Author:
Martineau, Matt J

Title:
On the Porting and Optimisation of Physics Simulations for Heterogeneous Parallel Processors

On the Porting and Optimisation of Physics Simulations for Heterogeneous Parallel Processors

Matt Martineau
University of Bristol, Merchant Venturers Building, Bristol, UK
m.martineau@bristol.ac.uk

June 3, 2019

Abstract

Modern science is increasingly reliant on computer simulations to model natural systems, and is limited by the available computational power. Modern supercomputers are regularly increasing in parallelism to meet the scientific throughput demands, while limited by power budgets and architectural restrictions such as heat emissions.

Those supercomputers now contain heterogeneous processors that range from CPUs that are latency optimised, and provide large complex cache hierarchies and DRAM, to GPUs that are latency hiding with many low power cores, and relatively simple caches and high bandwidth main memory. There is also a middle-ground offered by the Intel Xeon Phi, which is latency optimised and offers a modest number of low power cores with four hardware threads, a large but simplified cache hierarchy, and high bandwidth main memory. This thesis will consider the performance of all of these highly parallel processors, and the implications of the growing complexity of targeting modern processors.

Production physics simuations, of the kinds that simulate nuclear reactions, for instance, can often be monolithic, with millions of lines of code that can lack documentation and consistent coding style. Porting and optimising those applications to target modern supercomputers is a process of many choices, some with clearly defined options, and others requiring extensive investigation and research. Those choices are investigated in great depth in this thesis using a newly developed suite of exemplar applications that characterise important classes of physics applications: hydrodynamics, heat diffusion, and Monte Carlo neutral particle transport.

An informed choice of parallel programming model is essential to avoid inadvertently limiting future performance and portability. This thesis will consider some popular parallel programming models, and demonstrate their effectiveness and limitations in the context of the exemplar applications. The range of cutting edge algorithms for Monte Carlo neutral particle transport will be explored, and a novel approach to vectorising the application will be presented. With the search space of choices explored, a discussion is presented of those features of production applications often ignored in research codes, acknowledging the significant risks that are introduced with the complexity of real physics applications.

Dedication

I want to thank my supervisor, Professor Simon McIntosh-Smith, for his continued support and guidance throughout the entire post-graduate process. I would also like to thank Wayne Gaudin for his expert supervision, and enthusiasm for the craft. I have learnt a lot from both of them. Thank you to both of my examiners, Professor Stephen Jarvis, and Professor David May, for taking the time to read and critique my thesis, and providing insightful feedback. I am also grateful to the supervisors who mentored me on my internships: Carlo Bertolli with IBM Research at the T.J. Watson Resarch Center, and David Beckingsale and Richard Hornung at Lawrence Livermore National Laboratories. Thank you to the team at Intel, John Pennycook, Douglas Jacobsen, Jason Sewall, and Andy Mallinson for their continued collaboration on the `neutral` project. Finally, thanks to my wife, Sarah, who supported me in every way possible, making every step infinitely more enjoyable than if tackled alone.

Acknowledgements

Results presented in this research have been collected on the Swan XC50 supercomputer, where access was kindly granted by Cray Inc., as part of the Cray Marketing Partner Network. Results have also been collected on the Isambard test cluster, a GW4 collaboration, hosted by the Met Office. Extensive analysis was performed on the University of Bristol High Performance Group's Zoo testbed. Oxford University's Advanced Research Computing provided access to the IBM POWER8 system, Saffron. Simon Hammond at Sandia National Laboratories kindly arranged access to the Advanced Systems Technology Test Beds. Christopher Woods kindly setup and hosted the NVIDIA V100 GPU used in this thesis in the BlueGem compute cluster at the University of Bristol. Access to the Lawrence Livermore National Laboratories test clusters was provided by Richard Hornung. Alice Koniges arranged access to the Cori and Edison supercomputers at the National Energy Research Scientific Computing Center. This PhD was sponsored as part of a CASE converted DTP funded by EPSRC and the UK Atomic Weapons Establishment.

Author's Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the Universitys Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidates own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Signed:

Date:

Contents

List of Figures

List of Tables

Listings

Acronyms

ALE Arbitrary Lagrangian Eulerian.

AMR Adaptive Mesh Refinement.

AoS Array of Structures.

AoSoA Array of Structures of Arrays.

API Application Programming Interface.

ARB Architecture Review Board.

ASC Advanced Simulation and Computing.

AVX Advanced Vector Extensions.

BIOS Basic Input/Output System.

CBRNG Counter-Based Random Number Generation.

CFL Courant-Friedrichs-Lewy condition.

CG Conjugate Gradient.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

DOE Department of Energy.

DP Double Precision.

DRAM Dynamic Random Access Memory.

DSL Domain Specific Language.

ENDF Evaluated Nuclear Data File.

EOS Equation of State.

FLOP Floating Point Operation.

FMA Fused Multiply Addition.

FP Floating Point.

GCC GNU Compiler Collection.

GPU Graphics Processing Unit.

HBM High Bandwidth Memory.

HPC High Performance Computing.

ISA Instruction Set Architecture.

KNL Intel Xeon Phi Knights Landing.

LANL Los Alomos National Laboratories.

LCG Linear Congruential Generator.

LLNL Lawrence Livermore National Laboratories.

LOC Lines of Code.

MCDRAM Multi-Channel Dynamic Random-Access Memory.

MCNP Monte Carlo N-Particle Tranport Code.

MPI Message Passing Interface.

NUMA Non-Uniform Memory Access.

ORNL Oak Ridge National Laboratories.

PCI Peripheral Component Interface.

PDE Partial Differential Equation.

POSIX Portable Operating System Interface.

PTX Parallel Thread Execution.

SASS Streaming Assembler.

SDE Intel Software Development Emulator.

SIMD Single Instruction Multiple Data.

SIMT Single Instruction Multiple Thread.

SM Streaming Multiprocessor.

SMT Simultaneous Multi-Threading.

SoA Structure of Arrays.

TLB Translation Lookaside Buffer.

TRT Thermal Radiative Transfer.

WOC Words of Code.

Chapter 1

Introduction

Supercomputing is an essential component of modern scientific progress. Many areas of science reached the limits of analytical and numerical analysis on paper decades ago, and this pushed computer-assisted simulation to the forefront. The use of computers to solve complex mathematical problems spans the last century, and the prevalence of computational simulations in the sciences has lead to many scientists being directly involved in or responsible for the development of software projects.

The scale of the challenge from a computational perspective is astounding, and the continual increase in parallelism and architectural nuances increase the complexity greatly. To use any modern supercomputer, scientists are required to develop their code for parallel computation, notoriously one of the most challenging and error prone branches of software development. It is essential that the task is made as accessible as possible so that the majority of focus can be directed towards solving scientific problems. Over the last century, computing for science has grown from individual calculations on high speed single core processors, to computations spanning tens of thousands of nodes containing heterogeneous processors. The scientific applications can potentially span millions of lines of code, and can be expected to port to modern parallel processors and scale across millions of cores. A typical scientific workload might even involve multiple distinct packages co-operating in the solution of a system of equations.

Given a single persistent supercomputing architecture, it would be possible to develop scientific simulations while focusing purely on the computational concerns, optimising to the greatest possible extent for that particular platform. In reality, modern supercomputing resources are being constantly updated and replaced, to the extent that applications written for previous generations of CPU likely might not perform optimally on modern generations of CPU without tuning. The rapid rate of growth of computing in the sciences, and unpredictable technological changes, has introduced a multitude of problems for long standing scientific software applications.

With the repurposing of GPUs for general computing, compute architectures began to diversify even further, and targeting the new processors has become a highly challenging problem in of itself. Many supercomputers are now comprised of heterogeneous parallel processors, such as the 26000 NVIDIA V100 GPUs and 9000 POWER9 CPUs present in the world's fastest supercomputer, Summit, at Oak Ridge National Laboratory [154]. The Trinity supercomputer at Los Alomos National Laboratory will contain thousands of Intel Xeon and Intel Xeon Knights Landing CPUs. There is an expectation that the Department of Energy (DoE) simulations will be ported to both platforms; however, maintaining code bases for individual architectures

represents an unacceptable overhead, and so performance portable approaches are needed.

Many production scientific applications have been written to target clusters using the Message Passing Interface (MPI), which enables distributed computing and parallel execution on multi-core CPUs. The legacy codes written with MPI must be ported to enable threaded parallelism, which is in the best case an exercise in adding parallelisation to each computational loop, and in the worst case might require total redevelopment of the code and internal algorithms. Not only is the portability an important concern with porting legacy applications, but it is also imperative for scientific progress that the applications are not unduly inefficient. Enabling performance in large scientific software applications targeting modern parallel processors is a challenging area that requires an intimate understanding of the architecture, relevant algorithms, and the nuances of efficient parallel programming.

The size and structure of legacy applications makes it essentially impossible to perform agile experiments threading or optimising algorithms, without a large dedicated code team. The use of proxy applications has become the popular vehicle for such investigations, enabling research to quickly determine ideal algorithms, data structures, and parallel descriptions [63]. This thesis will concentrate on four new exemplar proxy applications that represent important classes of applications simulating physical processes: Eulerian and Lagrangian hydrodynamics, heat diffusion via conjugate gradient (CG) solve, and Monte Carlo neutral particle transport. Hydrodynamics and heat diffusion are quite general methods that can represent fluid motion and diffusive processes for a number of scientific areas. Monte Carlo neutral particle transport is more specific, and is particularly used in medical imaging and dosimetry, and reactor simulation [5] [133].

Although the principal focus of this thesis is the simulation of physical processes, it is expected that the techniques and concepts generalise to many areas of science, as the principles are relatively consistent. Most processes in science measure the phenomenon of change, requiring the numerical solution of partial differential equations, which is fundamentally the focus of the subsequent discussions. The expectation is that the work in this thesis will present important information about the state of existing parallel processors, optimisation techniques, and the performance portability of parallel programming models, in relation to applications that cover a sufficiently broad range of techniques for numerical solution of such PDEs. Several of the computational dwarves proposed by Asanovic et al. are represented within the thesis: Structured Grids, Unstructured Grids, Monte Carlo methods, and Sparse Linear Algebra [6].

