FPU) to be two orders of magnitude higher than the cost of performing a floating point operation on the same data [124].

One approach to hide the impact of the “Memory Wall”, is the introduction of memory hierarchies. Here, varying levels of memory technology are deployed ranging from small, expensive but fast, memory to larger cheaper slower memory. Consequently the temporal locality of data is becoming crucial, implying a trade off for increasing FLOP/s whilst reducing memory accesses.

Not only are technological improvements in the capacity and bandwidth of memory increasing at slower rates than those in compute, those memory improvements are manifesting themselves through more than one technical solution.

A number of new high bandwidth memory standards are emerging that ultimately aim to embed high bandwidth memory on the chip realising over five times the memory bandwidth of current Double Data Rate (DDR) Synchronous Dynamic Random Access Memory (SDRAM) memory. High Bandwidth Memory (HBM) and High Memory Cube (HMC) are two such current proposed standards, examples of which have already appeared in the NVIDIA®Pascal™GPU and as Multi-Channel DRAM (MCDRAM) in Intel's 2nd generation MIC architecture the Knight's Landing (KNL) Xeon Phi.

Similar developments in non-volatile memory (NVRAM) are enabling fast access to relatively small amounts of persistent memory. Primarily being considered as mechanisms to enable large HPC systems to checkpoint applications in acceptable times, the nature of such memory matches the requirement of writing large "bursts" of data and a slower constant write to traditional persistent memory on disk.

The outcome is a complex heterogeneous memory system. Such systems have a hierarchy structure, with cache-based static random access memory (SRAM), high memory bandwidth on chip, off-chip dynamic random access memory (DRAM), and NVRAM sitting between remote disk.

Where in the past, most memory performance gains were implicitly handled by the system (in the hardware or software, at runtime or through the compiler) parts of the hierarchy will need explicit management from the application developer to obtain the significant performance gains available.

1.1.5 System Interconnects

Analogous to reducing the operating frequency of a processing core when not in use, energy saving strategies are under consideration for current interconnect networks to disable or reduce power in links that are not being utilised [106]. For next generation interconnects, the move from copper-based electrical connectors to optical fibres (silicon photonics) aims to significantly increase density and performance (in terms of bandwidth) of connections, while also reducing the power required. Although it is not currently known whether these technologies will directly impact the application, the possibility of reduced reliability could indicate the requirement to build in greater resilience awareness into applications.

Off chip, the network is the backbone of the modern supercomputer. The issue of power at large scale is resulting in methods such as dynamic throttling, mapping point to point messaging to network and dynamic routing of messages to avoid congestion on the network. Optimisation for communication operations to enable the mapping of an application's communication pattern to that of a changing network architecture [107], is a growing research area.

This highlights the growing importance for an application to take advantage of such architectural features.

In summary, although tomorrow's HPC is striving to attain ever increasing performance, the basic hardware building blocks are no longer contributing to this by increases in speed, but by increasing power efficiency, resulting in heterogeneity throughout all levels of the system, from hardware to software.

The net result is greater levels of parallelism are required to exploit future machines potential. Where as once the techniques to exploit these levels of parallelism were implicitly utilised by the underlying hardware, or its compiler, these concepts are increasingly being required to be understood by an application developer. The developer needs to explicitly ensure their application is programmed in away that that exposes concurrency in their algorithms to be able to exploit these hardware implementations.

1.2 Impact of Architectural Changes

With such a diverse range of heterogeneous architectures coming to the fore, the challenge for HPC is complex; not only in the range of possible hardware and software options, but also due to the huge range of application domains.

More often than not, simulation of the natural world involves a combination of these algorithmic domains. Colella [74] identified and characterised the seven most prolific of these in relation to the then contemporary Defence Advanced Research Projects Agency (DARPA) program: Dense linear algebra, Sparse linear algebra, Spectral methods, N-body methods, Structured grids, Unstructured grids and Monte Carlo methods. With the emerging computational requirements to apply high-level abstractions to large data sets, algorithmic areas such as Graph Traversal, Map Reduce, Combinational Logic, Machine Learning and Finite State Machines are also growing in their use.

The field of Life Sciences encompasses drug development, DNA sequencing. Material Science research depends heavily on the concept of molecular dynamics, a simulation method developed with the emergence of HPC computational capabilities. The Oil and Gas Industry, along with vehicle and aircraft manufactures, needing to simulate the operability space of their engines through 3D unstructured fluid dynamic studies, depend on numerical algorithms to model the flow of fluids through a range of mediums, primarily falling into the field of Computational Fluid dynamics (CFD).

1.3 Sphere of Study and Research Questions

The Atomic Weapons Establishment (AWE) has played a central role in the defence of the United Kingdom for more than 60 years, providing and maintaining nuclear warheads for the United Kingdom's (UK) Continuous At Sea Deterrent.

In the absence of nuclear testing and the Comprehensive Test Ban Treaty

(CTBT), work to maintain and support a nuclear deterrent relies on cutting-edge science and computational methodologies to verify the safety and effectiveness of the warhead stockpile.

AWE's ability to understand the performance of a warhead and underwrite its safety depends crucially on numerical simulations for modelling both physics and engineering aspects. Information from hydrodynamics, laser experiments and data from material ageing studies plus previous nuclear test results are used in mathematical modelling.

Hydrodynamic computational models, described by the compressible Euler equations, simulate the flow of fluids by describing their properties in terms of pressure, velocity, temperature and density as functions of space and time.

To explore the use of HPC for hydrodynamics a suitable application was identified and its amenability was explored. Taking an industrial strength benchmark code, Shamrock (detailed in Section 4), an exploration to ascertain if its use case could be extended beyond that of a machine upgrade/procurement tool was carried out. This resulted in highlighting a number of shortcomings in the agility of a large benchmarking application for rapid turnaround to the questions of amenability of the hydrodynamic algorithm to a range of emerging architectures and programming paradigms.

The concept of the mini-application, or "mini-app" is a compact, self–contained application that embodies the essential performance characteristics of the main application it aims to be a proxy for. In Chapter 5 the process of developing an OpenACC-based performant version of the hydrodynamic mini-app CloverLeaf is described, detailing stage-by-stage the process required to enable the mini-app to utilise a Cray XK6, GPU-based supercomputer.

It has been demonstrated that a range of emerging hardware and software options can be explored with their relative performance and efforts highlighted.

With recognition of such changes in architectural designs and the subsequent implications to the applications that utilise HPC, this thesis aims to document a bounded study into this area (see Section 1.3). Each chapter in this thesis answers a set of research questions that guide its flow.

Chapter 1 describes the "state-of-the-nation" of HPC and covers why and how HPC architectures are changing, highlighting the increasing onus on the application developer to be architecturally aware and the need for applications to adapt to meet the changing HPC landscape.

Chapter 2 details those emerging hardware architectures that are beginning to exhibit such architectural changes. It considers these architectures in turn and then looks to identify commonalities in these emerging candidates.

Chapter 3 looks at the wide range of programming options available to the developer to exploit emerging hardware and (as with hardware) looks to see if there is any commonality across the programming choices.

Chapter 4 looks to assess the suitability of emerging hardware and programming paradigms, for a particular algorithmic domain through the use of existing software. By addressing the question "is it feasible to extend the use of an existing benchmark application to explore such suitability?", it indicates a new approach is required.

Chapter 5 introduces the idea of the mini-app and, by highlighting the step-by-step approach required to develop a particular variant of the mini-app, assesses if it is a feasible approach for an emerging technology assessment.

Chapters 6 and 7 then respectively demonstrate how both hardware and software comparisons can be made using such a mini-app based approach.

Finally, Chapter 8 details how the research undertaken as part of this study has further extended to other academic research opportunities, how it has been applied in practice in industrial environments and summarises the research and provides some thought provoking speculative ideas of possible future directions in the field of study, concluding the thesis.

Appendix A provides a dedicated, standalone description of all of the hardware systems utilised throughout this study.

The research contained in this thesis spans multiple years: 2010 until early 2017; with chapters written throughout this period. By the nature of such a relatively long period coupled with a fast changing high performance computing industry, by specifying when a chapter was composed and (in Appendix A when HPC systems were commissioned, the author aims to put the research into the context as the particular time of writing.

1.4 Thesis Contributions

As part of this research, this thesis describes the following novel contributions from the author:

1.4.1 Impracticalities of Using Production-Class Codes to Explore Architectures and Programming Models

Demonstrates lack of flexibility of an industrial-class benchmark code as a tool for the rapid exploration of emerging architectures and their associated programming models.

The standard practice of using benchmark codes to assess system upgrades to an incumbent platform and for comparisons for procurement of new platforms was demonstrated using a representative, industrial-class, benchmark application. This was utilised for exploring the feasibility of using such a benchmark to assess emerging technologies and reached the conclusion, due to time constraints and its practicality, that such an assessment requires a different approach.

The benchmark chosen is representative of all such benchmark applications, in that it contains non-localised compute sections of code, lack of good data parallelism, loop carried dependencies and non-optimised memory access patterns.

This work was published in the following research paper: Herdman, J. A., et al. "Benchmarking and Modelling of POWER7, Westmere, BG/P, and GPUs: An Industry Case Study."; which was presented at the 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 10) held in conjunction with IEEE/ACM Supercomputing 2010 (SC'10) New Orleans, LA, USA, and subsequently published in ACM SIGMETRICS Performance Evaluation Review 38.4 (2011): 16-22 [103]

1.4.2 Introduction and Extension of the Mini-Application Approach

Introduction of the CloverLeaf mini-application with development details to achieve a fully functional and portable OpenACC implementation.

This work introduced the concept of the mini-app, and described CloverLeaf, an explicit Eulerian hydrodynamics mini-app. The study detailed the step-by-step development process that produced a fully functional and portable version using a newly emerging standard OpenACC.

The resultant OpenACC versions of the mini-app were a subset of those versions of CloverLeaf which were accepted as part of the R&D 100 [172] award winning Mantevo test suite [104].

The development process detailed was also the subject of a Cray hosted, two day, XK6 programming workshop at Oak Ridge National Laboratory (ORNL) in October 2012 bringing together users of XK6 systems around the world to share experiences [77].

More recently the step-by-step approach was selected as a dedicated chapter in the technical book: "Parallel Programming with OpenACC" [88].

1.4.3 Demonstrating Performance Portability

Demonstration of the suitability of the mini-app as a tool for exploration of emerging architectures in the particular case of a GPU using three programming methodologies, namely OpenACC, OpenCL and CUDA.

The mini-application was demonstrated fully utilising a GPU-based architecture, with direct comparisons of performance, development time and "words of code" (WoC) when compared to the equivalent OpenCL and CUDA implementations.

This work was presented at SC'12 and subsequently published in the IEEE Companion: Herdman, J. A., et al. "Accelerating hydrocodes with OpenACC, OpenCL and CUDA." High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion: IEEE, 2012 [101].

1.4.4 Exploring Emerging Architectures

Extending the use of the CloverLeaf mini-app to explore a range of emerging architectures namely: GPUs, co-processor, APUs and current CPUs using OpenACC as a common baseline to compare against the performance of the best alternative programming models on each of the platforms analysed.

Contribution 1.4.3 was extended to further programming methodologies, enabling direct comparisons with regards to development time, maintenance effort, portability and performance on a GPU architecture. Subsequently, using OpenACC as a common baseline, further emerging hardware was assessed (co-processors, AMD APUs, GPUs) which enabled the optimal native programming methodology to be compared and contrasted against the OpenACC baseline.

This work was accepted and presented at the "First Workshop on Accelerator Programming using Directives" as part of SC14, winning the Best Paper award: Herdman, J. A., et al. "Achieving portability and performance through OpenACC." Proceedings of the First Workshop on Accelerator Programming using Directives. IEEE Press, 2014 [102].

Part I

Background Research

CHAPTER 2
Existing and Emerging Hardware

This chapter discusses those architectures that are at the forefront in today's HPC marketplace, and how they are beginning to evolve to address the power challenges of tomorrow. Additionally, it introduces those contending emerging technologies that can be thought of as a more disruptive solution to power constraints.

2.1 Established Hardware

This section discuses a number of the most prolific High Performance Computing (HPC) architecture options available to high-end HPC customers today. These have evolved over the years with x86 currently holding dominance.

2.1.1 x86-64

The x86-based architecture instruction set has existed since the late 1970s. Originating from Intel, but implemented by many manufacturers of microprocessors since. The first HPC machines to emerge based on x86 were clusters running the then emerging Linux operating system (OS). Spawning from the 1994 Beowulf project at NASA [57], to build a gigaflop (GFLOP/s) for under $ 50,000, Beowulf systems, reflecting the low cost, commodity component build of that original project began to emerge. These were more often than not based on standard building blocks connected together with a commodity interconnect. Typically these were modest sized platforms to fit a specific need and budget.

Sandia National Laboratory's (SNL) ASCI Red became the first true x86 based supercomputer based on commodity-off-the-shelf (COTS) components in 1997 and the first machine of any architecture to break the 1 TFLOP/s barrier. Since this landmark, the x86 architecture has seen a dramatic increase and today is a dominant presence in the list of the worlds top 500 supercomputers.

Addressing the unsustainable increase in power demanded by increasing unicore processor clock speeds, the x86 micro-architecture design evolved to multicore processors. The first of these available in dual-core, server form, appeared in 2005 with Intel's first Xeon branded dual-core processor *Paxville* [4], and AMD's first dual-core Opteron, the *Opteron 875* [1]. This multi-core strategy naturally evolved from dual-core to quad, hexa, octo, and deca-core CPUs.

In 2011 AMD released the 16-core Opteron 6272 Interlagos processors. However, these cores differed significantly to its predecessor the 12-core Magny Cours processor, in that they shared a subset of resources on the micro-architecture. Cores are paired into "modules" where each core shares a single FPU.

As of early 2017 the highest core count found in a server class processor is in the *Broadwell* generation Intel®Xeon®E7-8894 v4 [9], where a CPU has up to 24 cores.

These multi-core processors also support two-way simultaneous multi threading (SMT). This is essentially a mechanism that aims to keep as many of the processors functional and arithmetic logical units (ALUs) busy, by exploiting thread level parallelism (TLP) through the use of multiple hardware threads; and also has the potential to hide memory access latency. Figure 2.1 is a node diagram of Shepard (Appendix A.1.3) a 16 core Intel Xeon E5-2698 v3 [7] *Haswell* system based at SNL, which depicts the node-level complexity of a contemporary x86 processor.

2.1.2 IBM®POWER®

Once the dominant supercomputing processor, accounting for a quarter of the Top 500 supercomputers in November 2003 [146], the IBM®POWER® (Performance Optimized With Enhanced RISC) architecture emerged in 1990.

The first supercomputer built from the POWER architecture was the SP2, based on the IBM®POWER2 Architecture™. The IBM®POWER3®then saw the emergence of the SMP (Symmetric Multi Processor), with up to 16 processors on the Knighthawk2 node variant: the heart of the then number 1 system on the November 2000 Top 500 list ASCI White [145]. IBM®POWER4®had the distinction of being the first multi-core processor of any micro-architecture, to have two cores on a single die, in 2001. However, the relative gains in price to performance of mainstream x86 multiprocessors compared to the niche market of the POWER architecture, saw its dominance wane as the main supercomputing processor.

IBM®POWER5®introduced 2-way SMT (simultaneous multi-threading), the largest machine build from this generation of POWER was LLNL's ASC Purple in 2005 [135]. IBM®POWER6®again had 2-way SMT in a dual core format, but computed in-order rather than out-of-order instructions like previous POWER generations. The POWER6 also reached the highest clock speeds of any generation before or since, with variants available up to 5.0 GHz [71].

IBM®POWER7®, released in the last quarter of 2009 in its server variant, saw an increase in on die cores from 2 to 8, an increase in SMT from 2 to 4 and a switch back to out-of-order execution. However, the collapse of the planned

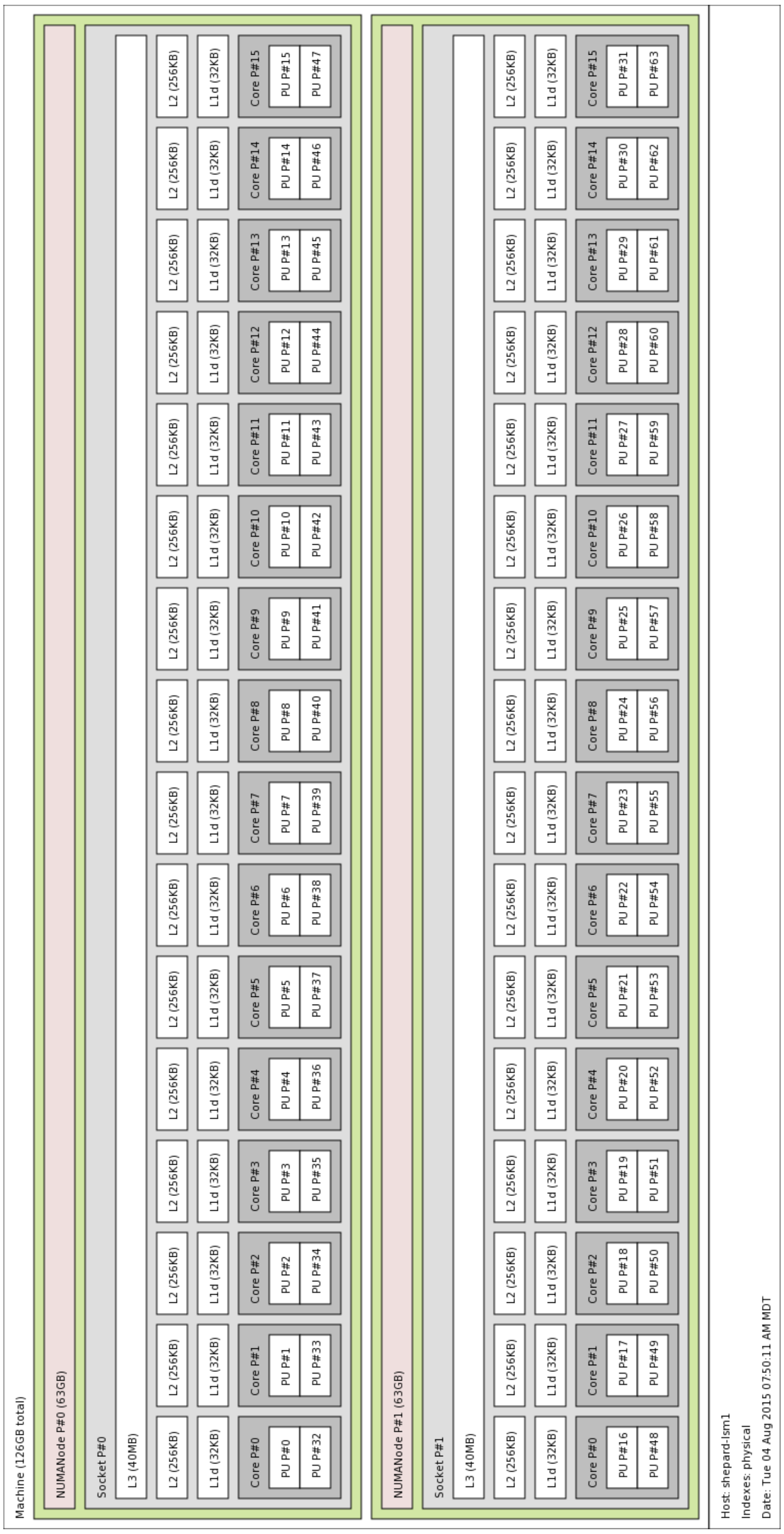


Figure 2.1: Example of a contemporary Intel®Xeon®x86 Node

2011, 10 PetaFlop (PFLOP) POWER7 based p775 Blue Waters system at the National Center for Supercomputing (NCSA) [13] struck a blow for the POWER PC architecture.

Today IBM offers the IBM®POWER8®processor; it is available with up to 12 cores and can run in 2, 4, or 8-way SMT mode. Figure 2.2 (Appendix A.2.5) is a node diagram of "White", a dual socket, 10 core POWER8, based at SNL.

By comparing with the x86 nodes in Figure 2.1 it show the differences in architecture are primarily down to core count, balance of memory hierarchies and levels of hardware multi-threading. These differences are contrasted in further detail in Section 2.5 with additional comparisons to today's "many-core" architectures. Despite the relative differences, there is also commonality in the levels of complexity found in such established hardware..

The architecture choices so far described can be thought of as mature, traditional, multi-core architectures.

2.2 The first "Many-Core" Architectures

2.2.1 IBM®Blue Gene®

The IBM®Blue Gene®architecture spans three generations: Blue Gene/L, Blue Gene/P and Blue Gene/Q. Blue Gene/Q is at the heart of the 1.6 million core Sequoia platform, a 20 PFLOP/s platform based at Lawrence Livermore National Laboratory (LLNL) which was qualified for production in 2012. Using the 1.6 GHz IBM®PowerPC®A2 processor, it has 4-way multi threading and a quad vector double precision (DP) fused multiply add (FMA), that IBM refer to as QPX (Quad Processing eXtension). Each chip has 18 cores, including one redunant core and one core which is used for OS functions, this presents itself to the end user as a 16 core processor. Blue Gene/Q follows the first and second Blue Gene generations: Blue Gene/L which had two 700 MHz IBM®PowerPC 440™, cache incoherent cores and Blue Gene/P with four 850 MHz IBM®PowerPC 450™, and 4-way SMT, now with cache coherency between cores. Initially the Blue Gene architecture was developed to target the specific field of protein folding but also with a secondary purpose to study extreme scale architectures to move towards incorporating many more parallel processing elements with a reduced power footprint compared to traditional CPUs. With the release of systems in 2012, Blue Gene/Q had the distinction of topping all three recognised supercomputing rating lists: Top500 (fastest floating point), Green500 (energy efficient: FLOP/s/Watt) and the Graph500 (for data intensive loads).

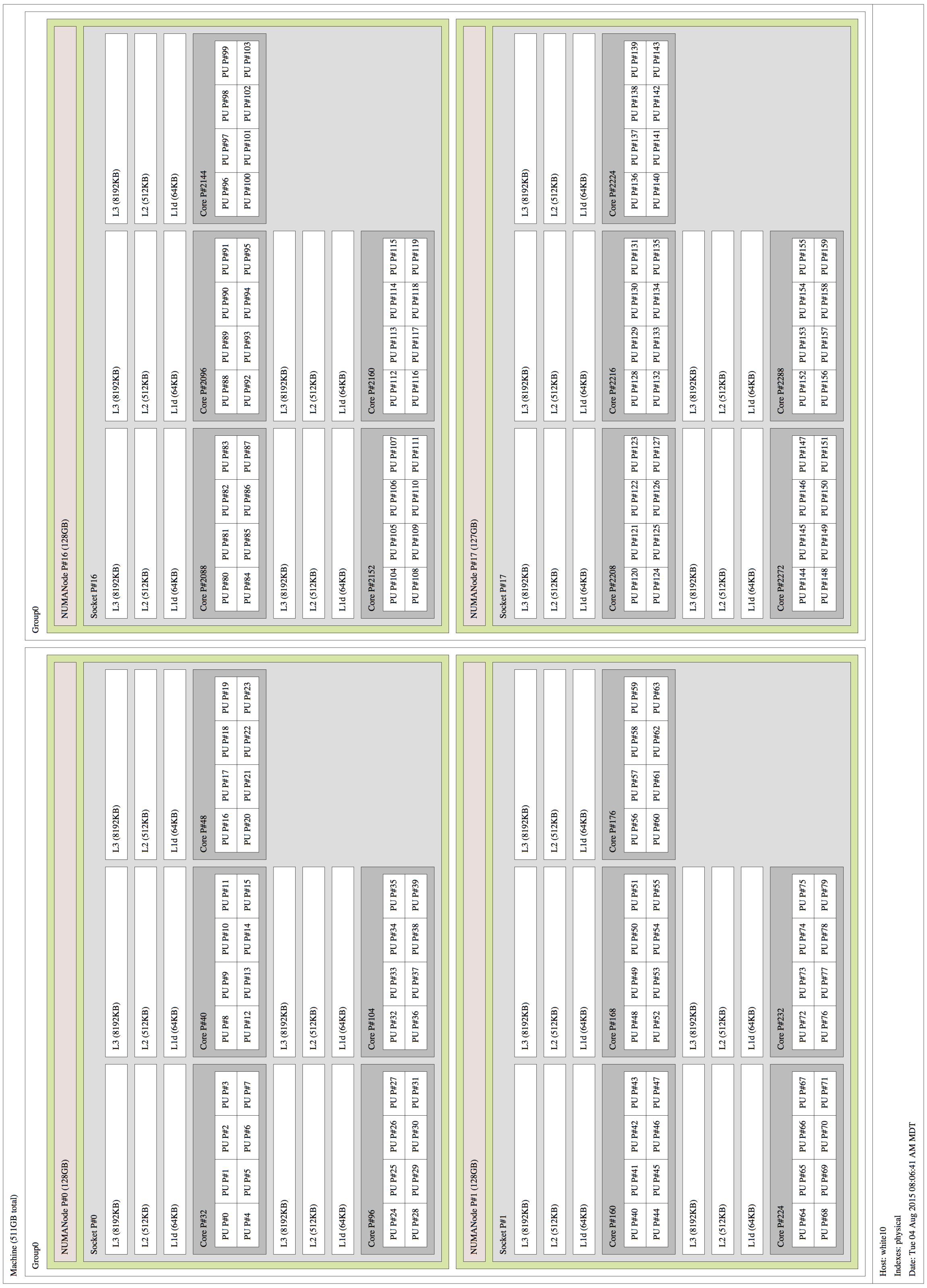

Machine (511GB total)
Group0
NUMANode P#0 (128GB)
Socket P#0
L3 (8192KB) L2 (512KB) L1d (64KB)
Core P#32: PU P#0 PU P#1 PU P#2 PU P#3 PU P#4 PU P#5 PU P#6 PU P#7
Core P#40: PU P#8 PU P#9 PU P#10 PU P#11 PU P#12 PU P#13 PU P#14 PU P#15
Core P#48: PU P#16 PU P#17 PU P#18 PU P#19 PU P#20 PU P#21 PU P#22 PU P#23
Core P#96: PU P#24 PU P#25 PU P#26 PU P#27 PU P#28 PU P#29 PU P#30 PU P#31
Core P#104: PU P#32 PU P#33 PU P#34 PU P#35 PU P#36 PU P#37 PU P#38 PU P#39
NUMANode P#1 (128GB)
Socket P#1
Core P#160: PU P#40 PU P#41 PU P#42 PU P#43 PU P#44 PU P#45 PU P#46 PU P#47
Core P#168: PU P#48 PU P#49 PU P#50 PU P#51 PU P#52 PU P#53 PU P#54 PU P#55
Core P#176: PU P#56 PU P#57 PU P#58 PU P#59 PU P#60 PU P#61 PU P#62 PU P#63
Core P#224: PU P#64 PU P#65 PU P#66 PU P#67 PU P#68 PU P#69 PU P#70 PU P#71
Core P#232: PU P#72 PU P#73 PU P#74 PU P#75 PU P#76 PU P#77 PU P#78 PU P#79
Group0
NUMANode P#16 (128GB)
Socket P#16
Core P#2088: PU P#80 PU P#81 PU P#82 PU P#83 PU P#84 PU P#85 PU P#86 PU P#87
Core P#2096: PU P#88 PU P#89 PU P#90 PU P#91 PU P#92 PU P#93 PU P#94 PU P#95
Core P#2144: PU P#96 PU P#97 PU P#98 PU P#99 PU P#100 PU P#101 PU P#102 PU P#103
Core P#2152: PU P#104 PU P#105 PU P#106 PU P#107 PU P#108 PU P#109 PU P#110 PU P#111
Core P#2160: PU P#112 PU P#113 PU P#114 PU P#115 PU P#116 PU P#117 PU P#118 PU P#119
NUMANode P#17 (127GB)
Socket P#17
Core P#2208: PU P#120 PU P#121 PU P#122 PU P#123 PU P#124 PU P#125 PU P#126 PU P#127
Core P#2216: PU P#128 PU P#129 PU P#130 PU P#131 PU P#132 PU P#133 PU P#134 PU P#135
Core P#2224: PU P#136 PU P#137 PU P#138 PU P#139 PU P#140 PU P#141 PU P#142 PU P#143
Core P#2272: PU P#144 PU P#145 PU P#146 PU P#147 PU P#148 PU P#149 PU P#150 PU P#151
Core P#2288: PU P#152 PU P#153 PU P#154 PU P#155 PU P#156 PU P#157 PU P#158 PU P#159
Host: white10
Indexes: physical
Date: Tue 04 Aug 2015 08:06:41 AM MDT

Figure 2.2: Example of a contemporary IBM®POWER8®Node

2.2.2 IBM® "Roadrunner"

The first true hybrid, or accelerated supercomputer where applications executed on all the heterogeneous components, was IBM's Roadrunner platform [48], installed at LANL.

It combined two different kinds of processors: 6,563 dual-core general purpose AMD LS21 Opteron blades, with each Opteron blade linked to two IBM BladeCenter QS22 PowerXCell 8i "Cell/B.E." blades.

This resulted in 12,960 "Cell" processors, each an enhanced / adapted version, of the specialised processor at the heart of the Sony Playstation 3.

Although a "one-off", the hybrid nature of Roadrunner was a forerunner to the hybrid systems, albeit not "Cell" based, that are emerging today.

2.3 Today's "Many-Core" Architectures

Today's conventional multi-core processors as described in Section 2.1 could be scaled up to build the exascale class machines that HPC is demanding. However, even when taking into account the predicted increases in technology, power demands on such a system would be in the region of 200 MW which equates to around $300M in annual operational electricity costs [84]; clearly this is not a feasible solution. To address this, the emergence of more simplistic, lower power highly parallel many-core processors are coming to the fore.

These manifest themselves in various ways depending on the type of parallelism they seek to exploit. However, they broadly fall into the two categories that were observed in the pioneering systems in Section 2.2: namely systems with lots of relatively slow low power cores (Blue Gene) or traditional CPU based systems with some form of attached "accelerator" (Roadrunner).

In many cases this results in a full system solution that contains different many-core devices resulting in heterogeneous architectures in which all hardware components are available to execute an application are not identical.

This section describes these architectures that are emerging into the HPC market.

2.3.1 Graphics Processing Units (GPUs)

As their name suggests, a Graphics Processing Units (GPUs) primary role is to process vast numbers of individual pixels extremely quickly. However, during their evolution from fixed function devices, to configurable devices, to programmable devices for graphics, and ultimately to enable the programmability of their floating point capabilities via NVIDIA®CUDA®(Compute Unified Device

Architecture) [78], the ability to utilise the GPUs computational resources can be now be realised for today's scientific computing demands.

Indeed, as attached devices via a PCIe (Peripheral Component Interconnect Express) interface, GPUs offer a way to build extremely large hybrid HPC platforms, with an increased FLOP/s/Watts ratio over a CPU only solution. Notable multi petaflop GPU-based HPC machines are Oak Ridge National Laboratory's (ORNL) Titan, Centro Svizzero di Calcolo Scientifico's (CSCS) Piz Daint, the Tokyo Institute of Technology's TSUBAME 2.5, and Tianjin's National Supercomputing Center's Tianhe-1A.

Essentially GPUs are a large (relative to a CPU) collection of multi-threaded SIMD processors that use their multi-threading capability to hide the latencies of accessing memory, rather than the cache hierarchy used for the same means on a CPU.

Although having significantly less memory than a traditional CPU; the GPU's memory has a much higher memory bandwidth. This is made possible, in part, due to the lack of constraints in its design that are required by the more general purpose memory found in the traditional CPU. A CPU needs to be able to access memory arbitrarily, while a GPU's memory can be laid out in simpler way with direct mapping of chunks of memory to blocks of ALUs. Furthermore, the CPU needs to manage OS interrupts, whilst the GPU does not. This results in wider memory buses and faster memory cycle than found on a CPU.

The most prevalent GPUs to be utilised in production level HPC architectures, aimed at scientific computing, with the necessary ECC (error correcting code) memory protection support, are NVIDIA's GPU cards.

The first generation of these GPU's was based on NVIDIA®Tesla®micro-architecture, consisting of a number (dependent on model) of what NVIDIA term its Streaming MultiProcessor (SM).

As hinted in the name: Streaming MultiProcessor, the SM is exactly that; issuing instructions to be executed and managing the data common to those instructions. In the case of the Tesla micro-architecture, each SM consisted of 8 GPU cores, where each GPU core is essentially a single precision Arithmetic Logic Unit (ALU).

The next generation was based on NVIDIA®Fermi®architecture, where a GPU could have (depending on model) between 7 to 16 SMs; where each SM has 32 GPU cores each which can configure its 64k local memory between shared memory or an L1 cache; all 32 GPU cores on the SM share a unified L2 cache.

The NVIDIA®Kepler®micro-architecture followed Fermi in the NVIDIA GPU range, introducing a new SM, known as the SMX. Here a high end Tesla GPU could have up to 15 SMX, where each of these contained 192 GPU cores. In addition to retaining the configurable 64k local memory as found in Fermi, a

48k read only data cache is utilised. Also the shared L2 cache was double over what was available with the Fermi.

Following Kepler the next micro-architectures with HPC variants planed are NVIDIA®Pascal®and NVIDIA®Volta®. In the case of the latter, machines are already planned for operation in the 2018 time frame; these will be in the region of 150 PFLOP/s. [136, 156].

2.3.2 Intel®Xeon Phi™

Intel Xeon Phi is a many-core multiprocessor system on-a-chip design. It evolved from an initial research project at Intel to investigate improved performance and power efficiency using x86 cores, which led to the Larrabee processor [185], aimed at graphics applications. Intel's "TeraScale" processor program looked at the feasibility and resultant issues from packing many-cores onto a die, this resulted in the 80-core Teraflops "Polaris" concept processor [193], which realised over 1 TFLOP/s from a single chip and the "Rock Creek" SCC (Single-chip Cloud Computer) concept [111], a processor containing 48 Intel iA cores connected via an on-chip mesh network, produced to explore the software needed to exploit such a many-core architecture.

These research activities ultimately led to the Intel Many Integrated Core (MIC) Architecture: in-order x86 cores, with wide vector units and greater SMT than the familiar Intel Xeon CPU, running a stripped down version of the Linux operating system (OS).

The first, proof of concept, device using the MIC architecture was known as Knights Ferry (KNF). This prototype contained 32 in-order, 1.2 GHz cores with 4-way SMT, but only supported single precision instructions.

The second generation based on the MIC architecture Knights Corner (KNC) was also the first to be branded under the Intel Xeon Phi name, and made commercially available in three variants the 3120, 5110, and 7120 with 1.1 GHz 57-cores, 1.05 GHz 60-cores, and 1.24 GHz 61-cores respectively. All three variants are available in either passive or actively cooled attached PCIe co-processor cards.

Each core has a 32KB L1 data cache and a local 512KB L2 cache, with access to all other core's L2 caches; unlike an Intel Xeon there is no shared L3 on the Intel Xeon Phi. At 512-bit, the vector instructions on the Xeon Phi are double that on the latest Intel Xeon CPUs. All cores are connected on the coprocessors via an on-chip bidirectional ring interconnect providing cache coherency.

As the Intel Xeon Phi is an x86-based co-processor, running a Linux OS, applications that compile and run under the standard x86 architecture will run "out-of-the-box" on the Intel Xeon Phi, requiring only a re-compilation to account for the instruction incompatibility with Intel Xeon.

Released in 2016, the second generation Intel Xeon Phi, Knights Landing (KNL), comes in a self bootable variant, in addition to an attached co-processor variant. Both variants have an increased core count over the KNC.

Already there are confirmed large system procurements that will utilise the KNL: Trinity at Los Alamos National Laboratory (LANL), Cori at the National Energy Research Scientific Computing Center (NERSC).

Additionally, the third generation Knights Hill (KNH) will be the basis for the "Aurora" machine, due to be delivered to the Argonne National Laboratory (ANL) in 2018.

2.3.3 Accelerated Processing Units (APU)

Initially known as the Fusion Accelerated Processing Unit (APU) architecture, AMD's on-chip combined CPU and GPU architecture aims to integrate the two more closely enabling the sharing of data between the CPU and GPU by way of a unified coherent memory space. Primarily aimed at the graphics market, these are analogous to the Intel®Core™CPUs (rather than HPC's server class, Intel Xeon, building block), introduced in their Sandy Bridge generation, these also contain their own integrated GPUs.

Although more tightly coupled than discrete GPUs linked via a PCIe, this is at the expense of losing the high memory bandwidth available from those discrete devices.

However, AMD's plans for future HPC architectures are via the building blocks of future generation APUs, hence there is merit in understanding their architectural characteristics, and the mapping of scientific applications to them.

Beginning with the Kaveri generation, support for the Heterogeneous System Architecture (HSA) is adopted. This aims to make the heterogeneous nature of the APU opaque to the scientific programmer, allowing those traditional applications using high level programming languages such as Fortran and C to utilise both CPU and GPU cores, without the necessity to re-code in CUDA or OpenCL.

The four generations of AMD's APUs are compared and contrasted in Table 2.1.

2.4 Emerging Technologies

2.4.1 ARM®

Although the most prevalent Instruction Set Architecture (ISA), running in billions of embedded devices world wide, it is only the current, 64-bit ARMv8,

Generation	**Llano**	**Trinity**	**Richland**	**Kaveri**
CPU	A8-3850	A10-5800K	A10-6800K	A10-7850K
Clock Speed (Turbo)	2.9 (N/A)	3.8 (4.2)	4.1 (4.4)	3.7 (4.0)
No. Cores	4	2	2	2
GPU	HD 6550	HD 7660D	HD 8670D	R7
No GPU “cores”	400	384	384	512
Released	Aug 2011	Oct 2012	Jan 2013	Jan 2014

Table 2.1: AMD’s APU Comparisons

instruction set that will enable ARM’s potential emergence into the server and HPC market.

At the time of writing, 64-bit ARM SoC’s (system-on-a-chip) are being touted from AMD, Applied Micro, Cavium Newtworks, Qualcomm and Texas Instruments, although to be successful, full system and software eco-systems need to be developed in order to enable the adoption from scientific applications.

2.4.2 OpenPOWER®

The OpenPOWER®Foundation, formed in 2013, is a group of technology members to which IBM have made available their POWER technology. The aim is to foster collaboration in the development of future systems based on the POWER CPU.

The first of these technologies targeting HPC is the Firestone server, released at the end of 2015. This incorporates two POWER8 processors alongside two NVIDIA Kepler K80 GPUs. These 2U servers have the ability to be scaled out via Mellanox IB.

It is this technology which will deliver the Leadership Class HPC platforms based on GPU acceleration discussed in Section 2.3.1, [136, 156].

2.5 Common Architectural Traits

Although Section 2.3 details a diverse range of emerging technologies, in an HPC context, these technologies all aim to achieve the same goal, which is to use their technology as the building blocks to deliver a high performing architecture, capable of enabling the largest scientific problems to be solved.

Although, each technology proposes a different solution to achieve this goal, each solution is essentially using different levels of components and parameters from a common design space.

This section breaks down, into their constituent components each of the architectures which were previously discussed. This helps to provide an understanding of the balance placed on underlying hardware solutions. It introduces some common "language" to describe characteristics, which are not consistent, and often conflicting between architectures.

This has previously been addressed by Gaster et al. [93] who describe the diverse range of hardware that OpenCL can operate on by describing the various methods to achieve performance on these architectures, and the trade-offs taken in the design space for each. Patterson and Hennessy [161] give a "GPU Rosetta Stone" which provides a quick guide to "translate" GPU terminology to that of CPUs, and vice versa. From a compiler point of view, the article from Leback et al. [130], presents the architectural aspects of an Intel Xeon CPU, NVIDIA Kepler GPU, Intel Xeon Phi, and an AMD Radeon$^{\text{TM}}$, firstly in their native terminology, and subsequently in that of a CPU or GPU. AMD [46] introduces the concept of a "compute core" irrespective of whether it is a CPU or GPU core, defining it as: *"any core capable of running at least one process in its own context and virtual memory space, independently from other cores".*

As the focus of this thesis is from the point of view of an application developer wishing to understand how best to exploit a given architecture, the comparison in design space choices for hardware solutions is mapped to the varying levels of parallelism required from an application to exploit such features.

One of the main issues when trying to compare and contrast different underlying hardware, is the inconsistency in terminology. For instance in GPU parlance a "core" is effectively a single precision SIMD processor, which in CPU terms is equivalent to a 32-bit vector lane in a modern CPU. While a core in CPU terms is a single processor, part of a microprocessor, made up of a control unit which issues instructions to integer and floating point vector (SIMD) processing units in order to execute these instructions. So, at a high level of abstraction it can be seen that a CPU core can be thought of as an equivalent to a GPU's SM.

If architectures are considered in terms of an *integrated circuit* or *chip*, broken down into multiple *processing cores*, or for brevity: *cores*. These *cores* consist of a control unit able to issue instructions; some vector, integer, and floating point processing units, capable of carrying out the instructions along with some memory hierarchy; effectively defining a *core* as an individual compute engine. When considered in this simplistic way, architectures can be compared and contrasted to show how their performance is achieved by using these basic building blocks in differing levels of degree. Although not drawn to scale, in the following Figures these building blocks are indicative of the sizes relative to each architectural technology.

Table 2.2 summarises the trade-off in the architectural design space. Ultimately, they are trade-offs in performance to power and what can physically

Table 2.2: Relative Architecture Design Dimensions

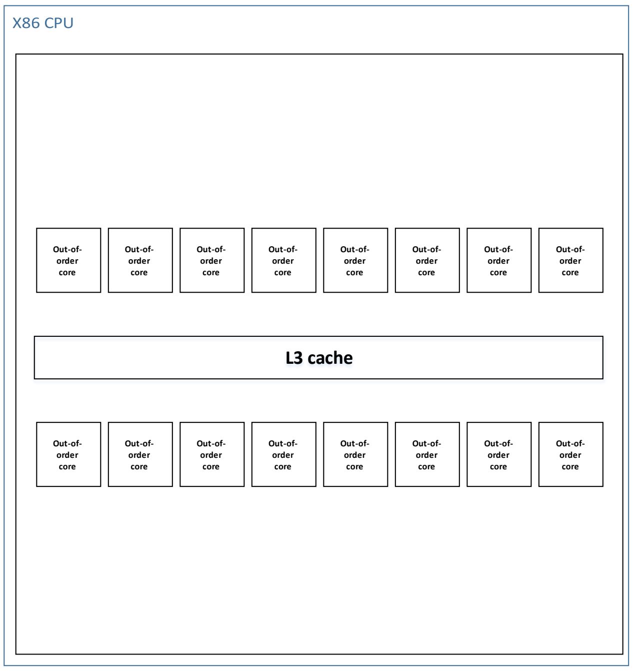
Architecture	**Clock Frequency**	**Instruction Cycle**	**LLC (Shared)**	**VPU**	**SMT**
Intel x86 CPU	16	out-of order	L3	8	2
NVIDIA Fermi GPU	16	in order	L2	32	N/A
NVIDIA Kepler GPU	15	in order	L2	192	N/A
Intel Xeon Phi	60	in order	N/A	16	4
IBM POWER8	12	out-of order	L3	4	8

fit into the finite space on the underlying silicon. However, there is also the trade off in productivity: can the end user of the hardware make good use of the architectural feature to make it worth while including.

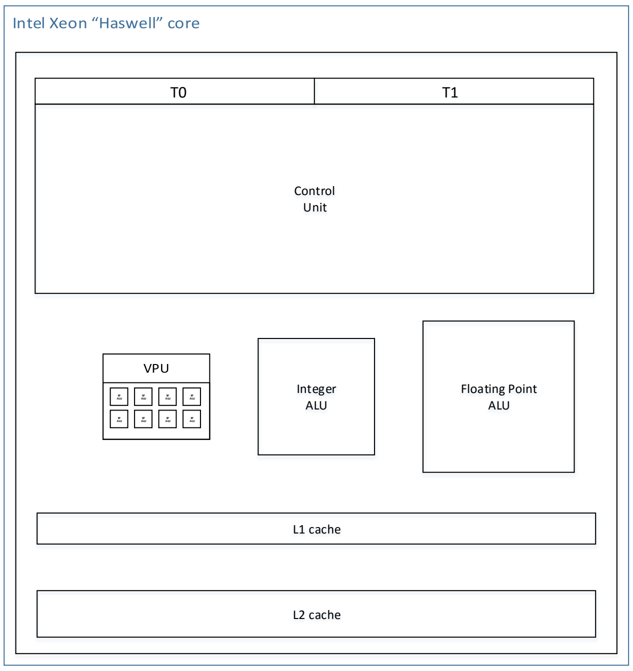
Figure 2.3 shows a generic 16-core variant of an Intel Xeon *Haswell* CPU. Figure 2.3a shows each *Haswell chip* has 16 out-of-order *cores* with a shared L3 cache, where each *core* is depicted in Figure 2.3b with two hyper-threads targeting a complex control unit which issues instructions that can be carried out by a 256-wide (advanced vector extension (AVX) supported) vector processing unit (VPU), an integer ALU, and a floating point ALU. Where the former is essentially eight 32-bit ALUs. For each processor, the data associated with these instructions has its own L1 and L2 cache.

Compare this with Figures 2.4 and 2.5, depicting an NVIDIA Fermi 16 SM M2090 and an NVIDIA Kepler 15 SMX GK110 generations of GPU respectively. Although displaying a similar number of *cores* per *chip* (where "core" is the local definition above of *processing core*, not to be confused with the GPU terminology of "core"), the *cores* themselves are very different to that of the Intel Xeon in Figure 2.3b. With a clock frequency of 1.1 GHz, around one third of that of the Intel Xeon, each *core's* computational units are primarily VPUs, on a much larger scale (192 32-bit ALUs in the case of the NVIDIA Kepler in Figure 2.5b) than in the Intel Xeon.

Figure 2.6 shows this concept for the Intel Xeon Phi 5110P, a 1st generation Intel Xeon Phi, familiarly called the "Knights Corner" or KNC. Comparing Figure 2.6a with the x86 Intel Xeon in Figure 2.3a, it is immediately apparent that the number of *cores* per KNC *chip*, is much higher. Although the 60 *cores* are simpler in-order and lower frequency, (1.0 GHz) than the Intel Xeon equivalent; they are also different in the balance of the computational units. Figure 2.6b show that the VPUs on the KNC are twice that of the Intel

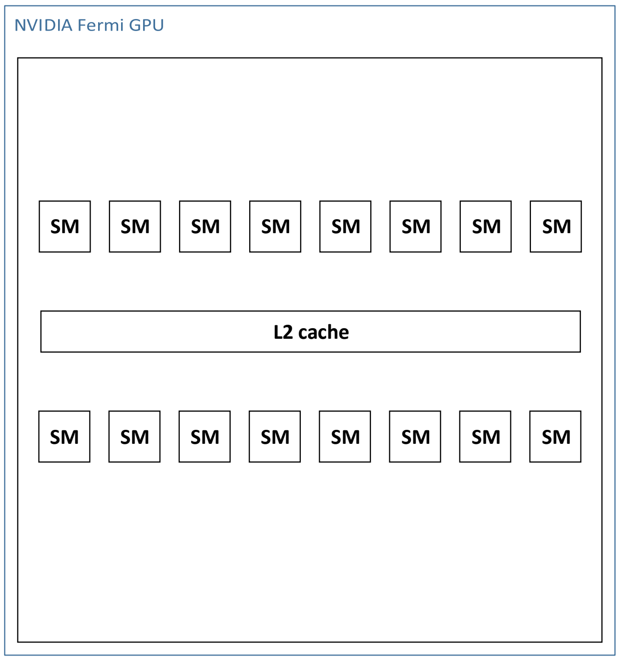


(a) 16-core Intel®Xeon®CPU

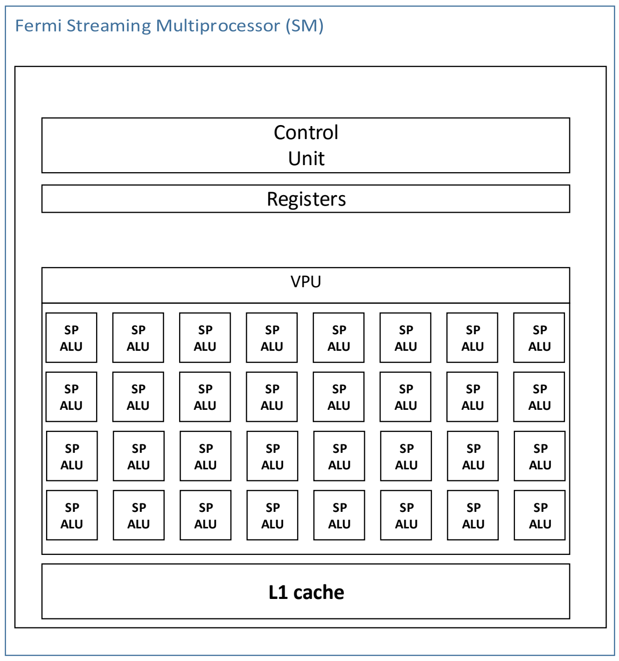


(b) Intel®Xeon® *Haswell* core

Figure 2.3: Intel®Xeon® *Haswell* Conceptual Diagram

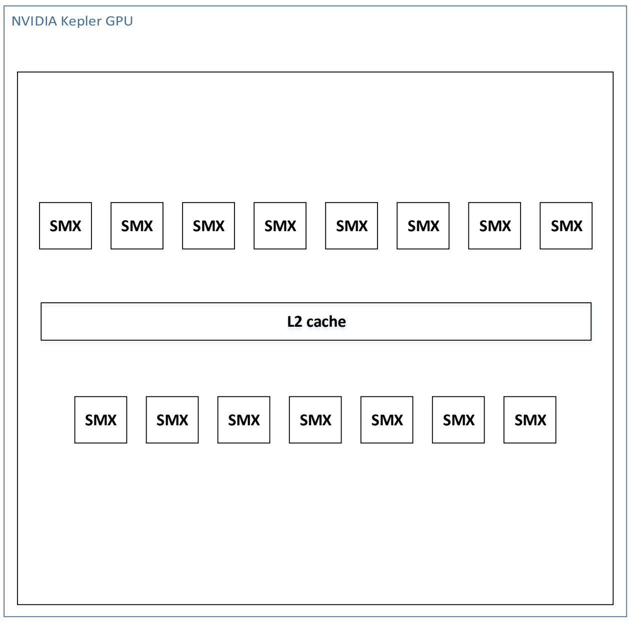


(a) NVIDIA®Fermi™GPU

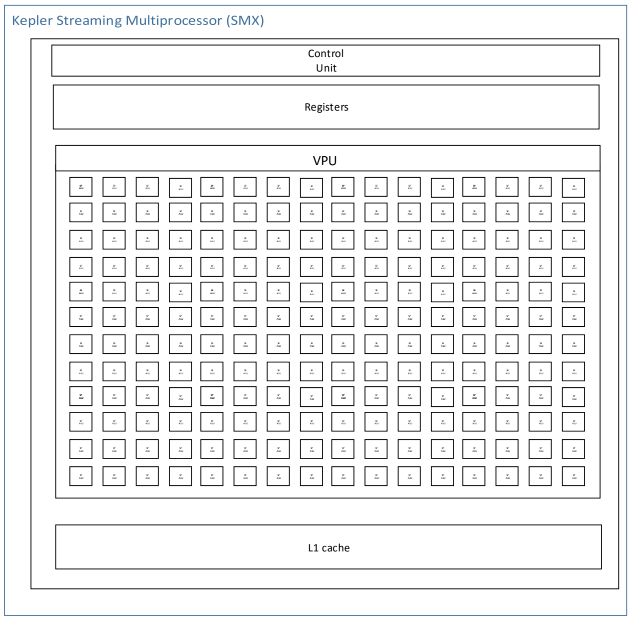


(b) NVIDIA®Fermi™SM

Figure 2.4: NVIDIA®Fermi™GPU Conceptual Diagram



(a) NVIDIA®Kepler™GPU



(b) NVIDIA®Kepler™SMX

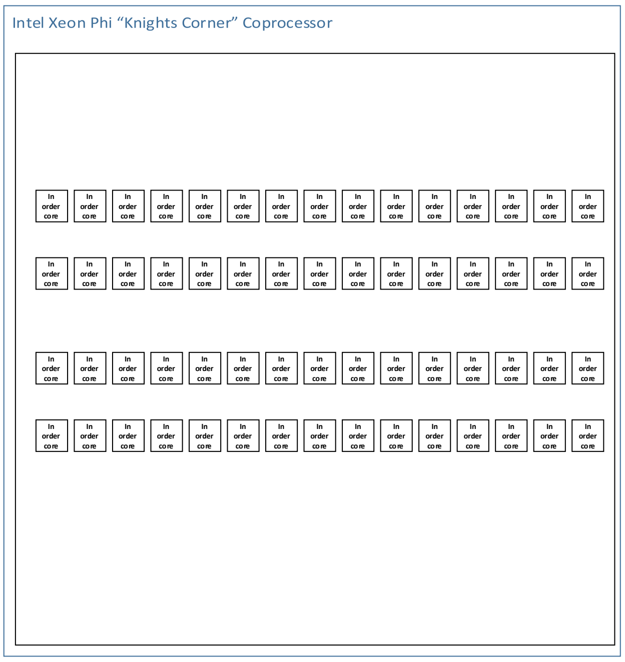
Figure 2.5: NVIDIA®Kepler™GPU Conceptual Diagram

Xeon, equating to 16 32-bit ALUs. Also each *core* supports four hyperthreads, analogous to the two supported in the Intel Xeon.

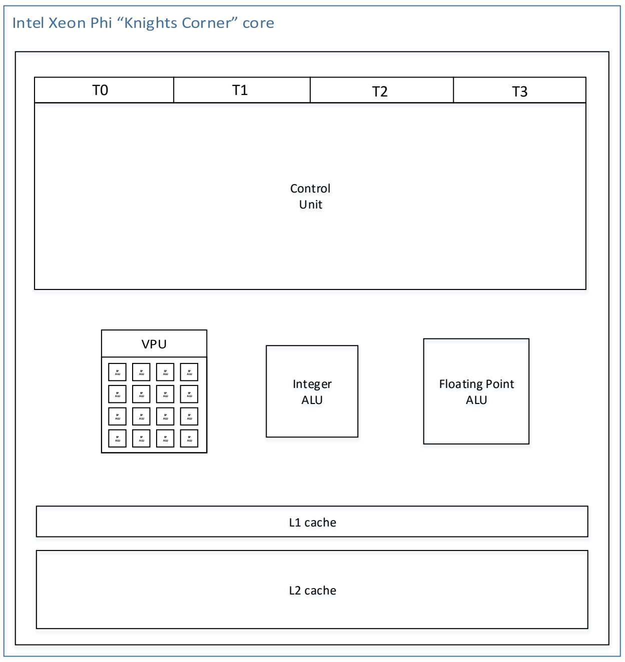
A 12-core IBM POWER8 is depicted in Figure 2.7. Here the 12 *cores* on a *chip* (Figure 2.7a), share a comparatively large L3 cache. Each *core*, shown in Figure 2.7b can store the state of up to 8 way SMT (simultaneous multi-threads) (cf. with the two in the Intel Xeon and the four in the Intel Xeon Phi in Figures 2.3b and 2.6b respectively), where computational units consist of complex floating point and integer units. There is also support for 128-bit vector multimedia extension (VMX) instructions, that is a POWER8 *core* can process four single precision floating point operations, depicted in Figure 2.7b, as a VPU with four 32-bit ALUs.

This diverse range of heterogeneous architectures represents a complex challenge for HPC. Understanding which hardware is best suited for an application mix involves a range of mitigating factors including application performance and application re-coding, to effective scheduling and running costs (e.g. power and cooling).

These diagrams notionally show the relative balance in the design space between the various “many-core” architectures previously described. With these conceptual comparisons in mind, Chapter 3 considers the alternative programming models available to the application developer and their potential suitability.

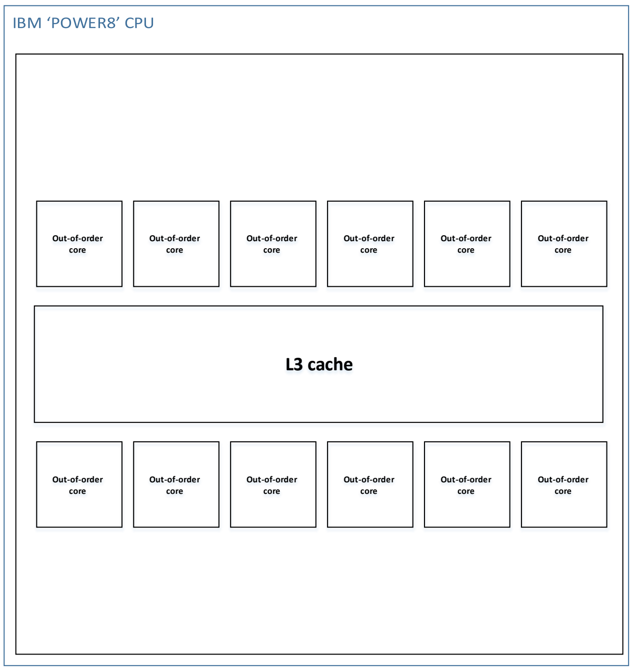


(a) 60-core Intel®Xeon Phi™ *KNC*

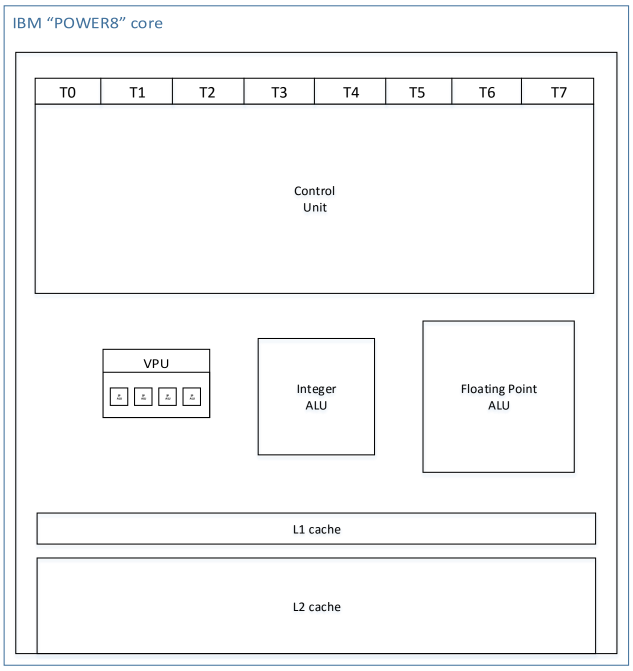


(b) Intel®Xeon Phi™ *KNC* core

Figure 2.6: Intel®Xeon Phi™ *KNC* Conceptual Diagram



(a) 12-core IBM®POWER8®CPU



(b) IBM®POWER8®core

Figure 2.7: IBM®POWER8®Block Diagram

CHAPTER 3
Programming Methodologies

Chapter 2 discussed the range of emerging technologies and their architectural traits. HPC application developers wishing to utilise these architectures, are faced with a dilemma relating to how they should develop application software for these competing hardware devices whilst still maintaining scientific productivity.

This is an important question, as outside the area of pure research, the goal is to reduce time to solution for production level applications. This implies that there needs to be a balance between the time taken to port and maintain an application in addition to any gains made in the solution time of the application.

This problem is compounded in that there is not a single emerging hardware architecture, but multiple. Porting and maintaining a different version of a production level application for different targeted platforms is not only undesirable, but impractical.

Programming for the large number of lightweight cores and vector units offered by these devices means departing from the traditional distributed MPI approach, to a tiered programming model which is designed to harness both coarse and fine grained parallelism.

Accelerated programming platforms and application interfaces like NVIDIA CUDA [78] and the Khronos Group's OpenCL [125] require explicit parallel programming using library calls and specially written compute kernels. While directive-based solutions like the OpenACC Application Programming Interface [23] offers a directive-based approach, similar to that found in OpenMP [21], for describing how to manage data and execute sections of code on the device.

These two approaches of exposing multi-level node parallelism within application programs have different effects on factors such as programmer productivity, the time required for modifying the code, programming language of choice, required application performance, and portability.

Runtime environments such as SGPU2 [159] and XKAAPI [95] aim to implicitly manage the resources available in heterogeneous node systems.

Additionally, there is the emergence of a myriad of "portability layer libraries" such as RAJA [109] from LLNL, Kokkos [87] from SNL, Bolt [183] from AMD, NVIDIA®Thrust® [105], University of Illinois's Charm++ [121] and Intel®Threading Building Blocks (TBB). All of which are primarily aimed at a specific problem of porting monolithic C/C++ legacy applications in a non-disruptive manner whilst maintaining performance portability. This is achieved by essentially providing a common interface to an application developer, while enabling the use of a range of programming abstractions within their lower level implementations or back-ends.

3.1 Low-Level Languages

Existing procedural structure applications need to be restructured or broken down into computationally distinct, self contained kernels and written in the low-level language of choice. Some, like CUDA, are only viable for specific architectures, whilst others, such as OpenCL are architecture agnostic but require architecture specific tuning to ensure performance portability.

3.1.1 NVIDIA®CUDA®

CUDA (Compute Unified Device Architecture) [78] is NVIDIA's parallel platform and programming model, and provides support for general purpose computing on NVIDIA GPUs [30]. It is based on the concept of a *host* and a *device*, where each, CUDA-based, computational kernel can be *accelerated* by being executed on the highly parallel *device*. This is achieved by treating the CUDA kernels as distinct entities to non-CUDA code, and compiling them using CUDA's C-based compiler, `nvcc`, for execution on the *device*, with non-CUDA code compiled for the *host* using its native compiler.

At runtime, the procedural code begins execution on the *host* and as each CUDA-based computational kernel is encountered, it is launched, moving its data and program code/instructions onto the *device* or *devices* attached to the *host*. Conversely, once computation of the kernel is completed, the associated data and program code/instruction are transferred back to the *host*.

In practice a *host* is usually a traditional x86 or ARM-based CPU, while a *device* is almost exclusively an NVIDIA-based GPU. This NVIDIA exclusiveness, results in the need to maintain duplicate code paths for applications: one for GPU and another for another architecture. An additional restriction is that CUDA is implemented by extending the C language's function declaration syntax, and hence is heavily dependent on the use of the C language. For non-C based applications to use CUDA, such as Fortran, then those computational kernels need to be C based, resulting in a Fortran host code calling C-based compute kernels. Although, there is a CUDA Fortran compiler [35] available exclusively from the Portland Group (PGI), who are in turn part of the NVIDIA Corporation.

3.1.2 OpenCL™

OpenCL is an open standard enabling parallel programming of heterogeneous architectures. Managed by the Khronos group and implemented by over ten vendors including AMD [25], Intel [29], IBM [33], and NVIDIA [34]; OpenCL code can be run on many architectures without recompilation. Emerging developments such as pocl (a Performance-portable OpenCL Implementation) [117], aspire to an architectural independent kernel-level compiler for OpenCL.

With such support it is possible to develop a single source that is hardware agnostic (CPU, GPU, coprocessors, APU) and can utilise all parts of a heterogeneous system that contains a mixture of such hardware, although support may be dropped for future generations of some of these architectures.

Also, the application developer is required to explicitly define the hardware and where to move the data, which requires programs to be explicitly developed for each architecture in order to achieve optimal performance portability. This results in a significant amount of boilerplate coding to do even a relatively simplistic program, albeit this boilerplate code can be re-used between programs so it only needs to be written once.

On inspection, OpenCL is simply an API to the C programming language, however this restricts direct interaction to applications which are Fortran based.

3.1.3 Intel®Cilk™Plus

Intel®Cilk™Plus is a language extension for C and C++ codes to handle task-based parallelism and vectorisation, through parallel loops, array notation and vectorised loops; it is also coupled with the Cilk Plus runtime.

Introduced in 2010 in version 12.0 of the Intel compiler, it consists of three main keywords (*cilk_for*, *cilk_spawn* and *cilk_sync*) which, respectively, enable loop iterations to execute in parallel, state that a function's caller can continue to execute without the need to wait for the function to return, and a synchronisation statement (cf. to MPI's *MPI_WAITALL*) that synchronises on all locally spawned functions.

Array notation enables sections of specified arrays to be vectorised; in a similar manner to vectorisation of computational loops, this is enforced via the *#pragma simd* directive (c.f. OpenMP 4.0's *#pragma omp simd*).

3.1.4 C++ AMP (Accelerated Massive Parallelism)

C++ AMP (Accelerated Massive Parallelism) [97] is a programming model from Microsoft®explicitly targeting GPUs for C++ applications running under Microsoft®Windows®OS. Available as of Version 2012 of Microsoft®Visual

Studio™, it contains extensions to C++ to enable the data movement between CPU and GPU hardware, and expressing parallelism through the *parallel_for_-each* function. Additionally, a parallel maths function library is also part of the model.

3.1.5 Pthreads

Pthreads (POSIX Threads) [153] is a portable threading standard available through a library for most C implementations. It provides a low level threading model, giving fine grained control over when tasks should fork or join via subroutine calls. Primarily aimed at managing system resources and carrying out individual tasks, it puts full responsibility onto the programmer to implement their own atomics and controls, hence it is not particularly applicable to an industrial strength scientific application.

3.2 High-level target-based directives

Directive-based solutions empower the application developer to explicitly parallelise their application through the introduction of directives, or pragmas, into their existing source code. Primarily, this is in the form of identifying suitably computationally intensive loops and parallelising each in turn by splitting their computation among the threads available on the targeted hardware.

When heterogeneous hardware platforms are being targeted, directives are used to transfer the relevant codes loops/sections, and their associated data, between the different processors from which the system is comprised.

For over two decades, the industry standard directive-based OpenMP [24], has dominated. However, with the emergence of heterogeneous hardware of a number of alternative solutions, from OpenHMPP (Hybrid Multicore Parallel Programming) [70], OpenACC®and Intel®Language Extensions for Offload [152] are coming to the fore. Although convergence between these models is beginning to occur. OpenMP 4.0 introduces new directives to target heterogeneous platforms, and OpenACC, in the latest PGI invocation, providing the ability to execute on the cores of a CPU.

Additionally, there are a number of niche solutions which extend the ideas developed in the previous models. StarSs [62] and OmpSs [86], for example, hide the synchronisation operations which are usually required to be carried out explicitly by the developer, by inferring when these operations should take place from the data dependencies inherent in the application code.

3.2.1 OpenMP

From the mid-1990's, OpenMP [24] has become the industry accepted paradigm for a directive-based parallel programming model. By inserting these directives into existing Fortran or C/C++ applications, primarily targeting computational intensive loops, the code developer can express the sharing of resources within the system and is thus able to express a shared memory parallel application.

By its nature, its main target has been shared memory, CPU-based systems, taking advantage of the shared memory available within CPU nodes. With the 4.0 release [21] support has been added to specifically target attached accelerated devices. Although supported in nearly all commercial and open-source compilers, the features which specifically relate to accelerated devices have only been implemented by a smaller subset of compiler vendor's offerings at the time of writing.

OpenMP allows an application to be implemented in a gradual manner and does not require the whole application to be parallelised. However, care is needed to ensure any OpenMP regions are thread safe (i.e. data independent), that is that modifications to global variables are appropriately synchronised and controlled. Limitations exist relating to the nesting of OpenMP directives, and issues in using OpenMP in some object oriented (OO) C++ applications. Lack of interoperability in a thread safe manner with the C++ standard template library (STL), has seen the emergence of some dedicated abstraction libraries (see Section 3.3) to support such applications via OpenMP.

3.2.2 OpenACC®

The OpenACC Application Program Interface (API) [23] is a high level programming model based on the use of directives. By applying directives to original Fortran, C or C++ source code it aims to provide increased architecture portability with minimal code modification. This increase in portability is offered without compromising code maintainability, a key consideration for existing complex industrial applications. At the time of writing three compiler vendors: CAPS[1] [26], Cray [31], and PGI[2] [131] support the initial OpenACC release.

Prior to the support of a common OpenACC Standard, Cray, PGI and CAPS each had their own bespoke set of accelerator directives from which their implementations of OpenACC is derived. A brief overview of each vendor's implementation, along with limitations, follows.

[1]As of June 27, 2014 CAPS Enterprise ceased trading, and the CAPS Compiler is no longer available.

[2]As of July 2013, PGI was acquired by NVIDIA.

Cray originally proposed accelerator extensions to the OpenMP standard [21] to target GPGPUs, through their Cray Compiling Environment (CCE) compiler suite. These evolved into the “parallel” construct in the OpenACC standard. Rather than creating a CUDA source for the kernels, CCE translates them directly to NVIDIA’s low-level Parallel Thread Execution (PTX) [155], a pseudo-assembly language subsequently compiled by the graphics driver into binary code. CCE is currently only available on Cray architectures, restricting portability.

As of version 10.4 of their compiler, PGI supported the PGI Accelerator model [19] for NVIDIA GPUs. This provided their own bespoke directives for the acceleration of regions of source code. In particular their “region” construct evolved into their implementation of the OpenACC “kernel” construct.

Initially, CAPS (Compiler and Architecture for Embedded and Superscalar Processors) provided support for the OpenHMPP directive model [70], which served as the basis for their implementation of the OpenACC standard. A major difference with CAPS is the necessity to use a host compiler. Here code is directly translated into the application developer’s choice of either CUDA or OpenCL [125]. In the case of the latter, this increases the range of architectures which can be targeted.

3.2.3 Intel®Language Extensions for Offload (LEO)

Intels Language Extensions for Offload (LEO) [152] consist of directive-based pragmas for use in C/C++ and Fortran based applications. These constructs are Intel specific and were introduced into the Intel compilers in order to target the Intel Xeon Phi as a way to run source code on a host Xeon CPU and “offload” marked sections, through the used of the *offload* pragma directive, onto the Xeon Phi co-processor.

3.3 Abstraction Libraries

Abstraction libraries, as their name suggests, are software layers, which through a one time implementation of their API aids in the detachment of software applications and the hardware platforms which they execute on. Ultimately the goal is to “future proof” an application irrespective of future hardware platforms.

3.3.1 Intel®Threading Building Blocks (TBB)

Aimed solely at enabling task based parallelism in C++ applications; Intel Threading Building Blocks (TBB) [177] is a template library extension to C++

that includes a global task scheduler and the ability to handle memory allocations and local thread storage. TBB consists of work-sharing expressions where everything is modelled as tasks, rather than using the fork/join paradigm prevalent in target-based directives or low-level programming language approaches.

3.3.2 RAJA

Developed at LLNL, RAJA aims to enable fine grained multi–threading to their legacy, multi-physics, MPI-distributed C++ applications, in order to enable the rapid transition and adaptation to the emerging architectural trends as described in Chapter 2. RAJA acts as an interface to the application code through the insertion of RAJA's *forall* loop template in place of C's traditional *for* loop. The RAJA form incorporates information regarding the execution policy and abstractions of the loop bounds to encapsulate loop execution details, thus decoupling specific hardware dependencies. With current back-end support for OpenMP and Cilk, RAJA is therefore able to interface with these back-end programming models transparently to the code developer.

To enable this via C++, RAJA needs to utilise C++ lambda functions, which enable the creation of anonymous objects which resemble functions. Albeit part of the C++ 11 standard, lambda functions are not supported fully by all vendor compilers, hence currently limiting the portability of RAJA enabled code.

3.3.3 Kokkos

Like RAJA, Kokkos is a library solution for C++ applications from SNL. Its back-end implementation allows mapping of existing C++ applications to a variety of alternative programming models. The choice of which programming model is to be enabled is based on the suitability of the model to the target architecture. Currently Kokkos supports OpenMP, CUDA and pthreads [153]. It also has the option to use the "hwloc" library [68] to assist with optimal mapping of threads to hardware cores.

Kokkos's interface to an application is through its library's API, which defines execution and memory spaces via a dedicated C++ class, which is imposed at compile time. Hence, an array and its layout in memory can be determined depending upon the hardware it is built on.

3.3.4 AMD®Bolt

Bolt [183] is a C++ standard template library (STL) designed to target GPUs. Developed by AMD, it contains a range of parallel primitive algorithms that can be customised by the code developer using C++ function objects. These objects are ultimately written in an OpenCL back-end to exploit the underlying targeted GPU hardware, yet abstracted away from the developer thus enabling a single source C++ to be targeted at either CPU or GPU.

3.3.5 NVIDIA®Thrust

Thrust [105] is template library from NVIDIA, for C++ CUDA enabled applications. Analogous to Bolt (see 3.3.4 above), it provides the application developer with a host of parallel primitives with which to abstractly describe their application.

3.3.6 Charm++

From the University of Illinois, Charm++ [121] extends C++, to provide an abstraction layer to application developers, hence hiding the underlying architecture.

3.3.7 OP2 / OPS

The OP2 stencil based framework from University of Oxford [96], provides an abstraction layer for the specific domain of unstructured mesh-based applications. Through the use of the OP2 API an application can utilise the numerous back-ends which the OP2 developers have implemented. OP2 currently supports OpenCL CUDA, OpenMP, MPI and OpenACC. OPS [176] is analogous to OP2 and is aimed at block structured mesh applications.

Part II

Research Contributions

CHAPTER 4
An Industry Case Study: Benchmarking and Modelling

Part I detailed the evolutionary trends driving the development of HPC hardware technology (Chapter 2) and described the main programming methodologies emerging to exploit them (Chapter 3).

By considering a current, production class application, this chapter examines the hurdles that are faced in determining the best way to migrate such an application to these emerging technologies.

An industrial strength benchmark, Shamrock, is introduced. Developed at the Atomic Weapons Establishment (AWE), Shamrock is a two dimensional (2D) structured hydrocode. One of its aims is to assess the impact of a change in underlying compute hardware, and (in conjunction with a larger HPC Benchmark Suite) to provide guidance in the procurement of future systems.

A representative test case and problem size is discussed, and subsequently executed on a local, high-end, workstation for a range of compilers and MPI implementations. Based on these observations, specific configurations are built and executed on a selection of HPC processor generations. These include Intel Xeon "Nehalem" and "Westmere" micro-architectures, IBM POWER5, POWER6, POWER7, Blue Gene/L, Blue Gene/P, and the AMD Opteron chip set. Comparisons are made between these architectures, for the Shamrock benchmark, and relative compute resources are specified that deliver similar time to solution, along with their associated power budgets.

Additionally, performance comparisons are made for a port of the benchmark to an Intel Xeon X5550 "Nehalem" based cluster, accelerated with NVIDIA Tesla C1060 GPUs. In addition to details of this port, this work also includes extrapolations to possible performance exploitation of the GPU.

The remainder of this chapter is organised as follows: Section 4.1 provides background information on the Shamrock benchmark, the purpose of the benchmark, a description of the test case examined in this work, a description of related work, and identifies the uniqueness of the benchmarking and predictions of this study. Section 4.2 introduces the performance of the benchmark on a local, high-end, workstation, which is followed by a description of the HPC platforms the code has been ported to. Section 4.3 examines the performance characteristics on these HPC platforms, and also covers the port of the benchmark to a GPU cluster, and its relative performance.

It concludes with a justification that the exploration of low power, emerging heterogeneous technologies via a traditional benchmarking application is time consuming and hence productivity limiting. It reaches the conclusion that to assess a range of architectures and associated programming methodologies a new approach is required.

In its original research paper form: Herdman, J. A., et al. "Benchmarking and Modelling of POWER7, Westmere, BG/P, and GPUs: An Industry Case Study."; this chapter was presented at the 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 10) held in conjunction with IEEE/ACM Supercomputing 2010 (SC'10) New Orleans, LA, USA, and subsequently published in ACM SIGMETRICS Performance Evaluation Review 38.4 (2011): 16-22 [103].

4.1 Shamrock

Prediction of the dynamic behaviour of materials as they flow under the influence of high pressure and stress is a key field of investigation at AWE. As a result, hydrodynamic simulations account for a large proportion of compute cycles on AWE's HPC systems. Representative benchmarks have existed for many years: two dimensional (2D) hydrodynamic code fragments were part of the original Livermore Loops [143], and have been used in earlier performance studies [201]. More recently, and primarily, due to the large HPC resources required to execute them, the focus has been on three dimensional (3D) benchmark codes, such as SAGE from LANL [122] and Hydra from AWE [79]. However, although not in the capability regime, 2D hydrodynamics accounts for a significant amount of capacity computing, with finer resolution and growth in the number of CPU hours continually increasing.

To reflect this, Shamrock was developed at AWE as an industrial-strength, domain decomposed, multi purpose benchmark. It is a 2D structured hydrocode, written predominantly in Fortran 90, using the Message Passing Interface (MPI) as its means of communication between sub-domains. The code has been designed for a number of purposes: (i) the assessment of the impact on code performance of system upgrades to an incumbent architecture; (ii) to be utilised as part of a larger HPC Benchmark Suite to assess application performance differences between alternative vendor offerings, primarily during machine procurement cycles, and (iii) to be used to assess current and emerging technologies. This chapter addresses each of these three categories of use and identifies problematic aspects in the use of a benchmark application for the latter category.

At the time of writing, the benchmarking and predictive modelling documented in this chapter differed significantly from earlier studies in that it was the first benchmarking presented in academic literature of the POWER7 platform using a 2D hydrodynamics benchmark. It encompassed the most diverse range of current architectures, compilers, and MPI invocations for such a benchmark, and is distinct in its comparison of these. Several studies have investigated the use of GPUs as accelerators in the field of 2D hydrodynamics , and have reported speed-ups of factors of 70 over a single threaded CPU [98, 123]. However, this chapter not only looks at speed-ups of the GPU over a single threaded CPU, but presents comparisons of running in an accelerated distributed MPI mode.

A representative, test case for this code is an interacting shock wave problem. Consisting of square inner, middle, and outer regions of ideal gas at differing initial densities and energies that cause the inner and outer regions to compress the middle region. This gives rise to shock fronts which collide and create a Rayleigh-Taylor instability [186].

Typical problem sizes are in the range of 300k to 5M cells. A representative problem of approximately 1.05M cells (1024 x 1024) was chosen as a problem size in the middle of this typical range, and by measuring the time to solution for 10 iteration time steps a workable turnaround time for benchmarking purposes was achieved.

It is known that, as with many hydrodynamics applications, as the cell quantities are updated each timestep, data reuse is limited and therefore the code is predominately memory bound, rather than CPU bound.

4.2 Architectures

Initially the code was developed and tested on a local, high-end, workstation for a range of compilers and MPI implementations. Based on these observations, specific configurations were subsequently built and executed on a selection of HPC architectures, including Intel Xeon "Nehalem" and "Westmere" micro-architectures; IBM POWER5, POWER6, POWER7, Blue Gene/P and Blue Gene/L, plus the AMD Opteron "Barcelona" chip sets. The following sections describe these architectures, and observations from benchmarking.

4.2.1 Assessment of System Upgrades and Software Environmental Changes

An initial performance assessment was carried out on an Intel Xeon E5405 [8], dual socket, quad core desktop workstation. The hardware has 12MB L2 cache, a 2.00 GHz clock speed, and a Front Side Bus (FSB) speed of 1333 MHz. Numerous compiler and MPI combinations were chosen to represent possible system changes. These included four diverse compilers: SUN's SUN Studio 12.1 [18], Portland Groups PGI 10.1.0 [22], Intel®Fortran 11.0.073 [20], and

Table 4.1: Local Compiler and MPI Build Versions

Compiler	**MPI**
Sun Studio 12.1	MPICH2 1.1.1
Sun Studio 12.1	OpenMPI 1.3.3
g95 3.0.4	MPICH2 1.1.1
PGI 10.0.1	MPICH2 1.0.7
Intel 11.1.046	MPICH2 1.1.1
Intel 11.1.046	OpenMPI 1.3.3

Table 4.2: Local Compiler Build Flags

Compiler	**Flags**
Sun Studio	-g -fast[2] -xtypemap=real:64 -xipo=2 -fsimple=0 -fns=no
g95	-g -O3 -march=native -fdefault-real-8 -ffloat-store -funroll-loops
PGI	-gopt -fast[3] -r8 -Kieee
Intel	-g -O3 -ip -xhost -r8 -fp-model strict -fp-model source -prec-div -prec-sqrt
XLF (Blue Gene)	-g -O4 -qunroll=yes -qipa=inline=auto -qipa=level=2 -qrealsize=8 -qfloat=nomaf -qstrict
XLF (Power)	-g -qfullpath -O3 -Q -qrealsize=8 -qfloat=nomaf -qstrict

GNU's g95 4.0.3 [3]; and two MPI variants: MPICH2 [2] and OpenMPI [17]. Although builds were not available for all compiler and MPI permutations. Those that were available, and their respective versions, can be found in Table 4.1.

The Shamrock build has some self imposed restrictions on compilation options. These are in place to ensure that the results between architectures and compilers are as numerically comparable as possible. A full list of compiler options used in this study is specified in Table 4.2.

Comparing the compilers, with the same MPI implementation, Figure 4.1 shows that the GNU compiler gives the poorest performance, 83.9% slower than the fastest compiler, the Sun Studio, for a single core run. Between these two, the Intel and PGI compilers are 26.5% and 15.1% slower than the Sun Studio respectively.

The discrepancies in compiler performance is due to how the restrictions, imposed through the build flags, affect how aggressive a particular compiler is

[2]With the Sun Studio complier, -fast is an alias for the following compiler options: -xtarget=native -O5 -libmil *-fsimple=2* *-dalign* -xlibmopt -depend=yes *-fns* -ftrap=common -pad=local -xvector=yes -xprefetch=yes -xprefetch-level=2 -nofstore.
Those in *italics* invalidate the IEEE Standard [189] and hence are manually disabled.

[3]-fast with the PGI compiler is an alias for: -O2 -Munroll=c:1 -Mnoframe -Mlre

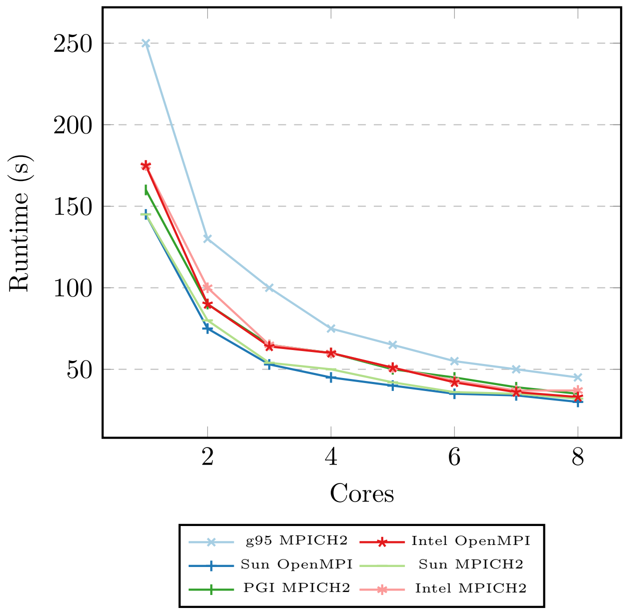


Figure 4.1: Shamrock: 1024^2-cells on Intel®Xeon®E5405

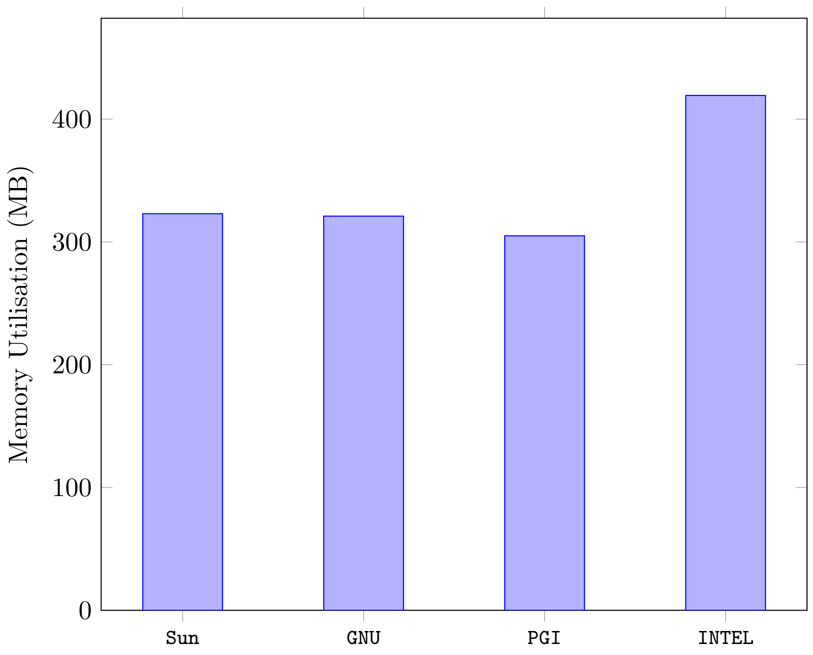


Figure 4.2: Intel®Xeon®E5405 Compiler Memory Footprint

in its level of optimisation. This can be seen if the restrictions are lifted with the Sun and Intel compilers. Removing the *-fp-model strict*, *-prec-div* and *-prec-sqrt* options from the Intel compiler, (by which enables more aggressive floating-point optimisations) and with the addition of the *-prof-gen* and subsequent *-prof-use* Profile-Guided Optimisation (PGO) options, the Intel compiler sees an 25% runtime improvement.

A performance metric, often overlooked, is the resultant memory footprint at runtime. Figure 4.2 shows the high memory watermark (HMW) of the application during its execution, for the four compilers tested. Sun and GNU show a 5.26% increase in memory over the PGI compiler, however it would appear that Intel's optimisations require a greater amount of memory to be executed, with a memory footprint 36.84% greater at 418MB, a not insignificant difference.