The Structured Grid and Sparse Linear Algebra applications, Eulerian hydrodynamics and heat diffusion via CG solve, are well understood and have previously been shown to achieve good performance on modern parallel processors [61] [42] [112]. In this thesis it has been possible to use those exemplar applications to evaluate a number of modern parallel programming models, considering the impact from the perspective of performance, portability and productivity. The Monte Carlo neutral particle transport problem was first published about in 1954, and the performance of the application on modern parallel architectures is an important and challenging topic [76]. During this thesis it has been possible to discover optimal approaches to parallelising Monte Carlo neutral particle transport applications on CPUs, GPUs, and KNLs. This required extensive experimentation at the algorithmic and data structure level, and the development of a novel sort-free algorithm to enable vectorisation on modern parallel architectures. Unstructured grids have been well considered in the literature, but the particular application considered in this thesis is Lagrangian hydrodynamics using a subcell discretisation for arbitrary polyhedra, which includes some interesting subtleties that will be discussed.

An challenge with all of the applications, but particularly Monte Carlo neutral particle

transport, is problem dependence. It is shown throughout that the results determined with proxy applications are greatly affected by changes in the target problem, and each of the applications is considered for a range of different parameters to account for this. There is also the challenge of faithful representation of the production application features, as missing important features in a proxy application could potentially lead to optimisations and parallel descriptions that do not scale into real applications. Where possible, the potential features of each of the exemplar applications are considered.

1.1 Contributions

The following contributions are complementary towards the core aim of presenting a thorough treatise of concerns related to the porting of production scientific applications.

1.1.1 The `arch` Project: Physics Proxy Applications

To support this thesis, a suite of physics proxy applications have been developed under a common architectural framework and permissive MIT license named the `arch` project[1]. Each of the proxy applications represents a reduced feature-set proxy for production applications solving a multitude of scientific problems, and analysis of those applications is presented in Chapters 5 to 7. The supporting infrastructural project, `arch`, provides cross-cutting concerns, such as MPI communications, memory management, and support for structured and unstructured meshes. Although the suite was intended to support and motivate the discussions in this thesis, it is becoming adopted as a tool for performance optimisation and algorithmic studies by the wider community [150].

1.1.2 Analysis of Performance Portability for Physics Applications

Performance portability has been shown to be a major challenge facing the future of large-scale scientific simulation, and achieving performance portability has been described as the gold standard for programming environments [82]. For scientific developers, performance portability starts with the choice of parallel programming model. Choosing an appropriate parallel programming model will have vast implications for the success of a large scientific applications. There are many models available, each presenting different characteristics and trade-offs, making the decision-making process highly challenging for scientific application developers. This thesis considers some of the most successful parallel programming models, OpenMP, CUDA, OpenACC, and RAJA, and their impact on performance, portability and productivity. The thesis contains recommendations for best practices when using performance portable parallel programming models, based on experiences porting the `arch` applications (Chapters 5 to 7).

1.1.3 Benchmarking of HPC Architecture Performance

In some cases there are publicly available details regarding the low level performance of particular processors, but it is not always possible to find this information for specific SKUs. Further, in some cases the vendors do not publicly expose such information or provide benchmarking tools. In order to reason about the performance of the applications, particularly the Monte Carlo neutral particle transport application (Chapter 5), it was necessary to benchmark the fine

[1] https://github.com/uob-hpc/arch

details of the architectures. This benchmarking process considers details like memory latency, memory bandwidth at all cache levels, and random memory access performance (Chapter 4). The results supported later reasoning and modeling of the performance of the `arch` applications.

1.1.4 Optimisation of Monte Carlo Neutral Particle Transport

This thesis considers a subset of proposed parallel computational patterns, their performance patterns, and the techniques required to optimise them on the most modern supercomputing resources. In particular, the best algorithms are found for the Monte Carlo neutral particle transport problem targeting NVIDIA GPUs, demonstrating impressive performance in spite of the divergent code (Chapter 5). Poor performance due to lack of vectorisation and challenging issues of latency on the CPU and KNL are improved through the development and optimisation of a novel sort-free algorithm for vectorising the particle tracking loop. The extent of problem dependence is demonstrated using a number of different case study problems, and results are presented in such a manner that they should be relevant to the transport of any neutral particle.

1.1.5 Analysis of Complex Production Concerns

Porting and optimising scientific software applications requires a rigorous consideration of key algorithms, often requiring the use of proxy applications to reduce the computational complexity to a minimal level so that a computer scientist can investigate optimisations. The success of those proxy apps is measured on their ability to translate optimisations back into their production counterparts, demanding careful consideration of the included features chosen as a subset.

Proxy apps tend to exclude complex features of production applications, and this thesis posits that this can inhibit the generalisation of results in many cases. For instance, there are few scientific simulations that handle only individual materials, with most instead requiring complex interfaces, which is something rarely included in proxy applications. The treatment of multi-material interfaces is a significant burden to the computer scientist but has major implications for the portability and performance on modern architectures. This issue is explored alongside the `arch` applications in Chapter 8.

1.2 Structure of Thesis

In this chapter the motivation for this thesis has been presented, and the subsequent chapters in this thesis address the following problems:

- **Background (Chapter 2):** This chapter includes the fundamental concepts of parallel and distributed computing, as well as some basic details regarding computational solution of partial differential equations necessary to follow the subsequent sections.
- **Programming Models and Performance Portability (Chapter 3):** Parallel programming models are an important aspect of porting applications to use modern hardware. This chapter provides a light background to the parallel programming models OpenMP (3.0 and 4.5), OpenACC, RAJA, and CUDA. A discussion about the current understanding and literature relating to the state of the art in performance portability is then presented.
- **HPC Architecture Performance (Chapter 4):** In order to optimise for the considered parallel processors, it is important to understand the performance characteristics of the

processors themselves. In this chapter, empirical results are presented for many aspects of the processors. This was either because accurate data was not available from the hardware vendor for the particular processor variant, or the results are markedly different between the marketed data.

- **Monte Carlo Neutral Particle Transport (Chapter 5):** This is the first application optimisation chapter, and focuses on the Monte Carlo neutral particle transport application `neutral`, which is part of the `arch` project. A thorough performance investigation is undertaken at the algorithmic, data structure and parallel programming level. An optimal GPU implementation is developed that greatly improves upon the performance of the traditional methods on the CPU. A novel algorithm is developed that enables vectorisation of the particle tracking loop without requiring sorting of particles. The problem dependence of the application is also considered, and a discussion is presented about those features not considered that could result in different performance characteristics.

- **Heat Diffusion via a Conjugate Gradient Solver (Chapter 6):** This chapter considers the performance of the conjugate gradient solver, through the `arch` application `hot`. In particular, the performance portability of the application with respect to modern parallel programming models is explored. The issue of solving problems with cache-resident meshes is explored with respect to the parallel programming models, to show that there are significant overheads present in the models that might show up in other application domains.

- **Hydrodynamics (Chapter 7):** Hydrodynamics is a particularly important application class that is used in the majority of areas of science and engineering. This chapter will explore two hydrodynamics proxy applications from `arch`, the 2D structured Eulerian hydrodynamics application `flow`, and the 3D unstructured Lagrangian hydrodynamics application `lags`. The `flow` application will be considered in terms of parallel programming models, and the capability of each to provide performance portability for the application. The `lags` application is used to explore the space of unstructured meshes and subcell computations, and the implications on parallel performance.

- **Production Challenges (Chapter 8):** This chapter presents some critical analysis of the work in the preceding chapters, by considering the impact of those features that might be present in production applications but that were not directly optimised for. Results are presented for a set of benchmarks directly targeting the complex production problem of multi-material data structures, and a consideration for how those features are extended into the dynamic structures of Eulerian flow fields. Through this discussion it is possible to consider the potential impact on the final efficacy of results from using proxy application in performance studies.

1.3 Reasoning for `arch`

The principal focus of this thesis will be directed towards a number of exemplar applications that have been chosen due to their relevance to the wider area of simulating physical processes. Each of the exemplar applications has been developed from scratch specifically for the project described in this thesis. There are a number of reasons that made it essential to use new

applications rather than rely upon existing applications, some specific to the application and others more generally.

- An open source Monte Carlo proxy application was not available with the particular characteristics required for the performance studies performed. Towards the end of the thesis project, the Quicksilver proxy application was released by Lawrence Livermore National Laboratories, which might have been a suitable candidate but the `neutral` application was already developed, and the majority of the research was already published.

- Many different hydrodynamical simulations were developed, including Eulerian, Lagrangian, and ALE. In order to maintain a fair comparison it was essential that they were all consistently developed and this cannot be offered by existing proxy applications. The proxy applications CloverLeaf and PENNANT offer similar features to the Eulerian and Lagrangian applications in the `arch` project, but are written in different programming languages and are many times larger than `flow` and `hot`. One important characteristic that was explored with `lags` was the concept of subcell computations, which had a significant impact on the performance and was not available in the alternative applications.

- Developing the applications to rely upon a single common infrastructural layer means that it is possible to make commentary about the issues of hosting multiple physics packages within a single framework. This will be shown to have important consequences in terms of portability in Chapter 8.

- The common infrastructural layer meant that the core computational code of each application was generally limited to 1000 lines of code, except for the ALE application `hal3d`, which was purposefully developed to consider the issue of large applications. This means that experiments could be performed in much less time than would be required to port larger applications such as the hydrocode PENNANT (5000 LOC) and the Monte Carlo application Quicksilver (13000 LOC).

It can be noted that the proxy applications CloverLeaf, TeaLeaf, and PENNANT were all used as part of this project and many of the relevant publications are relative to those applications. Although the `arch` applications have been developed from scratch using open source methods, the individual applications were optimised using already published techniques to avoid duplication of efforts.

Chapter 2

Background

2.1 Parallel Computing Architecture

Moore's law states that the number of transistors per chip grows at an exponential rate with a constant cost, doubling roughly every 18 months [118]. Dennard et al. observed that reducing the size of transistors meant that voltages could be decreased, thereby maintaining a constant power based on area rather than density [38]. More recently, since around 2006, transistors have become so small that leakage and threshold voltage restrictions have ended Dennard scaling, limiting the potential for single core performance.

There are a number of architectural adjustments that can aid in reducing the impact of heat while allowing an increase in performance. This section includes a succinct foundation in parallel computing to form a basis for the subsequent discussions in the thesis.

2.1.1 Instruction Pipelining

Instruction pipelining exploits inherent parallelism in the architectural processing of machine instructions.

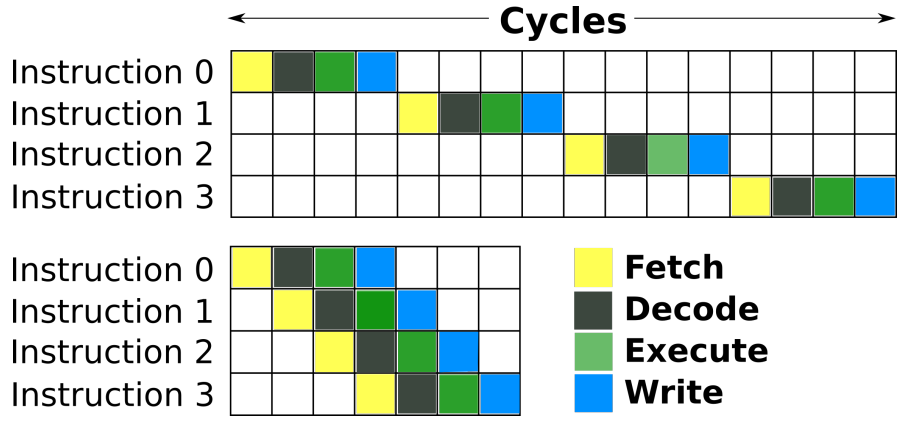

Figure 2.1: Instructions issued with (bottom), and without (top) instruction pipelining. The colours represent stages required to issue full instruction, e.g. fetch, decode, execute, write.

In modern processors, machine instructions are broken into micro-operations, which are the dependent stages in an instruction [60]. The top of Figure 2.1 depicts the scenario that

4 instructions are issued, requiring 16 micro-operations. Each stage, the coloured squares, of each instruction takes one cycle to complete before the next stage can occur, a throughput of 1 instruction per 4 cycles.

The benefits of pipelining can be realised from the observation that this approach under-utilises the available pipeline stages, as when decoding, for instance, the fetch and other units will sit idle. To improve the throughput, each micro-operation can be added to a pipeline, allowing overlapped processing of those micro-operations of independent instructions. The bottom of Figure 2.1 depicts 4 machine instructions issued to a processor that supports pipelining.

It can be seen that following an initial latency of 3 cycles, the 4 deep pipeline can keep all four pipeline stages active during a single cycle. Given an increasingly long stream of instructions the 4 deep pipeline can asymptote to a throughput of 1 instruction per cycle.

2.1.1.1 Superscalar Processing

Superscalar processing is an architectural feature of many processors, where multiple instructions can be issued within a single clock cycle [74]. Of course this requires that pipeline stages are duplicated, but it allows several instructions to be passed into the pipeline on each cycle, increasing instruction throughput. This is another important feature of modern processors that must be accounted for when modelling and analysing performance.

2.1.2 Vector Processing

Vector processing is an architectural design where vector registers can be filled with multiple operands, and arithmetic units operate on the set of operands with a single instruction, proven in early vector processors like the Cray-1 [142]. The benefit to this approach is that algorithms often apply the same instructions to multiple operands, and if those instructions are independent there is an inherent parallelism that vector processing exploits.

In modern CPUs, most core designs include vector registers and support SIMD instructions that can perform, for instance, a fused multiplication and addition on 16 words in a single cycle. For algorithms that are sensitive to computational performance this is a significant increase in throughput.

Code Sample 2.1: Example of vectorisable loop in C.

```
// C loop
for(int i = 0; i < 8; ++i) {
  a[i] = b[i] * c[i];
}
```

The C loop in Code Sample 2.1 is a canonical loop with a small constant trip count. On a scalar processor the loop iterations would need to be handled sequentially.

Code Sample 2.2 shows the C loop's x86 assembly code, where each individual element of the set of arrays is multiplied and stored in turn. Due to the small, constant trip count, the loop could be fully unrolled by an optimising compiler at high optimisation levels, removing the loop control instructions. This could potentially allow multiple instructions to be pipelined, taking advantage of the superscalar nature of the target processor. Considering that 'a', 'b' and 'c' are 8 `floats` in length, the arrays would be situated in L1 cache on current CPUs, and so the number of instructions is theoretically important to the performance of this loop.

Code Sample 2.2: Scalar loop in the x86 instruction set.

```
// Compiles to x86 scalar loop
..loop:
  movss (%rdx,%rax,4), %xmm0    // Move b[i] to register xmm0
  mulss (%rcx,%rax,4), %xmm0    // b[i] * c[i] (result in xmm0)
  movss %xmm0, (%rsi,%rax,4)    // Store result in a[i]
  incq  %rax                    // Increment counter 'i'
  cmpq  8, %rax                 // Compare counter 'i' with 8
  jl    ..loop                  // Loop back if 'i < 8'
```

Code Sample 2.3 depicts the same code but compiled with the Advanced Vector Extensions 2 (AVX2), an extension to the x86 instruction set for SIMD parallelism. There is no longer a loop, as the AVX2 instruction set includes 256-bit instructions capable of processing 8 floats with a single instruction. Given a possible instruction latency of 1 cycle for the multiplication, the vector processing approach is highly effective in algorithms where the operands are readily available.

Code Sample 2.3: Vector instruction in AVX2 instruction set.

```
vmovups (%rsi), %ymm0         // Move b[] to register ymm0
vmulps  (%rdx), %ymm0, %ymm1  // b[] * c[] (result in ymm1)
vmovups %ymm1, (%rdi)         // Move result to a[]
```

It is only correct to vectorise loops where there are no loop carried dependencies, or the loops can be transformed to have independent work for vector processing. Modern optimising compilers use a range of transformations to ensure that the majority of sane code is vectorised automatically, but there are many situations where programmer intervention is required. Enabling vectorisation or encouraging auto-vectorisation is discussed in the chapter discussing Programming Models (Chapter 3), and throughout the subsequent chapters dealing with individual scientific applications (Chapters 5 to 7).

2.1.3 Multi-core Computer Architecture

Once the practical limits of single core designs had been reached, the next source of processing power growth came from increasing core counts. In 2001, IBM designed the first dual-core processor, demonstrating the feasibility of placing multiple cores on a single die [156].

Increasing the number of cores in a CPU allows the processing power to be increased while maintaining consistent clock speeds and staying within a reasonable power envelope. There have been many different approaches to architecting multi-core processors: varying core counts, speeds, memory locations, and other factors. Once an additional core is added to a processor, the complexity of the architecture and programming approach is significantly increased, and modern CPUs can contain hundreds of cores.

2.1.3.1 Cache coherency

The introduction of multiple cores means that cache coherency mechanisms need to be added to ensure that the cores do not read or write incorrect data [60]. In Figure 2.2, a hypothetical multi-core architecture is presented, depicting the cores and cache hierarchy. Each core is directly connected to a private L1 cache, and it is possible for data to be duplicated from DRAM into both L1 caches at the same time.

In the event that there is duplicate data and one thread wants to write while the other wants to read from the same address, then there is a data race, but one which cannot be predicted as a programmer. The hardware has to include some mechanism for updating all caches when one of the cores attempts to change a value in an individual cache, which is known as write propagation.

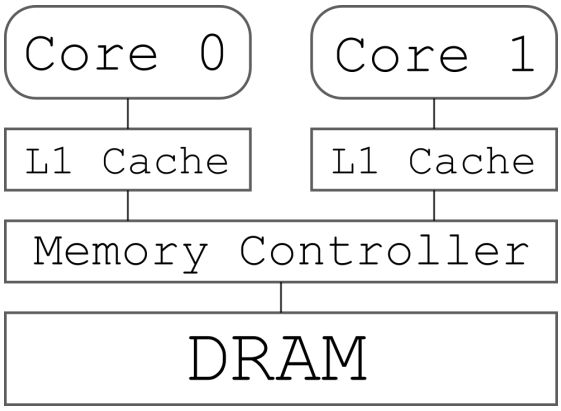

Figure 2.2: A cache layout for a hypothetical CPU architecture.

If both caches contain a duplicate memory entry and subsequently write back to that memory address there is a data race. The hardware is only responsible to ensure that those writes are fulfilled in cache in the original order requested. As such, if Core 0 writes a value to cache and Core 1 subsequently writes to the same location, the value residing within cache after the operations are complete must be the value output by Core 1. It is the province of the programmer to ensure that such output dependencies are avoided.

2.1.3.2 Multi-socketing

Many modern high performance computing platforms include two or more sockets within a single node, which is known as multi-socketing. The most common configuration in the largest supercomputers on the Top500 list[1] is two CPUs per socket, known as dual-socketing. This doubles the computational power available within one shared memory space, although Section 2.1.3.3 will discuss issues that arise from this partitioning. The two sockets can be programmed using the same techniques that are available for multi-core programming, as both sockets share main memory, and the operating system manages the attribution of threads to cores.

Multi-socketing introduces additional complexity in terms of selecting affinities on a supercomputer, as it is important to ensure that computational work is correctly balanced. If the programming environment provides control over the affinity then it is often best if the work can be evenly distributed across the cores of the pair of CPUs with a one-to-one correspondence between threads and cores. The ordering of threads on cores might have application-specific or problem-specific implications on the performance of a process.

2.1.3.3 Non-Uniform Memory Access

Depending upon the configuration of the architecture, levels of the memory hierarchy might exist at different distances between physical cores. The practice of dual-socketing CPUs means that pages of data can be allocated in the DRAM of a socket but accessed by cores on another socket. There will be a significant increase in the latency of accessing data from another socket. Taking care to bring data as close as possible to the cores that will be using it during a program's execution is an important optimisation techniques available for parallel programming, and careful allocation of data to account for NUMA might be necessary [174] [84].

Although Figure 2.2 depicts two separated caches, it is important to note that the cores would not access each other's L1 cache, rather the data would be duplicated as required. If a

[1] https://www.top500.org/

shared last-level cache were added to Figure 2.2, the most usual approach would be to layout the cores so that they are equidistant from that cache. In spite of this, there are processors that are architected with variable distances between cores and levels of the memory hierarchy, even within a single socket, for instance the AMD Ryzen ThreadRipper 1950X maintains two NUMA nodes on the same chip [33].

2.1.3.4 Simultaneous Multithreading

Simultaneous multithreading (SMT), also known by the Intel-specific term hyperthreading, is another architectural design technique that can improve performance without altering the clock speed [162]. Conceptually, SMT allows each core of a CPU to be considered as multiple logical cores, with shared execution resources, such as arithmetic units, but individual state, such as registers.

The consequence of this partitioning is that whenever the core stalls to wait for operands, the other logical cores can fill the pipeline with requests to utilise the unused execution resources. Essentially, the additional logical processors are there to fill bubbles in the pipeline, potentially increasing the overall efficiency of an application.