When considering the same compiler, but with a different MPI implementation, in both cases where this is possible: Sun and Intel, the OpenMPI build out performs the equivalent MPICH2 build, by 4.5% and 8.5% respectively, when averaged over multi-core runs. To gain best performance, an MPI implementation should be optimality tuned for the system in question, this can be achieved by disabling error checking in the MPI builds, or setting configuration options for a specific system. However, in this study both MPI implementations are default builds, so it would appear that, OpenMPI is able to outperform MPICH2, in handling the shared-message MPI queue that will be being used, for Shamrock, when running locally on the workstation.

The study shows significant differences are observed depending upon which compiler and MPI are used, indicating a benchmark application, such as Shamrock, is a suitable tool for tracking and exploring systems software upgrades and environmental changes.

4.2.2 HPC Benchmarking Platforms

At the time of this study, a range of HPC platforms were available for the benchmarking of the Shamrock code:

- Intel Xeon (Details Table A.1)
  - Nehalem (Willow A.1.1),
  - Westmere (Blackthorn A.1.2)
- IBM POWER (Details Table A.2):
  - POWER5 (Gollum A.2.1),
  - POWER6 (Milano A.2.2),
  - POWER6 (v60 A.2.3),

  - POWER7 (p90 A.2.4)

- IBM BlueGene (Details Table A.3):
  - Blue Gene/L (uBG/L A.3.1)
  - Blue Gene/P (DawnDev A.3.2)

- AMD Opteron (Details Table A.1):
  - Barcelona (Hera A.1.4)

Details of each platform, along with the compiler and MPI implementation of choice is given in the relevant sections of Appendix A. Based on performance gains observed with the Sun Studio and OpenMPI implementations on the E5405, where possible the same compiler and MPI implementation has been installed on the HPC platform. For future reference, each platform, its compiler and MPI implementation adopts the nomenclature: *Platform:Compiler:MPI.*

4.3 HPC Benchmark Performance

The problem set which was executed locally on the Intel Xeon E5405 workstation was run on the HPC architectures with compiler and MPI configurations as detailed in Section 4.2.2. As with the local builds, to ensure equality of results, restrictions were applied to the compilations. Table 4.2 again contains the compiler specific details which were employed in this part of the study.

4.3.1 Platform Comparisons

Figure 4.3 shows the runtimes, on a logarithmic scale, for all of the systems previously described. A number of observations can be deduced from this chart:

- The improvement seen, on the E5405, for the Sun Studio compiler over that of the Intel compiler, is also present on the Nehalem L5530. Single core runtimes for the *L5530:Intel:BullX* are 18.8% slower than the equivalent run on the *L5530:Sun:OpenMPI.*

- The use of IBM's SMT is demonstrated on the *Pwr6(4.2GHz):XL:IBM.* Using all 16 of the possible SMT threads gives little gain over an 8 core, no SMT, run. However, enabling the SMT, but only utilising a sub-set of the total SMT threads, a factor speedup of 1.42 over 8 threads on 12 cores, is close to a 1.5 maximum.

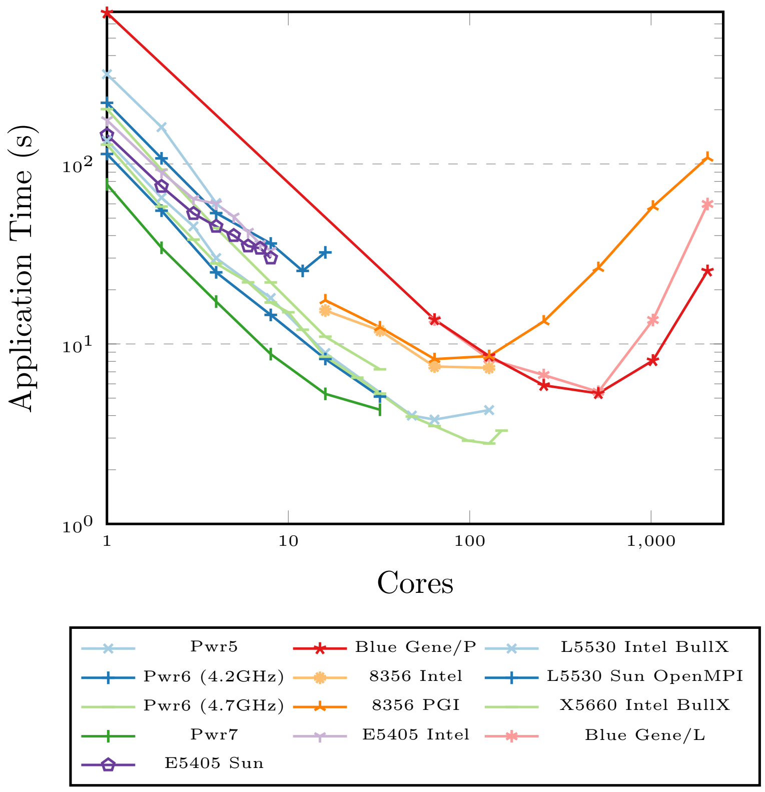


Figure 4.3: Comparative runtimes for Shamrock 1024^2-cells

Table 4.3: Relative Number of Cores and Power Consumption, for Equal Time to Solution

Equivalent # Cores	**Architecture**	**Power Consumption (kW)**
10	POWER7	0.61
20	Nehalem	0.72
20	Westmere	0.59
30	POWER6	4.79
64	Barcelona	3.52
160	Blue Gene/P	1.23

- Of most interest to the end-user, is the fastest time to solution for the problem. In the case of all the architectures benchmarked in this study, this is achieved for the *X5660:Intel:BullX* Westmere, on 128 cores for the 1024^2 cell problem size.

To enable additional platform comparisons a time to solution of 8.24 s was selected. This matches the execution time on 64 cores running on the *8356:Intel:OpenMPI*. The equivalent number of cores required from each of the architectures benchmarked, to match this 8.24 second turnaround can be inferred from Figure 4.3. This is captured in Table 4.3, together with the respective power consumption for that number of cores on a given architecture.

The power consumption figures given are extrapolations based on full system runs of LINPACK [166] for the systems specified in the (contemporary at time of study) June 2010 Top500 [147]. In the case of the POWER7 this figure is based on the maximum possible power draw for a system, as specified in IBM's p755 Redbook specification sheet [194]. This figure will be most certainly higher than an equivalent LINPACK run.

The power consumption figures clearly show an improvement from those architectures of an older generation (e.g. POWER6 and Barcelona, to a Westmere or POWER7) with the BlueGene solution sitting between these two ranges.

The Shamrock benchmark shows and allows comparison between currently available, production ready HPC machines. It enables assessments to be made on raw performance, time to solution, and with the availability of power consumption figures, the option to provide an overall cost to solution.

4.3.2 Assessing Emerging Technologies: GPU Comparisons

AWE has a modest GPU test bed architecture, codenamed "Dexter". Consisting of four nodes: one master, and three compute. Full specification details are given in Appendix A, under Section A.4.1, but essentially it provides access to a small number of NVIDIA®GeForce™GTX 285 [14] GPUs.

To enable a port of the Shamrock benchmark to the GPU, compute intense sections of the code were identified and subsequently turned into kernels. In total eight such kernels were identified, accounting for approximately 95% of compute time.

4.3.3 The "Re-Structure"

The process of kernelising the computationally intense code sections required a level of re-structuring within each of the identified code sections. As described by Gaudin [190], this consisted of exposing the inherent concurrency in the algorithm and optimising memory access.

In Shamrock, this was a case of finding concurrency at the loop level by re-factoring conditional heavy coding, which became a time consuming effort due to the requirement to identify variable scoping information. This was especially so, when variables were contained in shared data modules. Additionally, it was also found that objectives such as coding to save memory, heavy logic error checking and bookkeeping activities although well intentioned, were restrictive and resulted in artificially sequential code. Removing a number of Fortran90 features, such as derived types and the removal of ragged arrays and linked lists resulted in less elegant but simplified Fortran coding, enabling greater concurrency than the original implementation.

Once these issues were resolved the kernels were then optimised for memory bandwidth, floating point performance and memory access patterns.

The next step used the F2C utility, based on Feldman's Fortran to C converter [89], to convert the simplified Fortran kernels into C. Finally, through the use of AWE's Acrylic wrapper code parser, data management and placeholder code for the ported C routines were generated. This enabled the fast utilisation of the OpenCL [125] framework, to create GPU executable kernels from the C routines.

Even with this structured approach, the process was time consuming, taking a number of months to achieve. Particularly the kernelisation of the existing compute intense code sections identified areas that would require major refactoring of the entire code structure to obtain. However, a subset of the eight identified code regions were amenable to kernelisation in an incremental fashion.

As part of this work, four of these potential kernels were developed:

(i) *lagren* calculates adiabatic heating on a cell, based on the volume change in the cell and its pressure, using a predictor/corrector method

(ii) *lagrac* calculates nodal accelerations due to pressure gradients and subsequently updates the nodal velocities

(iii) *lagrqq* calculates an artificial viscous pressure around shock waves, smoothing out discontinuities and reduce oscillations

(iv) *lagrvf* which calculates the volume fluxes across cell faces, which are later used to carry out the advective remap

These four kernels account for 13.55%, 8.06%, 10.78%, and 1.5% of total runtime respectively. Although this gives 33.89% of the code resident on the CPU, there are some caveats which restrict claiming a fully distributed GPU enabled version of the code. Currently, for each kernel call, all data associated with the kernel is transferred to and from the accelerator device. Although, logic has been added enabling data to be shared between kernels there still exits a significant overhead from the data copies. Additionally, a number of boundary conditions are assumed for GPU execution, this severely restricts the problem range that Shamrock is capable of running. However, due to careful problem selection, the test case described in Section 4.1, and subsequently analysed is unaffected by this restriction. Hence some direct comparisons between non-accelerated and accelerated executions of the code can be made.

Figure 4.4 compares the average iteration time for each of the kernels when run in their original (Orig), simplified Fortran (F) and C versions on a single core of Dexter's X5550 Xeon, and the OpenCL version on the hosted GTX285 GPU. By re-writing the original Fortran, performance gains are apparent for each of the kernels, with the C versions demonstrating similar performance gains. For the OpenCL version, despite no optimisations being implemented, further performance gains are observed for the same three kernels over the C equivalents. In the case of the OpenCL kernels, two average iteration times are given for each kernel, the former (OpenCL(1)) includes data transfer to and from the device, the latter (OpenCL(2)) for compute time only of the kernel on the device. Comparing the compute-only OpenCL kernels against the original, yields an average increase in performance of 18.97x, with speedups of 25.24x, 16.53x, 14.63x, and 19.47x respectively for *lagren*, *lagrac*, *lagrqq*, and *lagrvf* routines.

The benchmark was run using MPI in a distributed fashion, in its entirety, in a non-accelerated and accelerated mode on Dexter, using the Intel 11.0 compiler. In the case of the accelerated mode, this includes all the data transfer times to and from the GPU and CPU host. The resultant figures are captured in Figure 4.5 as *X5550* and *X5550 Acc CT* where the former refers to the non-accelerated mode and the latter to the accelerated mode with *CT* denoting the "current transfer scheme".

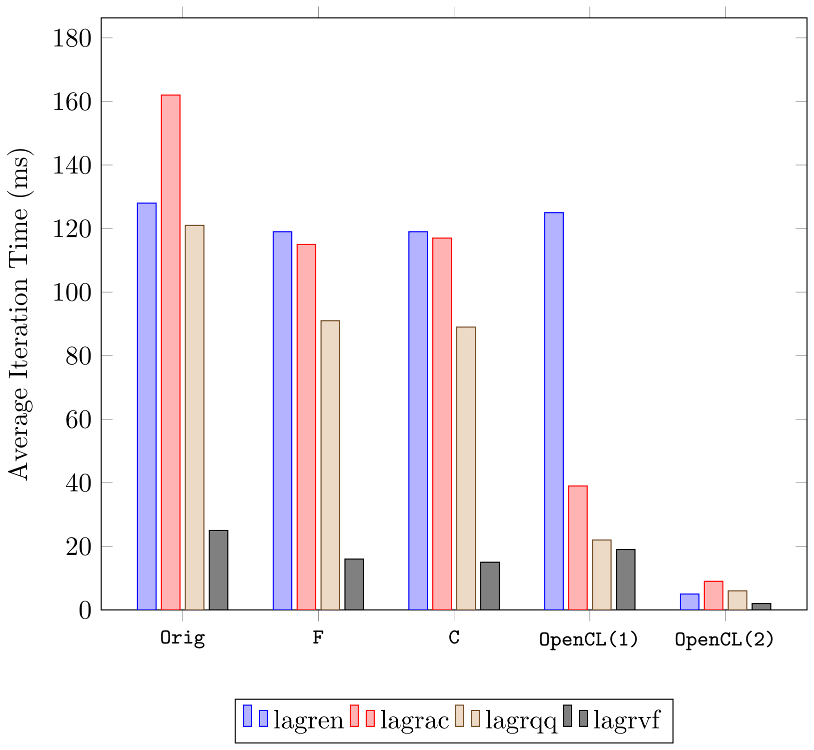


Figure 4.4: Kernel Performance

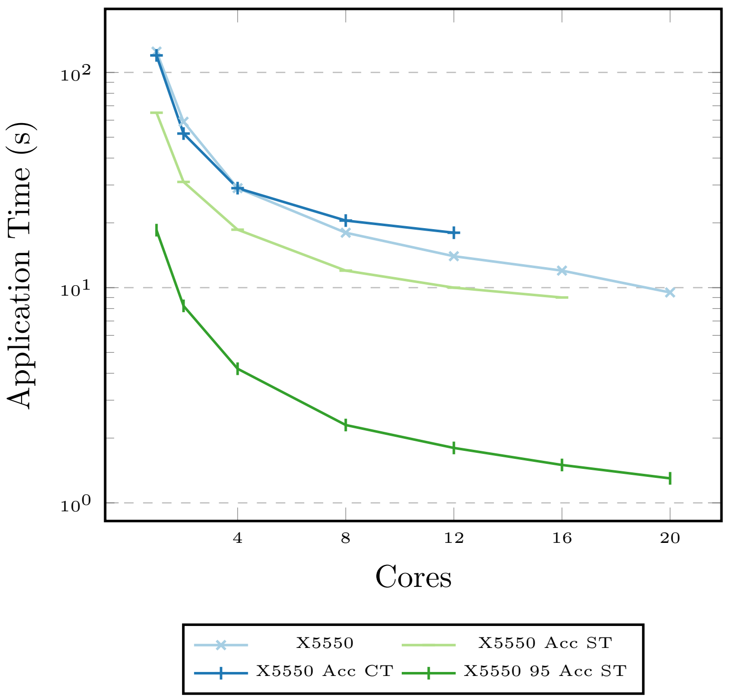


Figure 4.5: Comparison, including GPUs, of runtimes for Shamrock 1024^2-cells

Table 4.4: Data Bandwidths for Shamrock Transfer Sizes

Cores	**Transfer Size (MBs)**	**Host to Device (MB/s)**	**Device to Host (MB/s)**
1	460.80	4370.7	3362.7
2	230.40	4373.8	3086.0
4	115.20	4545.8	3064.3
8	57.60	4518.0	3030.4
15	30.72	4459.9	3016.7

Also shown is the execution times were only a single data transfer, denoted ST, to take place to and from the host. The assumptions taken to obtain these ST results is explained in the following paragraphs.

These show no gain in performance from using the OpenCL kernels, and indeed performance is worse for the GPU runs when more than 8 GPUs are utilised. This is not unexpected; as previously stated, the data copies, to and from the device each time a kernel is called, begin to dominate performance.

However when the remaining four kernels, of the original eight that were identified for kernelisation, are developed and ported providing the are reused in sequence and with no host processing in between) the need for the current data transfers would be negated, and would be replaced with a one-off initial copy to (and final copy from) the device. A worst case overhead for such a copy can be calculated. The total amount of data necessary for the entire test case stands at 0.45GB/N per processing core, where N is the number of processing cores used. This is well within the memory constraints of today's devices: NVIDIA's Tesla C2070 has 5.25GB of user available memory.[1] The memory bandwidth test, oclBandwithTest, which is shipped with NVIDIA's CUDA Toolkit [78] can be executed to measure the data transfer speeds for a range of transfer sizes. Table 4.4 shows these transfer speeds, using direct access and paged memory[2] averaged over ten runs.

Using these bandwidth figures, an estimate of the data transfer overheads can be made for the code if the data transfer per kernel is replaced with an initial copy to, and final copy from, the device. *X5550 Acc, ST*, in Figure 4.5, shows the estimated distributed runtime for the benchmark with this change of data transfer from the current transfer (CT) scheme, to a single transfer (ST) scheme.

The four kernels already ported to the GPU, accounting for 33.89% of the total code compute time, and their average speedup, over the original coding, is a factor of 18.97. If it is assumed the remaining 61.11% of compute, contained

[1]The C2070 has 6GB in total, however 0.75GB is reserved for error checking and correcting (ECC).

[2]It is possible to increased bandwidth using mapped access and pinned memory

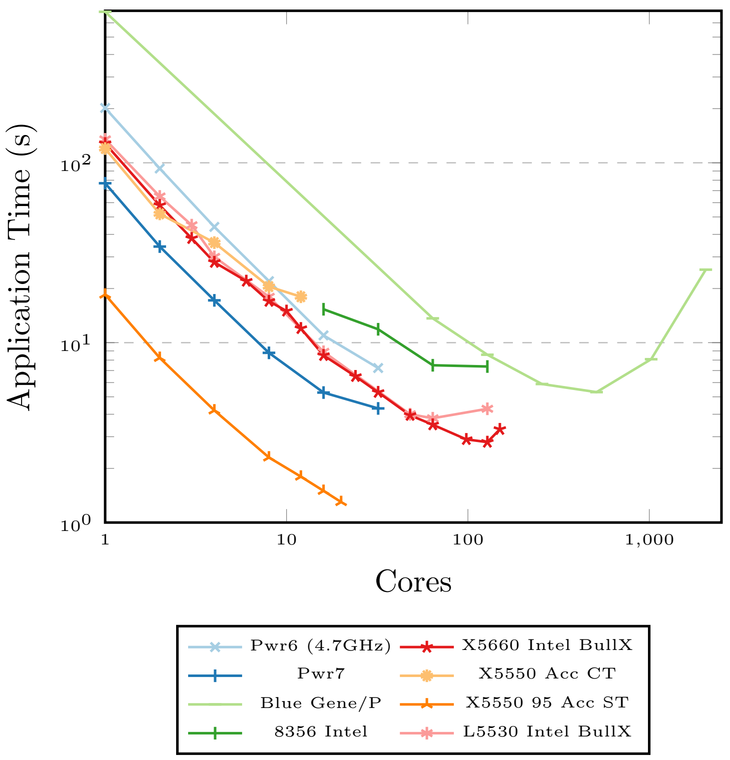


Figure 4.6: Comparison, including GPUs, of runtimes for Shamrock 1024^2-cells

in the remaining four kernels, achieves a similar average performance gain, Amdahl's law can be used to calculate an estimated speedup for a distributed, 95% GPU resident, version of the benchmark (Equation 4.1). Taking the 5% of the application which is not accelerated, each of the kernels with their percentage of compute and performance gains, and the estimated gain for the remaining 61.11%, an overall speedup of a 0.1006 is calculated.

$$\frac{0.05}{1} + \frac{0.1355}{25.24} + \frac{0.0806}{16.53} + \frac{0.1078}{14.63} + \frac{0.015}{19.47} + \frac{0.6111}{18.97} = 0.1006 \tag{4.1}$$

Shown in Figure 4.5 as *X5550 95 Acc, ST*; this gives an estimated order of magnitude gain over a non-accelerated x5550 run.

Figure 4.6 adds *X5550 Acc CT* and *X5550 95 Acc ST* to a subset of those benchmark runs depicted in Figure 4.3. Based on the prior assumptions, this shows that an equal time to solution of 8.24 s could be theoretically achieved using two X5550 cores, each accelerated with an NVIDIA Tesla C1060 GPU card. The resultant power consumption, as detailed in Table 4.5, shows savings compared to alternative platforms.

Table 4.5: Relative Number of Cores and Power Consumption, for Equal Time to Solution; Including GPU

Equivalent # Cores	**Architecture**	**Power Consumption (kW)**
10	POWER7	0.61
20	Nehalem	0.72
20	Westmere	0.59
30	POWER6	4.79
64	Barcelona	3.52
160	Blue Gene/P	1.23
2	Tesla Accelerated Nehalem	0.44 2x(0.036+0.1878)

4.4 Summary

This chapter introduced the industrial strength, multi-purpose 2D benchmark code Shamrock. Using a suitably defined test problem, it has been demonstrated as a suitable vehicle for assessing code performance variations due to system software changes, and also as a tool for benchmarking competing system offerings as part of an HPC procurement cycle.

To consider its suitability for assessment of emerging hardware technologies an OpenCL port to an NVIDIA Tesla C1060 accelerated Intel Xeon *Nehalem* based cluster was described for four of eight possible kernels within the benchmark, accounting for 33.89% of the total code compute time. Running with the accelerated kernels in a distributed mode, and comparing with the non-accelerated distributed code, experimental data showed increasing overheads due to the data transfer to and from the device. By applying an overhead based on the theoretical maximum data required to be transferred to and from an attached GPU device, a prediction was calculated for removing the current data transfer scheme, and replacing with a larger, single data transfer scheme. Additionally, by applying an average speedup factor of 18.97x on the remaining four kernels (based on an assumed average speedup for the ported kernels) a predicted execution time of a distributed, 95% resident version of the benchmark was derived. This means that an equal time to solution of 8.24 s is achievable with two C1060 accelerated *Nehalem* cores.

However, the removal of data transfer and speedup factors were artificial and not experimentally validated, hence can be argued an unreasonable comparison to the original application. For such verification in the Shamrock benchmark a significant implementation and code restructuring would be required. Such an effort was determined to be too time consuming to be profitable.

As Shamrock is representative of a typical procurement benchmarking tool, containing error handling, bookkeeping, I/O routines, loop carried dependencies and non-optimised memory access patterns; this indicates that a benchmark application is too unwieldy for the evaluation of an emerging architecture. To address this aspect, so far the study has been of one emerging technology, namely a GPU, and one programming methodology: OpenCL. If an industrial size benchmark is unsuitable for this selective case, then it is certainly unsuitable for the general case of interest: a rapid assessment of a range of programming methodologies on a range of emerging hardware. For such assessments a new approach is needed.

CHAPTER 5
Mini-Applications: The OpenACC Development of CloverLeaf

The previous chapter demonstrated the inefficiencies in using an industrial benchmark code as a mechanism to explore the changing HPC landscape. Due to the long turnaround time to assess just one programming model on one possible architecture a different approach is needed.

This chapter introduces the concept of a mini-application or mini-app, which is lighter-weight whilst still representative and are written with the concept of keeping algorithms explicit in their nature and purpose.

By ensuring the mini-app is a small, self-contained program that embodies essential performance characteristics of key applications, they provide a viable way to trial new programming methodologies and new architectures.

Specifically, this chapter makes the following key contributions:

- It provides a detailed documentation of the CloverLeaf mini-application which can be found as part of the Mantevo project [104], including its hydrodynamics scheme as well as the features which make it amenable to utilisation of many-core technologies.
- In particular, it describes the step-by-step development process to achieve a fully distributed and accelerated hybrid MPI/OpenACC implementation of the mini-app, using a Cray®XK6™ as the development platform.
- Finally, it introduces the various other implementations of CloverLeaf that are currently available and how they can be used in conjunction with the OpenACC variant to begin to explore suitability of emerging many core architectures.

This work was previously presented at the Cray Technical Workshop on XK6 Programming, hosted at ORNL in October 2012 [77]. Additionally, it was the focus for an OpenACC Standards Organisation White Paper and Case Study [100]. It has also been replicated in a forthcoming contribution-based book that will focus on teaching practical techniques for OpenACC parallel computing [88].

5.1 The Development of CloverLeaf

CloverLeaf, developed by the UK Mini-App Consortium (UK-MAC) [39], is part of the R&D 100 [172] award winning Mantevo test suite [104].

CloverLeaf has been written with the purpose of assessing emerging hardware and programming models. The simple hydrodynamics scheme is representative of the Shamrock benchmark, described in Chapter 4, but is written in such a way as to avoid unnecessary dependencies in key computational sections. This was achieved by encapsulating all scientific computation in small kernel functions, where the term "kernel" is used to refer to a self contained function which carries out one specific aspect of the overall algorithm, thus making long, complex loops containing many subroutine calls unnecessary. It incorporates the lessons learnt from the kernelisation of the four routines in the Shamrock benchmark application, as described in Chapter 4, Section 4.3.3. In particular, loop carried dependencies and deep loop logic was avoided; the use of allocations and print statements which cause synchronisation points within the kernels, were minimised. Additionally, the kernels were written to optimise memory access patterns and hence maximise the use of memory bandwidth. The resultant computationally intensive sections of CloverLeaf are implemented via twelve individual kernels.

5.1.1 Hydrodynamics Scheme

CloverLeaf is an explicit Eulerian hydrodynamic mini-app that solves the compressible Euler equations, a series of equations describing the conservation of energy, mass and momentum in a system. The equations are solved on a cartesian grid in two dimensions. Each grid cell stores three quantities: energy, density and pressure, and each cell corner, or node, stores a velocity vector. CloverLeaf solves the equations with second-order accuracy, using an explicit finite-volume method.

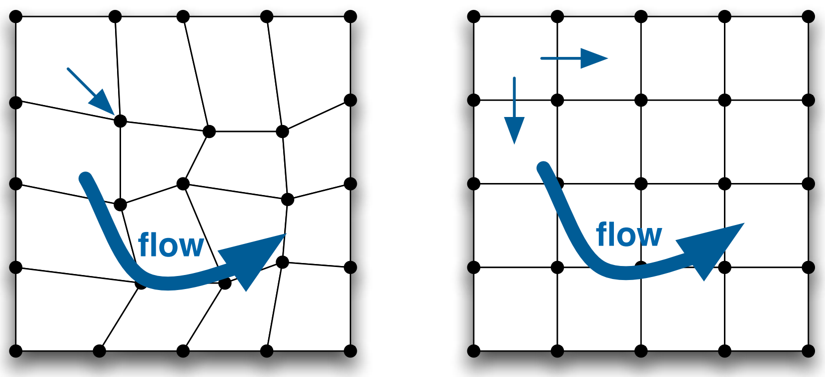
As depicted in Figure 5.1, each cycle of the application consists of two steps:

(i) a Lagrangian step advances the solution in time using a predictor-corrector scheme, distorting the cells as they move with the fluid flow

(ii) an advection step is used to restore the cells to their original positions

The initial implementation of CloverLeaf, developed by Gaudin, was in Fortran90 and was used to develop an optimised and highly vectorisable, hybrid MPI/OpenMP code.

This chapter focuses on the step-by-step development, a joint effort by Gaudin and the author, of an OpenACC implementation using the `parallel`



(a) Node movement during the Lagrangian step.

(b) Material movement during advective remap.

Figure 5.1: Lagrangian-Eulerian hydrodynamics cycle

construct using Cray's CCE compiler, on an NVIDIA GPU accelerated Cray XK6.

5.1.2 Test Case

A simple yet representative asymmetric test problem is used throughout the study, it consists of two regions of idealised gas; one of high density and energy, adjacent to that of a lower density and energy region. As the simulation proceeds a shock wave forms and penetrates the low density region.

Initially a simulation time of 0.5 µs, on a problem size of 0.25 million (500^2 cells) was created. This gave a relatively quick turnaround time, yet still long enough to see compute as the main work load. As the code was refined and improved larger cell counts for the same simulation were used to maximise the size able to fit onto a CPU core and subsequently a node. These are detailed at the relevant points throughout the chapter.

5.2 Development Platform: Cray®XK6™

As summarised in Appendix A.4.3, Chilean Pine is a Cray XK6 with 40 AMD, 16-core Opteron 6272 Interlagos processors. Each compute node has one of these Opteron 6272 CPUs plus a companion NVIDIA X2090 GPU. Each node has 32 GB of 1600 MHz DDR3 memory, supplying the CPUs. The XK6 utilises the "Cray Gemini Network" (Gemini) as its interconnect. The Opteron 6272 CPU shares resources at the "Bulldozer module" level. That is the two cores that make up a "Bulldozer module" both have access to the shared floating point unit (FPU). This FPU has two 128-pipelines which can be combined into one 265-bit pipeline that can then execute a single 256 AVX instruction. This still only provides four double precision FLOP/s/clock cycle. However the AMD does have a 256-bit fused multiply add (FMA) instruction, which can theoretically double the floating point performance to eight FLOP/s/clock cycle. The Opterons in Chilean Pine have a 2.1 GHz clock frequency, which equates to a total CPU peak performance of 10.75 TFLOP/s. The default Fortran and C compilers are the Cray Compiling Environment (CCE) version 8.0.7 (although for this study, a then contemporary beta release of CCE, 8.1.0.157, was made available from Cray) with the default MPI being MPICH2 via Cray's xt-mpich2 version 5.5.1. Thermal design power (TDP) for the Opteron and X2090 are 115W and 225W respectively.

5.3 Development of OpenACC CloverLeaf

This section describes the step-by-step approach, together with the incremental performance gains achieved as well as the issues which inhibit performance, of applying the OpenACC directive model to the CloverLeaf mini-app, resulting in a fully resident, multi-GPU version of the application.

CloverLeaf contains both C and Fortran implementations of the computationally intense code sections. The Fortran versions were targeted first and foremost as the basis of this study; the C implementations were subsequently produced once the fully optimal Fortran code was developed.

This was for two reasons: firstly the majority of applications developed within AWE are Fortran based, hence insights regarding the issues and processes required to take Fortran source code and accelerate it on a GPU architecture was of highest interest from an industry view point.

Secondly, at the time of development, Cray had focused on the Fortran implementation of OpenACC in their CCE programming environment which was therefore more mature, with greater support, than their C implementation.

The implementation of the OpenACC directives was greatly helped by the fact that the CloverLeaf code had an OpenMP based shared memory parallelisation scheme already implemented. This immediately identified those areas requiring the application of OpenACC directives to achieve acceleration. However, as will be demonstrated, this did not imply that simply adding or replacing OpenMP directives with OpenACC gives a suitably accelerated code. Ultimately, the number of kernels accelerated by applying the OpenACC directives to produce an efficient accelerated version are summarised as follows:

Subroutine	**% Runtime**
advec_mom	41.79
advec_cell	20.54
PdV	12.72
calc_dt	9.06
accelerate	5.32
viscosity	5.24

Table 5.1: CloverLeaf CPU Profile

- 12 unique kernels
- 25 ACC DATA constructs
- 121 ACC PARALLEL + LOOP regions
- 4 REDUCTION LOOPS
- 12 ASYNC
- 4 UPDATE HOST
- 4 UPDATE DEVICE

The following sections detail the progressive approach applied to the code to produce the efficient OpenACC accelerated version.

The first step was to identify those subroutines, or kernels, that are computationally intensive, or *hot spots*.

5.3.1 Hot Spots

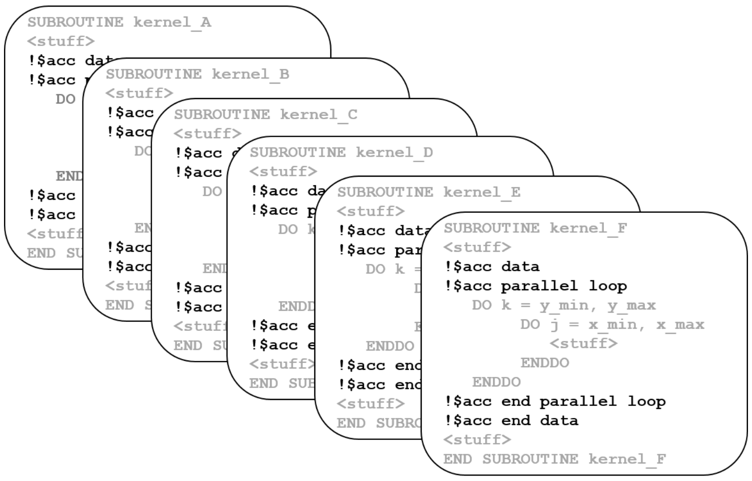
Using a simple profiler, Table 5.1 shows a flat, single CPU core profile for CloverLeaf.

These six computationally intense subroutines, or kernels, account for almost 95% of the code's execution time. On this basis, these kernels were targeted for initial acceleration.

5.3.2 Acceleration of Individual Kernels

Those six kernels identified were taken individually and OpenACC directives were applied. This was implemented by taking the existing OpenMP version of those kernels as a starting point, the advantage being that the development of the OpenMP version necessitated the scoping of the kernels variables, something also required for OpenACC implementation.

In the first instance, the `!$acc data` and `!$acc parallel loop`, in conjunction with their matching end directives, are all that is required to accelerate a particular computation loop within each kernel.



```
SUBROUTINE kernel_F
<stuff>
!$acc data
!$acc parallel loop
   DO k = y_min, y_max
         DO j = x_min, x_max
            <stuff>
         ENDDO
   ENDDO
!$acc end parallel loop
!$acc end data
<stuff>
END SUBROUTINE kernel_F
```

Figure 5.2: Pseudo Code: Individual Kernels Accelerated

Figure 5.2 schematically shows this applied to representative pseudo-code. However, this approach would implicitly copy all of the data to the device for execution, and subsequently copy all the data back to the host after completion of the computation of the kernel. This can be rectified by adding clauses to the OpenACC directives which describe the data dependencies between the CPU and GPU. With these clauses added, the code can be executed with one kernel running in its accelerated mode at a time. This approach was then repeated for each of the six kernels in turn.

Figure 5.3 shows a stacked bar chart for the individual kernels, executing the 500^2 cell test case. Each bar in the plot depicts the totality of execution time for that kernel, while each of the stacked segments represent the distinct categories the kernels spend their time in during execution, namely:

- "*No ACC'ed*" (code exclusively running on the CPU),
- "*Device Compute*" (pure execution time on the GPU),
- "*Data H2D*" (time taken to transfer data from the host to the device),
- "*Data D2H*" (data transfer time from device to host),
- "*Sync*" (synchronisation time on the device, that is time spent on the device not including computation, (e.g. allocations and waiting)).

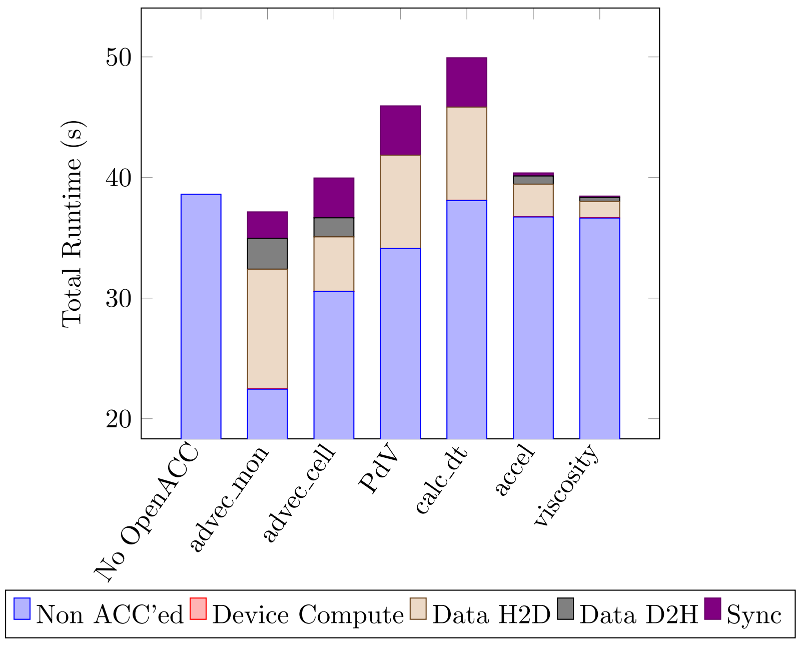


Figure 5.3: Breakdown of Individual Kernel Times

Irrespective of the kernel in question, it can be seen that the kernel time is dominated by the data transfer to the GPU from the CPU host. Also, synchronisation time is relatively high for some kernels. In the case of the latter, this will be addressed later in the optimisation steps. As for the former this is not unexpected; each time the kernel is called, all the state data is copied over for the kernel to execute.

5.3.3 Acceleration of Multiple Kernels

Once each kernel was checked for numerical correctness on the GPU, all the kernels were executed in their accelerated mode.

Figure 5.4 shows the comparison between the original, non-accelerated, version of the code executing on a single Opteron core and that with all the six computationally "hot" kernels accelerated, broken down into the respective categories of *No ACC'ed*, *Device Compute*, *Data H2D*, *Data D2H* and *Sync*.

Although now less than 5% of the code is executed on the CPU, the overall execution time is significantly greater than that of the non-accelerated original; and the data transfer effects are now even more apparent. Every timestep each kernel copies data (multiple times in the case of some kernels) to the device ready for computation. This data transfer can be reduced to a minimum by making the entire application resident on the GPU.

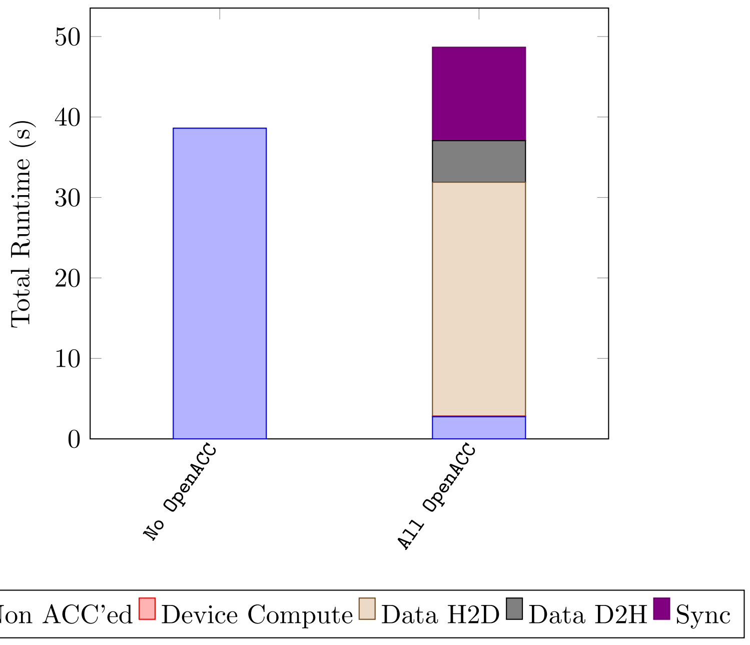


Figure 5.4: Breakdown of Multiple Kernel Times

5.3.4 Achieving Full Residency on the GPU

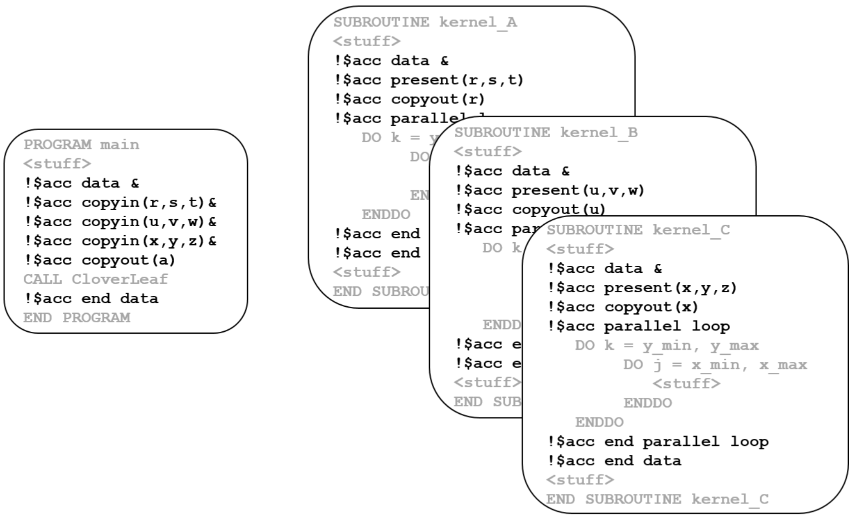
The pseudo-code in Figure 5.5 shows that by applying OpenACC data `copy` clauses at the highest level of CloverLeaf's call tree, namely the very start of the main program, a one off data transfer can be carried out. Subsequently, by use of the OpenACC `present` clause on the `data` construct of each kernel, it can be indicated that data is already on the device and a copy is not required.

In addition to restricting the data transfer to an initial copy, for full code residency to take place, additional kernels to the six identified, need to be placed on the GPU. These are non-computationally intensive, but without placement on the GPU, every time these sections of code are invoked, implicit data transfers will occur to and from the host.

With the exception of the initial set-up routine, a total of twelve unique kernels (including the previously identified six) are required to be accelerated. Figure 5.6 shows the impact of accelerating these additional kernels and applying an additional one off data transfer.

With the extra kernels now executing on the GPU there is virtually no computation remaining to be carried out by the CPU. This is reflected in the visible increase in device compute that is now observed. The most marked difference is that the data transfer overhead is dramatically reduced by implementation of the initial transfer.

As this data is a "one off" event, it would be reasonable to hypothesise that



```
PROGRAM main
<stuff>
!$acc data &
!$acc copyin(r,s,t)&
!$acc copyin(u,v,w)&
!$acc copyin(x,y,z)&
!$acc copyout(a)
CALL CloverLeaf
!$acc end data
END PROGRAM
```

```
SUBROUTINE kernel_A
<stuff>
!$acc data &
!$acc present(r,s,t)
!$acc copyout(r)
!$acc paralle
   DO k = y
      DO

      EN
   ENDDO
!$acc end
!$acc end
<stuff>
END SUBROU
```

```
SUBROUTINE kernel_B
<stuff>
!$acc data &
!$acc present(u,v,w)
!$acc copyout(u)
!$acc par
   DO k



   ENDD
!$acc e
!$acc e
<stuff>
END SUB
```

```
SUBROUTINE kernel_C
<stuff>
!$acc data &
!$acc present(x,y,z)
!$acc copyout(x)
!$acc parallel loop
   DO k = y_min, y_max
      DO j = x_min, x_max
         <stuff>
      ENDDO
   ENDDO
!$acc end parallel loop
!$acc end data
<stuff>
END SUBROUTINE kernel_C
```

Figure 5.5: Pseudo Code: Achieving Residency with OpenACC

if the problem size were large enough, or computation long enough, this data overhead would be a relatively smaller percentage of the overall execution time.

5.3.5 Increasing the Problem Size

The modest 500^2 cell problem size realises a speedup factor of 3.01 when executed on the GPU over the equivalent CPU run. Figure 5.7 shows this along with problems sizes of 960^2 cells, 2040^2 cells and 4096^2 cells. Where the relative performance gains over the CPU are 4.91x, 5.82x and 5.76x respectively. As expected, as the problem size is increased the percentage of runtime concerning the data transfer is reduced. Indeed, as can be seen in Table 5.2, for the 4096^2 cell problem the data transfer accounts for only 5.07%, compared to 18.24% for the 500^2 cell.

For a sufficiently large problem, the GPU performance is approaching 6x the performance of the non-accelerated, serial code on the Opteron CPU.

5.3.6 Comparison of Hybrid MPI/OpenMP

At face value, a factor of six improvement in the accelerated code over the original non-accelerated code sounds like a significant gain. However, this is comparing the performance of the entire X2090 GPU against that of a single Opteron core. A more realistic comparison would be to compare against the performance achievable by using all the cores available on the Opteron socket.

At the time of the study, using the hybrid MPI/OpenMP version of CloverLeaf, the optimal performance was achieved by using eight MPI tasks and one

Problem Size (Cells^2)	**No OpenACC (Serial)**	**Fully Resident**					
		Non Accelerated	**Device Compute**	**Data H2D**	**Data D2H**	**Sync**	**Total**
500	38.61 s (100%)	0.52 s (4.05%)	1.1 s (8.57%)	2.35 s (18.24%)	0.00 s (0%)	8.87 s (69.14%)	12.83 s (100%)
960	284.86 s (100%)	0.81 s (1.40%)	1.91 s (3.30%)	7.61 s (13.10%)	0.00 s (0%)	47.71 s (82.20%)	58.04 s (100%)
2,048	2800.50 s (100%)	6.03 s (1.25%)	3.80 s (0.79%)	35.19 s (7.31%)	0.00 s (0%)	436.16 s (90.65%)	481.18 s (100%)
4,096	23 757.43 s (100%)	15.20 s (0.37%)	6.32 s (0.15%)	209.28 s (5.07%)	0.00 s (0%)	3894.83 s (94.40%)	4125.63 s (100%)

Table 5.2: Execution Breakdown Comparison for Changes in Problem Size (Seconds) and Relative Percentage (%)

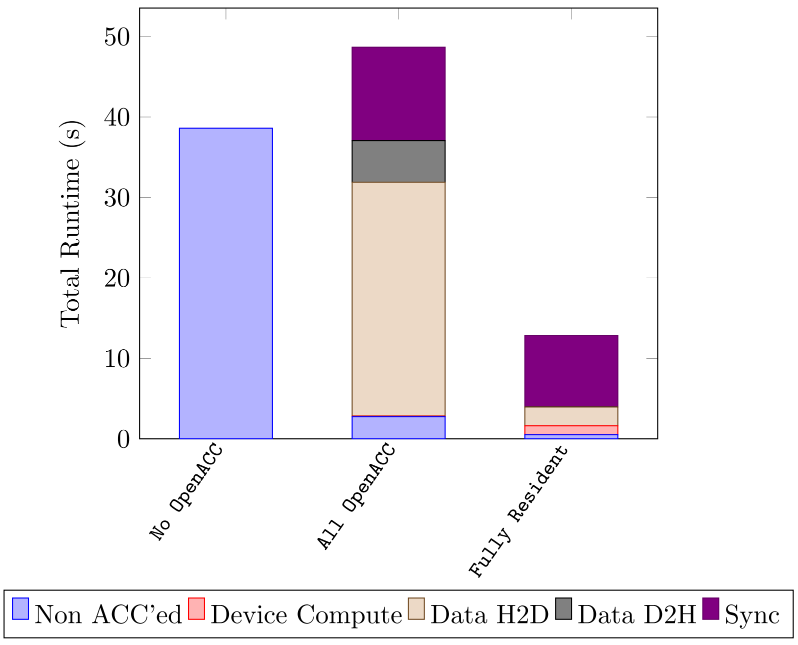


Figure 5.6: Breakdown of Resident Kernel Times

OpenMP thread and using one core per Bulldozer module. This gave a wall clock time of 43.52 s, in comparison to the 58.03 s for the GPU; a speedup factor of 0.88x.

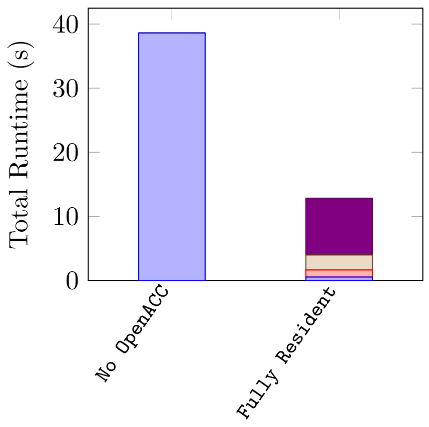
The factor of six over a serial implementation may have looked attractive, but with a distributed parallel implementation on the CPU outperforming the accelerated GPU variant, a GPU based architecture is no longer such an attractive proposition. Before returning to and addressing this performance, the OpenACC version of the code was extended to enable execution on multiple GPUs.

5.3.7 Hybrid MPI/OpenACC

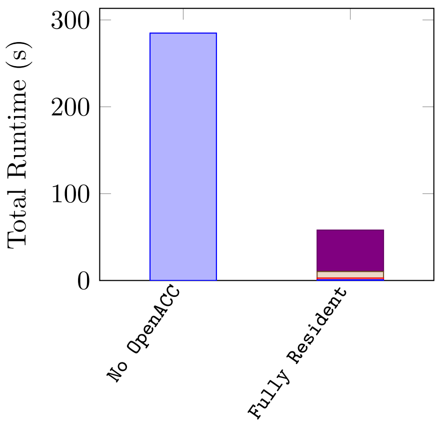
To enable the extension of CloverLeaf to execute on multiple GPUs, the OpenACC build of CloverLeaf was extended to use MPI. This required each GPU to be running fully resident on its section of the computational domain, with a traditional halo data exchange scheme implemented between each distributed sub-domain.

In practice this equates to first updating the host CPU with the latest halo data from its associated GPU, then using MPI to communicate that data between neighbouring CPUs, and finally each associated GPU obtaining the updated halo data from its CPU host.

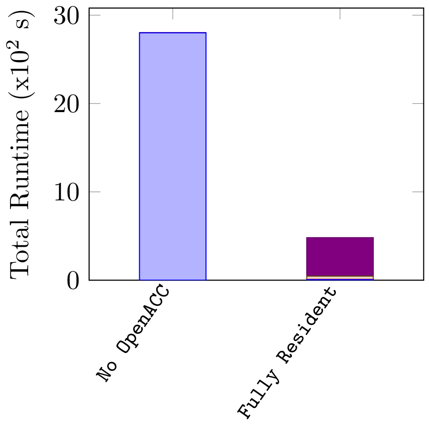
Figure 5.8 shows how this is implemented using the OpenACC `update`



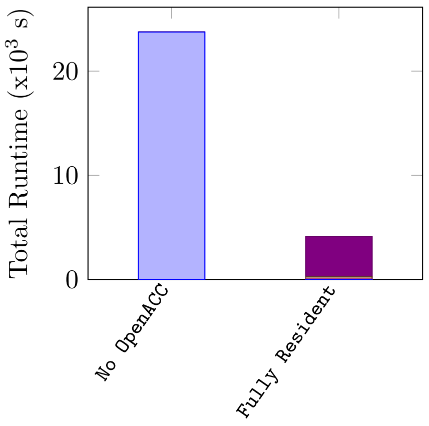
(a) 500^2 Cells



(b) 960^2 Cells



(c) 2048^2 Cells



(d) 4096^2 Cells

🟦 Non ACC'ed 🟥 Device Compute ▨ Data H2D ⬛ Data D2H 🟪 Sync

Figure 5.7: Increase in Problem Size

```
SUBROUTINE exchange
!$acc data &
!$acc present(snd_buffer)
!$acc parallel loop
   DO k = y_min_dpth, y_max_dpth
        DO j = 1, dpth
           <pack snd_buffer>
        ENDDO
   ENDDO
!$acc end parallel loop
!$acc update host (snd_buffer)
!$acc end data
CALL MPI_IRECV(rcv_buffer)
CALL MPI_ISEND(snd_buffer)
CALL MPI_WAITALL
!$acc data &
!$acc present(rcv_buffer)
!$acc update device(rcv_buffer)
!$acc parallel loop
   DO k = y_min_dpth, y_max_dpth
        DO j = 1, dpth
           <unpack rcv buffer>
        ENDDO
   ENDDO
!$acc end parallel loop
!$acc end data
END SUBROUTINE exchange
```

Figure 5.8: Pseudo Code: Hybrid MPI/OpenACC

directive. This results in a fully distributed and accelerated version of the CloverLeaf mini-app.

5.3.8 Version A: Initial Performance

Referred to as **Version A**, this initial, fully distributed and accelerated version can be taken as a baseline for analysing performance. Figure 5.9 shows the strong scaling performance characteristics of the initial multi-GPU version of the code. Here it is executing the 960^2 cell sized test case but with an increased simulation time to 15.5 µs rather than 0.5 µs as originally depicted in Figure 5.7. Increasing the simulation time allows the performance characteristics to be easily observed, especially once the problem is distributed over multiple CPUs and GPUs. Plots are for one Opteron CPU core, one Opteron socket, and one through six GPUs.

As described in Section 5.3.6, the key points to take away are:

- One X2090 GPU is a factor of 5.97 faster than one Opteron CPU core
- One X2090 GPU is 0.88 times faster than one Opteron socket
- Multi-GPU scaling turns over[1] once six GPU are utilised.

To see if these initial findings can be improved on, understanding these performance figures is crucial.

[1]The point, where increasing the number of processors to solve a problem results in a slower solution time;that is the problem no long demonstrates strong scaling.

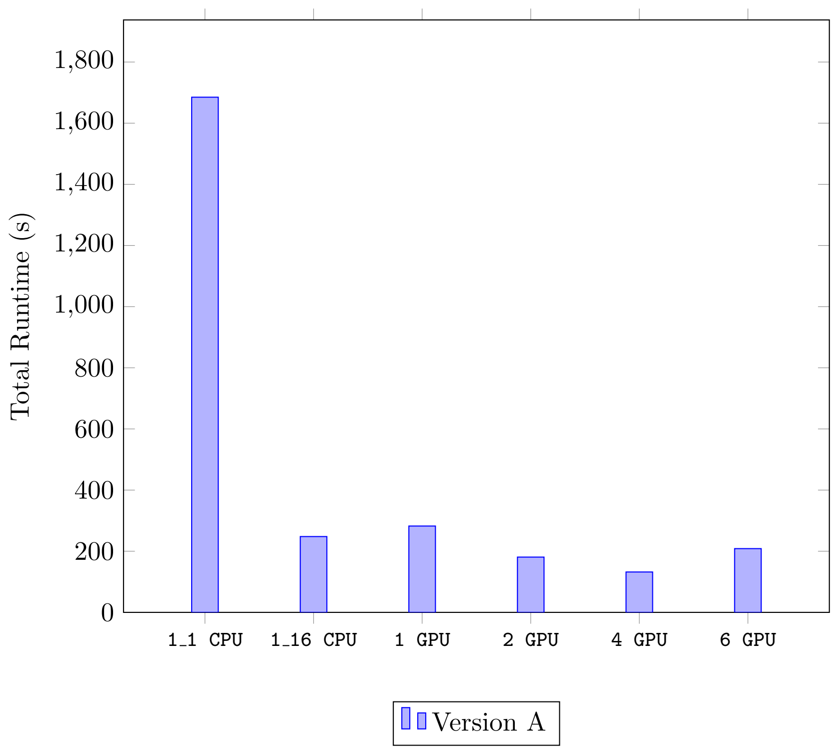
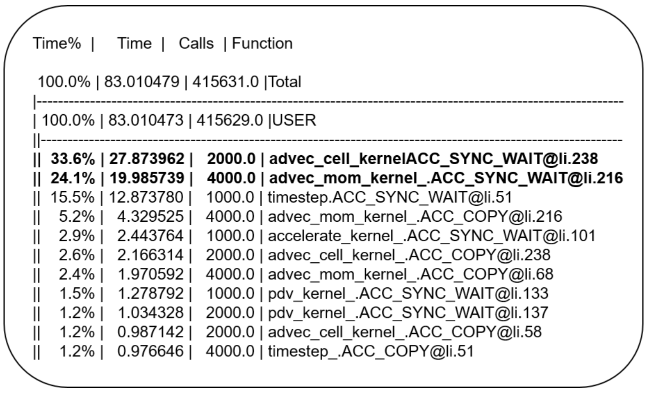


Figure 5.9: Version A Initial Performance of Multi CPU and GPU

5.3.9 Version B: Inner Loop Dependencies

On the XK6, Cray's analysis tool suite, Perftools, has been extended to measure GPU performance, including that of a flat profile, which shows the total execution times for each function within the application. Figure 5.10 shows a profile for Version A, that indicates *advec_cell* and *advec_mom* are the two routines that dominated runtime. This is not entirely unexpected; indeed, running the code exclusively on the CPU results in a not dissimilar profile.



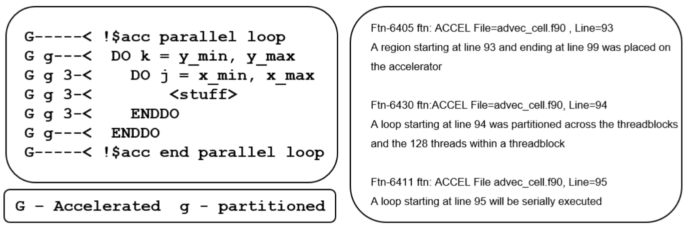
```
Time% |    Time  |   Calls  | Function

 100.0% | 83.010479 | 415631.0 |Total
|------------------------------------------------------------------------------------
| 100.0% | 83.010473 | 415629.0 |USER
||-----------------------------------------------------------------------------------
||  33.6% | 27.873962 |   2000.0 | advec_cell_kernelACC_SYNC_WAIT@li.238
||  24.1% | 19.985739 |   4000.0 | advec_mom_kernel_.ACC_SYNC_WAIT@li.216
||  15.5% | 12.873780 |   1000.0 | timestep.ACC_SYNC_WAIT@li.51
||    5.2% |   4.329525 |   4000.0 | advec_mom_kernel_.ACC_COPY@li.216
||    2.9% |   2.443764 |   1000.0 | accelerate_kernel_.ACC_SYNC_WAIT@li.101
||    2.6% |   2.166314 |   2000.0 | advec_cell_kernel_.ACC_COPY@li.238
||    2.4% |   1.970592 |   4000.0 | advec_mom_kernel_.ACC_COPY@li.68
||    1.5% |   1.278792 |   1000.0 | pdv_kernel_.ACC_SYNC_WAIT@li.133
||    1.2% |   1.034328 |   2000.0 | pdv_kernel_.ACC_SYNC_WAIT@li.137
||    1.2% |   0.987142 |   2000.0 | advec_cell_kernel_.ACC_COPY@li.58
||    1.2% |   0.976646 |   4000.0 | timestep_.ACC_COPY@li.51
```

Figure 5.10: Flat Profile Version A

However, understanding exactly how the underlying code is executing on the GPU is vital to realise if this performance is optimal or not.

CCE provides, by way of the `-r` option, the generation of a listing file. Depending on the sub-options invoked with this option, a set of compiler reports are concatenated to the listing file, detailing compiler listings, loopmark listings, source code listing and cross references. With the loopmark and source listing enabled, an understanding of how the compiler has translated the application can be ascertained.

Figure 5.11 shows the loopmark listing for Version A's *advec_cell*, but with pseudo code in place of the actual source for clarity.



```
G-----< !$acc parallel loop
G g---<  DO k = y_min, y_max
G g 3-<    DO j = x_min, x_max
G g 3-<        <stuff>
G g 3-<    ENDDO
G g---<  ENDDO
G-----< !$acc end parallel loop
```

```
G - Accelerated  g - partitioned
```

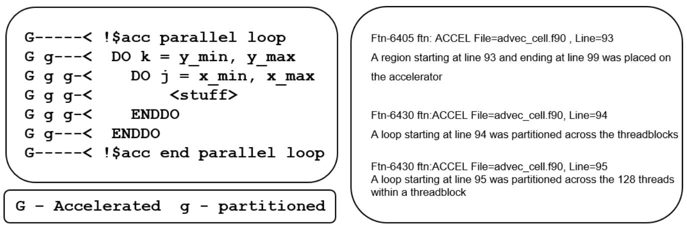
```
Ftn-6405 ftn: ACCEL File=advec_cell.f90 , Line=93
A region starting at line 93 and ending at line 99 was placed on
the accelerator

Ftn-6430 ftn:ACCEL File=advec_cell.f90, Line=94
A loop starting at line 94 was partitioned across the threadblocks
and the 128 threads within a threadblock

Ftn-6411 ftn: ACCEL File advec_cell.f90, Line=95
A loop starting at line 95 will be serially executed
```

Figure 5.11: Compiler Listing *advec_cell* Version A

The **G** indicates that the code block enclosed by the `!$acc parallel loop` and the `!$acc end parallel loop` is accelerated, this is also detailed in the associated dialogue: `A region starting at line 93 and ending at 99 was placed on the accelerator`. Additionally, the outer loop has the `g` loopmarking. This indicates that the loop has been distributed across the thread blocks and subsequently the threads within those blocks. Again, the associated dialogue indicates this: `A loop starting at line 94 was partitioned across the threadblocks and the 128 threads within a threadblock`.

Both of these statements and loopmarkings are the desired result, indicating that each thread is working on its own instance of `k`. However, on inspection of the inner loop the loopmarking specifies a numerical value (in this case "3"), along with the optimisation message: "`A loop starting at line 95 will be serially executed`". This indicates that `j` is split among the threads and all the threads are iterating the same value of `j` at the same time which is not the intended result.

To remedy this, the dependencies, or at least the dependencies the compiler perceives, in the code need to be addressed. In the case of *advec_cell* for each iteration of the inner loop a value is being calculated for pre and post mass, energy and volume. These updated values are then used in the same loop. This dependency is easily rectified by splitting the loop into two separate loops; one to calculate the values and a second to use these updated values.



```
G-----< !$acc parallel loop
G g---<  DO k = y_min, y_max
G g g-<    DO j = x_min, x_max
G g g-<        <stuff>
G g g-<    ENDDO
G g---<  ENDDO
G-----< !$acc end parallel loop
```

```
G - Accelerated  g - partitioned
```

```
Ftn-6405 ftn: ACCEL File=advec_cell.f90 , Line=93
A region starting at line 93 and ending at line 99 was placed on
the accelerator

Ftn-6430 ftn:ACCEL File=advec_cell.f90, Line=94
A loop starting at line 94 was partitioned across the threadblocks

Ftn-6430 ftn:ACCEL File=advec_cell.f90, Line=95
A loop starting at line 95 was partitioned across the 128 threads
within a threadblock
```

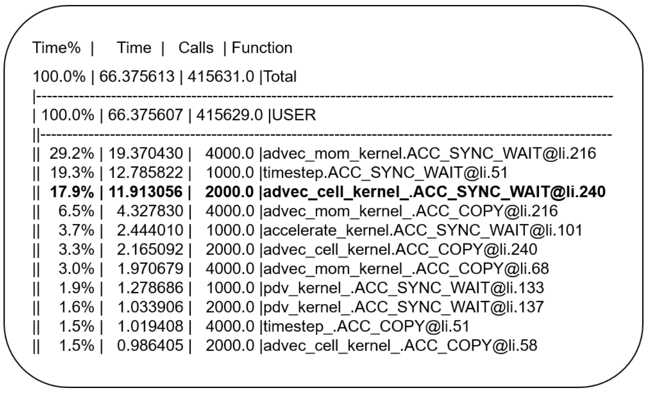
Figure 5.12: Compiler Listing *advec_cell* Version B

Once implemented an updated listing file (Figure 5.12) shows that the inner loop is now being correctly partitioned across the threads. The updated profile in Figure 5.13 shows *advec_cell* dropping from over 27 s to under 12 s.

This modified version of the code, denoted **Version B**, is compared to **Version A** in Figure 5.14.

5.3.10 Version C: Nested Loops and Global Variables

Figure 5.13 shows that what is now dominating runtime is *advec_mom*. Using the loopmarking in the listing file shows that loops containing multiple levels of nesting are not being accelerated. By removing these nested loops all but one are now being partitioned as desired. In the case of this non-partitioned loop, it can be forced to be scheduled across all the threads by addition of the



```
Time% |     Time |   Calls  | Function
100.0% | 66.375613 | 415631.0 |Total
|-----------------------------------------------------------------------------------------------------
| 100.0% | 66.375607 | 415629.0 |USER
||----------------------------------------------------------------------------------------------------
||  29.2% | 19.370430 |   4000.0 |advec_mom_kernel.ACC_SYNC_WAIT@li.216
||  19.3% | 12.785822 |   1000.0 |timestep.ACC_SYNC_WAIT@li.51
||  17.9% | 11.913056 |   2000.0 |advec_cell_kernel_.ACC_SYNC_WAIT@li.240
||   6.5% |  4.327830 |   4000.0 |advec_mom_kernel_.ACC_COPY@li.216
||   3.7% |  2.444010 |   1000.0 |accelerate_kernel.ACC_SYNC_WAIT@li.101
||   3.3% |  2.165092 |   2000.0 |advec_cell_kernel.ACC_COPY@li.240
||   3.0% |  1.970679 |   4000.0 |advec_mom_kernel_.ACC_COPY@li.68
||   1.9% |  1.278686 |   1000.0 |pdv_kernel_.ACC_SYNC_WAIT@li.133
||   1.6% |  1.033906 |   2000.0 |pdv_kernel_.ACC_SYNC_WAIT@li.137
||   1.5% |  1.019408 |   4000.0 |timestep_.ACC_COPY@li.51
||   1.5% |  0.986405 |   2000.0 |advec_cell_kernel_.ACC_COPY@li.58
```

Figure 5.13: Flat Profile Version B

OpenACC `vector` clause to the `!$acc loop` construct.

Figure 5.15 demonstrates the original nested loop structure along side its re-factored accelerated pseudo code. With the re-factored code the *advec_mom* kernel time drops from 19 s to 8 s, as highlighted in the profile presented in Figure 5.16.

What Figure 5.16 also shows is that the *timestep* routine is now dominating. Consulting the associated listing file for *timestep* reveals that the kernel is only executing on a single GPU thread.

The cause was traced to the use of global variables. As all threads have the potential to write to these global variables, the compiler takes a conservative approach and only allows the scheduling of the kernel on a single thread. The global variables in question are used in the timestep kernel to return the *(i,j)* coordinate of the cell which contains the minimum timestep values for the iteration. This functionally can be retained, without the need of global variables, by use of the Fortran intrinsic MINLOC.

Analysing the profile of **Version C**, (Figure 5.17) shows the timestep kernel's execution time is now reduced from 13 s to less than 1 s. Figure 5.18 shows the performance improvements delivered by **Version C** of the code, which includes the re-factored nested loops and the removal of the global variables. In comparison to **Version B** it can be seen that significant gains are now observed on comparative executions using a single GPU. However, although some gains are also observed on multiple GPU runs, it is still turning over once six GPUs are being utilised.

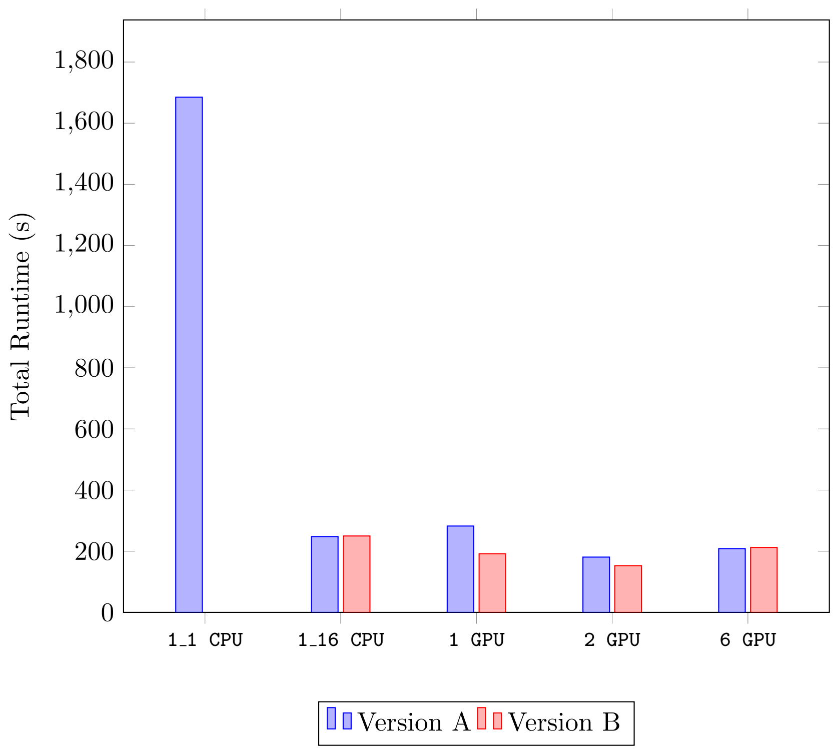


Figure 5.14: Version B Performance Comparison

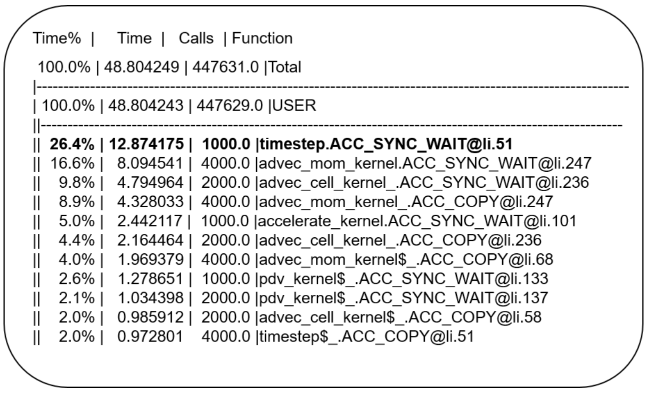
```
!$acc parallel loop
DO k = y_min, y_max
  DO j = x_min, x_max
     <flux stuff>
  ENDDO
  DO j = x_min, x_max
      <mass stuff>
  ENDDO
  DO j = x_min, x_max
     <vel stuff>
  ENDDO
ENDDO
!$acc end parallel loop
```

(a) Version B

```
!$acc parallel loop
DO k = y_min, y_max
  DO j = x_min, x_max
     <flux stuff>
  ENDDO
ENDDO
!$acc end parallel loop
!$acc parallel loop
DO k = y_min, y_max
  DO j = x_min, x_max
      <mass stuff>
  ENDDO
ENDDO
!$acc end parallel loop
!$acc parallel loop
DO k = y_min, y_max
!$acc loop vector
  DO j = x_min, x_max
     <vel stuff>
  ENDDO
ENDDO
!$acc end parallel loop
```

(b) Version C

Figure 5.15: Nested and Re-factored Pseudo Code for *advec_mom*



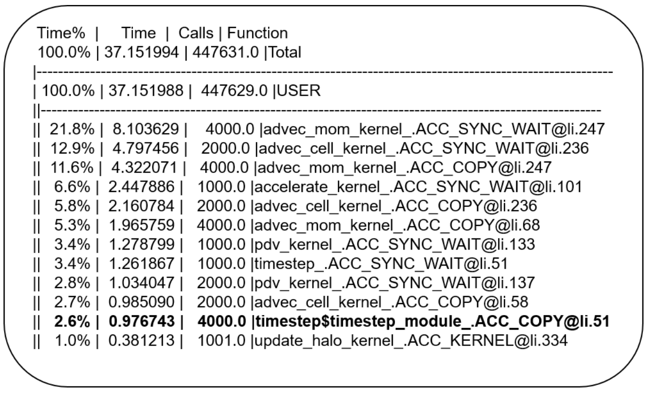
```
Time% |    Time  |   Calls  | Function
 100.0% | 48.804249 | 447631.0 |Total
|------------------------------------------------------------------------------------------------------------
| 100.0% | 48.804243 | 447629.0 |USER
||----------------------------------------------------------------------------------------------------------
||  26.4% | 12.874175 |   1000.0 |timestep.ACC_SYNC_WAIT@li.51
||  16.6% |   8.094541 |   4000.0 |advec_mom_kernel.ACC_SYNC_WAIT@li.247
||    9.8% |   4.794964 |   2000.0 |advec_cell_kernel_.ACC_SYNC_WAIT@li.236
||    8.9% |   4.328033 |   4000.0 |advec_mom_kernel_.ACC_COPY@li.247
||    5.0% |   2.442117 |   1000.0 |accelerate_kernel.ACC_SYNC_WAIT@li.101
||    4.4% |   2.164464 |   2000.0 |advec_cell_kernel_.ACC_COPY@li.236
||    4.0% |   1.969379 |   4000.0 |advec_mom_kernel$_.ACC_COPY@li.68
||    2.6% |   1.278651 |   1000.0 |pdv_kernel$_.ACC_SYNC_WAIT@li.133
||    2.1% |   1.034398 |   2000.0 |pdv_kernel$_.ACC_SYNC_WAIT@li.137
||    2.0% |   0.985912 |   2000.0 |advec_cell_kernel$_.ACC_COPY@li.58
||    2.0% |   0.972801     4000.0 |timestep$_.ACC_COPY@li.51
```

Figure 5.16: Flat Profile Version B with Re-factored *advec_mom*

5.3.11 Version D: Multi GPUs, Reducing Hidden Transfers

To understand why the scaling on the accelerator is limited to a few GPUs, attention is needed on the bottleneck affecting multi-GPU execution. The first area investigated was that of data transfers between the host CPU and GPU.

Depending on the variable in question, different depths of halo exchange cells are required for spatial domain decomposition. By default, the MPI distributed code sets the halo cell depth consistently for all variables. That default depth matches the maximum depth required for the worst case variable. Although implementation dependent, as MPI communication overheads are a stepping function, rather than a linear progression, the overheads in communication of one or two extra layers of data is negligible, if at all.



```
Time% |    Time  | Calls | Function
 100.0% | 37.151994 | 447631.0 |Total
|------------------------------------------------------------------------------------------------------------
| 100.0% | 37.151988 |  447629.0 |USER
||----------------------------------------------------------------------------------------------------------
||  21.8% |  8.103629 |    4000.0 |advec_mom_kernel_.ACC_SYNC_WAIT@li.247
||  12.9% |  4.797456 |   2000.0 |advec_cell_kernel_.ACC_SYNC_WAIT@li.236
||  11.6% |  4.322071 |   4000.0 |advec_mom_kernel_.ACC_COPY@li.247
||   6.6% |  2.447886 |   1000.0 |accelerate_kernel_.ACC_SYNC_WAIT@li.101
||   5.8% |  2.160784 |   2000.0 |advec_cell_kernel_.ACC_COPY@li.236
||   5.3% |  1.965759 |   4000.0 |advec_mom_kernel_.ACC_COPY@li.68
||   3.4% |  1.278799 |   1000.0 |pdv_kernel_.ACC_SYNC_WAIT@li.133
||   3.4% |  1.261867 |   1000.0 |timestep_.ACC_SYNC_WAIT@li.51
||   2.8% |  1.034047 |   2000.0 |pdv_kernel_.ACC_SYNC_WAIT@li.137
||   2.7% |  0.985090 |   2000.0 |advec_cell_kernel_.ACC_COPY@li.58
||   2.6% |  0.976743 |   4000.0 |timestep$timestep_module_.ACC_COPY@li.51
||   1.0% |  0.381213 |   1001.0 |update_halo_kernel_.ACC_KERNEL@li.334
```

Figure 5.17: Flat Profile Version C

However, any halo exchange data when running on distributed GPUs first needs to be transferred from the GPU to the host prior to CPU to CPU MPI communication and likewise the receiving CPU needs to transfer the new data to its host GPU ready for computation. As previously detailed, PCIe data transfers to and from the GPU are a major bottleneck, so any reduction should prove beneficial.

With this in mind the code was modified to only exchange data genuinely required. The CRAY_ACC_DEBUG runtime environment variable, documented from version 8.1.0.165 of CCE, assisted in this analysis. The variable has three informational setting levels, each providing increasing amounts of information relating to what is being transferred, and how large data is that is transferred, to and from the GPU, providing an informative runtime analysis.

Enabling this, showed unexpected data transfers in the *accelerate* kernel, not associated to the halo exchange data. Although relatively small in size (2,376 bytes) it stated that a Fortran derived type was being transferred and allocated on the GPU. On investigation, it became evident that scalar components of that particular derived type were being utilised in the *accelerate* kernel, and hence an implicit copy of the entire derived type was occurring to place it on the GPU. By creating local scalars and copying the appropriate fields from the derived type into these, resulted in the cessation of the implicit copy.

The resultant performance of **Version D** of the mini-app, with these two changes implemented, is shown in Figure 5.19. Although this is a significant improvement, multiple GPU scalability is still relatively modest, with two GPUs taking 73.99 s, while six take 67.01 s

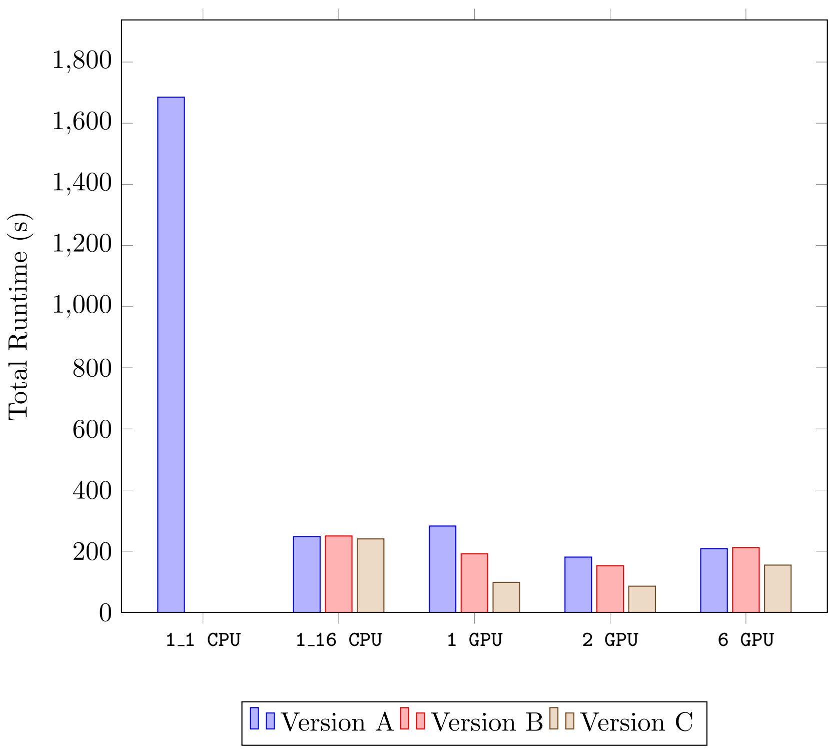


Figure 5.18: Version C Performance Comparison

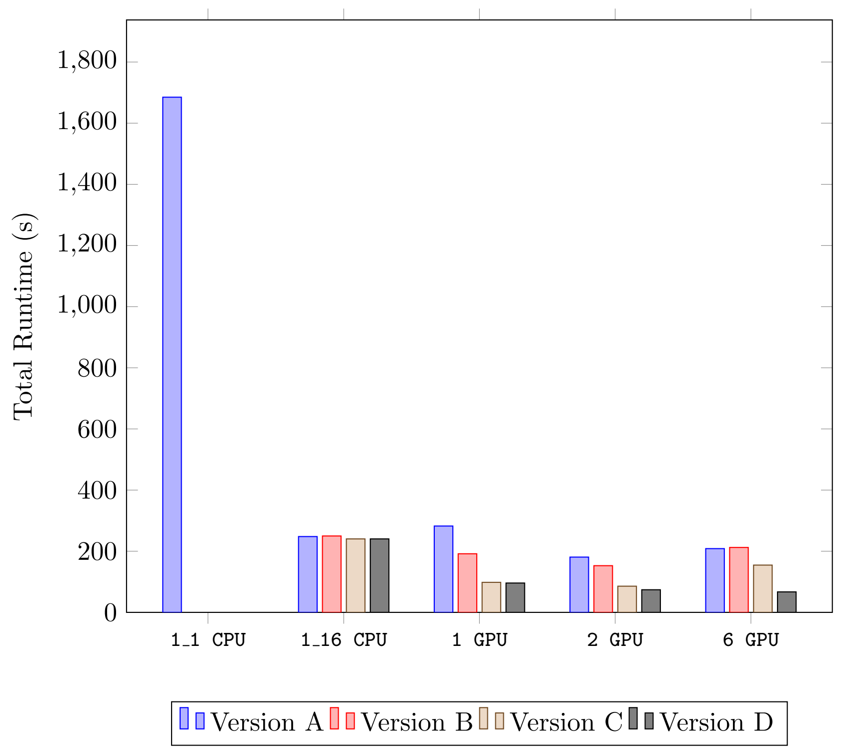


Figure 5.19: Version D Performance Comparison

5.3.12 Version E: "ACC_SYNC_WAITS"

A profile of **Version D** is shown in Figure 5.20. What is now dominating are a number of routines, all with the overhead of `ACC_SYNC_WAIT`. On further investigation, what all of these routines have in common is that they allocate data on the GPU.

```
Time% |     Time  | Calls | Function
 100.0% | 37.151994 | 447631.0 |Total
|------------------------------------------------------------------------------------------
| 100.0% | 37.151988 |  447629.0 |USER
||-----------------------------------------------------------------------------------------
||  21.8% |  8.103629 |   4000.0 |advec_mom_kernel_.ACC_SYNC_WAIT@li.247
||  12.9% |  4.797456 |   2000.0 |advec_cell_kernel_.ACC_SYNC_WAIT@li.236
||  11.6% |  4.322071 |   4000.0 |advec_mom_kernel_.ACC_COPY@li.247
||    6.6% |  2.447886 |   1000.0 |accelerate_kernel_.ACC_SYNC_WAIT@li.101
||    5.8% |  2.160784 |   2000.0 |advec_cell_kernel_.ACC_COPY@li.236
||    5.3% |  1.965759 |   4000.0 |advec_mom_kernel_.ACC_COPY@li.68
||    3.4% |  1.278799 |   1000.0 |pdv_kernel_.ACC_SYNC_WAIT@li.133
||    3.4% |  1.261867 |   1000.0 |timestep_.ACC_SYNC_WAIT@li.51
||    2.8% |  1.034047 |   2000.0 |pdv_kernel_.ACC_SYNC_WAIT@li.137
||    2.7% |  0.985090 |   2000.0 |advec_cell_kernel_.ACC_COPY@li.58
||    2.6% |  0.976743 |   4000.0 |timestep_.ACC_COPY@li.51
```

Figure 5.20: Version D: ACC_SYNC_WAIT Dominated Profile

By pre-allocating temporary arrays at the same high level of the calling tree as described in Section 5.3.4, and carrying out an initial data copy to the device, device memory can be re-used multiple times by passing the relevant arrays through subroutine arguments. This removes the need to check if the data is present and hence negates the need to create it on the device via an allocation.

```
SUBROUTINE advec_mom(a,b)
REAL ALLOCATABLE(:,:) :: node_flux

ALLOCATE(node_flux(xmin:ymax))

!$acc data present_or_create(node_flux)
!$acc parallel loop
DO k = y_min, y_max
  DO j = x_min, x_max
      <flux stuff>
  ENDDO
ENDDO
!$acc end parallel loop
!$acc end data
DEALLOCATE(node_flux)
END SUBROUTINE
```

(a) Version D

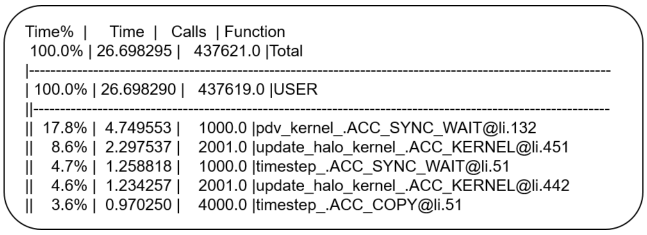
```
SUBROUTINE advec_mom(a,node_flux,b)
REAL DIMENSION(xmin,ymax) :: node_flux

!$acc data present(node_flux)
!$acc parallel loop
DO k = y_min, y_max
  DO j = x_min, x_max
      <flux stuff>
  ENDDO
ENDDO
!$acc end parallel loop
!$acc end data
END SUBROUTINE
```

(b) Version E

Figure 5.21: Pseudo Code for Pre and Post Pre-Allocations

Figure 5.21 shows pseudo code for both the original allocation method and the pre-allocation implementation. A new code profile of **Version E** (containing the implemented pre-allocations) is displayed in Figure 5.22. This shows the implementation of the pre-allocation has removed the `ACC_SYNC_WAIT` overheads.

Adding **Version E** runtimes to the plot of previous versions (Figure 5.23)



```
Time% |    Time  |   Calls  | Function
 100.0% | 26.698295 |   437621.0 |Total
|------------------------------------------------------------------------------------------------
| 100.0% | 26.698290 |   437619.0 |USER
||-----------------------------------------------------------------------------------------------
||  17.8% |  4.749553 |     1000.0 |pdv_kernel_.ACC_SYNC_WAIT@li.132
||    8.6% |  2.297537 |     2001.0 |update_halo_kernel_.ACC_KERNEL@li.451
||    4.7% |  1.258818 |     1000.0 |timestep_.ACC_SYNC_WAIT@li.51
||    4.6% |  1.234257 |     2001.0 |update_halo_kernel_.ACC_KERNEL@li.442
||    3.6% |  0.970250 |     4000.0 |timestep_.ACC_COPY@li.51
```

Figure 5.22: Flat Profile Version E

shows a significantly improved accelerated version of the code. Indeed, **Version E** gives a 67% improvement in total turnaround time over that of the initial **Version A** on a single GPU.

It is also apparent, comparing Versions A and E in Figure 5.23, that the described optimisations applied to achieve GPU performance have a significant impact on CPU performance; these will be addressed in Section 5.3.13.

Recall from Section 5.3.8, the performance figures for **Version A**:

- One X2090 GPU is a factor of 5.97 faster than one Opteron CPU core
- One X2090 GPU is 0.88 times faster than one Opteron socket
- Multi-GPU scaling turns over once six GPU are utilised.

These same metrics for **Version E** are now:

- One X2090 GPU is a factor of 19.34 faster than one Opteron CPU core
- One X2090 GPU is 4.91 times faster than one Opteron socket
- Multi-GPU scaling is still scaling when six GPU are utilised.

With a TDP of 115W for the Opteron and 225W for the X2090, this implies a performance increase of over 2.5 for the X2090, for the same power footprint.

5.3.13 Impact of GPU Optimisations on CPU

An interesting, yet relevant aside is to look at the relative performance of the final, GPU optimal, incarnation of the code (**Version E**) against that of the initial version (**Version A**) when running exclusively on the CPU. Recall, all optimisations carried out between these two versions were aimed at improving and fixing issues detrimental to performance on the GPU. Yet, comparing single CPU-only performance **Version E** gives over a 17% improvement over that of **Version A**.