Architecture	Cores	SMT
Intel Xeon Skylake (Platinum 8176)	28	2
Intel Xeon Phi Knights Landing (7210)	64	4
IBM POWER9	12 or 24	4 or 8

Table 2.1: Number of simultaneous threads per core.

Table 2.1 shows some examples of SMT counts in different multicore processors available today. The logical cores are made available to the operating system and can be programmed in the same manner as multiple physical cores. Enabling SMT in software is relatively straightforward, but the determination of the optimal choice of SMT utilisation has to be performed on a per-application basis. One of the main challenges with correctly leveraging SMT is expressing the affinity between threads and cores, but modern advances in standards such as OpenMP are greatly improving the situation. Throughout this thesis there will be results of using SMT across a range of architectures and application types, and a case where hyperthreading is particularly advantageous in Chapter 5.

2.1.4 Distributed Computer Architecture

Another important form of parallel processing comes from connecting multiple independent nodes together, scaling out to form a network of processors. The nodes can be comprised of multiple heterogeneous parallel processors, for instance, the Summit supercomputer nodes contain 6 NVIDIA V100 GPUs and 2 POWER9 CPUs. If software can be designed to take advantage of those processors in parallel, it is possible to greatly increase scientific throughput by the connection of large clusters. In terms of cluster size, it is possible to scale to the limits of financial budgets and power consumption commitments, as seen with the current drive towards *exascale* computing [8] [114].

Programming distributed computer architectures generally involves the use of a message passing library, the most well known being the Message Passing Interface (MPI), which will be

discussed in Section 2.2.1.3. There are major challenges introduced by distributing data across different nodes of a supercomputer, and the current drive towards exascale computing means that the problems observed at scale will become more prevalent in the future.

Supercomputer	Nodes
Sierra (LLNL)	4320 nodes (2 x POWER9 CPU and 4 x Volta GPU)
Summit (ORNL)	4600 nodes (2 x POWER9 CPU and 6 x Volta GPU)
Trinity (LANL)	9436 nodes (Haswell CPU), and 9984 nodes (KNL)

Table 2.2: Figures for the newest Department of Energy (DoE) supercomputers, for Lawrence Livermore National Laboratories (LLNL), Oak Ridge National Laboratory (ORNL), and Los Alomos National Laboratories (LANL).

Table 2.2 shows the node counts and processors in the newest Department of Energy (DoE) supercomputers [12] [64] [166]. There is an expectation that codes written for one supercomputer should execute and scale adequately on each of the supercomputers with minimal changes, which is a major challenge given the volume of resources in each machine.

2.1.5 Many-core Computer Architecture

Many-core architectures, while essentially an extension to the multi-core processor approach, are treated separately as they introduce new programming paradigms and performance characteristics.

2.1.5.1 Graphics Processing Units

Graphics Processing Units (GPUs) were developed for the graphics processing market and support a 3D rendering pipeline, manipulating images and outputting them to a display adapter. The key to the success of those architectures was that they were fast enough to support real time rendering, and this was achieved by using many simplified low power cores that can perform the same task on many pieces of data.

With some alterations, and the development of appropriate APIs, it was possible to leverage the GPU processing power to handle general purpose computation. This introduced many-core processing as an alternative approach to traditional CPU-based supercomputing, for those applications that could take advantage of the particular style of parallel processing. Today, GPUs have been shown to be highly capable processors for handling scientific simulation and machine learning, which is the reason that they feature in two of the main DoE pre-exascale supercomputers, as shown in Table 2.2.

NVIDIA currently leads the market in compute on GPU, offering processors specifically tuned for computational workloads, with high bandwidth memory and double precision compute throughput. GPUs are considered one solution to the performance problems of exascale computing, as they support high FLOP-per-watt, allowing greater performance for a particular power budget. In the June 2018 Green500 list, 7 of the top 10 supercomputers were comprised of NVIDIA GPUs [154].

The most recent NVIDIA GPU, the NVIDIA V100, is comprised of 80 streaming multiprocessors. A V100 streaming multiprocessor includes 4 warp schedulers, each containing 16 FP32 units, for a total of 5120 FMAs per cycle. GPUs were designed as separate processing

components that are connected to the CPU via a PCI connection, or more recently, a high speed connection called NVLink [50]. As such, programming GPUs falls into the realm of co-processing, where the CPU is used as a host device that *offloads* commands to the GPU for processing.

The CUDA programming API supports this with the CUDA C/C++ extensions, which results in a kernel oriented language with similarities to programming shaders. Many new issues arise when attempting to port existing applications to take advantage of GPUs, including finding large parallel data streams in existing algorithms, minimising data movement, and direct programming of shared caches. The introduction of GPUs and other accelerators has also introduced many problems for performance portability, which will be explored in Chapters 5 to 7.

2.1.5.2 Intel Xeon Phi

Since the rise of popularity of GPUs for computational processing, Intel has attempted to replicate the approach with their Xeon Phi line of processors. The Xeon Phi CPUs are closer to GPUs in that they use a higher count of lower clocked cores than their server grade Xeon CPUs; however, they are still similar to the Xeon CPUs in that they have SIMD units and many other features not present in NVIDIA GPUs.

The newest Xeon Phi, the Knights Landing (KNL), has 64-72 cores with 4 hyperthreads and two AVX-512 SIMD units per core. As such, a 72 core KNL can process 2304 FMAs per cycle. One of the major benefits of the Xeon Phi over CPUs is the inclusion of high bandwidth memory called MCDRAM. At the time of writing this is not a feature of the Xeon CPUs, and offers a significant performance improvement for some applications. When programming a KNL, it is generally possible to compile code that is written for a Xeon CPU and expect it work on the KNL without changes. In practice, this is not necessarily the case, and this thesis will demonstrate that there are implications on the performance of the resulting application that mean it is often necessary to perform extensive optimisation to fully exploit the KNL.

2.2 Parallel Software

Transforming sequential programs into robust, correct, portable and performant parallel applications is a complex discipline with many pitfalls.

2.2.1 Fundamental Approaches to Parallel Programming

As discussed in the preceding sections, there are many different architectures and there have been a number of different approaches developed to target them. There are three broad subcategories that can be used as umbrella terms to distinguish different approaches to embedding parallelism into an application: threads, tasks, and message passing. Although each of the terms describe different approaches to parallel computation, they have a number of similarities.

2.2.1.1 Threads

The term 'thread' can be used at varying granularities, the definition in this thesis will be:

> *"Threads are independently executable subsets of a process that share memory but maintain some private state."*

As such, threads are a software concept that encapsulate instruction streams that can be interleaved on a single processor core, or performed in parallel on multiple cores [105]. Considering the simplest case of threading on a multi-core CPU, an individual thread can be bound to each of the available cores, and each thread can concurrently perform independent work. Section 2.1.3.4 showed that threads can also be pinned to the logical cores of a CPU when the hardware provides SMT.

Threads are managed by the programmer using some API, for instance, POSIX threads (pthreads), OpenMP, etc. It is necessary for the programmer to consider the correctness of an application based on the non-deterministic nature of the thread execution, and their use of shared memory.

Code Sample 2.4: Example of a simple parallel loop.

```
parallel_for(int i = 0; i < n; ++i) {
  a[i] = b[i];
}
```

Code Sample 2.4 demonstrates the use of a hypothetical loop parallelism construct called '`parallel_for`' to execute a copy loop in parallel. The implementation would need to divide up the iteration space of the loop, the set of iterations from 0 to n - 1, so that each thread could operate independently on a subset.

In many cases it is not possible to parallelise every line of a program, with traditional parallel programming focusing upon loop-level parallelism. In early implementations of OpenMP, threads would be spawned and de-spawned after each structured block executed in parallel. Modern implementations typically use thread pools to remove those overheads.

2.2.1.2 Tasks

As with threads, tasks can be considered at different granularities. Coarse grained tasks might encapsulate whole sections of a parallel program, for instance, loops, algorithms or even entire physics packages can be wrapped into tasks. Alternatively, tasks can be used for fine-grained parallelism, where the iterations of a loop can be converted into tasks with data dependency and ordering constraints, or tasks can be used to construct complex graphs for applications with tree-based structures.

Tasks can offer increased productivity by mapping more directly to the natural description of an algorithm, especially in the case of tree-based applications [7]. Asynchronous tasking also might offer some benefit for load balancing and resilience across large exascale platforms [28]. The challenge with tasks is that they require the use of a scheduler that dequeues tasks in an optimal order. Of course, this introduces some overhead on a single node, and even more when considering node-to-node scheduling. The Exascale Compute Project (ECP) includes the subproject 'PARSEC: Distributed tasking for exascale', which aims to investigate the development of an exascale tasking execution model [41].

2.2.1.3 Message Passing

Distributed architectures cannot share memory in the traditional sense, which means that other mechanisms are required to either emulate shared memory across the network, or use message passing to communicate data. Fundamentally, applications can use message passing as a means to send data from one shared memory space to a separate shared memory space. The most

common message passing API is the Message Passing Interface (MPI), which is a mature open standard adopted by the majority of scientific applications for inter-node communication [43].

It is also possible to abandon the concepts of threads and tasks in favour of using message passing within a shared memory environment, for instance, using MPI to communicate between the cores of a multi-core CPU. In fact, many scientific applications have been written using MPI only, and limits on the scaling of message passing in particular applications has fueled the desire to exploit shared memory.

2.3 Parallel Performance

A core focus of this thesis is on the performance of parallel applications. The subsequent chapters will demonstrate that it is a multi-faceted issue with a number of contradictory practices and theories between different application types, based on their performance characteristics. This section will introduce some of the key performance optimisation concepts and nomenclature.

2.3.1 Amdahl's Law

Amdahl's law succinctly states that the scalability of an application when given additional resources is limited by the part of the system that does not benefit from those resources [3].

$$S = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.1}$$

Equation 2.1 gives an expectation of the speedup possible if the proportion p of the application speeds up by s. The most important implication for optimising parallel applications is that it motivates the parallelisation of the most expensive parts of an application. In many scientific simulations, parallelisation efforts will focus on the core solvers, or computational kernels.