The main contributors to this improvement are the optimisations to allocate, deallocate and subsequently re-allocate the arrays as described in Section

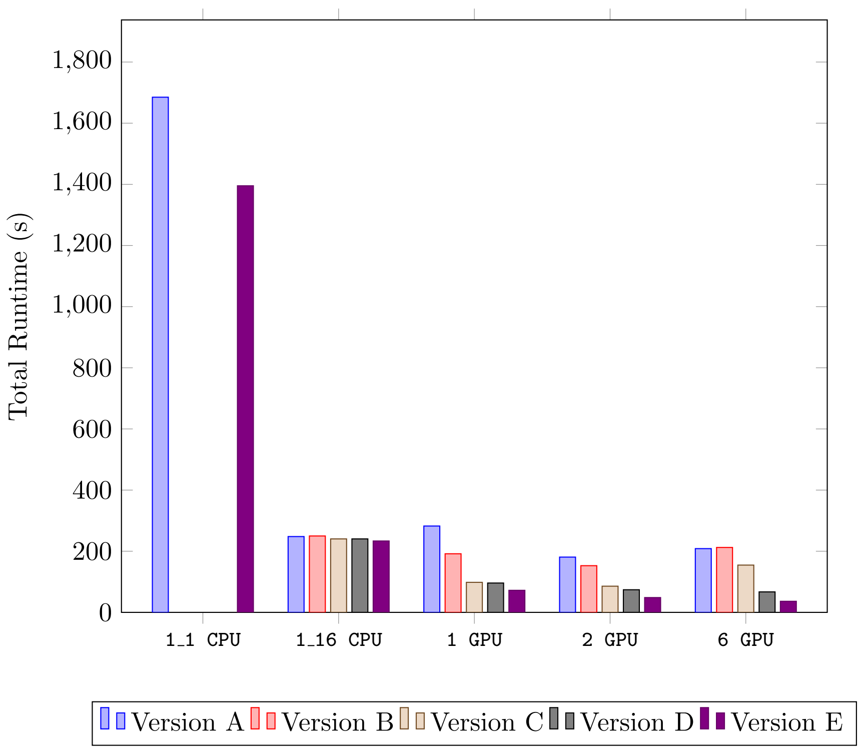
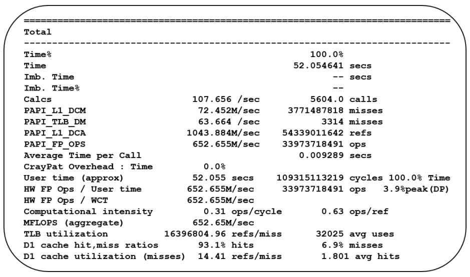


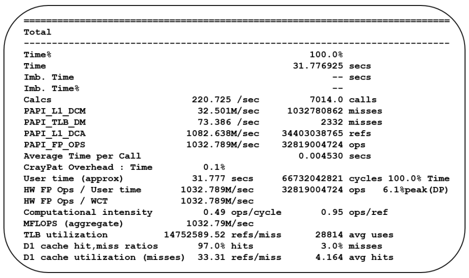
Figure 5.23: Version E Performance Comparison

5.3.12. Additionally, this is further emphasised when the derived metrics for each version's performance are compared. Although **Version A** is realising a respectable 652 MFLOP/s (Figure 5.24), this increases to over 1 GFLOP/s for Version E (Figure 5.25). This increase comes from an increase in L1 cache utilisation (up by a factor of 2.3), which is now averaging 4.164 uses per operand.



```
=============================================================================
Total
-----------------------------------------------------------------------------
Time%                                               100.0%
Time                                             52.054641 secs
Imb. Time                                               -- secs
Imb. Time%                                              --
Calcs                         107.656 /sec          5604.0 calls
PAPI_L1_DCM                    72.452M/sec      3771487818 misses
PAPI_TLB_DM                    63.664 /sec            3314 misses
PAPI_L1_DCA                  1043.884M/sec     54339011642 refs
PAPI_FP_OPS                   652.655M/sec     33973718491 ops
Average Time per Call                             0.009289 secs
CrayPat Overhead : Time          0.0%
User time (approx)            52.055 secs     109315113219 cycles 100.0% Time
HW FP Ops / User time        652.655M/sec      33973718491 ops   3.9%peak(DP)
HW FP Ops / WCT              652.655M/sec
Computational intensity          0.31 ops/cycle       0.63 ops/ref
MFLOPS (aggregate)            652.65M/sec
TLB utilization          16396804.96 refs/miss       32025 avg uses
D1 cache hit,miss ratios        93.1% hits            6.9% misses
D1 cache utilization (misses)  14.41 refs/miss        1.801 avg hits
```

Figure 5.24: Derived Metrics: Version A



```
=============================================================================
Total
-----------------------------------------------------------------------------
Time%                                               100.0%
Time                                             31.776925 secs
Imb. Time                                               -- secs
Imb. Time%                                              --
Calcs                         220.725 /sec          7014.0 calls
PAPI_L1_DCM                    32.501M/sec      1032780862 misses
PAPI_TLB_DM                    73.386 /sec            2332 misses
PAPI_L1_DCA                  1082.638M/sec     34403038765 refs
PAPI_FP_OPS                  1032.789M/sec     32819004724 ops
Average Time per Call                             0.004530 secs
CrayPat Overhead : Time          0.1%
User time (approx)            31.777 secs      66732042821 cycles 100.0% Time
HW FP Ops / User time       1032.789M/sec      32819004724 ops   6.1%peak(DP)
HW FP Ops / WCT             1032.789M/sec
Computational intensity          0.49 ops/cycle       0.95 ops/ref
MFLOPS (aggregate)           1032.79M/sec
TLB utilization          14752589.52 refs/miss       28814 avg uses
D1 cache hit,miss ratios        97.0% hits            3.0% misses
D1 cache utilization (misses)  33.31 refs/miss       4.164 avg hits
```

Figure 5.25: Derived Metrics: Version E

5.3.14 Multi-GPU Scalability

To better observe the strong scaling of the code, the 960^2 cell test problem executing on one Opteron core, was scaled out to all 16 cores resulting in a test size of 3840^2 cells. To reduce wall clock times during experiments, the simulation time of the system was also reduced back to 0.5 µs.

Results for scaling this larger case on the GPU are presented in Figure 5.26. Utilising all 32 GPUs on Chilean Pine, parallel efficiency dips just under 50% with a speed up of 15.42 over one GPU. This is not entirely unexpected, as the data sizes per GPU at this point (as a fraction of GPU memory) are relatively small. With a higher ratio of communications relative to computation, the parallel efficiency would not be expected to be large.

Weak scaling figures based on average cell execution time (micro-seconds per cell) are presented in Figure 5.27. Once nine GPUs are deployed, communication is instigated in both directions, in both dimensional planes; that is to say there

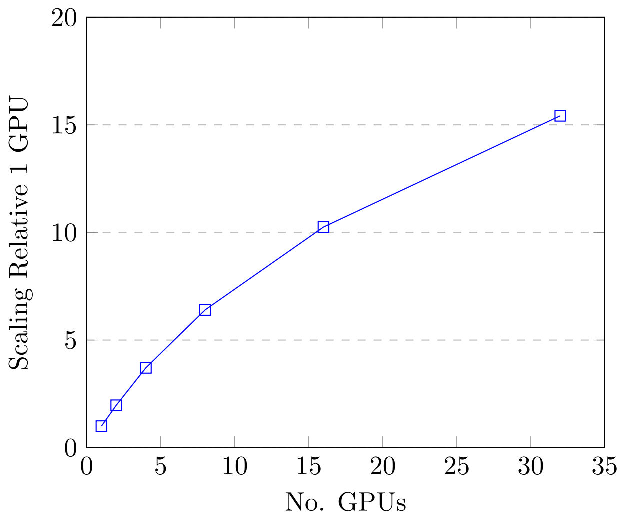


Figure 5.26: 0.5 µs, 3840^2 cells, Strong Scaled

is a “central” GPU which will communicate in all four directions. Once this occurs, the cost per cell per timestep stays constant as the number of GPUs are increased. At over 96% parallel efficiency, this shows good scaling on the whole of the machine.

Comparing the strong and weak scaling performance, this demonstrates that the GPU is very good at compute as long as it is kept “filled”, that is keeping the ratio of computation to communication high.

5.4 Summary

Once identified, OpenACC was applied to each kernel on an individual basis, and the breakdown of compute and data transfer to and from the CPU can be assessed on a kernel by kernel basis.

Once all the main compute kernels were accelerated, they were then executed in unison. This showed the overall performance being achieved from execution on the GPU. However, as each kernel was still transferring data to and from the host, it also highlighted the overhead of the data transfer.

To remove the impact of the data transfers, the code was made fully resident on the GPU device. This removed all but an initial copy to the GPU, the copies from the MPI communications and a final copy back to the host CPU.

However, to make fully resident, and run exclusively on the GPU, not only did all of the data need to be identified for the initial transfer, but also those parts of the code that were not necessarily computationally intense also needed to be made to execute on the GPU; that is extra kernels needed to have

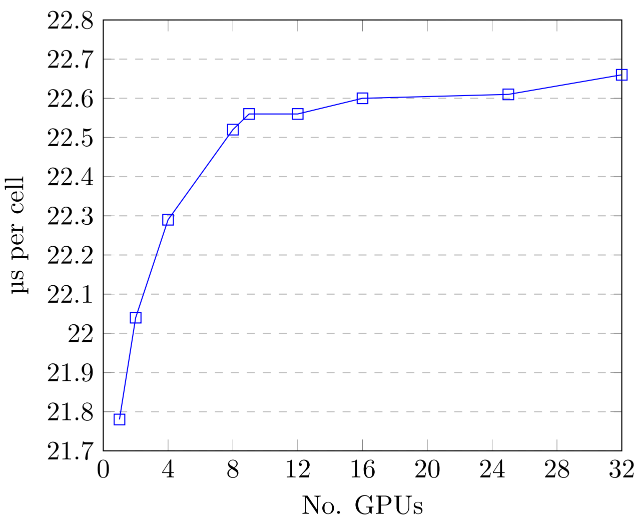


Figure 5.27: 0.5 µs, 3840^2 cells, Weak Scaled

OpenACC applied.

Once the entire application was resident and running exclusively on the GPU, the performance was then investigated. Firstly, the impact of increasing the problem size was explored, and once a large enough case identified that suitably occupied the GPU, the performance was compared against the best CPU performance possible from the most optimal variant of the code (MPI, OpenMP, MPI/OpenMP hybrid) available at the time of study. Based on these comparisons, a range of optimisations were applied by analysing how each kernel was utilising the GPU's threads.

Finally, attaining a robust, optimal fully resident OpenACC version of the code, the application was extended to use multiple GPUs by developing a hybrid MPI/OpenACC implementation. This resulted in a performance increase of 5.82x over a single Opteron CPU core, but showed detrimental performance compared to an entire CPU socket.

A study was subsequently carried out which identified those areas inhibiting performance, as summarised in Table 5.3

Once rectified, a single GPU now realises a factor of 19.34 over a single Opteron CPU core, and a factor of 4.91 over an entire socket.

These optimisations also benefited CPU only performance with an increase from 652 MFLOP/s to in excess of 1 GFLOP/s.

Increasing the problem size, strong and weak scaling showed almost 50% parallel efficiency for the former, and over 96% for the latter when executed across the entirety of the Chilean Pine XK6 platform.

Issue	**Remedy**
Inner loop dependencies, “calculating and using”	Split loops into “calculating” and “use”
Multiple nested loops not accelerating due to outer loop fusion	Re-factor loops into own entity and use of loop vector directive
Global variables, single thread execution	Remove
Hidden data transfers	Identified and removed/minimised
“ACC_SYNC_WAITS”	Temporary array allocation and reuse

Table 5.3: Summary of Performance Issues

5.5 Chapter Summary

OpenACC is a directive based programming model to allow the code developer to identify areas of code to be accelerated on a hosted device. Adding directives to an existing code base is an attractive proposition when compared to re-writing in a new language and realises some of the benefits from an open, non-disruptive approach. Vendors are developing backends that can still utilise their proposed methodology, yet have the OpenACC standard as a common interface. By investigating the OpenACC programming model, through the hydrodynamic mini-app CloverLeaf, an idea of the steps and understanding needed to accelerate an application has been gained. It has been shown that it is not just the case of working on the “hot spots” of an application and applying directives, but that the time has to be taken to understand what is happening at the hardware level in those targeted subroutines and kernels.

The re-factoring of compute kernels to be data parallel is key. As discussed in Chapter 2, the number of processing elements on accelerated hardware can run into the thousands, it is essential to make sure all kernels are scheduled across all the device threads. In some cases this will be at the expense of additional memory usage which can be required in order to remove dependencies.

Data transfers severely limit achievable performance; it has been shown that this can even be true for the smallest pieces of data, which instigates a much larger implicit data transfer.

Optimisations required for a performant GPU implementation can also give rise to significant benefits when applied to a CPU implementation. These optimisations enabled the kernels to be expressed in a data parallel manner, which not only meant they threaded well on the GPU, but also when running multi-threaded via OpenMP on a CPU. Additionally, the re-factoring of the code also allowed the compiler to perform better scale and vector optimisations.

A number of discoveries that were made through the process of developing a performant OpenACC mini-app are applicable to the physics domain of structured 2D hydrodynamics applications in general. The technique utilised to minimise the data communicated during halo cell updates makes a significant difference on the GPU compared to the CPU, something that is commonplace in all structured hydrodynamics schemes.

In the case of multiple nested loops not accelerating due to outer loop fusion, this is a point of focus for all applications with similar loop structures, not only restricted to the 2D hydrodynamics domain. This would indicate the application developer should take the time to investigate that their code is indeed behaving optimally in any nested loop regions when executing on the GPU. Likewise, issues limiting performance issues were identified with the allocation, deallocation and re-alloaction style of programming that is prevalent in many applications. This should be a point of focus for any application that uses this structure wishing to execute on a GPU.

This development of a performant OpenACC implementation has subsequently formed the basis for a number of further research activities. This initial OpenACC version was used as the basis of an implementation that would compile and perform under the PGI and CAPS OpenACC compilers. This ultimately led to two OpenACC implementations, the "parallel" and "kernel" versions.

The "kernel" version has been adopted by The PGI Compiler Group for internal regression testing of their compiler. Also as part of the inaugural release of the Standard Performance Evaluation Corporation's (SPEC) High-Performance Group (HPG) SPEC ACCEL$^{\text{TM}}$benchmark [44, 119], both the C and Fortran OpenACC versions of CloverLeaf [43] are one of the 15 OpenACC benchmark applications used to provide a comparative performance measure across a range of accelerator hardware.

Work led by Mallinson [137] performed a number of MPI specific optimisations: pre-posting receives, MPI rank re-ordering and overlapping of communications and computation, resulting in weak scaling the MPI+OpenACC hybrid CloverLeaf code to all 16,384 GPUs on ORNL's Titan [65] supercomputer.

Although this chapter has demonstrated the development of an optimal OpenACC version of the mini-app, it does not address the question of whether there are better options for utilising accelerated hardware. How the performance of alternative programming methodologies compares and what are the relative costs in terms of development and intrusiveness are open questions.

The following chapters take alternative variants and begin to explore their performance and productivity on a particular architecture, and their performance portability on a range of many-core architectures.

The current releases of CloverLeaf's OpenACC versions discussed in this chapter are available for download from dedicated GitHub web pages [75].

CHAPTER 6
Using mini-apps to Explore Performance Portability

6.1 Introduction

The previous chapter introduced the idea of the mini-app, in particular that of CloverLeaf. The detailed step-by-step approach taken was highlighted resulting in a performant OpenACC version of the code. Analysis of the code during the creation of a performant version identified a number of key factors in achieving such a performant version: re-factoring, data residency, local buffer packing, minimal data transfer and focusing on code sections containing multiple nested loops.

Reference was also made to the whole range of implementations of CloverLeaf in various programming models and paradigms. This chapter presents a comparative study of the CloverLeaf hydrodynamics mini-app ported to GPUs using three of those technologies: OpenACC, OpenCL and CUDA. Specifically, it makes the following key contributions:

- In the context of the CloverLeaf mini-application, the first direct comparison between OpenACC, OpenCL and CUDA, the three dominant programming models for GPU architectures, is given.
- A quantitative and qualitative comparison of these three approaches with regard to code development, maintenance, portability and performance on two problem sizes of interest on a GPU-based cluster (Cray XK6), is given.

The remainder of this chapter is organised as follows: Section 6.2 discusses related work in this field; Section 6.3 provides details on each of the three implementations used in this study, as well as the changes needed to make the overall algorithm more amenable to parallelisation on the GPU architecture. The results of the study are then presented in Section 6.4 together with a description of the experimental set-up and a description of the current ecosystems for each of the programming paradigms covering tool support, developer communities and the future directions of the standards is given in Section 6.5. Finally, Section 6.6 concludes the chapter by discussing the relative merits of the alternative programming methodologies in each of the categories described: performance, development, maintenance and future.

6.2 Related Work

NVIDIA CUDA is currently the most mature and widely used technology for developing applications for GPUs. However, directive-based approaches such as OpenACC, driven by the work from the Center for Application Acceleration Readiness (CAAR) team at Oak Ridge National Laboratory (ORNL) are becoming increasingly used [65]. Fewer studies have been carried out on the assessment of OpenCL.

At the time of study, the only direct comparison between OpenACC and OpenCL available was that of Wienke *et al.* [195]. Their work, however, is focused on two applications from significantly different domains: the simulation of bevel gear cutting, and a neuromagnetic inverse problem.

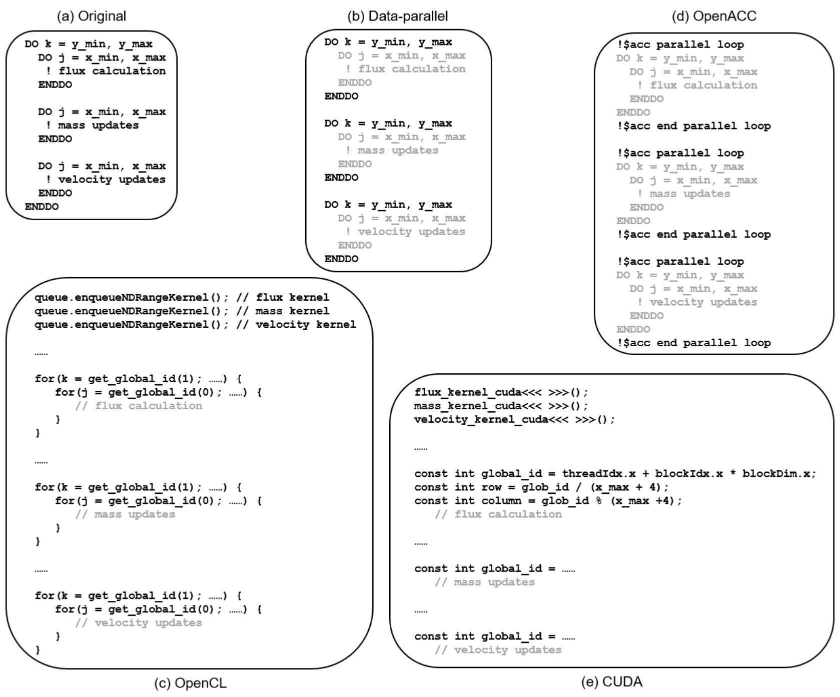
A number of studies have evaluated the applicability of non-mesh based hydrodynamical methods (unlike CloverLeaf's structured grid), to GPU accelerated hardware [92, 120, 169, 182]. Although these studies have limited their scope to CUDA and not compared performance, productivity or portability with alternative approaches such as OpenCL or OpenACC, which is a key focus of this work.

An OpenCL implementation of a compressible gas dynamics code was developed and described by Bergen [63] and similarly an OpenCL version of a single code-based structured library for multi-science applications by Shukla [188] have been developed. However, these both solely focus on OpenCL implementations and do not address performance or alternative approaches.

The port of a Euler-based solver application and a Boltzmann based solver application were described by Brook [67]. Although there is some analogy between the former application and that of CloverLeaf, Brook's focus was exclusively on an OpenMP implementation specifically targeting an Intel Xeon Phi based architecture.

Lattice Quantum Chromodynamics (QCD) is an additional domain which has seen numerous applications successfully ported to GPUs. However, a number of these studies employ the QUDA library [36, 53, 187]. This library is based on NVIDIA CUDA technology and therefore these studies do not examine alternative approaches such as OpenCL or OpenACC.

While all the above works primarily discuss the development of the application in question, the work described in this chapter takes three alternative programming methodologies and compares their relative merits.



```
DO k = y_min, y_max
  DO j = x_min, x_max
   ! flux calculation
  ENDDO

  DO j = x_min, x_max
   ! mass updates
  ENDDO

  DO j = x_min, x_max
   ! velocity updates
  ENDDO
ENDDO
```

```
DO k = y_min, y_max
  DO j = x_min, x_max
   ! flux calculation
  ENDDO
ENDDO

DO k = y_min, y_max
  DO j = x_min, x_max
   ! mass updates
  ENDDO
ENDDO

DO k = y_min, y_max
  DO j = x_min, x_max
   ! velocity updates
  ENDDO
ENDDO
```

```
!$acc parallel loop
DO k = y_min, y_max
  DO j = x_min, x_max
   ! flux calculation
  ENDDO
ENDDO
!$acc end parallel loop

!$acc parallel loop
DO k = y_min, y_max
  DO j = x_min, x_max
   ! mass updates
  ENDDO
ENDDO
!$acc end parallel loop

!$acc parallel loop
DO k = y_min, y_max
  DO j = x_min, x_max
   ! velocity updates
  ENDDO
ENDDO
!$acc end parallel loop
```

```
queue.enqueueNDRangeKernel(); // flux kernel
queue.enqueueNDRangeKernel(); // mass kernel
queue.enqueueNDRangeKernel(); // velocity kernel

......

for(k = get_global_id(1); ......) {
   for(j = get_global_id(0); ......) {
      // flux calculation
   }
}

......

for(k = get_global_id(1); ......) {
   for(j = get_global_id(0); ......) {
      // mass updates
   }
}

......

for(k = get_global_id(1); ......) {
   for(j = get_global_id(0); ......) {
      // velocity updates
   }
}
```

```
flux_kernel_cuda<<< >>>();
mass_kernel_cuda<<< >>>();
velocity_kernel_cuda<<< >>>();

......

const int global_id = threadIdx.x + blockIdx.x * blockDim.x;
const int row = glob_id / (x_max + 4);
const int column = glob_id % (x_max +4);
   // flux calculation

......

const int global_id = ......
   // mass updates

......

const int global_id = ......
   // velocity updates
```

Figure 6.1: Key differences between implementations of the advection computational kernel in CloverLeaf.

6.3 Implementations

The profiling of CloverLeaf as described in Chapter 5, shows that approximately 95% of the execution time is contained in six computationally intense kernels (Table 5.1). However, to achieve full GPU residency, *i.e.* the physics algorithm executed exclusively on the GPU with necessary data residing in device rather than host memory, all twelve unique kernels are required to be ported to the accelerator device, leaving only control code to executed on the host CPU.

The "data-parallel" nature of the OpenMP implementation of the mini-application was an ideal basis to create each of the new implementations. Exposing the loop-level parallelism required by OpenMP required a combination of loop splitting and adding extra temporary data storage to enable additional temporary data to be reused. Whilst porting the code to the accelerator, the data parallelism within each kernel was improved, and these changes were applied back to the OpenMP version to increase CPU performance.

The development of the advection kernel in each of the three programming models is shown in Figure 6.1; a similar approach was used for each of the remaining kernels. The original Fortran code was first modified in order to remove dependencies between loop iterations. The loops, however, must still be completed sequentially, as each loop uses data calculated by the previous loop.

6.3.1 OpenACC®

As fully detailed in Chapter 5, in order for the author to convert the data-parallel version of the kernel to OpenACC, loop-level pragmas were added to specify how the loops should be run on the GPU, and to describe their data dependencies.

For effective use of the GPU, data transfers between the host processor and the accelerator must be kept to a minimum. CloverLeaf is fully resident on the device; this was achieved by applying OpenACC data *"copy"* clauses at the start of the program, which results in a one-off initial data transfer to the device. The computational kernels exist at the lowest level within the application's call-tree and therefore no data copies are required. This is achieved by employing the OpenACC *"present"* clause to indicate that all input data is already available on the device.

As in any block-structured, distributed MPI application, there is a requirement for halo data to be exchanged between MPI tasks. In the accelerated versions, however, this data resides on the GPU local to the host CPU, hence the data which is to be exchanged is transferred from the accelerator to the host via the OpenACC *"update host"* clause. MPI send/receive pairs exchange data in the usual manner, and then the updated data is transferred from the host to its local accelerator using the OpenACC *"update device"* clause. A key point to note is that the explicit data packing (for sending) and unpacking (for receiving) is carried out on the device for maximum performance.

6.3.2 OpenCL™

An OpenCL [125] port of CloverLeaf was developed by Mallinson, then of the University of Warwick. The C bindings that form the interface to the functionality described by the OpenCL standard mean that integrating directly with the Fortran codebase of CloverLeaf is difficult. To ease programmability, a C++ header file is provided by the Khronos Group which allows access to the OpenCL routines in a more object-oriented manner [32]. This header file was used by a static C++ class to manage the interaction between the original Fortran code and the new OpenCL kernels. The class holds details about all the buffers and kernels used by the application, allowing C functions (which are easily callable from Fortran) to be written that initiate kernels and transfer data as needed.

As with the OpenACC version of the code, data transfers between the host processor and the device must be minimised in order to maximise performance. This is achieved by creating and initialising all data items on the device, and

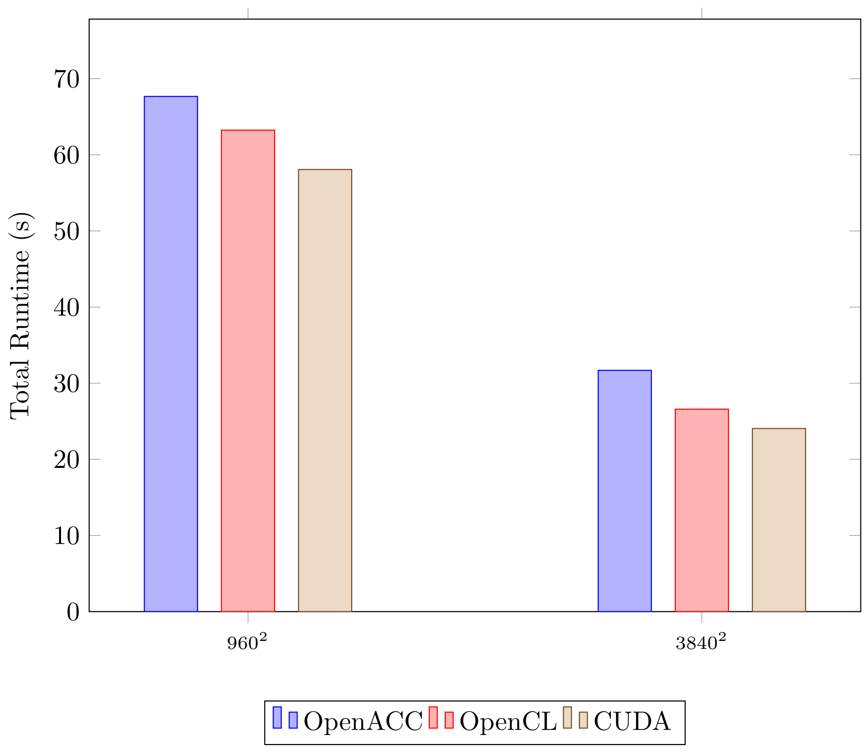


Figure 6.2: Total runtimes for 960^2 and 3840^2-cell problems

allowing these to reside on the GPU throughout execution. Data is only copied back to the host in order to write out visualisation files, and for MPI communications.

6.3.3 NVIDIA®CUDA®

Developed in conjunction with the University of Bristol (Michael Boulton and Simon McIntosh-Smith) and the NVIDIA Corporation (Tom Bradley), the CUDA [78] implementation of CloverLeaf is almost identical in design to the OpenCL implementation. It was implemented using a global class that coordinated data transfer and computation on the GPU, with helper functions to handle interoperability between the CUDA and Fortran code.

6.4 Results

6.4.1 Experimental Setup

All experiments were conducted on Chilean Pine, a Cray XK6 hosted at the Atomic Weapons Establishment (AWE) (see Appendix A Section A.4.3 for details). The default Fortran and C compilers are the Cray Compiling Environment (CCE) and the choice of MPI is MPICH2. The OpenCL version, however, was built with the GNU compiler environment, as utilisation of the Cray compiler with the C++ OpenCL constructs, proved unsuccessful. The CUDA kernels were compiled with the appropriate flags to enable double precision calculation capability on the Fermi architecture[1]. Cray's *CrayPAT* profiling tool was used to produce the timing profile for the OpenACC version, whereas for the OpenCL and CUDA versions kernel timings were derived by subsequently querying the *event* objects returned by each kernel invocation.

6.4.2 Performance Analysis

The performance of the three accelerated implementations of CloverLeaf was tested using the test case described in Chapter 5, Section 5.1.2: a representative asymmetric shock wave problem. Two test configurations, described in terms of the number of cells in the computational mesh, were used in the experiments: a smaller 960^2-cell problem (introduced in Chapter 5, Section 5.3.5) and a large 3840^2-cell problem (introduced in Chapter 5, Section 5.3.14). The performance of these two problems was analysed using one node of Chilean Pine containing a single NVIDIA X2090 GPU.

Figure 6.2 shows the overall runtimes for the OpenACC, OpenCL and CUDA versions for each problem set. Although a similar pattern for both test cases is observed, in that the the CUDA outperforms the OpenCL, which in turn outperforms the OpenACC, the relative differences are larger for the larger problem. OpenCL is 6.5% faster than OpenACC for the 960^2-cell problem and 16.1% for the 3840^2-cell problem, while CUDA is 14.19% and 24.11% faster than the equivalent OpenACC for the 960^2 and the 3840^2-cell problems respectively.

The CUDA version has been targeted for the specific Fermi hardware as part of compilation and as such should be greater optimised for the architecture.

The increase in data in the larger problem set could be a contributing factor in the different relative performance due to the reduction operations inherent in each programming model. OpenACC reduction's will be implemented within Cray's CCE compiler and expectations would be that this would have been highly optimised. The CUDA version of the code uses a reduction coded partially by hand, and partially provided by the Thrust library [105]. Meanwhile, the OpenCL version of the code uses a hand-coded reduction. To investigate this hypothesis, the runtime of the code would need to be broken down into its kernel components. This enabled the relative performance differences on a per kernel basis to be assessed to determine if there is a correlation to those kernels

[1] `-gencode arch=compute_20,code=sm_21`

Version	**WOC**			**Kernel Language**	**Tools**	**Portability**
	Total	**Device**	**Host**			
OpenACC	1 510	-	-	Fortran	Good	Average
OpenCL	17 930	4 327	13 608	OpenCL C	Poor	Good
CUDA	13 628	5 830	7 798	CUDA C	Average	Poor

Table 6.1: Key development metrics for the three versions.

containing reduction operations. This is explored further in Chapter 7.

6.4.3 Productivity Analysis

In order to assess the programmer productivity offered by each approach a number of factors were taken into consideration: the number of *words* of code (WOC) added for each version, considered whether computational kernels needed rewriting, and the tool support available for each version. By considering WOC (not including symbols such as braces and parentheses) a metric was derived that overcame the variations caused by different programming styles, something which affects the lines of code (LOC) metric.

In terms of programmer productivity, OpenACC proved superior to both OpenCL and CUDA, requiring the addition of only 184 OpenACC pragmas (1,510 WOC). The OpenCL and CUDA versions of the code required an additional 17,930 and 13,085 WOC respectively. However, of the 17,930 words required by OpenCL, 3,958 can be attributed to the static class created to manage the OpenCL objects. This static class could be used in other similar applications with little modification, meaning the total amount of OpenCL code unique to CloverLeaf is 13,972 words. Additionally, OpenCL and CUDA both required extra work to re-write the computational kernels in C-style code. However, the simple design of the Fortran kernels eliminated much of the work that might be required in a legacy code.

Developing the OpenACC version in an incremental manner (*i.e.* one kernel at a time) proved to be a straightforward process, which made validating and debugging the code considerably easier. Whilst it was also possible to develop the OpenCL and CUDA versions of the code in an incremental manner, the significantly larger code volumes required for each increment and the immaturity of the tool support, particularly for OpenCL, made debugging problems harder and more time consuming.

CloverLeaf requires the use of several reduction operations. Under OpenACC, reductions were described by pragmas, and implemented by the OpenACC compiler. In CUDA, the reductions were implemented using a simple two stage approach. The first stage reduces within each block (a block being the CUDA equivalent to an OpenCL work group), producing an array of partial reductions (*i.e.* one per block). The second stage then combines the elements of the resulting partial-reduction array; this was implemented using the Thrust C++ library. At the time of the study an optimised library within the OpenCL "ecosystem" which provided equivalent functionality was unavailable and therefore Mallinson implemented a hand-crafted reduction in a similar multi-stage approach.

6.4.4 Portability Analysis

At the the time of study, the Cray OpenACC implementation, using OpenACC's "parallel" construct, is the only implementation to have been utilised extensively in this work.

The experience at the time indicated that the full implementation of the OpenACC standard was not present in all vendor compiler offerings, with different vendors focused on implementing different aspects of the standard, resulting in only Cray having a full "parallel" construct implemented.

Therefore, the then current implementation was constrained to the Cray platform, under the CCE compiler environment, using OpenACC's "parallel" construct. This limitation is described further in Section 3.2.2 of Chapter 3.

Chapter 7 subsequently addresses this by developing a fully compiler agnostic OpenACC code, for both OpenACC's "parallel" and "kernel" constructs following greater vendor compiler adoption of the standard.

Similarly, utilising CUDA as a mechanism to take advantage of accelerator devices limits the choice of officially supported hardware platforms available to an organisation, as NVIDIA only supports CUDA on their own hardware. The Ocelot project [27], and PGI's CUDA Fortran [35] compiler do however provide alternatives for other languages and hardware.

The OpenCL version of the code exhibited the highest portability, and using this version it was possible to execute the application on both AMD and NVIDIA GPU devices, AMD and Intel CPUs and also a pre-production Intel Xeon Phi.

These observation are picked up and discussed in Chapter 7 with a more detailed investigation into OpenACC and alternative programming methodologies including a direct comparison with OpenCL on an Intel Xeon Phi.

6.5 Supporting Infrastructure

Another consideration of the relative merits of each implementation is the supporting infrastructure and user communities that exist. Strong communities enable the development of supporting tools and ancillary software that can assist in the development and subsequent maintainability of an application.

The two major commercial debugging platforms are Rougue Wave's Totalview and Allinea's DDT. Both CUDA and OpenACC are supported in Totalview; beginning with version 8.10 of the debugger, released mid 2012 [37], support was limited to CUDA and OpenACC (only under Cray's CCE) and only on Cray's XK6 architecture. Indeed, beta versions of v8.10 were instrumental in the development stages detailed in Chapter 5. At the time of writing CUDA [41] and OpenACC [42] support from PGI, in addition to CCE, is available on Linux x86-64. Likewise, DDT fully supports CUDA and all OpenACC compiler providers [40].

A CUDA debugger is one component of NVIDIA's Nsight development platform enabling debugging of CUDA code, albeit restricted to NVIDIA hardware.

To have similar levels of debugging for OpenCL, that is the ability to step through source code, setting breakpoints and inspect variables, there are only hardware specific tools available. For Intel hardware debugging is possible through plug-ins to Intel's Visual Studio IDE (Integrated Development Environment), while for AMD hardware there is AMD's developer tool suite, CodeXL, of which one of its component is a graphical debugger. At the time of writing there is no equivalent available for debugging OpenCL code on NVIDIA hardware.

To profile the resulting code, with the aim of identifying code hot spots, or areas of poor performance, there are a range of options depending on the programming implementation. In the case of OpenCL, again the AMD specific CodeXL developer tool suite offers a profile tool which will gather performance data at runtime.

PGI's OpenACC runtime 15.7 or later, contains the ability to profile OpenACC code with *GPROF*, which allows a breakdown of performance of an OpenACC instrumented application into its kernel components. Such a breakdown can be tailored to give details on the number of calls to, and the min, max and average and total time spent in the kernel. As demonstrated in Chapter 5, Cray's analysis tool suite, Perftools contains functionality to measure the performance of OpenACC instrumented code giving profiles and information analogous to *GPROF*. For the CAPS compiler, the environment variable HMPPRT_LOG_LEVEL is available which gives overall execution time on an accelerated region basis; additionally the CUDA_PROFILE environment variable will give execution time on a kernel by kernel basis.

The standard CUDA toolkit comes with a visual profiler (*nvvp*) plus the command line driven *nprof* tool which is analogous to *GPROF* for OpenACC. As with the debugger component, NVIDIA's Nsight contains a profiler tool, plus additional features such as a replay mode to gain finer profiling information, and the ability to visualise the concurrency of kernel execution on the device, which provides a visual insight into the utilisation of a the device.

The user/developer communities also vary for each standard. Since 2013, the International Workshop on OpenCL (IWOCL) [38] has been held annually, rotating between North America and Europe; a non-profit community led workshop it consists of a technical program with peer reviewed research papers. Supported by NVIDIA, the GPU Technology Conference (GTC) series encompasses all related NVIDIA GPU activities, with a strong coverage of developments in CUDA and OpenACC. Held annually in San Jose California since 2009, with a European version beginning in 2016. In addition to GTC, OpenACC Hackathons are held annually, allowing teams of developers to port their applications to accelerated devices using OpenACC under the supervision of industry and academic experts. These have been held in both North America: Oak Ridge National Laboratory (ORNL), National Centre for Supercomputing Applications (NCSA), University of Delaware (UDel) and Europe: Swiss National Computing Centre (CSCS) and , Forschungszentrum Julich (Research Centre Julich).

6.6 Chapter Summary

This chapter took the performant OpenACC version version of the CloverLeaf mini-app, and performed a direct comparison with two alternative programming methodologies (OpenCL and CUDA) on a GPU accelerated architecture (a Cray XK6).

A quantitative and qualitative comparison was carried out assessing not just raw performance, but portability, ease of maintenance, and respective supporting environments.

In all cases the key to improving the performance of the code on the GPU architecture was to maximise data parallelism within each of the main computational kernels, restructuring the loops to remove data dependencies between iterations. Whilst time consuming, this activity was necessary regardless of the programming model employed and was therefore constant across the three implementations. Feedback from the Cray compiler (CCE) proved to be crucial in understanding the partitioning of the threaded code on the accelerator. Use of the generated listing files and runtime debugging options to capture which data items were actually being transferred was also vital.

Producing functionally equivalent CUDA and OpenCL versions of CloverLeaf required considerably more programmer effort compared to the OpenACC version, both having an order of magnitude increase in the *"words of code"* over that of OpenACC, required to produce compliant implementations. This revealed OpenACC superior to both OpenCL and CUDA, requiring the addition of only 184 OpenACC pragmas (1,510 words of code), compared to OpenCL and CUDA versions of the code requiring an additional 17,930 and 13,085 words of code respectively. However, of the 17,930 words required by OpenCL, 3,958 can be attributed to the static class created to manage OpenCL objects. This static class could be reused in other similar applications with little modification, meaning the total amount of OpenCL code unique to CloverLeaf is 13,972 words. They also required the computational kernels to be in a C-style code which required additional effort for a Fortran based application, while OpenACC could be applied direct to the original Fortran.

In terms of performance the findings shown in Figure 6.2 show around a 10% improvement in performance for OpenCL over that of the equivalent OpenACC implementation and around 20% for CUDA.

Comparative levels of infrastructure exist for CUDA and OpenACC, both having significant industry (NVIDIA) and National Laboratories (ORNL) support. Plus there is the availability of debugging / profiling tools available on agnostic hardware platforms for both of these standards. OpenCL is more community led and is restricted in its development tool support.

Although there is new adoption from NVIDIA for Kepler and AMD have announced v3.0 of their SDK, other vendors are removing support; Intel have indicated than OpenCL will not be supported on the 2nd generation Xeon Phi, the Knights Landing.

The findings indicate that OpenACC is an extremely attractive and viable programming model for accelerator devices going forward. It offers an acceptable level of performance when traded off against a significant gain in programmer productivity compared to both CUDA and OpenCL. It also has a relatively active user community and acceptance from major tool providers which see value in supporting the standard in their compiler, profiler, and debugging offerings.

The future plans for OpenACC [157] look positive, with indications of growing tool support, adoption from academia with fee compiler access and plans for features based on community requests. The question around whether OpenACC will be made redundant should its key functionalities become implemented in OpenMP is often muted. If this were the case then the efforts involved in developing a fully performant OpenACC application would not have been in vain, as the real difficulties, as demonstrated in Chapter 7, lie in understanding the data parallelism inherent in the code. Thus replacing OpenACC directives with OpenMP equivalents would be a relatively easy exercise.

However, there are limitations of this chapter's study, in that the outcomes are restricted to the use of one compiler: CCE, using one possible construct:

"parallel", and one architecture: NVIDIA GPU (Cray XK6). These are addressed in Chapter 7 which extends this work to include the OpenACC "kernel" construct in addition to its "parallel" construct. It also compares compiler portability via PGI, and extends the hardware architectures that execute the resulting OpenACC code to more that just a GPU system. The latter of which also allows comparison between not just OpenCL and CUDA, but also to the most performant programming methodology available on each of the hardware platforms.

CHAPTER 7
Extending the Study to a Range of Emerging Architectures

Chapter 5 introduced the concept of mini-applications and demonstrated the development of a performant OpenACC mini-app deployed on a GPU based system. Subsequently Chapter 6 assessed the alternative programming methodologies on the GPU architecture, namely CUDA and OpenCL, by comparing metrics such as development, performance and supporting infrastructure with OpenACC.

Chapter 7 extends analysis to examine the use of the mini-app for exploring a range of emerging architectures using OpenACC as the baseline to compare and contrast with the alternative methodologies available.

As detailed in Section 3.2.2, OpenACC is a directive-based programming model designed to allow easy access to emerging advanced architecture systems for existing production-class codes based on Fortran, C and C++. It also provides an approach to utilising contemporary technologies without the need to learn complex vendor specific languages, or understand the hardware at the deepest level. Portability and performance are the key features of this programming model, which are essential to productivity for real scientific applications.

OpenACC support is provided by a number of vendors and is defined by an open standard. However, the standard is relatively new, and the implementations are relatively immature. This chapter experimentally evaluates the currently available compilers by assessing two approaches to the OpenACC programming model: the `parallel` and `kernels` constructs. The implementation of both of these constructs is compared, for each vendor implementation which shows performance differences of up to 84%. Additionally, performance differences of up to 13% between the best vendor implementations were observed. OpenACC features which appear to cause performance issues in certain compliers are identified and linked to differing default vector length clauses between vendors. These studies are carried out over a range of hardware including NVIDIA GPU, AMD APU, Intel Xeon and Intel Xeon Phi based architectures. Finally, OpenACC performance, and productivity, is compared against the alternative native programming approaches on each targeted platform, including CUDA, OpenCL, OpenMP 4.0 and Intel Offload, in addition to MPI and OpenMP.

In this chapter both a portability and performance study of OpenACC, using the hydrodynamic mini-application (mini-app) CloverLeaf, is presented. To study portability an evaluation of the relative performance across a wide range of supported architectural platforms for each OpenACC compliant compiler is carried out. For the performance study, each compliant compilers implementation of the OpenACC `parallel` and `kernels` constructs are compared and contrasted. Each vendors highest performing construct is in turn compared against the highest performing construct from other vendors. Performance differences are investigated by analysing the application at the kernel level.

Performance comparisons are presented and discussed, comparing the OpenACC implementation against alternative programming methodologies for those particular architectures. This is demonstrated on a kernel by kernel basis where appropriate.

This work differs from other studies in open literature in that it provides a comprehensive and objective evaluation of the current implementations of commercially available OpenACC compilers, using both OpenACC's `parallel` and `kernels` constructs, these evaluations are carried out on a range of diverse and competing hardware architectures. Additionally, the OpenACC implementations available are compared against the best established native programming alternatives. No other studies have compared all such variations.

In addition performance portability is discussed; that is whether OpenACC provides a level of abstraction that is essential for enabling existing large code bases to exploit emerging many-core architectures, whilst being sufficiently simple and non-intrusive to be viable in a production-class environment.

The remainder of this chapter is organised as follows; Section 7.1 discusses related published work in this field. Section 7.2 gives a brief overview of the CloverLeaf test utilised. Section 7.3 gives further details on the various commercially available OpenACC implementations available and the differences in the two compute OpenACC constructs, namely `kernels` and `parallel`. Section 7.4 details the architectural specifics of the hardware platforms used in the study. The results of the study are then presented in Section 7.5, and finally, Section 7.6 concludes the chapter and outlines future research.

This work would not have been possible without the considerable assistance from each of the three compiler vendors: Cray, PGI and CAPS. In particular John Levesque and Alistair Hart from Cray Inc; Doug Miles, Craig Toepfer and Michael Wolfe of The Portland Group (PGI) and Romain Dolbeau of CAPS Enterprise.

This work was previously presented at WACCPD '14, the first Workshop on Accelerator Programming using Directives in conjunction with SC14: The International Conference for High Performance Computing, Networking, Storage, and Analysis [102], where it was awarded the best workshop paper.

7.1 Related Work

Chapter 6 describes a first comparison between OpenACC, OpenCL and CUDA for the CloverLeaf mini application in terms of its performance over multiple GPUs. Productivity is also considered by analysing several development metrics for the three programming models. However, the OpenACC comparison is only made under a single vendor implementation: Cray's CCE, and a single hardware architecture: a Cray XK6 with NVIDIA X2090 "Fermi" GPUs.

This was extended in [137] to investigate extreme scale across four generations of Cray platforms, showing the utility of hybrid MPI with OpenMP, CUDA, OpenCL and OpenACC under both PGI and Cray compilers.

An increasing number of application code developers are utilising the OpenACC directive approach to allow established industrial codes to take advantage of accelerator enabled architectures. Levesque et al. demonstrate the approach employed with S3D, a current MPI application, and expose greater levels of parallelism using OpenACC as the vehicle to exploit the GPUs on Oak Ridge National Laboratory's (ORNL) Titan supercomputer [132]. Whilst illustrative of OpenACC's capabilities they fail to provide a comparative performance analysis of alternative OpenACC implementations or alternative approaches to acceleration.

Baker et al. [54] look at a hybrid approach of OpenSHMEM and OpenACC for the BT-MZ benchmark application. The focus of the research is on hybridising the application rather than an assessment of the OpenACC implementation used (a beta version of CAPS 3.3), and results are focused on a single architecture, namely ORNL's Titan.

There are a small number of case studies presenting direct comparisons of OpenACC against alternative programming models. Reyes et al. present a direct comparison between hiCUDA [191], PGI's Accelerator model and OpenACC using their own novel implementation of OpenACC: accULL [178]. Again, this focuses on a single type of accelerator, and a single instance of an architecture: the NVIDIA Tesla 2050.

Comparison of OpenCL against OpenACC can be found in [195] by Wienke et al. The paper compares OpenACC against PGI Accelerator and OpenCL for two real-world applications, demonstrating OpenACC can achieve 80% of the performance of a best effort OpenCL for moderately complex kernels, dropping to around 40% for more complex examples. The study only uses Cray's CCE compiler and OpenACC's `parallel` construct. Additionally, it is also limited to a single hardware architecture, an NVIDIA Tesla C2050 GPU.

7.2 Test Cases

This study uses the same test case described in Chapter 5, Section 5.1.2, and utilises the large 3840^2-cell problem introduced in Chapter 5, Section 5.3.14 and a smaller 960^2-cell problem introduced in Chapter 5, Section 5.3.5. The test cases compute 2,955 and 87 iterations respectively, which for their respective sizes, gives sufficient compute time to ensure reliable timing measurements.

Although CloverLeaf is capable of multiple accelerated node runs, the nature of this study is focused on a single accelerator's performance.

7.3 Implementations

The CUDA and OpenCL implementations of the code, as described in Sections 6.3.3 and 6.3.2, were used in this study.

OpenACC provides two constructs `parallel` and `kernels` to launch, or execute, accelerated regions. The main differences are down to how they map the parallelism in the region to be accelerated to the underlying hardware [197]. In the case of the `parallel` construct this is explicit, requiring the programmer to additionally highlight loops within the region, while in the case of the `kernels` construct parallelisation is carried out implicitly.

In the case of single loops, as in CloverLeaf's kernels, the two constructs are virtually interchangeable, although the compiler is able to automatically generate vector code for the `parallel` construct variant. With the simple restructuring required for the PGI and CAPS implementations, this produced a single source code capable of compilation with both the `parallel` and `kernels` construct implementations with all three compiler vendors. This enabled the additional comparison of how the two alternative constructs were implemented by the different compiler vendors.

Additionally a version of the code using Intel's Language Extensions for Offload (LEO) [152], [114], developed by Victor Gamayunov and Stephen Blair-Chappell of Intel [91], and an OpenMP variant (also by Gamayunov), using the accelerator constructs as available in version 4.0 of the OpenMP standard [115], enabled a further comparison on those architectures which supported this standard.

7.4 Targeted Architectures

Five test architectures were deployed in this study: *Chilean Pine* a Cray XK6 with NVIDIA "Fermi" X2090 GPUs; *Swan* a Cray XK7 with NVIDIA Kepler K20X GPUs; *Shannon* a cluster with both NVIDIA Kepler K20X and K40

GPUs; *Teller* an AMD APU cluster and *PillowB* an Intel based cluster with 1st generation MIC Intel Xeon Phi (KNC) coprocessors. Full architectural details can be found in Appendix A, but for convenience the targeted architectures and their system software stacks are summarised in Table 7.1.

7.5 Results

7.5.1 Experimental Studies

With the exception of the *Teller* AMD APU cluster (where memory constraints prevented execution of the larger test case), platform results are presented for both the 960^2 cell and $3{,}840^2$ cell test cases.

For each OpenACC implementation available on the targeted platforms, results are presented for both the OpenACC `parallel` and `kernels` construct. This allows a comparison of not only each vendor's best OpenACC implementation, but also any differences between a vendor's best and worst implementation. Additionally it highlights those implementations where collaborative development efforts with vendors should be focused, or which implementations are to be avoided.

For each alternative programming model available on the target architecture, comparable performance figures are presented. This gives an OpenACC performance baseline with which to compare alternative approaches. Additionally comparative performance of hand-coded CUDA and OpenCL implementations on the XK6, XK7, K20x and K40, and the AMD APU are presented.

Finally, performance at the compute kernel level is examined. The aim is to identify under-performing kernels and determine if certain OpenACC features have performance issues under a particular vendor's implementation. The aim is to identify if particular kernels exhibit good or poor performance rather than the application as a whole. Subsequent analysis of these kernels will likely identify particular OpenACC features that require vendor attention within their respective implementations.

For all cases in this chapter, the hardware was run in dedicated mode; that is the author has exclusive access to the hardware and its resources. This is reflected in the resultant execution times showing insignificant differences in the multiple runs measured (in the region of 10 to 100 depending on test case and hardware device).

7.5.2 Analysis

Figures 7.1, through 7.7 show the comparative performance of each OpenACC implementation, the alternative programming models available and the hand-

Platform	**Chilean Pine (XK6)**	**Swan (XK7)**	**Shannon**	**Pillow B**	**Pillow B**	**Teller**
Architecture	X2090	K20x	K20x/K40	E5-2450	5110P	A10/7660D
OpenACC	CCE 8.1.7 PGI 13.7 CAPS 3.3.2	CCE 8.3.0 PGI 13.10	PGI 13.9.0 CAPS 3.3.4	CAPS 3.3.2	CAPS 3.3.2	CAPS 3.3.3
CUDA	5.0	5.5	6.0	N/A	N/A	N/A
OpenCL	AMD APP SDK v2.7	CUDA 5.5	CUDA 5.0.0	Intel OpenCL SDK 1.2.3.0	Intel OpenCL SDK 1.2.3.0	AMD APP SDK 2.8.0
Offload	N/A	N/A	N/A	N/A	Intel 13.1.3	N/A
OpenMP 4.0	N/A	N/A	N/A	N/A	Intel 13.1.3	N/A
Hybrid (MPI/OMP)	N/A	N/A	N/A	Intel MPI 4.1.1	Intel MPI 4.1.1	N/A

Table 7.1: System Software Stack

coded CUDA and OpenCL implementations on the XK6, XK7, K20x & K40, and the AMD APU respectively; these results are summarised in Table 7.2.

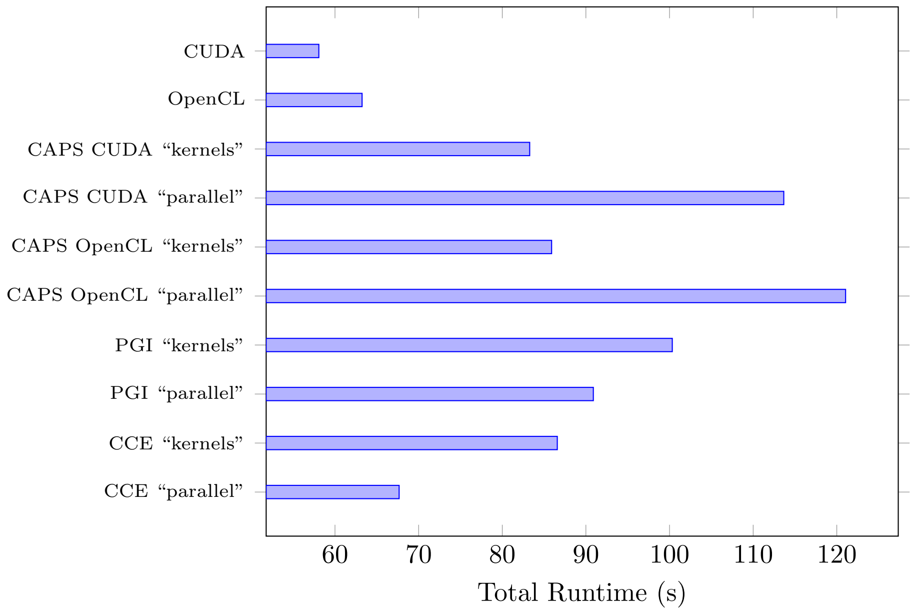


Figure 7.1: 960^2 cells, 2955 Timesteps: XK6 Runtimes (s)

With the exception of the XK6, which uses older compiler revisions from each vendor, comparing each vendor's best OpenACC implementations (be it `parallel` or `kernels` constructs, with either the CUDA or OpenCL backends) performance differences are within 13%. However, choosing the least optimal OpenACC implementation can result in significant differences up to 84% when using the `parallel` construct via the CAPS compiler on a K40.

The raw CUDA and OpenCL versions do out perform any OpenACC implementation: by 15% to 20% for CUDA and by 10% to 20% for OpenCL. The exception to this is raw OpenCL on the AMD APU, where differences are only around 1%. This is attributed to the raw OpenCL not being optimised for either the APU's A10 CPU or HD-7600D GPU. When considered in the overall context of productivity, portability, maintainability including the number of words of code (WOC) metric, as devised in Chapter 6 and considering whether computational kernels needed to be rewritten, 10% to 20% lower performance from an OpenACC implementation is an acceptable trade off.

Figures 7.8 and 7.9 depict the same comparisons for the Intel Xeon E5-2450 and the Intel Xeon Phi 5110P, for the 960^2 cell and $3{,}840^2$ cell test cases respectively. For both platforms a traditional MPI/OpenMP hybrid versions of the code (running natively on the card in the case of the Intel Xeon Phi 5110P) is used as a baseline on this architecture. Additionally, in the case of

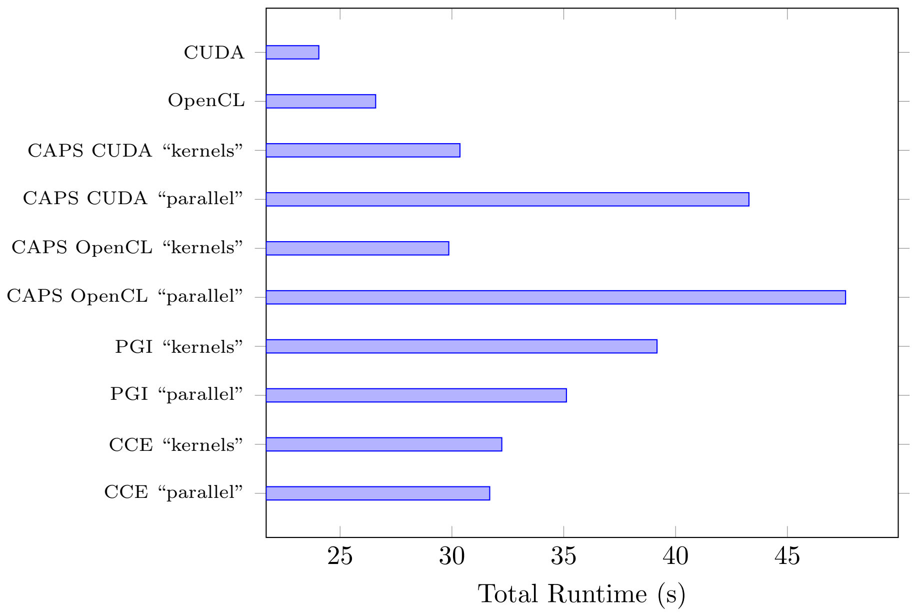


Figure 7.2: 3840^2 cells, 87 Timesteps: XK6 Runtimes (s)

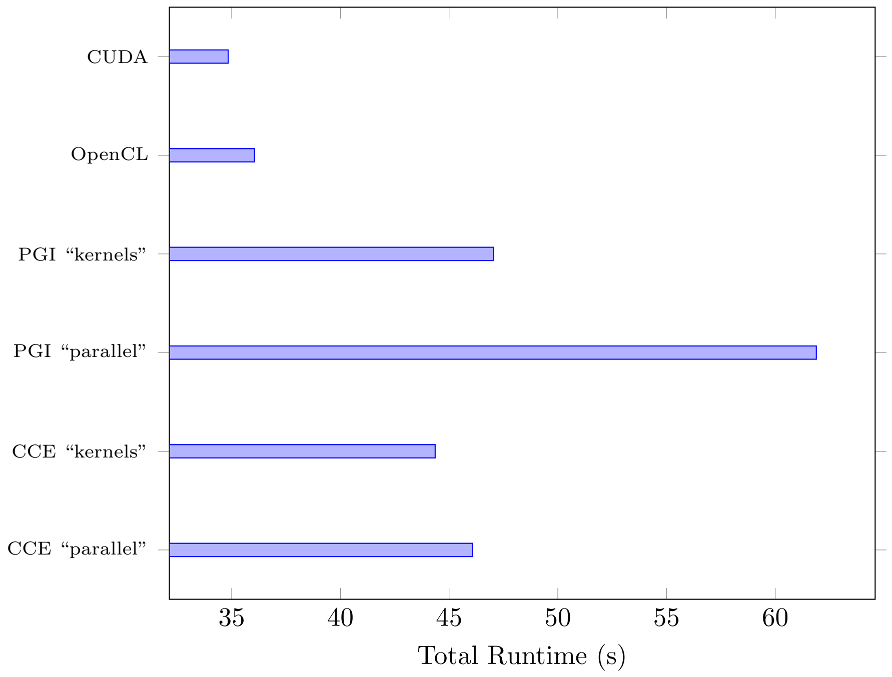


Figure 7.3: 960^2 cells, 2955 Timesteps: XK7 Runtimes (s)

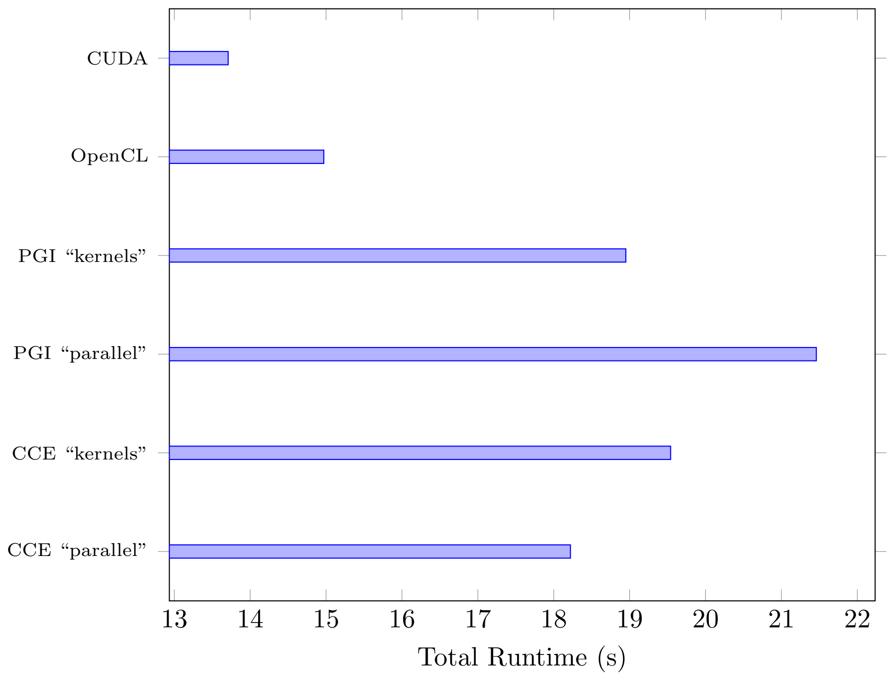


Figure 7.4: 3840^2 cells, 87 Timesteps: XK7 Runtimes (s)

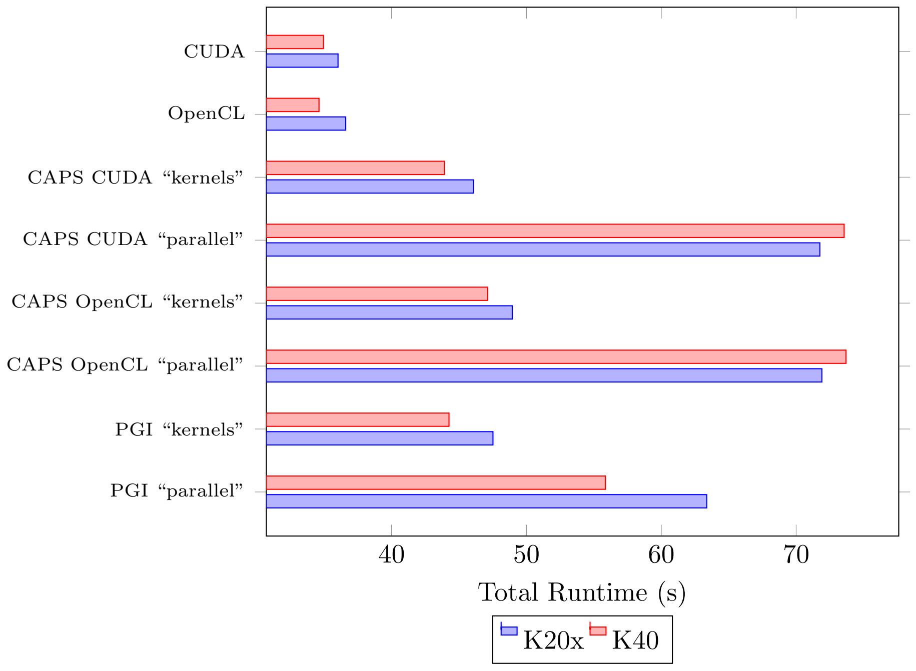


Figure 7.5: 960^2 cells, 2955 Timesteps: K20x & K40 Runtimes (s)

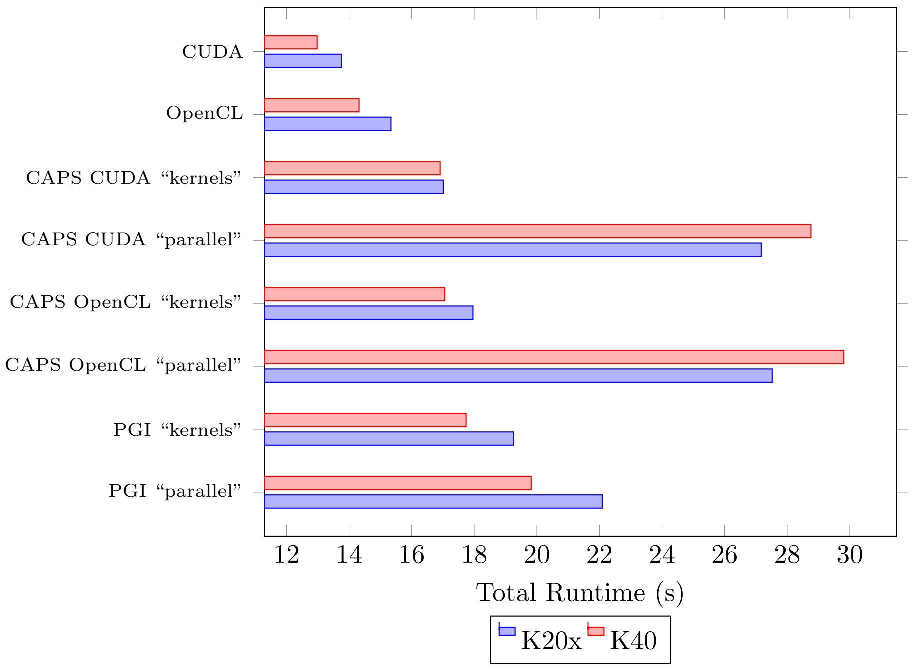


Figure 7.6: 3840^2 cells, 87 Timesteps: K20x & K40 Runtimes (s)

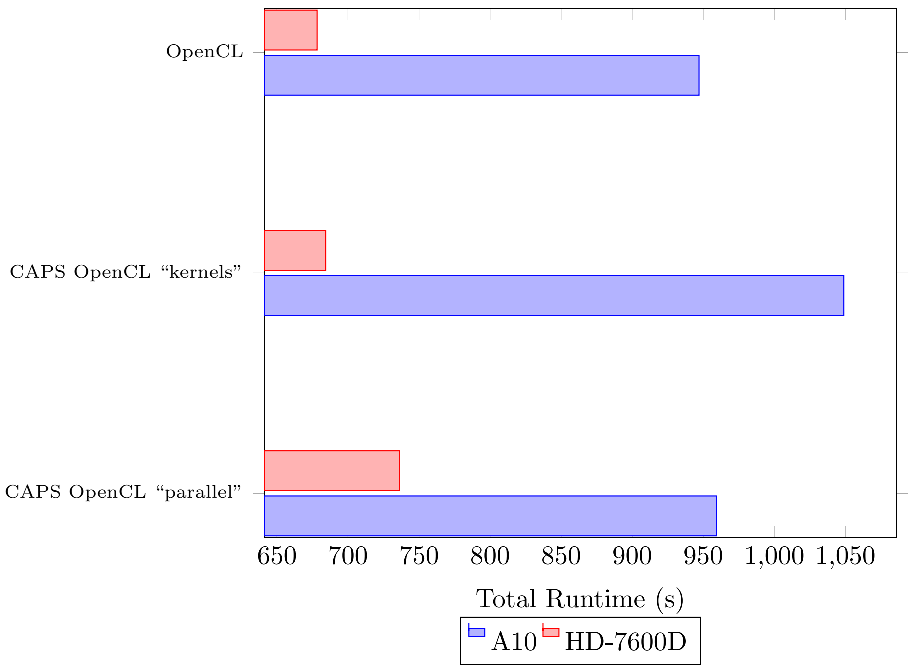


Figure 7.7: 960^2 cells, 2955 Timesteps: A10 & HD-7600D Runtimes (s)

Platform / Problem	**Best OpenACC**	**Slowest Best Alt OpenACC % diff**	**Slowest OpenACC % diff**	**CUDA % Diff**	**OpenCL % Diff**
XK6 / 960^2	CCE "parallel"	PGI "parallel" -34.31	CAPS OpenCL "parallel" -78.88	14.19	11.41
XK6 / 3840^2	CAPS OpenCL "kernels"	PGI "parallel" -17.62	CAPS OpenCL "parallel" -56.79	19.46	10.95
XK7(K20x) / 960^2	CCE "kernels"	PGI "kernels" -6.04	PGI "parallel" -34.34	21.46	18.73
XK7(K20x) / 3840^2	CCE "parallel"	PGI "kernels" -4.00	PGI "parallel" -17.78	24.75	17.84
K20x / 960^2	CAPS CUDA "kernels"	PGI "kernels" -3.13	CAPS OpenCL "parallel" -56.09	21.79	20.56
K20x / 3840^2	CAPS CUDA "kernels"	PGI "kernels" -13.17	CAPS OpenCL "parallel" -61.79	19.11	9.82
K40 / 960^2	CAPS CUDA "kernels"	PGI "kernels" -0.80	CAPS CUDA "parallel" -54.40	20.41	21.16
K40 / 3840^2	CAPS CUDA "kernels"	PGI "kernels" -9.57	CAPS OpenCL "parallel" -84.13	19.83	11.55
A10 / 960^2	CAPS OpenCL "parallel"	CAPS OpenCL "kernels" -9.36	CAPS OpenCL "kernels" -9.36	N/A	1.27
HD-7600D / 960^2	CAPS OpenCL "kernels"	CAPS OpenCL "parallel" -7.60	CAPS OpenCL "parallel" -7.60	N/A	0.89

Table 7.2: OpenACC Performance Comparisons

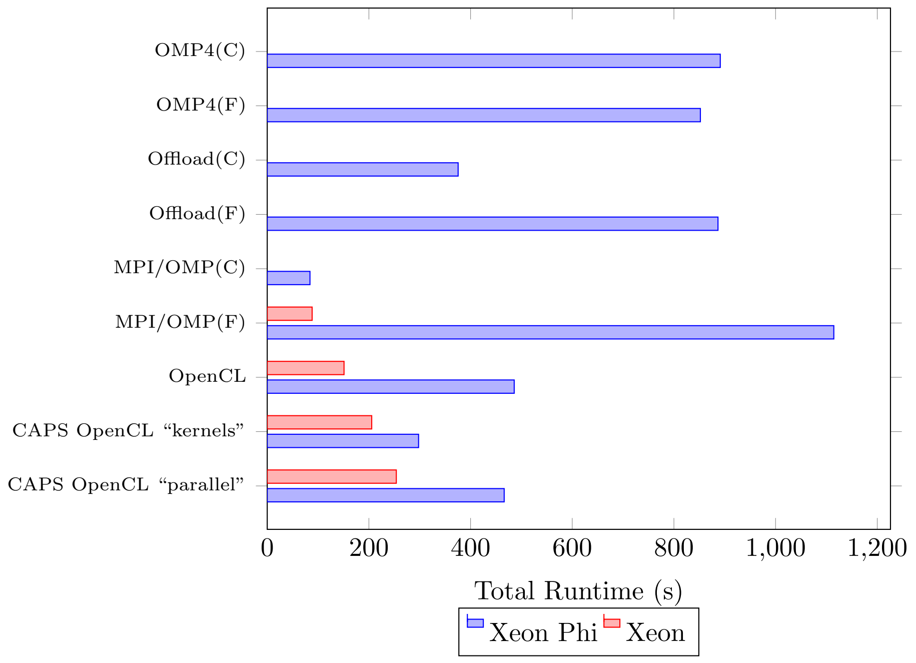


Figure 7.8: 960^2 cells, 2,955 Timesteps:
Intel Xeon E5-2450 & Intel Xeon Phi 5110P Runtimes (s)

the Intel Xeon Phi 5110P, offload models of OpenMP 4.0 and Intel's LEO are also presented.

Although the study focuses on OpenACC directives applied to Fortran source code, the perceived poor performance observed on the Intel Xeon Phi 5110P of the Fortran based MPI/OpenMP hybrid led to the development and comparison of a C-based implementation. This showed significant differences between C and Fortran which have been attributed to the fact that the Intel compiler was not vectorising the Fortran source code when compiled for the Intel Xeon Phi 5110P, despite accomplishing this when compiling for the Intel Xeon E5-2450. This is also true for Intel's LEO model when using Fortran, hence in the summary of the results, the C implementations on the Intel Xeon Phi 5110P are taken as reference points rather than the original Fortran implementations.[1]

Table 7.3 shows an overview of the results. The only OpenACC implementation available on the Intel chipsets is via CAPS using its OpenCL backend. Hence the only OpenACC-to-OpenACC comparison that can be made is between the `parallel` and `kernels` constructs. This shows a significant performance difference for both test cases, on both architectures. On the Intel Xeon E5-2450 the CAPS OpenCL `kernels` version is 23% more performant for the 960^2 cell problem which rises to 30% for the $3{,}840^2$ cell case. On the Intel

[1]This issue has been reported as resolved in Intel 15.0

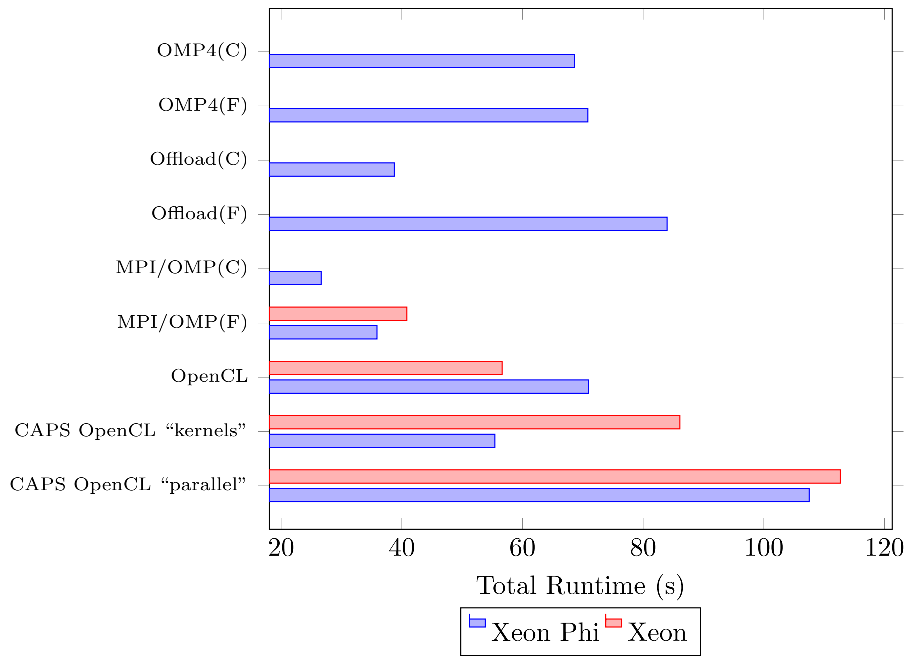


Figure 7.9: 3840^2 cells, 87 Timesteps:
Intel Xeon E5-2450 & Intel Xeon 5110P Runtimes (s)

Xeon Phi 5110P the performance differences are 56% and 94% for the 960^2 cell and $3{,}840^2$ cell problems respectively. As previously observed on other platforms the performance of CAPS `parallel` is significantly worse; as this is the only comparison these differences are to be expected.

Comparing the optimal OpenACC implementation against the raw OpenCL code on the Intel chipsets shows a very different pattern between the Intel Xeon E5-2450 and the Intel Xeon Phi 5110P. On the Intel Xeon E5-2450 the raw OpenCL gives a gain of either 26% to 34% depending on problem size, while on the Intel Xeon Phi 5110P the OpenCL gives performance *degradation* of over 60%. In the case of the Intel Xeon Phi 5110P this is attributable to the performance portability of the hand-coded OpenCL, in that it has not been optimised for the Xeon Phi. However, given that the OpenACC is a single source and no code modifications have taken place here either, this points to CAPS producing an efficient Intel Xeon Phi OpenCL backend.

All offload models perform poorly when compared against a hybrid MPI /OpenMP implementation. On the Intel Xeon E5-2450 this is between 52% and 57% more performant, depending on problem size, while the gains on the Intel Xeon Phi 5110P are 52% to 72% when using the C implementation as a reference point.

All of the above analysis considers the code as a whole; that is the overall

Platform / Problem	**Best OpenACC**	**Slowest Alt OpenACC % diff**	**Best Alt Offload Model % Diff**	**OpenCL % Diff**	**MPI/OpenMP Hybrid % Diff**
Xeon E5-2450 / 960^2	CAPS OpenCL “kernels”	CAPS OpenCL “parallel” -23.53	N/A	26.49	57.02
Xeon E5-2450 / 3840^2	CAPS OpenCL “kernels”	CAPS OpenCL “parallel” -30.87	N/A	34.22	52.56
Phi 5110P / 960^2	CAPS OpenCL “kernels”	CAPS OpenCL “parallel” -56.55	Offload -26.13 (C)	-63.18	71.75 (C)
Phi 5110P / 3840^2	CAPS OpenCL “kernels”	CAPS OpenCL “parallel” -93.97	Offload 30.09 (C)	-27.95	51.94 (C)

Table 7.3: Xeon and Xeon Phi OpenACC Performance Comparisons

execution time, with each of its twelve kernels accelerated. Further insight is gained if the performance of each kernel is analysed in isolation. What is observed is the relative performance of each kernel can vary substantially depending on construct and vendor implementation.

The methods available to obtain a kernel by kernel breakdown are programming paradigm dependent. As described in Section 6.5 of Chapter 6, for the OpenACC implementations GPROF was used to measure the kernel times for the PGI `parallel` and `kernels` constructs, Cray's Perftool utility was used to extract the same information from the CCE compiled kernels, while the CUDA_-PROFILE environment variable provided the information when executed under the CAPS compiler builds. For CUDA, *nprof* which is provided as part of the CUDA Toolkit was utilised to record the kernel breakdowns. Kernel by kernel breakdown was not possible for the OpenCL builds; although AMD's CodeXL has the functionality to achieve this, it is only available on AMD hardware and hence not compatible with the NVIDIA GPUs. Direct hand profiling was implemented for the OpenCL kernels, however the timing routines adversely affected the performance and hence rendered them useless.

Figures 7.12 and 7.13 show the percentage difference of cumulative runtimes of each of the twelve CloverLeaf kernels: *Timestep*, *IdealGas*, *Viscosity*, *PdV*, *Revert*, *Accelerate*, *Fluxes*, *CellAdvection*, *MomAdvection*, *Reset*, *Halo*, and *Summary*. These are recorded on the 960^2 cell problem running on one of Chilean Pine's X2090 GPUs and are normalised against the performance of the Cray CCE `parallel` construct time for that particular kernel. Hence, a positive percentage value indicates a shorter time spent in the kernel, and a negative a longer execution time for that particular compiler / construct combination.

Data is also available for the larger $3{,}840^2$ cell problem, which is presented in Figures 7.14 and 7.15, which show similar behaviour.

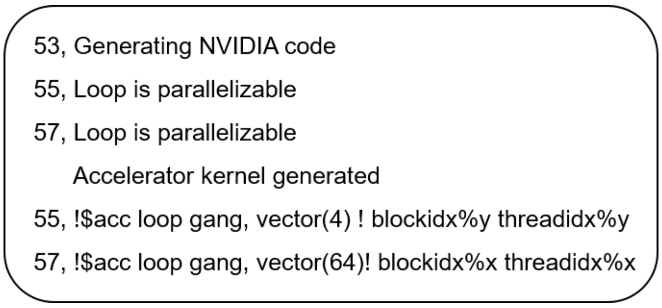
This closer analysis backs up some of the previous observations. In particular CAPS implementation of the `parallel` construct underperforms on all twelve kernels irrespective of whether it is a CUDA or OpenCL backend, Cray's `parallel` implementation is almost always equal to or outperforms its `kernels` construct for all CloverLeaf kernels. However it also reveals that the CAPS `kernels` implementation, which is the best performing OpenACC on all of the non-Cray architectures, makes these gains as it performs well on the following computationally intense kernels: *MomAdvection*, *PdV*, *CellAdvection*, and *Accelerate*, which account for 60% overall execution time, while underperforming on those with a lesser contribution.

PGI `kernels` outperforms all implementations in seven of the twelve kernels on the 960^2 test case, and six on the $3{,}840^2$ cell problem. However, these kernels: *Timestep*, *IdealGas*, *Revert*, *Fluxes*, *CellAdvection*, *Reset*, *Halo* and

Summary account for less than a quarter of execution time. There are a subset of kernels where PGI performs badly: *viscosity*, *PdV*, *Accelerate*, *mom_advec*, which account for over 85% of overall execution time, hence showing PGI's underperformance when comparing the code as a whole. Indeed, on some of these kernels the relative performance is surprisingly low. In particular the viscosity kernel, where PGI's `kernels` construct shows more than a 180% decrease compared to CCE's `parallel` implementation.

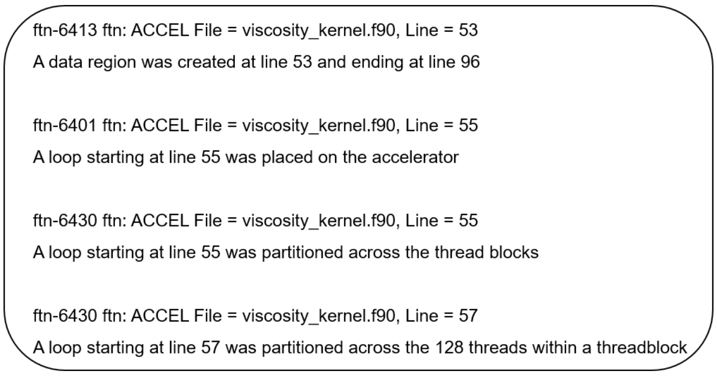
Such a disparity warrants further investigation. Both PGI and CCE provide a means to provide information regarding what the compiler has generated for the OpenACC accelerated regions of code. By use of the `-Minfo=accel` compile time flag with PGI and the `-ra` flag with CCE, information is sent to `stderr`, and a source listing file respectively.

Figures 7.10 and 7.11 present output snippets from both PGI and CCE compilers respectively for CloverLeaf's viscosity kernel.



```
53, Generating NVIDIA code
55, Loop is parallelizable
57, Loop is parallelizable
    Accelerator kernel generated
55, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
57, !$acc loop gang, vector(64)! blockidx%x threadidx%x
```

Figure 7.10: PGI OpenACC Source Listing



```
ftn-6413 ftn: ACCEL File = viscosity_kernel.f90, Line = 53
A data region was created at line 53 and ending at line 96

ftn-6401 ftn: ACCEL File = viscosity_kernel.f90, Line = 55
A loop starting at line 55 was placed on the accelerator

ftn-6430 ftn: ACCEL File = viscosity_kernel.f90, Line = 55
A loop starting at line 55 was partitioned across the thread blocks

ftn-6430 ftn: ACCEL File = viscosity_kernel.f90, Line = 57
A loop starting at line 57 was partitioned across the 128 threads within a threadblock
```

Figure 7.11: Cray OpenACC Source Listing

It indicates a difference in the maximum vector length which the loop iterations are executed over. The `vector` clause is one such optional clause that is allowable on the OpenACC `kernels` construct, however in the case of the CloverLeaf kernels this clause is absent, hence the values reported by the compilers are default /assumed values. In the case of the viscosity kernel, the compiler information indicated a difference in the assignment of vector length. Further investigation across all of the twelve kernels gives a similar account for the six kernels for which the PGI compiler has disproportionate performance. Explicitly adding the `vector` clause to the `kernels` construct in these cases delivers a match between the compiler outputs, although this change only delivers improvement in kernel execution time.

7.6 Chapter Summary

This chapter set out to evaluate the performance and portability of OpenACC. This was achieved by comparing OpenACC's `parallel` and `kernels` constructs in each vendor's implementation which enabled performance comparison between vendors as well as how OpenACC compares with alternative programming methodologies whilst balancing performance against programmer productivity.

By evaluating both the `parallel` and `kernels` constructs the results indicate that each vendor has primarily focused efforts on one of the two constructs. This is most apparent in CAPS implementation of the `parallel` construct, with an 84% difference in runtime over the `kernels` implementation.

When comparing each vendor's quickest construct, differences in execution time are within 13% when the latest compiler versions are invoked. This indicates, as long as the application developer is aware of a vendor's relative performance on the `parallel` and `kernels` construct, the choice of a vendor is not particularly crucial.

Each vendor does however have their relative merits: the CAPS's implementation of the `parallel` construct is deficient, and their focus has concentrated on the `kernels` construct. However, the CAPS compiler is the most widely supported on the diverse range of architectures tested, and in some cases is the only available option to compile and execute OpenACC applications. CCE is the best performing compiler, in all but one case, on all architectures where it is available. However, it is only available on Cray systems, limiting its accessibility. Looking at overall runtimes PGI trails the other vendor's compilers. However, when broken down into the individual kernels, PGI's `kernels` construct outperforms the rest on more occasions than any other implementation. This is counterbalanced by its significant poor performance for one particular kernel (*viscosity*) in CloverLeaf which is 175% and 225%, for the 960^2 cell problem and the $3{,}840^2$ cell problem respectively, slower than the best implementation which severely impacts the code when it is considered as a whole.

At the time of writing, the "deep copy" required to support Fortran 90 derived types was not supported in either the PGI or CAPS compilers. Hence some minor code reconstruction was required to remove derived types and explicitly pass data through the relevant Fortran routines. This also required relocating the directives to locations higher up in the call tree to reflect this reconstruction. Also under the PGI compiler, compile time for the `parallel` construct variant is significantly longer than that for the `kernels` construct.