2.3.2 Limiting Bounds

One of the most important factors in performance optimisation for parallel applications is recognising and improving upon the limiting factor of an application. For each of the applications discussed in this thesis, significant time is devoted to understanding the bounds of an application or algorithm, taking into account the fact that the bounds might be dynamic in some cases, changing as the algorithms or problems are tuned. Some of the key examples of limiting bounds that will be discussed are:

- **Compute bound:** Algorithms that have a large number of computations compared to memory operations might be limited by the computational performance of the hardware. The application performance will be restricted by the maximum number of floating point instructions per second (FLOP/s) achievable on a particular system.
- **Memory bound:** The performance of computation has been improved dramatically over the past decade, and memory bandwidth has not improved to the same extent [108]. Many scientific applications are memory bandwidth bound, as they process large sets of data that cannot be maintained within fast caches. Applications can also be memory latency bound, where the time it takes to perform individual memory accesses limits the performance of the application. Some applications can be memory footprint bound, where

the memory capacity required to solve problems at the desired accuracy is beyond the available resources.

- **Communication bound:** If an application spends the majority of its time waiting for memory to be communicated between different memory spaces, then the application can be considered communication bound.

This list is not exhaustive and there are many strategies that can be adopted for each of the different computational bounds. One route to improving performance is purchasing new hardware that offers improved performance for the limiting characteristic; for example, choosing processors with the highest achievable FLOP/s for compute bound applications. It is also sometimes possible that algorithms can be adapted or replaced within an application to change the computational bound, potentially improving throughput. There has been speculation that computational methods for the sciences will need to focus on compute-bound approaches in order to compensate for the lack of progress in memory performance relative to the compute performance [10].

A useful formal tool for the analysis of performance is the roofline model, which can accept machine and application parameters and predict what aspect of the target architecture is the limiting factor [173]. Later work by Ilic et al. extended the roofline model to be cache-aware, greatly improving its efficacy on modern cache-based architectures [68].

The interoperation between diverse processor architectures, and algorithms in applications, means that it can be challenging to determine a bound at all. Many low-level details of modern architectures are not well documented beyond marketing, and algorithms can be long and complex, making performance modelling of their behaviour cumbersome and inaccurate. Throughout this thesis, performance analysis will be supported by the empirical results of benchmarks to better understand observed performance characteristics.

2.3.3 Scaling

One of the figures of merit of parallel applications is their ability to *scale*. This term could refer to a vector loop's propensity to take advantage of increasing vector widths, or an applications' ability to scale up to thousands of CPU cores. This section briefly discusses varying perspectives of application scaling.

2.3.3.1 Vector and Thread Scaling

Parallel programs targeting modern processors can scale at the thread and vector level. For compute bound problems it might be expected that a roughly linear increase in performance is seen when introducing additional cores or widening SIMD units. For memory bandwidth bound problems, it is not unusual for sub-linear scaling when increasing cores and SIMD widths, as vectorisation benefits are obscured by the large overheads for fetching data from main memory. Further, as memory bandwidth is a shared resource, scaling is limited as the core count increases. Examples of this will be presented in Chapters 6 and 7.

2.3.3.2 Inter-processor Scaling

Regardless of how an application scales at the vector and thread level, it can have vastly different scaling performance when increasing the number of distributed memory resources. This is a key

problem when programming for supercomputers as the performance of an application at scale is limited by its propensity to amortise the costs of communicating memory between non-shared memory spaces.

There are two perspectives on inter-processor scaling that can offer different insights into the efficacy of the optimisation routines of a particular application, *strong scaling* and *weak scaling*. Strong scaling fixes the problem size, and adds additional resources to speedup the calculation of that particular problem, which relates to Amdahl's law [3]. Weak scaling is where the size of the problem proportionally increases with the resources, for instance, doubling the problem size when going from 1 to 2 nodes, which relates to Gustafson's law [56].

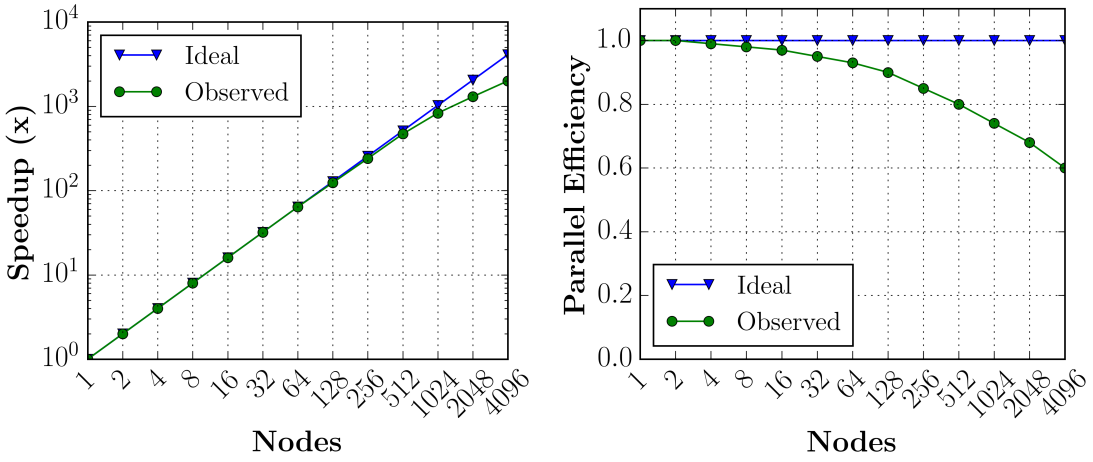

Figure 2.3: Hypothetical scaling graphs for strong scaling (left) and weak scaling (right).

Figure 2.3 depicts strong scaling and weak scaling graphs for a hypothetical application scaled from 1 node to 4096 nodes of a supercomputer. The example strong scaling graph is plotted as a log-log graph of node count vs speedup, and it shows that the application scales nearly perfectly up to 32 nodes, but that the performance begins to drop well below ideal by the time the problem is run on 4096 nodes. There can be many causes for a performance profile like this, but the most likely issue is that the level of scaling has resulted in each node having a small chunk of the overall problem, and communication costs are no longer amortised by the computation.

The weak scaling graph on the right of Figure 2.3 shows a greatly reduced parallel efficiency once executed on 4096 nodes, which demonstrates that the communication costs are increasing with the introduction of additional nodes. There are many potential reasons for this behaviour, but some interesting examples are: (1) the application requires some all-to-all communications which can rapidly limit parallel scaling, or (2) the performance of the network degrades due to some nodes being located in different racks of the supercomputer.

2.4 Numerical Simulations

It is generally intractable to simulate physical processes perfectly, and instead physicists employ approximations in their numerical models. Numerically solving equations describing physical processes requires careful handling of those approximations, and the methods of solution introduce a number of issues of accuracy and efficiency. The algorithms discussed in this thesis will

not be altered to improve their numerical properties, but some understanding of the fundamental issues in numerical methods is useful to the dialogue and will be presented in this chapter.

Further, scientific applications are intended to serve a practical purpose, which means that many features are required to interpretation and validity of the results. There are a number of issues including visualisation, file handling, reproducibility, etc. that are not discussed in this section, but are briefly discussed in Chapter 8.

2.4.1 Partial Differential Equations

In scientific simulations, the general goal is to accurately describe or predict some process or processes involving rates of change, potentially evolving in time or stabilising to some steady state. Such processes can be mathematically described using a system of partial differential equations (PDEs), an example of which is the wave equation (Equation 2.2) [86]. There are few practical cases where PDEs can be solved using a closed form solution, rather numerical methods are typically employed. The cost of numerically solving PDEs for real life processes can be high, and PDEs must be formulated in a manner in which they can be solved by a computer efficiently.

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \tag{2.2}$$

It is often the case that the PDEs, while accurate in their description of some approximation of the truth, require additional computational fixups to account for phenomena not captured by the approximations. An example that will be discussed in Chapter 7 is the numerical fix-up of introducing artificial viscosities when solving the inviscid Euler equations of hydrodynamics. Without the numerical fix-up, shock boundaries are discontinuous and lead to unphysical oscillations, that comes about from the fact that the continuum is discretised and so does not perfectly represent the shock boundaries. Rather than an accurate representation of the physical concept of viscosity, the resolution is often a more ad-hoc adjustment to the momentum at discontinuities, introducing steep but numerically resolvable gradients that do not result in oscillations [95].

2.4.1.1 Boundary Conditions

Due to the finite simulation of some physically continuous space it is necessary to control the values at the boundaries of the discrete space, in order to maintain stability within a simulation. There are many different possible boundary conditions including periodic and vacuum, and the applications discussed in this thesis have all been developed to use reflective boundary conditions. The reflective boundary condition means that conserved variables should be perfectly conserved throughout the entire simulation, making it simple to validate the applications.

2.4.2 Discretisation

Discretisation is a fundamental process that helps to make partial differential equations computationally solvable. Some approximation is used to describe a continuous function or space using a number of discrete points prior to some solution, generally involving a numerical method of integration.

The three most prevalent discretisation approaches are finite volume, finite difference and finite element discretisation [19] [86]. Each technique can be employed to manipulate a continuous partial differential equation into a discrete equation by using some assumptions or approximations of the limiting behaviour as discrete elements are reduced in size, increasing the precision of the integration.

The different approaches lead to different mathematical and computational challenges. Considering a transport equation, for instance the flow of fluid, a finite difference discretisation would begin from the differential equation and expand it using series representation. A finite volume discretisation begins from the integral formulation and operates on the principal of balancing fluxes between closed volumes. The finite element approach starts with a weak formulation of an equation, and discretises the computational domain into elements comprised of multiple nodes that are described by shape functions [71].

2.4.3 Decomposition

In order to solve a computational problem in parallel it is necessary to perform some level of decomposition, the process of logically breaking a problem into parts that can be independently processed. There may be significant differences between the complexity of decomposition for shared-memory and distributed processing purposes.