The different basis of each vendor implementation manifested itself most notably in the varying levels of support for OpenACC's two compute constructs `kernels` and `parallel`. Cray targeted the `parallel` construct with a direct mapping from its OpenMP `parallel` extension, while PGI and CAPS targeted the `kernels` construct with a direct mapping from the PGI Accelerator `region` construct.

It is worthy of note that the two constructs are not mutually exclusive, and can be mixed and matched depending on individual accelerator region performance, and optimal constructs could be used depending on the compiler and architecture as part of an optimisation strategy.

Although a caveat exists that automatic compiler analysis and compilation of the `parallel` construct is only applicable for non-nested loops.

An artefact of this study highlighted issues with the Intel Fortran compiler on the Intel Xeon Phi, which required the use of a C-based implementation to give a more realistic comparison. Also, all Xeon Phi offload models perform poorly when compared to hybrid MPI/OpenMP.

When compared to alternative offload models on the Intel Xeon Phi and Intel Xeon architectures, OpenACC outperforms both Intel's Heterogeneous LEO model and their current OpenMP 4.0 implementations in all but the large $3{,}840^2$ cell test case when using the former. Although in the case of these two architectures, a hybrid MPI/OpenMP implementation outperforms all offload based programming models. On the three NVIDIA GPU based architectures a native CUDA implementation outperforms the best OpenACC by 15% to 20%. OpenCL is the only alternative programming model to OpenACC that spans all the target architectures in this study. Comparing performance, OpenCL outperforms OpenACC by 10% to 20% with the exception of the AMD APU and the Intel Xeon Phi where performance is on par with the former and it delivers a significant under-performance of over 60% against OpenACC on the latter.

Empirically, these performances differences are more than acceptable when offset against programmer productivity measured in the number of words of code. While performance is important, for a new programming standard like OpenACC, the convergence of the standard on a range of compilers is the most important factor for portability.

CloverLeaf now has a portable single source for both the `parallel` and the `kernels` construct versions, that works without modification on all compilers and across a diverse range of architectures. Indeed, PGI have adopted the `kernels` OpenACC version of CloverLeaf as part of their compiler regression testing suit, and it has been accepted as one of the OpenACC applications [43] that make up the SPEC ACCEL$^{\text{TM}}$benchmark [44]. This is major step forward and would indicate that OpenACC provides a good level of abstraction.

CAPS Enterprise announced that as of June 27, 2014 they are to cease trading. This loss of one of the three vendors, and hence future CAPS OpenACC compiler support, reduces the options for OpenACC developers. However, OpenACC v1.0 support has been implemented into a branch of GCC Fortran [168], which as an open source compiler, bodes well for the standard.

Irrespective of OpenACC's future, the development effort expended in producing a portable OpenACC implementation of an application is easily translated into alternative pragma based offload models such as OpenMP 4.0's accelerator offload support constructs, should this become the de facto standard.

OpenACC has matured significantly in both its portability and performance. The ability to use a higher level language like C, C++, or Fortran on CPUs, attached co-processors, GPUs and APUs is a major step forward in future proofing production-class scientific applications.

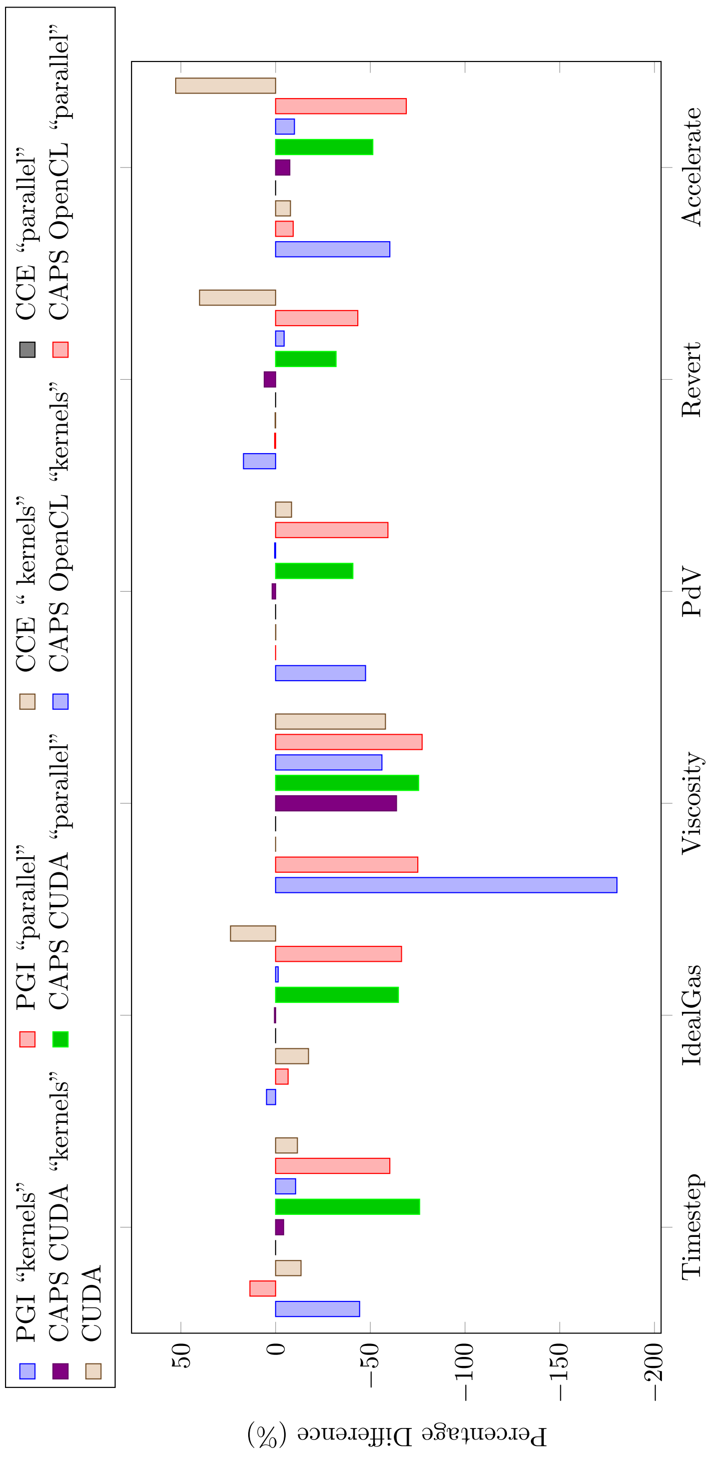


Figure 7.12: 960^2 2,955 TS, Relative Kernel Performance Normalised to CCE `parallel` Construct

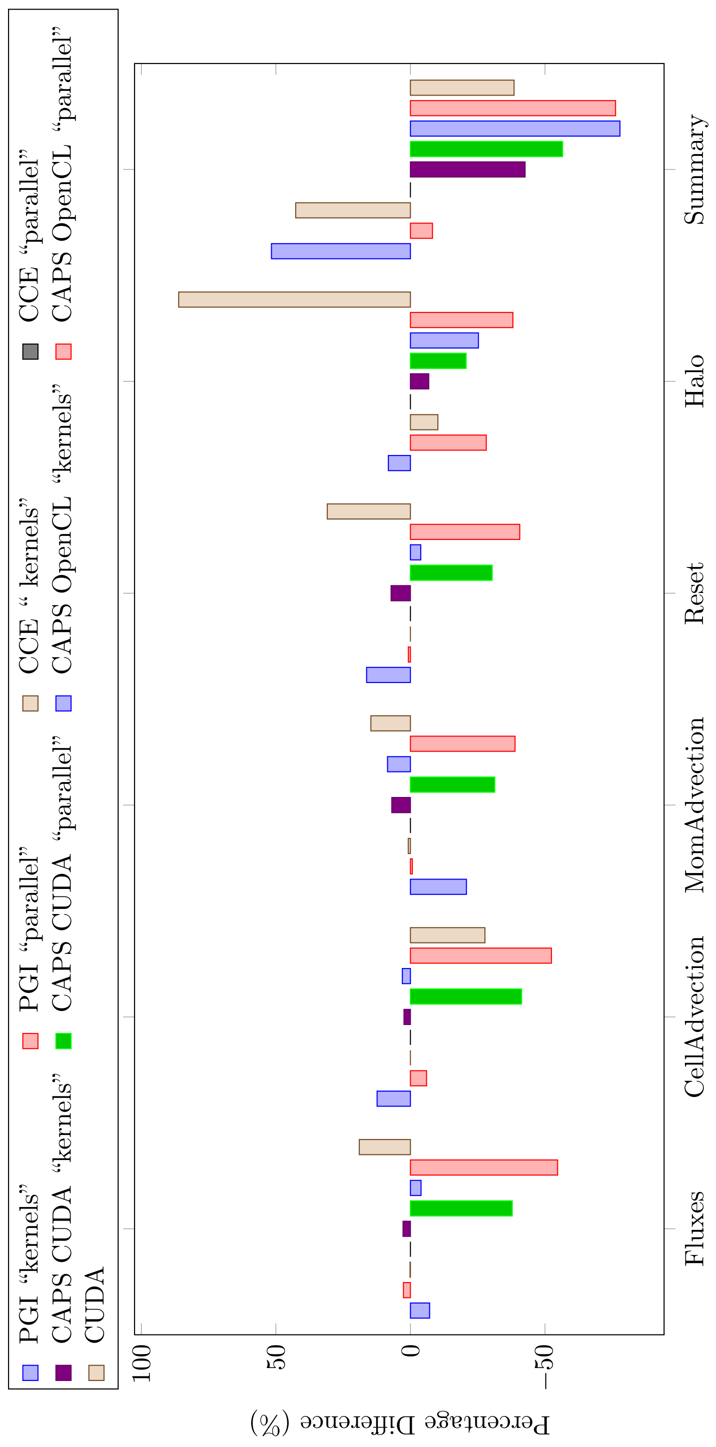


Figure 7.13: 960^2 2,955 TS, Relative Kernel Performance Normalised to CCE "parallel" Construct

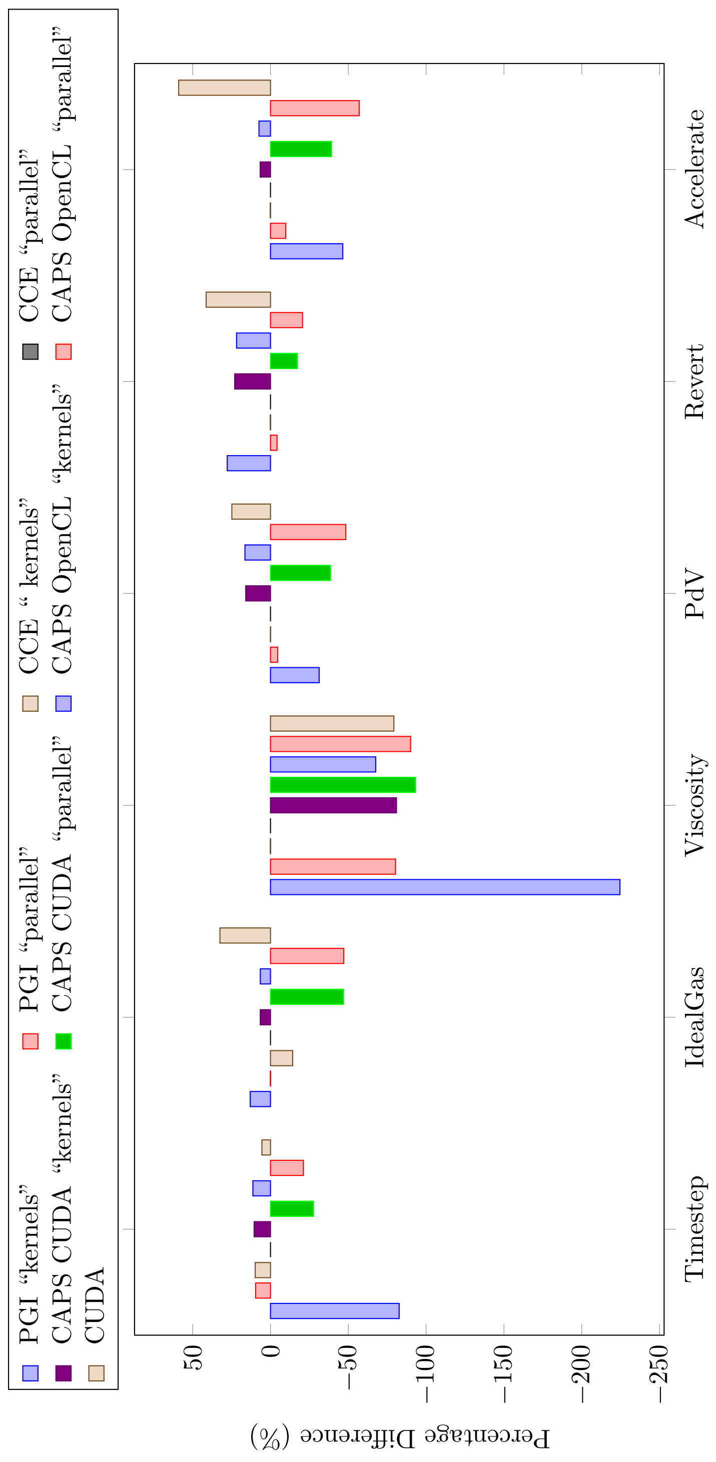


Figure 7.14: $3{,}840^2$ 87 TS, Relative Kernel Performance Normalised to CCE "parallel" Construct

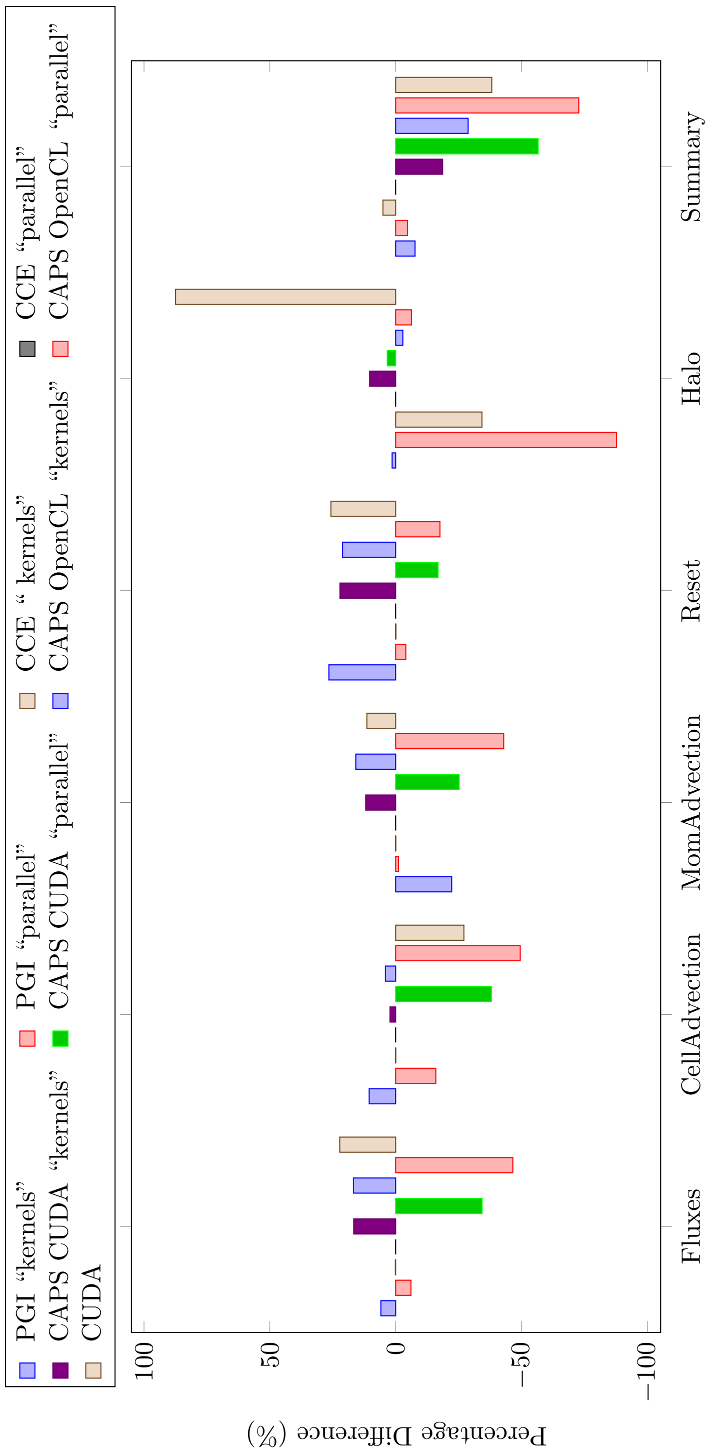


Figure 7.15: $3{,}840^2$ 87 TS, Relative Kernel Performance Normalised to CCE "parallel" Construct

Part III

Theory into Practice

CHAPTER 8
Conclusions and Further Developments

This chapter concludes this thesis with the lessons learnt through this research and how these have been applied in practice within an industrial setting; both within the author's own institution as well as with similar independent approaches elsewhere across the HPC industry and academia.

It shows the independent adoption of mini-apps as the tool of choice not just for exploring emerging HPC architectures and programming languages, but also how they are beginning to play their part in future architectural design.

The numerous collaborative research activities that have been spawned as a result of this research work are also highlighted.

An overview of some of the architectures which are at present becoming available to the market are detailed. An emphasis is placed on the changes relative to those utilised throughout this study. Additionally, a hypothesis on how potential longer term systems may look is provided.

Finally, in conclusion, a re-cap of the main contributions which this research has delivered is presented.

8.1 Lessons Learnt

With the constraints on the lack of accessibility to suitably large disruptive systems and their inflexibility to experiment with applications due to commercial or proliferation reasons, an alternative approach is needed to assess emerging HPC architectures.

Chapter 4 investigated the hypothesis that an existing procurement benchmark application could be used to explore emerging architectures and the range of associated programming methodologies. It demonstrated that the time required for substantial code restructuring to explore one programming methodology, on one potential emerging architecture, was not efficient or cost effective and hence a new approach was required.

Chapter 5 introduced the concept of the mini-app: a lightweight, but representative application written with a particular focus on algorithmic solutions. It demonstrated, through a step-by-step approach, how a directive-based version of the mini-app which is able to execute on a GPU accelerated system, could be developed.

Chapter 6 examined how performant the resultant directive-based mini-app was compared to the alternative approaches on the GPU accelerator, and holistically considered the benefits when taking into account relative development time and the supporting infrastructures of alternative methods.

Chapter 7 then expanded the mini-app with an additional OpenACC directive construct and compared both OpenACC construct implementations across a range of emerging architectures. This demonstrated acceptable levels of trade-off between performance and having the portability of a single source code capable of execution on multiple hardware environments. This indicated that a directive-based solution is amenable for a hydrodynamic computational model.

It also demonstrated the usefulness of the mini-app based approach in enabling rapid prototyping and assessment of an algorithm's ability to map onto emerging hardware. As part of this process, a number of key lessons have been learnt that are not limited to hydrodynamic schemes, but generalise across a large number of scientific domains. These key lessons are listed below.

8.1.1 Kernelisation

Those areas of the application doing computation need to be appropriately broken down and isolated. These need to be placed at the lowest level, and dedicated solely to compute. Ensuring that they do not call subroutines or perform additional functions, minimising pointers, excluding the use of derived types and Fortran array syntax generally makes everything as explicit as possible for the compiler. This also structures the application for any architectures requiring the launch of independent chunks of work such as for the GPU.

8.1.2 Data Parallelism and Thread Safety

Achieving good data parallelism is helped by striving for stride-one data accesses, or at least data accesses in a regular pattern which is recognisable by the compiler, enabling optimised gather/scatter operations to be deployed. Even moderately computationally intense kernels need good data parallelism and threading when executed on accelerated devices otherwise they become the bottlenecks for performance. Good data parallel code needs to be thread safe, that is, the actions or order of execution of one thread should not impact that of another. Data parallel regions of kernels which can show different results when executed in threaded mode will need to be restructured, or re-factored, in order to make them thread safe.

Care is also needed when implicit synchronisation activities occur in data parallel models. This can have little impact, and hence go undetected, on familiar hardware, such as a CPU; however, when executed on highly parallel devices, these soon become bottlenecks.

8.1.3 Data Residency

Achieving optimal computation on an attached device is one concern, however for computation to take place it needs data on which to operate. The mechanisms for transferring data to and from an attached device are significantly slower than passing data within a device and slower still relative to the computation itself. Hence, the continued passing of data to and from any hosted device becomes a significant bottleneck, quickly swamping any computational gains that are achieved. Ensuring that as much of the data that is necessary for the compute to take place always resides on the computing device, enables the data transfers to be minimised. An ideal situation would be for all of the data to be fully resident on a device for the duration of all the computation. However, this would only occur if a single device was being utilised. In practice an application needs to span multiple devices, in this case limiting the transfer of data to that of the neighbouring device's boundary data is an optimal minimum.

Although this is in the context of minimising data transfers to an attached device, it is equally applicable to the changes in the memory hierarchy appearing on emerging many-core self hosted devices. That is, to achieve optimal performance, data needs to reside in high bandwidth memory; which is analogous to keeping data resident on an attached processor.

8.1.4 Halo Data Transfer and Local Buffer Packing

In the case of halo exchange, a common approach in distributed algorithms, minimising the data which needs to be transferred between any device is crucial. Whilst this is good practice for a distributed algorithm which executes on a MPP (Massively Parallel Processing) system, even small excesses can have significant impacts on performance on systems utilising attached accelerator devices. Small accesses in a MPP distributed mode may still keep MPI messages within the same MPI protocol boundary, while due to the overhead of the data transfers, even small excesses soon become the bottleneck on hosted devices.

Additionally, the process of packing MPI halo buffers ready for exchange should be carried out locally on the device, rather than transferring the data back to the host to perform the operation together with the exchange itself.

8.1.5 Limited Device Memory Capacity

Although memory capacity is increasing, the high bandwidth memory necessary for performance, is in limited supply on accelerated devices. This requires careful management by the developer to ensure it is utilised effectively. If a particular simulation demands more than exists, then the application needs to be able to scale over a greater number of devices, hence reducing the memory requirements needed per device.

8.1.6 Memory Allocation Overheads

The cycle of memory allocation and deallocation, familiar in many applications, was traced to the cause of synchronisation overheads that were detrimental to performance. Effectively these were serialised memory allocation operating system calls in which one thread was able to allocate memory whilst the remaining threads had to wait until the original thread had completed its operation. Although the solution which was devised in the CloverLeaf mini-app employed the creation of temporary arrays that were subsequently re-used for multiple purposes, this may not be applicable in all situations. Such allocation/deallocation cycles should be a point of investigation in any application aiming to achieve performance on an accelerated device.

8.1.7 Use of Global Variables

The presence of globally addressable variables in computational kernels implies an atomic update as the compiler is forced to protect these data values from concurrent accesses from different threads. A solution to this problem is to create separate instances which are private to each thread and contain a separate reduction operation on those variables.

8.2 Putting these Lessons into Practice

Although these lessons were discovered through the examination of a hydrodynamics application, they are equally valid for any algorithmic domain. Whilst this has been extremely useful as a research investigation, it is also of interest how this relates back to the practical problem facing institutions with industrial strength HPC applications.

As a direct result of this research a new application design strategy has been developed at AWE to explore and make ready applications for emerging architectures and as a means to explore suitable programming paradigms.

The “Path to Many Core” (PtMC) is the resultant project to meet this strategy. At its heart is the concept of mini-apps, proven through this research as a suitable methodology for hydrodynamical computational models, it expands the method to cover a range of algorithmic areas of interest. These algorithmic areas are broken down into dedicated workstreams and resourced appropriately: assigning a mix of computer scientists, specific domain specialists and mathematicians.

The research contained herein has evolved from assessing those methods and rapidly exploring emerging HPC hardware and programming methods, to a formal strategy with an associated programme of work. The PtMC project has identified a number algorithmic areas on which to focus efforts in order to enable their performant use on emerging many-core architectures. The strategy is to explore these areas through the development of, or the use of existing, mini-apps.

Complementing the PtMC is the "Path to Agile Coding" (PtAC), this provides the means, by way of restructuring real applications into a framework, which make the application of the lessons learnt in PtMC amenable.

Following the success of CloverLeaf, the UK Mini-App consortium (UK MAC) [39] has grouped together a number of mini-apps, developed as part of collaborations with a number of UK based institutions. Although a non-exhaustive list, a number of the complementary mini-apps to CloverLeaf are:

8.2.1 CloverLeaf3D

CloverLeaf3D is a 3D implementation of CloverLeaf, solving the compressible Euler equations in three dimensions, using an explicit, second-order method. Released into Version 3.0 of the Mantevo Suite, it has been extended beyond its original use and has been the focal tool to "roadtest" a batch-based in-situ visualisation infrastructure for multi-physics simulation codes [129].

8.2.2 CleverLeaf

CleverLeaf [61] is an adaptive mesh refinement implementation of CloverLeaf using the Structured Adaptive Mesh Refinement Application (SAMRAI) [110] toolkit library. Developed and extensively used as part of Beckingsale's PhD Thesis assessing the scalability of AMR on future parallel architectures [60]. It was introduced as a mini-driver as part of the Version 2.0 release of the Mantevo Suite.

8.2.3 TeaLeaf

TeaLeaf is a mini-app that solves the linear heat conduction equation on a spatially decomposed regularly grid using a five point stencil with implicit solvers. Released into Version 3.0 of the Mantevo project, it is been used as a research vehicle by the University Bristol to evaluate its suitability for emerging programming models such as Kokkos, RAJA, OpenACC, OpenMP, CUDA and OpenCL [140], [141].

8.2.4 BookLeaf

Bookleaf is an unstructured Lagrangian Hydro mini-app to which the University of Oxford have applied their stencil based framework OP2 to [174]. The resultant version has been used to perform one of the first assessments of performance on IBM POWER8 CPUs using the software toolchain: Little-Endian Ubuntu, the GNU and the XL compilers and OpenMP runtimes, on IBMs POWER8 CPUs [173].

It has also been taken as a proxy, with added I/O patterns, to be used as a tool for investigating I/O library paradigms [83].

8.2.5 MINIO

Developed by Dickson at the University of Warwick, MINIO [81] is a mini-app to enable the investigation of the overheads of high-level IO libraries. By inputting representative IO patterns of production-class scientific applications it can explore the relative benefits and/or inefficiencies of a range of common IO data format libraries.

8.3 The Spread of Mini Apps

The idea of isolating the dominant numerical kernels contained in a multi-million line source code application has emerged independently across the spectrum of HPC research. Whether as described in this study, or for the purpose of hardware co-design with hardware vendors, their purpose is varied and growing.

The drivers for exascale systems, as detailed in Chapter 1, has seen nearly all of the international states which are developing an exascale strategy include the idea of the mini-app: RIKEN's Application Development Team [179] from Japan, Europe's CEA Hydrobench [28], and the US led Mantevo [104] suite.

8.4 Collaborative Research

Although initially independent, some of these efforts are pooling resources. Mantevo pioneered the idea of an integrated collection of mini-apps. Its initial Version 1.0 contained seven mini-apps, including CloverLeaf, from the three US National Laboratories: SNL, LANL and LLNL, plus NVIDIA and AWE (in collaboration with the University of Bristol and the University of Warwick).

A common format, size of code, build procedure and collection of results is stipulated for all mini-apps wishing to be part of Mantevo. This has delivered the HPC user community a tool with which to be able to track and predict application performance on existing and emerging hardware with minimal time and effort.

This cooperative effort saw Mantevo receive a 2013 R&D 100 award [172] and an FLC (Federal Laboratory Consortium) Regional Technology Transfer award. The project puts out annual releases of the suite prior to each US Supercomputing Conference in November.

At the time of writing there are over 250 papers related to Mantevo, with over 20 specifically referencing CloverLeaf [142], [94], [101], [139], [56], [102], [49], [204], [175], [140], [192], [137], [76], [126], [149], [150], [59], [58], [60], [160], [55].

8.5 Beneficiaries and Future Work

Subsequent to the programming models and architectures explored through CloverLeaf, additional models of the code have been developed and explored.

Work led by Mallinson [137] performed a number of MPI specific optimisations: Pre-posting receives, MPI rank re-ordering and overlapping of communications and computation, resulting in weak scaling the MPI+OpenACC hybrid CloverLeaf code to all 16,384 GPUs on ORNL's Titan [65] supercomputer. Additionally, a Co-Array Fortran (CAF) version of CloverLeaf was developed and compared against the MPI+OpenACC version.

In addition to the OpenMP and OpenACC variants, Gaudin's original Fortran / MPI parallel mini-app, now boasts numerous instances covering a wide range of programming models and paradigms, developed by a wide range of collaborators: CUDA (Boulton and McIntosh-Smith (University of Bristol) and Bradley (NVIDIA)), CUDAFortran [181] (Toepfer, Miles and Wolfe of The Portland Group (PGI)), Partitioned Global Address Space (PGAS) Programming Models [167]: CAF, SHMEM (Mallinson formally of the University of Warwick and now at Intel), OpenCL (Mallinson), LEO (Gamayunov and Blair-Chappell (Intel)), and OpenMP4 (Gamayunov).

8.6 External Industry Take Up

In addition to collaborative efforts with similar research activities, the research has been an enabler for building collaborations with HPC hardware and software industrial partners.

8.6.1 Intel®

The porting and performance optimisations implemented in CloverLeaf when targeting Intel's second generation Intel Xeon Phi, the KNC (Appendix A.5.1

and A.5.2) resulted in collaborations with Intel resulting in requests to present and showcase the work. The Intel Xeon Phi performance details of Chapter 7 were presented at SC12 in November 2012 and subsequently at an invitation event from Intel in March 2013: “Intel Xeon Phi Programming Methods and Tools” [154]. This was followed in issue 14 of Intel’s “The Parallel Universe”, June 2013, which featured an article on the development and results CloverLeaf achieved on their Intel Xeon Phi coprocessor [73].

8.6.2 ARM®

The readiness and potential of Cavium®ThunderX®ARM based platforms for High Performance Computing, in particular CFD, has been explored, on behalf of ARM, in a White Paper from the University of Cambridge [51]. CloverLeaf was chosen as one of two computational fluid dynamics applications that demonstrated scalability and competitive performance using the open-source toolchain based on GCC.

8.6.3 OpenACC®Organisation

In addition to presenting at ORNL [77], an OpenACC Standards Organisation White Paper and Case Study [100] from the OpenACC Standards were developed. These documented the step-by-step development process to achieve a fully distributed and accelerated hybrid MPI/OpenACC implementation of CloverLeaf, as documented in Chapter 5, and was adapted by the author as a chapter in the book Parallel Programming With OpenACC [88]. This aims to be used as a practical guide to show how to use OpenACC with CPUs, GPUs and other accelerators to improve application performance without significant programming effort.

8.6.4 PGI®

Following introductions during the ORNL workshop [77] with PGI, efforts to produce a single source OpenACC version of the CloverLeaf mini-app were carried out resulting in error free compilation under all OpenACC commercially available compilers. Subsequently, PGI have adopted the OpenACC version of CloverLeaf as part of their compiler regression testing suit and a common benchmark comparison when discussing their OpenACC enabled compiler [198].

With the only CUDA Fortran compiler, PGI were keen to have applications demonstrating a performant implementation. With a number of alternative reference versions available, they chose to develop a CUDA Fortran version of the mini-app and make it generally available [180].

8.6.5 International Supercomputing Conference (ISC)

The 2016 ISC Student Cluster competition [163] used CloverLeaf as the mystery application in its 2016 edition of its annual challenge [196]. The goal was to run the application using the lowest peak power, but achieving completion to solution within one hour.

8.6.6 SPEC ACCEL™

The SPEC ACCEL benchmark suite contains 19 application benchmarks running under OpenCL and 15 under OpenACC [119]. The OpenCL suite is derived from the well-respected Parboil benchmark from the IMPACT Research Group of the University of Illinois at Urbana-Champaign and the Rodinia benchmark from the University of Virginia. The OpenACC suite includes tests from NAS Parallel Benchmarks (NPB), SPEC OMP2012, and others derived from high-performance computing (HPC) applications.

The OpenACC `kernels` construct version of CloverLeaf's Fortran implementation has been accepted as one of the initial 15 OpenACC applications [43] that make up the Standard Performance Evaluation Corporation's (SPEC) High-Performance Group (HPG) SPEC ACCEL™benchmark [44].

8.7 Evolution of Architectures and their Programming Models

Heterogeneous computing is changing the HPC architecture landscape; indeed a number of those emerging technologies introduced in Chapter 2 are now becoming established.

Bottlenecks currently associated with the attached accelerator model include data transfers, via PCIe, between the host and device; however this is being addressed in a number of ways depending on the accelerated system in question.

Through the OpenPOWER Foundation [158] NVIDIA and IBM have coupled their respective technologies with the release of the S822LC for the HPC "Minsky" system [113]. Two POWER8 CPUs and up to four NVIDIA Tesla P100 GPUs are connected via NVIDIA's bespoke NVLink interconnect. This significantly reduces the time required for data transfers by enabling over five times faster data transfers compared to PCIe.

As documented in Chapter 2, Section 2.3.2 Intel have also released the second generation Intel Xeon Phi, Knights Landing (KNL) as a self bootable processor no longer requiring a "host", hence entirely removing the attached co-processor bottleneck. However, the addition of high memory bandwidth, in the form of MCDRAM, brings with it the additional complication of a more complex memory hierarchy. The MCDRAM can be utilised in a number of different ways depending on the memory mode the system is configured to, however fully exploiting this technology will place an additional burden on application developers.

The OpenMP standard has been further extended since the OpenMP work presented in Chapter 7, with directives enabling accelerator devices to be targeted. This provides greater control for mapping and unmapping variables to and from a device's data environment. This includes unstructured data mapping functionality as well as enabling the specification of the `private` and `firstprivate` clauses. Although this increase in accelerator device support is welcomed, OpenMP still relies heavily on the developer to describe the specific details for the device in question.

Meanwhile OpenACC now has extended its support to cover more architectures, including ARM CPUs, Sunway CPUs and OpenPOWER CPUs. Additionally, following the demise of CAPS, support returns for x86 CPUs and Intel Xeon Phi processors; although, not all of these architectures are as yet supported by all OpenACC compilers. Additional compiler support is now available from PathScale (ENZO 2015) and full OpenACC 2.0 support will be available in GCC 6. New features which have been added to OpenACC since this research was undertaken include nested parallelism, the ability to target multiple devices and asynchronous data movement. Future plans for OpenACC include a manual deep copy to assist with moving deeply nested data structures, routine error callback, array reductions and unstructured data regions. (The latter directly addressing the deep copy issue observed in section 7.6).

OpenCL is no longer supported on the 2nd generation Intel Xeon Phi, (the KNL), reducing its main benefit, that of portability.

New features in existing high-level programming languages offer some of the functionality found in the low-level languages and directive based approaches. As of Fortran 2008 the DO CONCURRENT construct enables the developer to specify that individual loop iterations have no interdependencies, hence in theory providing enough information to expose loop parallelism to the compiler.

Looking beyond what is available today, both hardware and software continue to evolve, with the trends outlined in Chapter 1 persisting. The core count which is available on Intel's latest generation of x86 processors has increased to a 24 core variant [9]. Whilst the latest GPU from NVIDIA, the P100 [15], based on the "Pascal" micro-architecture consists of 6 Graphics Processing Clusters (GPC) each with 10 SMs, where with 384 GPU cores per SM, an overall 3,840 GPU cores is delivered, up from the 2,880 found in "Kepler".

While these different aspects of many-core devices still exist as separate entities, future heterogeneous systems will most likely have a mixture of these "large" and "small" cores on the same silicon. They will also be able to dynamically adjust their clock speeds, depending on workload and possess the ability to reconfigure the system on-the-fly, keeping the power required to a minimum will increasingly become an important goal. Although research efforts and proof of concept devices already exist such as the 1,000 core KilioCore, from University of California Davis [66], in which each of the one thousand cores can be independently clocked and is able to shut down completely when idle. The overall mix of these cores will likely not be all general purpose, with specialist types becoming available, catering for bespoke tasks which sections of an algorithm, or application will map on to.

Further developments in hardware design will see greater integration within the die, with optical fabrics driving down the costs of interconnects. Additionally, on-package high bandwidth memory with sufficient capacities to provide greatly improved performance for applications that can utilise it, will become common on a range of hardware, not just those seen today.

Power consumption costs for moving data to the sites of computation will become increasingly dominant compared to the costs of actual computations themselves. This will drive the need to co-locate the computation and the data potentially bringing the compute to the data rather than the common methods today for moving the data to the compute engine.

Looking even further into the future will likely see revolutionary changes rather than evolutionary. Different materials which complement silicon together with more flexible materials such as those which can stretch whilst still conducting, will enable advances in micro-architecture design. The 3D stacking of processors is also proposed, which will reduce power consumption for multiple chips, whilst making them more dense.

Non-traditional Turing/Von-Neumann machines may emerge, such as Neuromorphic Computing systems where the system is presented with training data from which it ultimately learns from that data to be able to identify and classify new data it is presented with. Quantum computing which has been muted for many years is beginning to become more than theory with experimental hardware existing in the laboratories of IBM [45] and University of Maryland [80]. Together with a small but growing number of identified quantum algorithms [118], [118], enabling early comparisons of the two different technologies [134]. Although unlikely to fully replace Turing/Von-Neumann based machines, they may emerge as systems that can significantly reduce problems spaces prior to processing on traditional computer systems. Currently, these are speculative designs and would most likely require complete rethinks on algorithmic inputs and languages to program such systems.

8.8 Thesis Contributions

To conclude, this research has focused on a study of the impact on legacy scientific applications, and potential mitigation options, of the emerging changes in high performance computing architectural design. It has set out the drivers for technological hardware change, discussed the prominent architectures that are emerging and, by identifying common architectural traits, enabled a high level conceptual comparison between those architectures. Similarly from a software point of view, those nascent programming methodologies have been categorised and their relative status and maturity assessed. In particular, this thesis makes the following contributions:

8.8.1 Impracticalities of Using Production-Class Codes to Explore Architectures and Programming Models

Demonstrates lack of flexibility of an industrial-class benchmark code as a tool for the rapid exploration of emerging architectures and their associated programming models.

Chapter 4 demonstrated, via the use of a representative, industrial-class, benchmark application, the standard practice of using benchmark codes to assess system upgrades to an incumbent platform and for comparisons for procurement of new platforms. It subsequently showed the shortcomings of development time and practicalities, that a different approach is needed for the assessment of emerging technologies.

8.8.2 Introduction and Extension of the Mini-Application Approach

Introduction of the CloverLeaf mini-application with development details to achieve a fully functional and portable OpenACC implementation.

Chapter 5 introduced the concept of the mini-app, and described CloverLeaf, an explicit Eulerian hydrodynamics mini-app. The study detailed the step-by-step development process that produced a fully functional and portable version using a newly emerging standard, OpenACC.

8.8.3 Demonstrating Performance Portability

Demonstration of the suitability of the mini-app as a tool for exploration of emerging architectures in the particular case of a GPU using three programming methodologies, namely OpenACC, OpenCL and CUDA.

In Chapter 6 the mini-application was demonstrated fully utilising a GPU-based architecture, with direct comparisons of performance, development time and "words of code" (WoC) when compared to the equivalent OpenCL and CUDA implementations. It's finding indicated that a directive based approach, such as OpenACC, is an attractive programming model for accelerator devices both from a productivity and performance perspective.

8.8.4 Exploring Emerging Architectures

Extending the use of the CloverLeaf mini-app to explore a range of emerging architectures namely: GPUs, co-processor, APUs and current CPUs using OpenACC as a common baseline to compare against the performance of the best alternative programming models on each of the platforms analysed.

Contribution 8.8.3 was extended in Chapter 7 to assess further programming methodologies, enabling direct comparisons with regards to development time, maintenance effort, portability and performance on a GPU architecture. Subsequently, using OpenACC as a common baseline, further emerging hardware was assessed (coprocessors, AMD APUs, GPUs) which enabled the optimal native programming methodology to be compared and contrasted against the OpenACC baseline. This work showed the good portability of the directive based method with acceptable levels of performance, but highlighted the need for wariness in variances in compiler implementations.

References

[1] AMD®Opteron 875. http://www.cpu-world.com/CPUs/K8/AMD-Dual-Core%20Opteron%20875%20-%20OST875FAA6CC.html.

[2] Argonne National Laboratory: MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.

[3] GNU Fortran. http://gcc.gnu.org/fortran/.

[4] Intel®Xeon®Paxville. http://ark.intel.com/products/codename/6191/Paxville?q=Paxville.

[5] Intel®Xeon®Processor E5-2450. http://ark.intel.com/products/64611/Intel-Xeon-Processor-E5-2450-20M-Cache-2_10-GHz-8_00-GTs-Intel-QPI.

[6] Intel®Xeon®Processor E5-2670. http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI.

[7] Intel®Xeon®Processor E5-2698v3. https://ark.intel.com/products/81060/Intel-Xeon-Processor-E5-2698-v3-40M-Cache-2_30-GHz.

[8] Intel®Xeon®Processor E5405. http://ark.intel.com/Product.aspx?id=33079.

[9] Intel®Xeon®Processor E7-8894v4. https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz.

[10] Intel®Xeon®Processor L5530. http://ark.intel.com/Product.aspx?id=41755.

[11] Intel®Xeon®Processor X5550. http://ark.intel.com/Product.aspx?id=37106.

[12] Intel®Xeon®Processor X5660. http://ark.intel.com/Product.aspx?id=47921.

[13] National Center for Supercomputing Applications (NCSA) at the University of Iiinois. http://www.ncsa.illinois.edu/.

[14] NVIDIA®GeForce™GTX 285. http://www.nvidia.com/object/product-geforce-gtx-285-us.html.

[15] NVIDIA®Tesla P100. http://www.nvidia.com/object/tesla-p100.html.

[16] NVIDIA®Tesla™C1060 Computing Processor. http://www.nvidia.com/object/product-tesla-c1060-us.html.

[17] OpenMPI: OpenSource High Performance Computing. http://www.open-mpi.org/.

[18] Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/ss12u1-241645.html.

[19] PGI Fortran & C Accelerator Compilers and Programming Model. http://www.pgroup.com/lit/pgiwhitepaperaccpre.pdf.

[20] The Intel Fortran Compiler. http://software.intel.com/en-us/articles/intel-fortran-compiler-professional-edition-for-linux-documentation/.

[21] The OpenMP API Specification for Parallel Programming. http://openmp.org/wp/.

[22] The Portland Group Fortran Compiler. http://www.pgroup.com/resources.

[23] The OpenACC Application Programming Interface version 1.0. http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, November 2011.

[24] The OpenMP Application Program Interface version 3.1. http://www.openmp.org/mp-documents/OpenMP3.1.pdf, July 2011.

[25] AMD OpenCL Accelerated Parallel Processing (APP) SDK. http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/, November 2012.

[26] CAPS OpenACC Compiler The fastest way to many-core programming. http://www.caps-entreprise.com, November 2012.