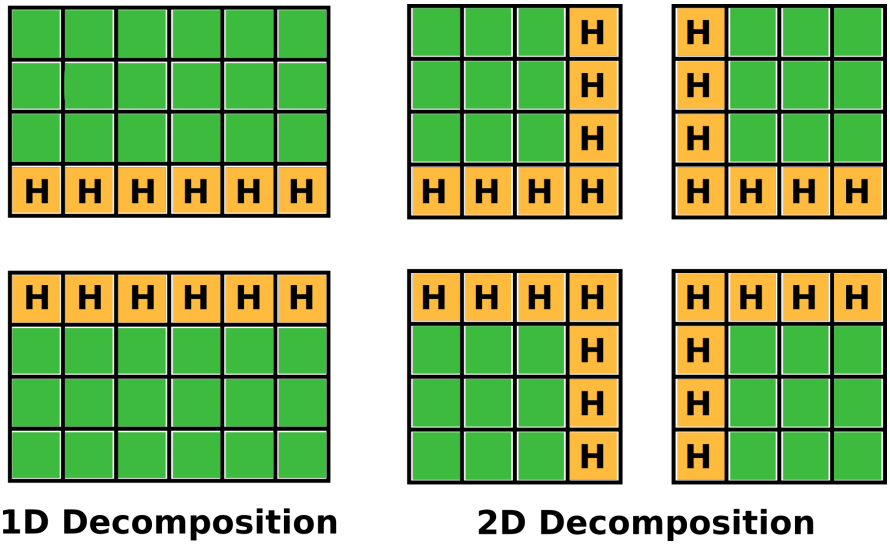

Figure 2.4: Two approaches to decomposing a two dimensional space: 1D decomposition (left), and 2D decomposition (right) where the orange cells containing '**H**' are halo cells.

In a distributed memory environment, the problem might be decomposed into parts that limit the amount of communication that is necessary between processes. Further, load balancing issues may be managed by carefully selecting an initial decomposition, or through dynamic rebalancing [132]. Even a simple and structured problem might need to be distributed into multiple levels on modern architecture, for instance, decomposition across compute nodes, scheduling to multiple threads on each nodes, and grouping into SIMD instructions to be executed by a core [151].

Figure 2.4 demonstrates a decomposition of a 6×6 computational mesh for distributed processing, where the communication approach between independent domains would likely be a

halo exchange [112]. When considering unstructured meshes, for instance, the complex connectivity of the mesh might make decomposition more complicated to setup. In applications with dynamic mesh connectivity this problem is worsened as domain partitioning might need to change along with the changes in connectivity, which will be discussed in Chapter 8.

2.4.4 Numerical Accuracy and Reproducibility

In many scientific simulations, the current approach is to use double precision floating point representations in order to maintain a high level of accuracy through precise arithmetic. Some applications have been developed to fulfil the requirement of bitwise identical reproducibility, the strictest possible form. Applications using threading and tasks cannot efficiently guarantee the order of operations, therefore there are major issues with using thread-based parallelism when an application has strict reproducibility requirements.

There is recognition that reproducibility and accuracy are important concerns for future scientific growth and much research is focused on hardware and software techniques to address the problem, as discussed by Demmel et al. with respect to the progress towards exascale [37]. It can be understood that all discussions about optimisation through the thesis maintain a suitable level of accuracy, but that reproducibility is not necessarily enforced, and results may change depending upon the target architecture or technique applied.

2.4.5 Structured Mesh

The process of decomposing a physical problem into discrete units will generally have a significant impact on the computational approach and results, but also has implications for the performance of the application [111]. Many approaches can be categorised as structured mesh discretisations, which use squares, parallelepipeds, cuboids, etc. to represent discrete chunks of a problem space. The connectivity is designed such that the discrete chunks are non-overlapping, with highly regular and 'structured' mesh representations that enable evaluation of connectivity from spatial location in constant time.

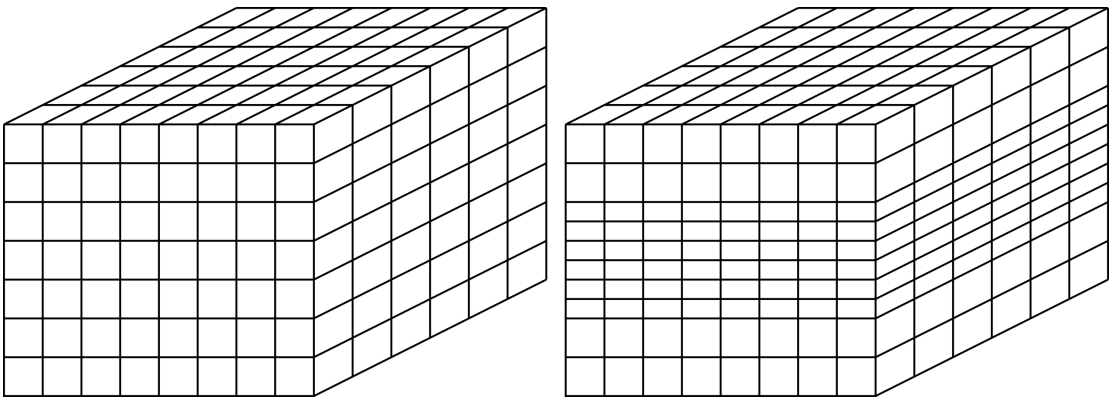

Figure 2.5: Three-dimensional structured meshes: a Cartesian mesh with congruent cells (left); a rectilinear mesh with non-congruent cells (right).

There are two examples of structured meshes presented in Figure 2.5. On the left a Cartesian mesh, where each of the mesh cells can be adequately described as a unit cube, potentially

pending transformation. On the right, the mesh is rectilinear due to the non-congruent band of mesh cells parallel to the x-z plane [19].

Where structured meshes are appropriate from a computational perspective, there are significant performance benefits available due to the regularity of connectivity. The computational methods include less geometric burden, where necessary quantities such as volumes and derivatives between mesh cells are trivially calculated. Furthermore, spatial connectivity requirements such as neighbour lists can be inexpensively determined from spatial location.

The Chapters discussing the heat diffusion and Eulerian hydrodynamics applications, in Chapters 6 and 7, investigate the performance of structured mesh applications.

2.4.6 Unstructured Meshes

Accurately meshing physical objects does not necessarily map well onto structured meshes. In some cases the meshing process itself can become an expensive part of the scientific workflow that needs optimising. Furthermore, some problems require vastly different levels of refinement between different locations of the simulated region, which leads to structured meshes simulating the entire space at the most accurate level of refinement.

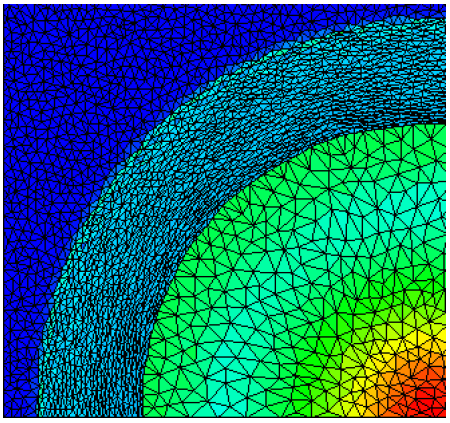

Figure 2.6: An example of a unstructured mesh in `lags` (Chapter 7).

A solution to those problems is to instead use an unstructured mesh, as seen in Figure 2.6, which can in theory allow arbitrarily connected polyhedra of arbitrary construction [97]. There are many consequences of using unstructured meshes, for instance, calculating volumes is significantly more challenging, and connectivity between mesh cells often cannot be determined analytically from spatial location, but needs to be fetched from neighbour lists stored in memory. It is necessary to maintain many indirections, which leads to increased memory footprints and scattered memory accesses within the computational kernels, which might have a significant impact on performance.

The Lagrangian hydrodynamics application in Chapter 7 discussed the implications of unstructured meshes on modern parallel processors.

2.4.7 Explicit Solvers

Explicit methods determine a solution of a future state from a current state. Take the following example differential equation:

$$\frac{dy}{dt} = \alpha \frac{dy}{dx} \tag{2.3}$$

Approximate the temporal derivative with a forward difference and the spatial derivative with a central difference:

$$\frac{y_i^{t+1} - y_i^t}{\Delta t} = \alpha \frac{y_{i+1}^t - y_{i-1}^t}{2\Delta x} \tag{2.4}$$

Rearrange to:

$$y_i^{t+1} = y_i^t + \alpha\Delta t \left(\frac{y_{i+1}^t - y_{i-1}^t}{2\Delta x} \right) \tag{2.5}$$

The equation has gathered all of the known data, allowing each of the discretised mesh cells in the next timestep to be computed independently using the right hand side (RHS) update equation. Explicit solvers apply operators to the computational mesh on a cell by cell basis using known data. The approach is typically highly parallelisable as the operators can generally be performed independently.

2.4.8 Implicit Solvers

An implicit formulation of a problem is where the dependent variables are related by a coupled system of equations. Using the example given for explicit solvers, Equation 2.3, we can instead approximate the spatial derivative at the following timestep:

$$\frac{y_i^{t+1} - y_i^t}{\Delta t} = \alpha \frac{y_{i+1}^{t+1} - y_{i-1}^{t+1}}{2\Delta x} \tag{2.6}$$

Collecting the terms for the next timestep, and substituting γ in place of the scalar terms:

$$y_i^t = y_i^{t+1} - \gamma y_{i+1}^{t+1} + \gamma y_{i-1}^{t+1}, \quad \gamma = \frac{\alpha\Delta t}{2\Delta x} \tag{2.7}$$

From Equation 2.7, it is possible to describe the operation across the whole computational mesh using a single matrix equation in the form $Ax = b$. The vector b is an N-length vector, where N is the number of cells in the computational domain, and represents the known state of the computational domain, y_i^t, while x is an N-length vector containing the state in the subsequent timestep, y_i^{t+1}. The matrix A is an $N \times N$ coefficient matrix containing at most three non-zero entries of the coefficients: 1, $-\gamma$, and γ for each of the rows, depending upon whether cells fall on the boundaries or not.

Formulating problems in this manner can avoid timestep restrictions that are imposed by the inherent numerical stability of the problem. The computational characteristics of the solution are significantly different from those of the explicit solution, as a linear solver is typically required to find a numerical solution of an implicit method. The problem described above is inherently sparse, allowing for the use of sparse linear algebra methods in the solution, and the matrix size of $N \times N$ is generally large, meaning that approximate methods are preferred to direct methods for the linear solve.

2.4.9 Stencil Operations

As seen in the equations in Section 2.4.7, an explicit method is formulated into a set of equations that update a mesh cell using a relation to other known mesh cells. This might lead to a spatial relation resulting from local calculations involving gradients between attached cells, which are described as stencils. The example in Section 2.4.7 describes a one dimensional stencil where the quantity in each cell is updated with a scaling of the difference between the right and left neighbouring cells.

Figure 2.7 presents a two-dimensional stencil calculation on a structured mesh. The operation denoted by '**a**' is a calculation involving the surrounding coloured neighbours, and lies on a boundary. Both the '**a**' and '**b**' stencil computations can be performed independently and in any order, making it straightforward to parallelise the operation, especially if the storage of the results is to a new location.