[27] gpuocelot - A dynamic compilation framework for PTX. http://code.google.com/p/gpuocelot/, November 2012.

[28] hydrobench. https://github.com/HydroBench/Hydro, Oct 2012.

[29] Intel SDK for OpenCL Applications 2012. http://software.intel.com/en-us/vcsource/tools/opencl-sdk, November 2012.

[30] NVIDIA® CUDA API Reference Manual version 4.2. https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_Toolkit_Reference_Manual.pdf, April 2012.

[31] OpenACC accelerator directives. http://www.training.prace-ri.eu/uploads/tx_pracetmo/OpenACC.pdf, November 2012.

[32] OpenCL 1.1 C++ Bindings Header File. http://www.khronos.org/registry/cl/api/1.2/cl.hpp, November 2012.

[33] OpenCL Lounge. https://www.ibm.com/developerworks/community/alphaworks/tech/opencl, November 2012.

[34] OpenCL NVIDIA Developer Zone. https://developer.nvidia.com/opencl, November 2012.

[35] PGI — Resources — CUDA Fortran. http://www.pgroup.com/resources/cudafortran.htm, November 2012.

[36] QUDA: A library for QCD on GPUs. http://lattice.github.com/quda/, Oct 2012.

[37] RougeWave Releases TotalView 8.10. http://www.roguewave.com/company/news/2012/rogue-wave-releases-totalview-8-10, May 2012.

[38] The OpenCL Conference. http://www.iwocl.org/, May 2013.

[39] UK Mini-App consortium. http://uk-mac.github.io/, October 2013.

[40] CUDA Debugger and Profiler - Advanced Debugging and Performance Optimization Tools for CUDA and OpenACC. https://www.allinea.com/cuda-debugger-and-profiler-advanced-debugging-and-performance-optimization-tools-cuda-and-openacc#features, March 2014.

[41] RougeWave CUDA Debugger. http://www.roguewave.com/products-services/totalview/features/cuda-debugging, March 2014.

[42] RougeWave OpenACC Debugger. http://www.roguewave.com/products-services/totalview/features/openacc-debugging, March 2014.

[43] SPEC ACCEL: 353.clvrleaf. https://www.spec.org/auto/accel/Docs/353.clvrleaf.html, March 2014.

[44] SPEC ACCEL Benchmark Suite. https://www.spec.org/accel/, March 2014.

[45] IBM Q: Building the first universal quantum computers for business and science. http://www.research.ibm.com/ibm-q/, March 2017.

[46] Advanced Micro Devices Inc. AMD white Paper: Compute Cores. https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf, 2014.

[47] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[48] D. L. Andrew Komornicki, Gary Mullen-Schulz. *Roadrunner: Hardware and Software Overview.* IBM Redbooks, 2009.

[49] J. A. Ang, R. F. Barrett, S. D. Hammond, and A. F. Rodrigues. Emerging high performance computing systems and next generation engineering analysis applications. *SAND-2013-0054P*, 2013.

[50] ANL. Aurora. http://aurora.alcf.anl.gov/.

[51] ARM. HPC Case Study: CFD Applications on ARM. https://community.arm.com/processors/b/blog/posts/arm-hpc-case-study-university-of-cambridge/, 2017.

[52] N. Attig, P. Gibbon, and T. Lippert. Trends in supercomputing: The european path to exascale. *Computer Physics Communications*, 182(9):2041–2046, 2011.

[53] R. Babich, M. Clark, and B. Joo. Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics. *IEEE*, 2010.

[54] M. Baker, S. Pophale, J.-C. Vasnier, H. Jin, and O. Hernandez. Hybrid Programming using OpenSHMEM and OpenACC. *OpenSHMEM, Annapolis, Maryland. March 4-6, 2014*, 2014.

[55] D. W. Barnette, R. F. Barrett, S. D. Hammond, J. Jayaraj, and J. H. Laros III. Using miniapplications in a mantevo framework for optimizing sandias sparc cfd code on multi-core many-core and gpu-accelerated compute platforms. In *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, page 1126, 2012.

[56] R. F. Barrett, P. S. Crozier, D. Doerfler, M. A. Heroux, P. T. Lin, H. Thornquist, T. Trucano, and C. T. Vaughan. Assessing the role of mini-applications in predicting key performance characteristics of scientific and engineering applications. *Journal of Parallel and Distributed Computing*, 75:107–122, 2015.

[57] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, volume 95, pages 11–14, 1995.

[58] D. Beckingsale, W. Gaudin, J. A. Herdman, and S. Jarvis. Resident block-structured adaptive mesh refinement on thousands of graphics processing units. In *Parallel Processing (ICPP), 2015 44th International Conference on*, pages 61–70. IEEE, 2015.

[59] D. Beckingsale, W. Gaudin, R. Hornung, B. Gunney, T. Gamblin, J. Herdman, and S. Jarvis. Parallel block structured adaptive mesh refinement on graphics processing units. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), 2014.

[60] D. A. Beckingsale. *Towards scalable adaptive mesh refinement on future parallel architectures*. PhD thesis, University of Warwick, 2015.

[61] D. A. Beckingsale, O. Perks, W. Gaudin, J. Herdman, and S. A. Jarvis. Optimisation of patch distribution strategies for amr applications. In *Tribastone, Mirco and Gilmore, Stephen, 1962-, (eds.) Computer Performance Engineering. Lecture Notes in Computer Science , Volume 7587 .*, pages 210–223. Springer, 2013.

[62] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 5–5. IEEE, 2006.

[63] B. Bergen, M. Daniels, and P. Weber. A Hybrid Programming Model for Compressible Gas Dynamics using OpenCL. *39th International Conference on Parallel Processing Workshops*, 2010.

[64] R. F. Bird, P. Gillies, M. Bareford, J. Herdman, and S. A. Jarvis. Mini-app driven optimisation of inertial confinement fusion codes. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 768–776. IEEE, 2015.

[65] A. Bland, J. Wells, O. Messer, O. Hernandez, and J. Rogers. Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory. In *Cray User Group*, 2012.

[66] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. A 5.8 pj/op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*, pages 1–2. IEEE, 2016.

[67] R. Brook, B. Hadri, V. Betro, R. Hulguin, and R. Braby. Early Application Experiences with the Intel MIC Architecture in a Cray CX1. In *Cray User Group*, 2012.

[68] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. IEEE, 2010.

[69] BULL. Bull sequana supercomputers. http://www.bull.com/sequana.

[70] CAPS Enterprise. HMPP: A Hybrid Multicore Parallel Programming Platform. http://www.caps-entreprise.com/en/documentation/caps_hmpp_product_brief.pdf.

[71] C. C. Charlie Cler. *IBM Power 595 Technical Overview and Introduction.* IBM Redbooks, 2008.

[72] K. China. Chinas 12th five-year plan: overview. *China: KPMG Advisory*, 2011.

[73] N. Clemons. A complete development solution for intelligent systems, 2013.

[74] P. Colella. Defining software requirements for scientific computing, 2004.

[75] P. Computing and U. Visulisation Group, University of Warwick. CloverLeaf: A Lagrangian-Eulerian hydrodynamics mini-app. http://warwick-pcav.github.io/CloverLeaf.

[76] J. Cownie and S. McIntosh-Smith. Leverage your opencl investment on intel® architectures. *Graduate from MIT to GCC Mainline*, page 42, 2014.

[77] Cray Technical Workshop on XK6 Programming. XK6 Workshop. https://www.olcf.ornl.gov/training-event/cray-technical-workshop-on-xk6-programming/, 2012.

[78] C. CUDA. Programming Guide v5.5. *NVIDIA Corporation, July*, 2013.

[79] J. Davis, G. R. Mudalige, S. D. Hammond, J. 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, 2011.

[80] S. Debnath, N. Linke, C. Figgatt, K. Landsman, K. Wright, and C. Monroe. Demonstration of a small programmable quantum computer with atomic qubits. *Nature*, 536(7614):63–66, 2016.

[81] J. Dickson, S. Maheswaran, S. A. Wright, J. Herdman, and S. A. Jarvis. Minio: an i/o benchmark for investigating high level parallel libraries. *27th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), Austin, Texas, USA*, Nov. 2015.

[82] J. Dickson, S. A. Wright, S. Maheswaran, J. Herdman, D. Harris, M. C. Miller, and S. A. Jarvis. Enabling portable i/o analysis of commercially sensitive hpc applications through workload replication. *Cray User Group 2017 Proceedings*, 2017.

[83] J. Dickson, S. A. Wright, S. Maheswaran, J. Herdman, M. C. Miller, and S. A. Jarvis. Replicating hpc i/o workloads with proxy applications. *1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS'16), Salt Lake City, Utah, USA*, Nov. 2016.

[84] DOE. Exascale Strategy: Report to Congress June 2013. Technical report, Department of Energy , 2013.

[85] J. Dongarra. Report on the Sunway TaihuLight System. http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf, June 2016.

[86] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[87] H. C. Edwards and D. Sunderland. Kokkos Array performance-portable manycore programming model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10. ACM, 2012.

[88] R. Farber. *Parallel Programming with OpenACC, 1st Edition.* Morgan Kaufmann, 2016.

[89] S. I. Feldman. A Fortran to C converter. In *ACM SIGPLAN Fortran Forum*, volume 9, pages 21–22. ACM, 1990.

[90] A. Fog. The microarchitecture of intel, amd and via cpus/an optimization guide for assembly programmers and compiler makers, 2012.

[91] V. Gamayunov and S. Blair-Chappell. Shockwaves! CloverLeaf Meets the Intel Xeon Phi Coprocessor. http://www.sos-software.com/

download-sos/Intel-Parallel-Universe-Magazin-Ausgabe-14_web.pdf, 2012.

[92] X. Gao, Z. Wang, H. Wan, and X. Long. Accelerate Smoothed Particle Hydrodynamics Using GPU. *IEEE*, 2010.

[93] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.* Newnes, 2012.

[94] W. Gaudin, A. C. Mallinson, O. F. J. Perks, J. A. Herdman, D. A. Beckingsale, J. Levesque, M. Boulton, S. McIntosh-Smith, and S. A. Jarvis. Optimising Hydrodynamics applications for the Cray XC30 with the application tool suite. *Cray User Group 2014, Lugano, Switzerland. CUG2014 Final Proceedings pp. 4-8.*, May 2014.

[95] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1299–1308. IEEE, 2013.

[96] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly. Performance analysis of the op2 framework on many-core architectures. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):9–15, 2011.

[97] K. Gregory and A. Miller. C++ amp: accelerated massive parallelism with microsoft visual c++, 2014.

[98] T. Hahn, V. Heuveline, and B. Rocker. GPU accelerated scientific computing: Fluid and particulate flows with CUDA.

[99] S. D. Hammond, G. R. Mudalige, J. Smith, J. A. Davis, S. A. Jarvis, J. Holt, I. Miller, J. Herdman, and A. Vadgama. To upgrade or not to upgrade? catamount vs. cray linux environment. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[100] J. A. Herdman. Applying OpenACC to the Cloverleaf Hydrodynamics Mini-App. http://www.cray.com/sites/default/files/resources/OpenACC_213462.7_OpenACC_Cloverleaf_CS_FNL.pdf.

[101] J. A. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. A. Jarvis. Accelerating hydrocodes with OpenACC, OpenCL and CUDA. In *SC Companion 2012 High Performance Computing, Networking Storage and Analysis, Salt*

Lake City, UT, 10-16 Nov 2012. Published in: SC Companion, pages 465–471. IEEE, 2012.

[102] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Achieving Portability and Performance through OpenACC. In *Proceedings of the First Workshop on Accelerator Programming using Directives (WACCPD '14), held as part of SC14 The International Confernece for HPC, Networking, Storage and Analysis. New Orleans, LA, USA.*, pages 19–26. IEEE Press, Nov. 2014.

[103] J. A. Herdman, W. P. Gaudin, D. Turland, and S. D. Hammond. Benchmarking and Modelling of POWER7, Westmere, BG/P, and GPUs: An Industry Case Study. *In: 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 10), New Orleans, LA, USA. ACM SIGMETRICS Performance Evaluation Review*, 38(4):16–22, 2011.

[104] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

[105] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010.

[106] T. Hoefler. Software and hardware techniques for power-efficient hpc networking. *Computing in Science & Engineering*, 12(6):30–37, 2010.

[107] T. Hoefler, C. Siebert, and A. Lumsdaine. Group operation assembly language-a flexible way to express collective communication. In *2009 International Conference on Parallel Processing*, pages 574–581. IEEE, 2009.

[108] J. P. Holdren. National strategic computing initiative strategic plan. https://www.whitehouse.gov/sites/whitehouse.gov/files/images/NSCI%20Strategic%20Plan.pdf.

[109] R. Hornung and J. Keasler. The RAJA portability layer: overview and status. *Lawrence Livermore National Laboratory, Livermore, USA*, 2014.

[110] R. D. Hornung and S. R. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14(5):347–368, 2002.

[111] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010.

[112] W. Hu, Y. Zhang, and J. Fu. An introduction to cpu and dsp design in china. *Science China Information Sciences*, 59(1):1–8, 2016.

[113] IBM. IBM S822LC for High Performance Computing. https://public.dhe.ibm.com/common/ssi/ecm/po/en/pod03117usen/POD03117USEN.PDF, 2016.

[114] Intel Developer Zone. The Heterogeneous Programming Model. http://software.intel.com/en-us/articles/the-heterogeneous-programming-model, 2012.

[115] Intel Developer Zone. OpenMP 4.0 Features in Intel Fortran Composer XE 2013. http://software.intel.com/en-us/articles/openmp-40-features-in-intel-fortran-composer-xe-2013, 2013.

[116] ISC. Isc high performance. http://www.isc-hpc.com/id-2016.html, 2016.

[117] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.

[118] S. Jordan. The Quantum Algorithm Zoo. http://math.nist.gov/quantum/zoo/.

[119] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu, et al. Spec accel: A standard application suite for measuring hardware accelerator performance. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 46–67. Springer, 2014.

[120] J. Junior, E. Clua, A. Montenegro, and P. Pagliosa. Fluid simulation with two-way interaction rigid body using a heterogeneous GPU and CPU environment. *Brazilian Symposium on Games and Digital Entertainment*, 2010.

[121] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[122] 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 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 37–37. ACM, 2001.

[123] P. Kestener, F. Château, and R. Teyssier. Accelerating Euler equations numerical solver on graphics processing units. In *Algorithms and Architectures for Parallel Processing*, pages 281–288. Springer, 2010.

[124] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE international symposium on workload characterization (IISWC)*, 2013.

[125] Khronos OpenCL Working Group. The OpenCL Specification Version 1.2. http://www.khronos.org/opencl/, 2008.

[126] H. Kim, H. Kim, S. Yalamanchili, and A. F. Rodrigues. Understanding energy aspects of processing-near-memory for hpc workloads. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 276–282. ACM, 2015.

[127] LANL. Crossroads: A critical element for improved predictive capability. http://www.lanl.gov/projects/crossroads.

[128] LANL. Trinity: Advancing predictive capability for stockpile stewardship. http://www.lanl.gov/projects/trinity/.

[129] M. Larsen, E. Brugger, H. Childs, J. Eliot, K. Griffin, and C. Harrison. Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 30–35. ACM, 2015.

[130] B. Leback, D. Miles, and M. Wolfe. Tesla vs. Xeon Phi vs. Radeon A Compiler Writers Perspective. *Technical News from The Portland Group*, 2013.

[131] B. Lebacki, M. Wolfe, and D. Miles. The PGI Fortran and C99 OpenACC Compilers. In *Cray User Group*, 2012.

[132] J. Levesque, R. Sankaran, and R. Grout. Hybridizing S3D into an Exascale Application using OpenACC. *SC12, November 10-16, 2012, Salt Lake City, Utah, USA (2012)*, 2012.

[133] X. Liao, L. Xiao, C. Yang, and Y. Lu. Milkyway-2 supercomputer: system and application. *Frontiers of Computer Science*, 8(3):345–356, 2014.

[134] N. M. Linke, D. Maslov, M. Roetteler, S. Debnath, C. Figgatt, K. A. Landsman, K. Wright, and C. Monroe. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences*, page 201618020, 2017.

[135] LLNL. 100 teraflops dedicated to capability computing. https://asc.llnl.gov/computing_resources/purple/.

[136] LLNL. Lawrence Livermore National Laboratory Machine Catalog: Sierra. http://computation.llnl.gov/computers/sierra-advanced-technology-system.

[137] A. Mallinson, D. Beckingsale, W. Gaudin, J. Herdman, J. Levesque, and S. Jarvis. CloverLeaf: Preparing hydrodynamics codes for Exascale. *In: A New Vintage of Computing : CUG2013, Napa, CA, 6 - 9 May 2013. Published in: A New Vintage of Computing : Preliminary Proceedings*, 2013.

[138] A. Mallinson, D. A. Beckingsale, W. Gaudin, J. Herdman, and S. A. Jarvis. Towards Portable Performance for Explicit Hydrodynamics Codes. *Proceedings of the 1st International Workshop on OpenCL (IWOCL 2013). ACM (May 2013)*, 2013.

[139] A. Mallinson, S. A. Jarvis, W. Gaudin, and J. Herdman. Experiences at scale with PGAS versions of a Hydrodynamics application. In *In: PGAS '14 : 8th International Conference on Partitioned Global Address Space Programming Models, Eugene, Oregon, USA, 7-10 Oct 2014. Published in: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models pp. 1-11.* ACM, 2014.

[140] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin. An evaluation of emerging many-core parallel programming models. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10. ACM, 2016.

[141] M. Martineau, S. McIntosh-Smith, M. Boulton, W. Gaudin, and D. Beckingsale. A performance evaluation of kokkos & raja using the tealeaf mini-app. *Poster session presented at Poster session presented at IEEE/ACM SuperComputing, Austin, United States.*, 2015.

[142] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price. On the performance portability of structured grid codes on many-core computer architectures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).*, pages 53–75. Springer, 2014.

[143] F. McMahon. Livermore Fortran Kernels: A computer test of numerical performance range. Technical report, Lawrence Livermore National Laborotory, 1986.

[144] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Tanhe-1A. https://www.top500.org/system/176929.

[145] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. TOP500 November 2000. http://www.top500.org/lists/2000/11/, 2000.

[146] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. TOP500 November 2003. http://www.top500.org/lists/2003/11/, 2003.

[147] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. TOP500 June 2010. http://www.top500.org/lists/2010/06/, 2010.

[148] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. TOP500 June 2011. http://www.top500.org/lists/2011/06/, 2011.

[149] G. Mudalige, I. Reguly, M. Giles, A. Mallinson, W. Gaudin, and J. Herdman. Performance Analysis of a High-Level Abstractions-Based Hydrocode on Future Computing Systems. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, held as part of SC14 The International Confernece for HPC, Networking, Storage and Analysis. New Orleans, LA, USA. Revised Selected Papers*, pages 85–104. Springer, Nov 2014.

[150] J. Nelson. Analyzing papi performance on virtual machines, 2013.

[151] NERSC. Cori. http://www.nersc.gov/users/computational-systems/cori/.

[152] C. J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire. Offload compiler runtime for the intel® xeon phi coprocessor. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1213–1225. IEEE, 2013.

[153] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing.* " O'Reilly Media, Inc.", 1996.

[154] Nicholson, Matt. Gray Matter: HardCopy. Issue 60 Summer 2013. http://www.greymatter.com/corporate/wp-content/uploads/HardCopy-PDF/HC-60.pdf, 2013.

[155] NVIDIA Corporation. Parallel Thread Execution ISA Application Guide v3.2, July 2013.

[156] Oak Ridge National Laboratory. Summit. https://www.olcf.ornl.gov/2014/11/14/oak-ridge-to-acquire-next-generation-supercomputer/.

[157] OpenACC Org. OpenACC Reviews Latest Developments and Future Plans. https://www.hpcwire.com/2015/11/11/openacc-reviews-latest-developments-and-future-plans/, 2015.

[158] OpenPOWER Foundation. The Open POWER Foundation. https://openpowerfoundation.org/, 2016.

[159] M. Ospici, D. Komatitsch, J.-F. Mehaut, T. Deutsch, et al. Sgpu 2: a runtime system for using of large applications on clusters of hybrid nodes. In *Second Workshop on Hybrid Multi-core Computing, held in conjunction with HiPC*, pages 1–8, 2011.

[160] J. Ouyang, B. Kocoloski, J. R. Lange, and K. Pedretti. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 149–160. ACM, 2015.

[161] D. A. Patterson and J. L. Hennessy. Computer Organization and Design. *Morgan Kaufmann*, pages 474–476, 2007.

[162] S. J. Pennycook, S. D. Hammond, S. A. Wright, J. Herdman, I. Miller, and S. A. Jarvis. An Investigation of the Performance Portability of OpenCL. *Journal of Parallel and Distributed Computing*, 73(11):1439–1450, 2013.

[163] I. H. Performance. ISC 2016 Student Cluster Competition. http://www.hpcadvisorycouncil.com/events/2016/isc16-student-cluster-competition/index.php, 2016.

[164] O. Perks, D. A. Beckingsale, A. Dawes, J. Herdman, C. Mazauric, and S. A. Jarvis. Analysing the influence of infiniband choice on openmpi memory consumption. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 186–193. IEEE, 2013.

[165] O. Perks, D. A. Beckingsale, S. D. Hammond, I. Miller, J. Herdman, A. Vadgama, A. H. Bhalerao, L. He, and S. A. Jarvis. Towards Automated Memory Model Generation via Event Tracing. *The Computer Journal, Volume 56 (Number 2). pp. 156-174. ISSN 0010-4620*, 2012.

[166] A. Petitet. HPL-a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. *http://www.netlib-. org/-benchmark/hpl/*, 2004.

[167] PGAS Working Group. Partitioned Global Address Space. http://www.pgas.org/, 2015.

[168] Phoronix. Samsung Brings OpenACC 1.0+ Support To GCC Fortran. http://www.phoronix.com/scan.php?page=news_item&px=MTU4MjQ, 2014.

[169] J. Pier, I. Figueroa, and J. Huegel. CUDA-enabled Particule-based 3D Fluid Haptic Simulation. *Electronics, Robotics and Automotive Mechanics Conference*, 2011.

[170] T. N. Platform. Inside japans future exascale arm supercomputer. http://www.nextplatform.com/2016/06/23/inside-japans-future-exaflops-arm-supercomputer/.

[171] PRACE. PRACE Resources. http://www.prace-ri.eu/prace-resources/.

[172] R&D Magazine. The 2013 R&D 100 Award Winners. http://www.rdmag.com/award-winners/2013/07/2013-r-d-100-award-winners, 2013.

[173] I. Z. Reguly, A.-K. Keita, R. Zurob, and M. B. Giles. High performance computing on the ibm power8 platform. In *International Conference on High Performance Computing*, pages 235–254. Springer, 2016.

[174] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. Kelly, and D. Radford. Acceleration of a full-scale industrial cfd application with op2. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1265–1278, 2016.

[175] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Design and development of domain specific active libraries with proxy applications. In *2015 IEEE International Conference on Cluster Computing*, pages 738–745. IEEE, 2015.

[176] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith. The ops domain specific abstraction for multi-block structured grid computations. In *Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on*, pages 58–67. IEEE, 2014.

[177] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[178] R. Reyes, I. Lopez, J. Fumero, and F. de Sande. Directive-based Programming for GPUs: A Comparative Study. *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012.

[179] RIKEN. RIKEN Advanced Institute for Computational Science: Application Development Team. http://www.riken.jp/en/research/labs/aics/exascale_comput_proj/app_dev/, 2013.

[180] G. Ruetsch. S5379 - Porting CloverLeaf to CUDA Fortran. In *GPU Technology Conference*, March 2015.

[181] G. Ruetsch and M. Fatica. Cuda fortran for scientists and engineers. *NVIDIA Corporation*, 2701, 2011.

[182] E. Rustico, G. Bilotta, G. Gallo, and A. Herault. Smoothed Particle Hydrodynamics simulations on multi-GPU systems. *20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2012.

[183] B. Sander. Bolt: A c++ templater library for hsa. *AMD Fusion Developer Summit*, 12:18–64, 2012.

[184] R. R. Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[185] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008.

[186] D. H. Sharp. An overview of rayleigh-taylor instability. *Physica D: Nonlinear Phenomena*, 12(1-3):3IN111–10IN1018, 1984.

[187] G. Shi, S. Gottleib, and M. Showerman. Tuning And Understanding MILC Performance In Cray XK6 GPU Clusters. In *Cray User Group*, 2012.

[188] H. Shukla, T. Woo, H. Schive, and T. Chiueh. Multi-Science Applications with Single Codebase - GAMER - for Massively Parallel Architectures. *IEEE*, 2011.

[189] D. Stevenson et al. An American National Standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.

[190] S. Swaminarayan, J. A. Herdman, and M. W. Glass. Migrating Legacy Applications to Emerging Hardware. http://sc15.supercomputing.org/schedule/event_detail-evid=bof138.html, 2015.

[191] D. Tianyi and T. Abdelrahman. hiCUDA:High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed System , vol. 22, pp. 7890, 2011.*, 2012.

[192] E. Totoni, J. Torrellas, and L. V. Kale. Using an adaptive hpc runtime system to reconfigure the cache hierarchy. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1047–1058. IEEE Press, 2014.

[193] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, 2008.

[194] S. Vetter, G. Anselmi, B. Blanchard, Y. Cho, C. Hales, M. Quezada, et al. *IBM Power 750 and 755 (8233-E8B, 8236-E8C) technical overview and introduction*. IBM Redbooks, 2012.

[195] S. Wienke, P. Springer, C. Terboven, and D. Mey. OpenACC - First Experiences with Real-World Applications. *Euro-Par 2012, LNCS, Springer Berlin/Heidelberg(2012)*, 2012.

[196] H. Wire. South Africa Team Claims Third ISC Student Cluster Championship. https://www.hpcwire.com/2016/06/23/south-africa-team-scores-third-isc-scc-win/, 2016.

[197] M. Wolfe. PGI Insider: OpenACC Kernels and Parallel Constructs. http://www.pgroup.com/lit/articles/insider/v4n2a1.htm, 2012.

[198] M. Wolfe. Programming NVIDIA GPUs with OpenACC Directives. http://on-demand.gputechconf.com/supercomputing/2013/presentation/SC3106-Accelerated-Computing-OpenACC.pdf, March 2013.

[199] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. Herdman, I. Miller, A. Vadgama, A. Bhalerao, and S. A. Jarvis. Parallel File System Analysis through Application I/O Tracing. *The Computer Journal*, 56(2):141–155, 2013.

[200] S. A. Wright, S. D. Hammond, S. J. Pennycook, I. Miller, J. A. Herdman, and S. A. Jarvis. LDPLFS: Improving I/O Performance without Application Modification. In *In: 13th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing, Shanghai, China, 21-25 May 2012. Published in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW) pp. 1352-1359.*, 2012.

[201] X. Wu. Parallel Performance Measurement and Analysis. In *Performance Evaluation, Prediction and Visualization of Parallel Systems*, pages 103–162. Springer, 1999.

[202] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[203] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe. The k computer: Japanese next-generation supercomputer development project. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pages 371–372. IEEE Press, 2011.

[204] Y. Zhou, B. Subramaniam, K. Keahey, and J. Lange. Comparison of virtualization and containerization techniques for high-performance computing. *Proceedings of the 2015 ACM/IEEE conference on Supercomputing*, 2015.

Appendices

APPENDIX A

Hardware Platforms and Architectures

Chapter 2 gave an overview of the current HPC architecture options available, or becoming available, to the high-end HPC application user. In this Appendix, specifics are provided of the supercomputing resources utilised throughout this thesis. These have ranged from local Linux based workstations to some of the largest distributed supercomputers in the world. This chapter categorises and describes these resources. As this work has spanned a number of years, the rapid pace of HPC technological progress is reflected in the numerous generations of hardware utilised in each category; to assist the reader, the year of commission of the system is supplied.

A.1 x86-64

A.1.1 Willow (AWE, Bull Distributed Commodity Capacity Cluster)

Commissioned: 2010

"Willow" is an Intel based cluster from Bull, based at the Atomic Weapons Establishment (AWE). It is built on Bull's BullX B500 Extreme Computing blade technology. "Willow" is actually two distinct platforms: "WillowA" and "Willow B" each of which consist of 468 blades, or nodes, housed in 26 chassis. Each node contains two quad-core Intel Xeon Nehalem processors. This gives a total of 3,744 cores. Willow employs the Nehalem L5530 [10]. This has a clock speed of 2.4 GHz, with 8 MB cache and a theoretical peak of four floating point operations (FLOP/s) per clock cycle. As is common with the Nehalem micro architecture, it utilises Intel's QuickPath Interconnect (QPI). The L5530 has a

QPI speed of 5.86 GigaTransfers per second (GT/s). Willow's interconnect is Quad Data Rate (QDR) InfiniBand (IB), with 40 Gb/s full-duplex bandwidth, providing an MPI bisectional bandwidth of 748.8 GB/s. Each node has 24GB of DDR3 memory, which equates to 3GB/core. The total peak performance of each Willow platform is 35.942 TeraFlops (TFLOP/s). The default Fortran compiler and MPI implementation of choice on the system is Intel 11.0.073 and BullXMPI 1.0.1 respectively. Test builds of SunStudio12u1 and OpenMPI 1.4.1, for SunStudio, have also been installed on the "Willow" systems.

A.1.2 Blackthorn (AWE, Bull Distributed Commodity Capability Cluster)

Commissioned: 2010

"Blackthorn" is a large, capability, platform based at AWE. The cluster is a Bull platform consisting of 1,080 compute nodes. Each node consists of two Intel Xeon X5660 hex-core "Westmere" processors. The X5660 [12] has a 2.8 GHz clock[1], with 12 MB of cache, and a QPI speed of 6.40 GT/s. Each node has 48GB of DDR3 memory, which equates to 4GB/core. Blackthorn's interconnect is QDR IB, with 40 Gb/s full-duplex bandwidth, providing 1728 GB/s MPI bisectional bandwidth. With each of the X5660 cores providing a theoretical four FLOP/s per clock cycle; Blackthorn has a peak of 145.1 TFLOP/s. Intel 11.0.073 is the default Fortran compiler, with BullXMPI 1.0.2 the default MPI implementation.

A.1.3 Shepard (SNL, Penguin TestBed Cluster)

Commissioned: 2014

"Shepard" is a Penguin®integrated testbed based at SNL. It consists of 36 Dual Intel Xeon *Haswell* E5-2698 v3 [7] @ 2.30GHz, 16 cores, 2 SMT HW threads per socket. 128 GB DDR4-2133 MHz (split at 64 GB per socket). Mellanox

[1]It is possible to increase this to 3.2 GHz via Intel's TurboBoost, but this is currently disabled on this platform

	Willow	**Blackthorn**	**PillowB**	**Hera**
Integrator	BULL	BULL	BULL	Appro
Processor Type	Intel L5530	Intel X5660	Intel E5-2450	AMD Opteron 8356
Clock Speed GHz (Turbo Freq)	2.4 (2.66)	2.8 (3.2)	2.1 (2.9)	2.3 N/A
Compute Nodes	486	1,080	40	847
CPUs/ Node	1xNehalem 8-core	1xWestmere 12-core	2xSandyBridge 8-core	4xBarcelona 4-core
Interconnect	IB	IB	IB	IB
Compilers	Intel 11.0 Sun 12.1	Intel 11.0	Intel 13.1	Intel 11.0 PGI 8.0.1
MPI	BullX 1.0.1 OpenMPI 1.4.1	BullX 1.0.2	IntelMPI 5.0.3	OpenMPI 1.3.2

Table A.1: x86 Platform Resources

FDR IB. Red Hat 6.5.

A.1.4 Hera (LLNL, Appro Distributed Commodity Capacity Cluster)

Commissioned: 2009

"Hera" is an Appro integrated cluster based at LLNL. Consisting of 790 nodes of AMD quad-core Opteron 8356 (Barcelona) processors, it contains 13,824 cores, with a clock speed of 2.3 GHz, and 32 GB/node, or 2 GB/core. Hera's peak performance equates to 127.2 TFLOP/s. MPI is provided by a build of OpenMPI 1.3.2 A choice of Fortran compilers are Intel 11.1 and PGI 8.0-1.

All x86 based platforms utilised during this research are detailed in Table A.1

A.2 IBM®POWER®

A.2.1 Gollum (AWE, IBM POWER5)

Commissioned: 2006

"Gollum" is an IBM 550 POWER5 based server. It consists of a single node with 4 processors clocked at 1.65 GHz. Fortran is provided by XL Fortran 13.1.0.2.

A.2.2 Milano (IBM, IBM POWER6)

Commissioned: 2008

"Milano" is an IBM Power 550 Express system. The four socket system is populated with dual-core POWER6 processors. Each core has a clock speed of 4.2 GHz. The system is running AIX (Advanced Interactive eXecutive) 5.3 ML11, and the Fortran compiler is IBM's XL Fortran, Version 12.1.0.6. Milano has two-way simultaneous multithreading (SMT) enabled.

A.2.3 v60 (IBM, IBM POWER6)

Commissioned: 2006

"V60" an IBM Power 575. With 16 dual-core sockets this gives rise to 32 cores, each clocked at 4.7 GHz, and 128 GB memory. The operating system (OS) is AIX 5.3 ML10, and the XL Fortran compiler is Version 12.1.0.4. As with "Milano", two-way simultaneous multithreading (SMT) enabled.

A.2.4 p90 (IBM, IBM POWER7)

Commissioned: 2010

"p90", is an IBM POWER7, p755 server; it is packaged as a 4-way quad-chip-module (QCM) with 32 physical cores and 128 SMTs. Each core has a clock speed of 3.3 GHz and 128 GB of accessible memory. The system has 4-way SMT enabled by default.

	Gollum	**Milano**	**v60**	**p90**	**White**
Processor Type	POWER5 p550	POWER6 p550 Express	POWER6 p575	POWER7 p755	POWER8 SL822L SL824L
Clock Speed (GHz)	1.65	4.2	4.7	3.6	3.42
Compute	1	1	1	1	8 9
CPUs/ Node	2x 2-core	4x 2-core	16x 2-core	4x 8-core	2x 8-core 10-core
Compilers	XLF 13.1	XLF 12.1	XLF 12.1	XLF 13.1	XLF 13.1.3
MPI	IBM MPI	IBM MPI	IBM MPI	IBM MPI	OpenMPI

Table A.2: POWER Platform Resources

For all the above POWER platforms, MPI is provided via IBM MPI.

A.2.5 White (SNL, IBM POWER8)

Commissioned: 2015

"White" is a POWER8 based system consisting of one S822L and nine S824L two socket servers. Each socket is subdivided into two NUMA (Non-Uniform Memory Access) regions, each with their own memory controller. MPI is served via OpenMPI.

All POWER based platforms utilised during this research are detailed in Table A.2

A.3 IBM®Blue Gene®

Results in this thesis were obtained on examples of first and second generation IBM Blue Gene supercomputers: Blue Gene/L and Blue Gene/P respectively. The instances of these machines were located at LLNL. Table A.3 displays the characteristics detailed below.

	uBG/L	**DawnDev**
Processor Type	PowerPC 450	PowerPC 450(d)
Clock Speed (MHz)	700	850
Compute Nodes	40,960	1,024
Cores/Node	2	4
Total Cores	81,920	4,096
Memory/Node (GB)	0.5	4
Interconnect	Proprietary	Proprietary
Peak TFLOPS	229.4	13.9
Compilers	XLF 11.0	XLF 11.0
MPI	IBM BlueGene MPI	IBM BlueGene MPI

Table A.3: BlueGene Platform Summary

A.3.1 uBG/L (LLNL, IBM BlueGene/L)

Commissioned: 2007

"uBG/L" is based at the Lawrence Livermore National Laboratory (LLNL), its building block is the 700 MHz, 32-bit PowerPC (450). 40,960 dual core compute nodes combine to provide 81,920 cores, each with 256 MB / core. This gives "uBG/L" a peak performance of 229.4 TFLOP/s. Fortran is provided by XL Fortran 10.1.0.4.

A.3.2 DawnDev (LLNL, IBM BlueGene/P)

Commissioned: 2009

"DawnDev" is a BG/P architecture, again based at LLNL. BG/P is built from the 850 MHz 32-bit PowerPC (450d) processor, with four cores per node, and 1 GB/core. As a test system, DawnDev contains 1,024 nodes, giving a peak of 13.9 TFLOP/s. Fortran is provided by XL Fortran 11.1.0.5. For both BlueGene systems, IBM BlueGene MPI is the MPI implementation.

A.4 Graphics Processing Units (GPUs)

A.4.1 Dexter (AWE, NVIDIA GPU Testbed)

Commissioned: 2010

AWE has a modest GPU test bed architecture, codenamed "Dexter". Consisting of four nodes: one master, and three compute. The master node contains two quad core Nehalem X5550 [11] 2.67 GHz processors, and four NVIDIA Tesla C1060 [16] GPUs. Where each C1060 contains 240 streaming processor cores with a frequency of 1.3 GHz. The compute nodes consist of one quad core X5550, two of the three with four NVIDIA GeForce GTX 285 [14] GPUs and the third with three GTX 285's and one AMD Radeon$^{\text{TM}}$HD5870. For the purposes of this study, only the NVIDIA cards were considered, and the C1060 and GTX 285's treated as one. Dexter is running OpenSUSE as its OS and the GNU compiler and OpenMPI provide the Fortran and MPI environments respectively.

A.4.2 Shannon (SNL, NVIDIA GPU Cluster)

Commissioned: 2013

"Shannon" is a 32 node, dual socket oct-core Intel Xeon E5-2670 with either 2 NVIDIA Kepler K20X per node, each with 2,688 cores clocked at 732 MHz, or 2 NVIDIA K40 per node, each with 2,880 cores clocked at 745 MHz. As part of NNSA's Advanced Simulation and Computing (ASC) project, it is one of a number of advanced test-bed architectures based at Sandia National Laboratories (SNL). OpenACC implementations available on Shannon are PGI 13.9.0 and CAPS 3.3.4. Alternative approaches available to application acceleration on the system are native implementations in CUDA.

A.4.3 Chilean Pine (AWE, Cray NVIDIA GPU Cluster)

Commissioned: 2011

"Chilean Pine" is a 40 node Cray XK6 hosted at the Atomic Weapons Estab-

	Dexter	**Chilean Pine**	**Shannon**	**Swan**
GPU	GeForce GTX 285	Tesla X2090	Tesla K20x	Tesla K20x
Architecture	Fermi	Fermi	Kepler	Kepler
GPU Chip	GT200B	GF110	GK110	GK110
GPU Clock (MHz)	648	1,150	732	732
# Active SMs	30	16	14	14
# SPs	8	32	192	192
Total CUDA Cores	240	512	2,688	2,688
Memory/Node (GB)	4	6	6	6

Table A.4: GPU Platform Summary

lishment (AWE). Each node consisting of one 16-core AMD Opteron 6272 CPU and one NVIDIA "Fermi" X2090 GPU, with 512 cores clocked at 1.15 GHz. Although an earlier technology generation to the contemporary systems detailed in the rest of this Appendex, "Chilean Pine" is the only Cray architecture available to the author containing a full range of OpenACC compilers.

A.4.4 Swan (Cray, NVIDIA GPU Cluster)

Commissioned: 2013

"Swan" is primarily a Cray XC series system, provided by Cray's Marketing Partner Network. A subset is configured as an XK7 consisting of 8 nodes, each with an 8 core Intel Xeon E5-2670 and an attached NVIDIA Kepler K20X, with 2,688 732 MHz cores and 6 GB of memory. OpenACC is available on Swan via CCE 8.3.0 and PGI 13.10. Alternative approaches available to application acceleration on the system are native implementations in OpenCL and CUDA.

A.5 Intel®Xeon Phi™

A.5.1 PillowB (AWE, Intel Xeon Phi Cluster)

Commissioned: 2010

Hosted at AWE, "PillowB" consists of 40 nodes of dual 2.1 GHz Intel Xeon E5-2450 [5] processors, each node having two Intel Xeon Phi 5110P cards.

	PillowB	**Compton**
Integrator	BULL	Intel
Chip Manufacturer	Intel	Intel
Processor Type	E5-2450	E5-2670
Clock Speed (GHz)	2.1 (2.9)	2.6 (3.3)
CPUs/ Node	2xSandyBridge 8-core	2xSandyBridge 8-core
Compute Nodes	40	42
Accelerator	KNC 60 core 5110P	KNC 57 core C0
Interconnect	IB	IB
Intel Compilers	Intel 13.1	Intel 13.1
MPI	IntelMPI 5.0.3	IntelMPI 5.1.0

Table A.5: Xeon Phi Platform Summary

OpenACC can be used on both the Intel Xeon and the Intel Xeon Phi via CAPS 3.3.2, using Intel OpenCL SDK v1.2.3.0. Likewise both Intel Xeon and Intel Xeon Phi have support for OpenCL, MPI and OpenMP. Specifically on the Intel Xeon Phi, with the release of Intel's Fortran Composer XE 2013 Update 2 (compiler version 13.1), Intel's Heterogeneous LEO model and Intel's implementation of OpenMP 4.0's new features for controlling execution on coprocessors can also be assessed.

A.5.2 Compton (SNL, Intel Xeon Phi Cluster)

Commissioned: 2012

Part of SNL's Heterogeneous Advanced Architecture Platforms (HAAPs), is a 42 node Intel Xeon Sandy Bridge based cluster where each compute node additionally contains two, pre-production (stepping C0) Intel Xeon Phi co-processor cards. The Sandy Bridge SKUs (Stock Keeping Units) are dual 2.6 GHz Intel Xeon E5-2670 [6] processors.

	Teller
Integrator	Penguin
CPU	AMD A10-5800K
CPU: # cores	1 x quad
CPU: Clock Speed (GHz)	3.8
GPU	Radion HD-7660D
GPU: Clock Speed (MHz)	800
GPU: # Active Compute Units	6
GPU: # SPs	64
GPU: Total cores	384
Compute Nodes	104
Memory/Node (GB)	16
Interconnect	Qlogic QSFP QDR IB
Compilers	PGI 13.4.0 CAPS 3.3.3 GNU 4.8.1
OpenCL	AMD APP SDK 2.8.0
MPI	OpenMPI 1.6.4

Table A.6: APU Platform Summary

A.6 AMD APU

A.6.1 Teller (SNL, AMD APU Cluster)

Commissioned: 2012

Also based at SNL is “Teller”: a cluster of AMD, second generation “Trinity”, AMD Fusion Accelerated Processing Unit (APU) processors. Each APU consists of an AMD A10-5800K (Piledriver) 3.8GHz Quad-core with one Radeon HD-7660D (Northern Islands) with on-die integration containing 384 x 800MHz cores. OpenACC is provided via CAPS 3.3.3, the only alternative for exploiting the Radeon is via raw OpenCL using AMD’s APP SDK 2.8.0.