If the mesh depicted were a tile of a larger computational domain then the '**a**' stencil might not lay on a computational boundary but rather a tile boundary. In order to fulfil this stencil computation, there is an independent working set of data required by each of the non-shared memory processes and a, generally limited, set of data that needs to be duplicated between processes. This duplicated subset of data is called a halo region, as was seen in Figure 2.4.

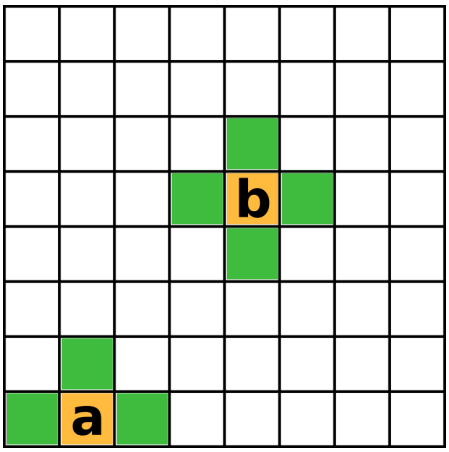

Figure 2.7: A Cartesian mesh with two five point stencil computations depicted.

There is no restriction on the domain of dependence in a stencil operation; as such, the stencils can include many more cells than the immediate neighbours. An important benefit of stencil-based computational methods is that there will be spatial locality and, given correct ordering of operations, temporal locality. Note that stencil operations appear in both explicit and implicit formulations [112] [62].

2.5 Application Domains

Throughout this thesis, a number of applications will be considered for their performance and portability on modern parallel processors. A short introduction to each is presented below, and more specific details can be found in their corresponding chapters. The application domains were chosen because they span a number of important parallel patterns, as discussed in the preceding sections. Each application represents a typical package you might see in the multi-physics applications of, for instance, astrophysics, or reactor simulation [30] [46] [145].

2.5.1 Heat Diffusion

Heat diffusion is a canonical and straightforward example of a second order PDE and requires an implicit solve to maintain a practical timestep [112]. The PDE is expressed as follows:

$$\frac{\delta \vec{u}}{\delta t} = \alpha \nabla^2 \vec{u} = \alpha \left(\frac{\delta^2 \vec{u}}{\delta x^2} + \frac{\delta^2 \vec{u}}{\delta y^2} \right) \tag{2.8}$$

Equation 2.8 describes the transition of heat in a medium as the curvature of the temperature derivative across the spatial domain. The result is that any temperature spikes within a domain will quickly smooth out and then the solution will slowly progress towards a steady state equilibrium, in the absence of a continuous source term. The equation can be discretised using an approximate differencing method, but typically has to be solved implicitly as the problem is stiff. A stiff equation is one where some terms lead to numerical instability unless an, often prohibitively, small timestep is chosen. Implicit linear solutions to such equations lead to

faster time to solution, as time step restrictions can be greatly relaxed [87]. In this thesis, the computational method employed is the Conjugate Gradient (CG) method, as it is a simple but fast converging approximate method for solving linear systems of equations. The CG method is an iterative method for determining the solution of a matrix equation, and interested readers can refer to Shewchuk's clear introduction [146]. The heat diffusion application considered in this thesis is `hot`[2], which uses a stripped back CG solver.

2.5.2 Eulerian Hydrodynamics

In this thesis, the exemplar for explicit structured mesh computations will be a finite volume Lagrangian-Eulerian remap code that is staggered in time and space. An example of a solution calculated with `flow` is shown in Figure 2.8.

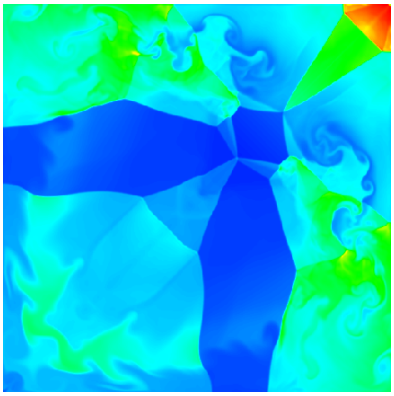

Figure 2.8: Example output of the `flow` hydro proxy application.

The equations solved are the Eulerian equations for conservation of compressible flow, presented in two dimensions.

$$\frac{\delta\rho}{\delta t} = -\left(\vec{u}\cdot\nabla\rho + \rho\nabla\cdot\vec{u}\right) \qquad \text{Conservation of mass} \tag{2.9}$$

$$\rho\frac{\delta\vec{u}}{\delta t} = -\left(\rho\vec{u}\cdot\nabla\vec{u} + \nabla\rho\right) \qquad \text{Conservation of momentum} \tag{2.10}$$

$$\rho\frac{\delta e}{\delta t} = -\left(\rho\vec{u}\cdot\nabla e + p\nabla\cdot\vec{u}\right) \qquad \text{Conservation of energy} \tag{2.11}$$

Where $\vec{u}$ is the velocity vector, ρ is the density, e is the energy, and p is the pressure. The conservation equations are functions of those four unknowns, meaning that the three equations are not complete. A relation can be made between the energy, density and pressure, called an equation of state (EOS), that closes the set of equations.

$$p = (1-\gamma)\rho e \tag{2.12}$$

In the described application, `flow`, the equation is simply the ideal gas EOS, as seen in Equation 2.12 [97]: There are many possible discretisations of the problem, and issues that arise in the accurate calculation of differential quantities and interpolations, as well as numerical fixes for problems arising from the physical approximations. At a high level, the computational

[2]https://github.com/uob-hpc/hot

method uses a number of stencils to calculate the fluxes of each of the dependent quantities across the faces of the cells in the structured mesh. Although the fine details are not presented within this thesis, an interested reader could refer to [19], and the `flow` source code [3]. Further, a thorough description of the computational profile of the `flow` application will be presented in Chapter 7.

2.5.3 Lagrangian Hydrodynamics

The equations of hydrodynamics (as in Equations 2.9 to 2.11) can be also described using a Lagrangian representation, which considers that the discretised problem, composed of cells, deforms due to momentum and pressure gradients in the system, while the mass of each cell is conserved. As such, a mesh that begins as a Cartesian hexahedral domain can deform such that the cells all have different volumes with asymmetric faces.

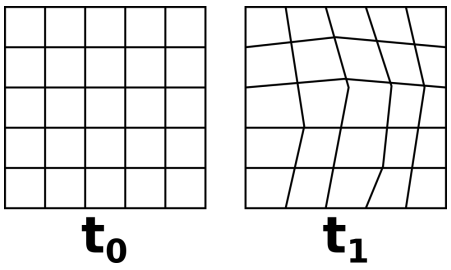

Figure 2.9: An example of a Lagrangian mesh deforming after a single timestep.

The Lagrangian mesh in Figure 2.9 begins as a Cartesian mesh and deforms in the first timestep. The location of each node must be continuously stored in memory, to be updated one or more times per timestep, while the connectivity in this case might be statically determined during the application initialisation phase. In all applications discussed within this thesis only static connectivity will be explicitly solved; however, some discussions about the implications of dynamic connectivity will be included in Chapter 8.

The `lags` application[4] is a Lagrangian solver that will be discussed in Chapter 7, the solver was written from the ground up based on a number of related texts [19] [119]. The application assumes that the computational mesh could be fully unstructured and comprised of arbitrary polyhedra. Only a single type of polyhedra is allowed per mesh, as restricting the mesh to a single type of polyhedra is an effective optimisation, introducing useful constraints on the connectivity. The transition to unstructured meshes makes many of the computational tasks that are simple in the structured case, such as finding volumes and spatial derivatives, significantly more challenging.

2.5.4 Probabilistic Methods

Probabilistic methods typically employ robust parallel random number generation to sample enough random quantities to converge upon an accurate solution to some problem, relying upon the Central Limit Theorem [155]. The approach is applicable to many problems including: solving complex multi-dimensional integration, and the transportation of particles [76]. Taking a probabilistic approach can offer some benefits compared to strict deterministic numerical schemes, potentially making calculations possible that would be untenable due to computational and/or memory capacity limitations.

A simple but popular motivating example of the use of Monte Carlo approaches is the calculation of π using random sampling, a visualisation of which can be seen in Figure 2.10. In the simulation, points are randomly placed inside a square of length $2r$ that perfectly surrounds

[3] https://github.com/uob-hpc/flow
[4] https://github.com/uob-hpc/lags

a circle of radius r. The ratio between the area of the square and circle is known to be $\pi/4$ so the random distribution of N particles within the square should lead to an expected $N\frac{\pi}{4}$ particles landing inside the circle. The Monte Carlo simulation therefore distributes N particles randomly and then solves for π, with the accuracy increasing based on the size of N.

Particle transport can be solved using probabilistic and deterministic methods; where the approaches are most clearly distinguished by the types of source problems that they are suited to [155]. The Monte Carlo approach to particle transport does not use averaging approximations like the deterministic approach, and is instead consistent with the natural physical interpretation of the laws of particles.

In this thesis, the `neutral`[5] application will be used to investigate the performance and portability of Monte Carlo neutral particle transport applications. The application has been written from the ground up based on a number of publicly available research materials [88] [158] [23] [24].

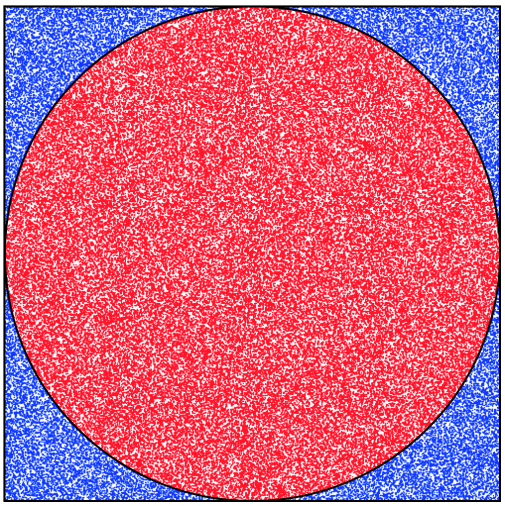

Figure 2.10: Monte Carlo calculation of π.

[5] https://github.com/uob-hpc/neutral

Chapter 3

Programming Models and Performance Portability

Key Publications

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, 2016.

M. Martineau and S. McIntosh-Smith. *The Productivity, Portability and Performance of OpenMP 4.5 for Scientific Applications Targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs.* In Proceedings of the 13th International Workshop on OpenMP (IWOMP), 2017.

M. Martineau, S. McIntosh-Smith, C. Bertolli, A. C. Jacob, S. F. Antao, A. Eichenberger, G.-T. Bercea, T. Chen, T. Jin, K. OBrien, et al. *Performance analysis and optimization of Clangs OpenMP 4.5 GPU support.* In Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), International Workshop on, 2016.

M. Martineau, S. McIntosh-Smith, and W. Gaudin. *Evaluating OpenMP 4.0s Effectiveness as a Heterogeneous Parallel Programming Model.* In Proceedings of 21st International Workship on High-Level Parallel Programming Models and Supportive Environments, 2016.

M. Martineau, S. McIntosh-Smith, and W. Gaudin. *Assessing the performance portability of modern parallel programming models using TeaLeaf.* Concurrency and Computation: Practice and Experience, 2017.

Key Publications (cont)

M. Martineau, J. Price, S. McIntosh-Smith, and W. Gaudin. *Pragmatic performance portability with OpenMP 4.x.* In International Workshop on OpenMP, 2016.

S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, et al. *Offloading support for OpenMP in Clang and LLVM.* In Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC, 2016.

R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis. *Achieving performance portability for a heat conduction solver mini-application on modern multi-core systems.* In Cluster Computing (CLUSTER), 2017 IEEE International Conference on, 2017.

3.1 Introduction

One of the first, and most influential, choices that must be made when porting a large scientific application, is the selection of parallel programming model. The parallel programming model is the interface between the software and the server hardware, and HPC applications often require fine grained controls in order to exploit the available hardware resources. Demand for expressive and powerful parallel programming models means that constructing new models has become a popular area of research, resulting in an explosion of the available choices [40] [105].

Selecting the correct programming model for a particular scientific project is a challenging process that, if left to chance, can have major implications for the future performance, portability of the project. Further, different programming models provide different features for productivity and levels of accessibility, which can affect the development and maintenance costs of large scientific applications.

Each of the parallel programming models in this section will be utilised later to port the exemplar applications in `arch`, uncovering the efficacy of each model, and any challenges posed by particular algorithms. The code samples and discussions focus on the C programming language, but the majority of the concepts translate directly to Fortran.

3.2 Non Performance Portable Programming Models

Each of the applications and kernels discussed in this thesis have highly optimised versions of the code written in low-level languages such as CUDA, OpenMP, and Intel Intrinsics. While CUDA and Intel Intrinsics are specific to particular technologies, OpenMP is in fact a performance portable model. The reason that it is discussed as a "non performance portable model" is that it is only since version 4.0 that heterogeneous processors could be targeted with the offloading model [17]. OpenMP 3.1 is one of the most popular high-performance options for targeting multi-core CPUs, and provides an optimal CPU-specific baseline for the other performance portable models. To readers familiar with the typical parallel programming stack, note that

distributed parallel programming is briefly discussed in Section 3.5, but that threaded and on-node performance is the principal focus of this thesis.

3.2.1 OpenMP 3 and Intrinsics

The OpenMP specification was first released in 1997, supporting thread-based parallelisation of Fortran codes, with C/C++ support added shortly after. OpenMP 3 is the last specification of the directive based programming model that did not contain the facilities for targeting heterogeneous processors using the `target` offloading model [16]. The specification is designed to allow prescribed parallelism targeting multi-core CPUs and is readily adopted for threaded parallel programming [32]. OpenMP provides an alternative to MPI for on-node parallelism, reducing the amount of decomposition by leveraging the shared memory on a node.

The compilers available on modern supercomputers provide highly efficient implementations of OpenMP, meaning that code written for one CPU is likely to be portable to other platforms without significant changes to the parallel code [98]. When optimal CPU implementations are presented in this thesis written in OpenMP, they are typically compiled with compilers providing some level of OpenMP 4.5 support, but using only features and directives from the OpenMP 3 standard, and `omp simd` from OpenMP 4.0.

To achieve the best performance on CPUs it is possible to use architecture-specific intrinsics that map more directly to machine instructions, or even inline assembly code. This can overcome deficiencies in the compiler's vectorising and optimising code generation passes and support complex tasks such as explicit software prefetching. The intrinsic calls are not necessarily portable between different generations of hardware, and are generally non-portable between processors from different vendors.

Often, intrinsics routines are only used in high performance libraries, that have the resources to support different versions for different processors. In HPC applications there would have to be a strong motivation for a particular kernel to be translated into intrinsics, and typically to motivate improvements in the compiler and libraries, rather than as a long-term solution. For instance, the molecular dynamics code GROMACS uses a custom abstraction layer to allow the use of SIMD instructions while retaining ease of portability [1].

3.2.2 CUDA

The potential for using GPUs in general-purpose computing meant that it was essential that a software ecosystem was developed to enable research and adoption of such technologies [126]. The CUDA parallel programming platform and model was first released in 2006 to support the offloading of computational tasks to NVIDIA graphics processors. The CUDA programming model is a low-level C/C++ language extension that allows for fine-grained control of a GPU's hardware resources. If a developer requires maximum performance then CUDA and PTX provide the greatest flexibility and control when targeting NVIDIA GPUs.

The adoption of the CUDA programming model within the scientific community is limited, in part due to the complexity of the model, and also due to the proprietary nature of the specification. The model is designed only for NVIDIA GPUs, meaning that there is good functional portability between generations of those processors, but portability to GPUs or CPUs from other vendors is unsupported. Typically, scientific application developers will rely on higher level abstractions and high performance libraries. The Oak Ridge National Laboratory supercomputer Titan was, at release, the largest supercomputer in the world, ranking first in

the Top500 list, and was comprised of NVIDIA K20X GPUs [153]. The chosen strategy for porting Oak Ridge's scientific applications to run on Titan was via the OpenACC programming model which will be discussed in Section 3.3 [15]. It does not appear that pure CUDA was ever considered as a viable alternative for porting the scientific workloads.

In spite of this, expert developers may rely on the low-level power of the CUDA programming model to develop high performance kernels where necessary. Further, the CUDA framework, and in some cases the CUDA programming model, are essential for the development of high performance abstractions and programming models. In Sections 3.3 and 3.4, models utilising the CUDA framework will be discussed that offer portability to other processors while maintaining a single codebase.

3.2.2.1 Execution Model

A brief description of the model will be provided, but the interested reader can refer to the CUDA Programming Guide for more details [31].

Code Sample 3.1: Simple CUDA example.

```
1  __global__ void func(const int n, double* arr) {
2    const int i = blockDim.x*blockIdx.x+threadIdx.x;
3    if(i < n) {
4      arr[i] += 1.0;
5    }
6  }
7
8  void test(const double* h_arr) {
9    double* d_arr;
10   cudaMalloc(&d_arr, sizeof(double)*n);
11   cudaMemcpy(&d_arr, &h_arr, sizeof(double)*n, cudaMemcpyHostToDevice);
12
13   const int nthreads = 128;
14   const int nblocks = n / nthreads;
15   func<<<nblocks, nthreads>>>(n, d_arr);
16   cudaDeviceSynchronize();
17 }
```

Code Sample 3.1 presents a simple CUDA kernel, called from a host routine, and demonstrates some of the most important features of CUDA programming. The `__global__` specifier on Line 1 tells the compiler to generate device code for the subsequently defined routine. CUDA kernels are issued to the hardware using hierarchical parallel structures: grids of blocks of threads. There are limits on the size of each level of the hierarchy, but the threads are the most constrained, allowing a maximum 1024 threads per block on the P100 and V100 compute capabilities, for instance.

The number of threads is passed as a kernel launch parameter in Line 15 as 128 threads for this kernel, and the number of blocks is then determined to be the length of the iteration space '`n`' partitioned into chunks of length `nthreads`, where the chunks are named CUDA blocks. Each thread is able to determine its location in the grid from the block ID and thread ID, as seen in Line 2. It is important to note that the grid is issued at the block granularity, meaning that if `n % nthreads != 0`, then the total number of threads in `nblocks` would be less than `n` due to the truncation by division. To overcome this, it is possible to issue more than `n` CUDA

threads (in blocks), perhaps by incrementing `nblocks`. The last block in the grid could access out-of-bounds memory locations, which is why the code executed by the kernel is guarded with a condition that ensures `i < n`.

The array `d_arr` is allocated on the device in Line 10, and populated with the contents of `h_arr` on Line 11. Note that two pointers must be maintained within the application, a host and a device pointer. This can increase the overall burden of managing data structures in the application as the number of pointers is immediately doubled. In a proxy application this is generally a minimal overhead but for large applications it can introduce a significant burden to productivity unless abstracted away. The CUDA programming model also provides the mechanisms to share the virtual address space of the host and device, meaning only a single pointer is necessary, and memory accesses are managed with bi-directional page faults.

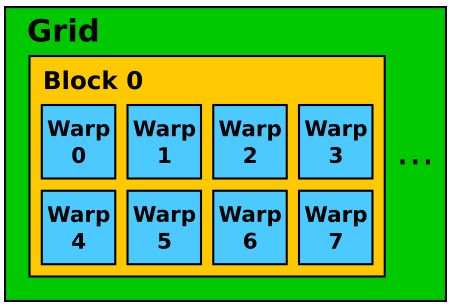

Figure 3.1: The parallel hierarchy exposed by the CUDA programming model.

There is an additional layer of parallelism not exposed by Code Sample 3.1, which is the concept of a warp, as seen in Figure 3.1. Warps are the finest level of parallelism, groups of 32 threads that are scheduled as a single entity to the target processor. In previous hardware generations the warps have been guaranteed to operate in lockstep; however, the Volta architecture and CUDA 9.0 now support more fine grained synchronisation, with each thread of a warp maintaining an individual program counter. Often, CUDA codes can be developed to be warp-agnostic; however, all applications must choose a number of threads that is a multiple of the warp size, 32.

3.2.2.2 Memory Model

The GPU manages multiple memory spaces, several of which will be considered in later chapters. The key memory spaces will be briefly defined in relation to the Volta architecture:

- **Global Memory**: Stored in the GPU main memory, the highest latency and lowest bandwidth memory space.
- **Local Memory**: Private to a thread, but accesses are cached in L1 and use interleaved addressing so that accesses are coalesced if threads in a warp all access the same memory location.
- **Shared Memory**: A programmable partition of the L1 cache that is shared amongst CUDA blocks.
- **Texture Memory**: Device memory that is cached in the non-coherent texture cache, allowing for higher bandwidth for memory accesses expressing some spatial locality.
- **Constant Memory**: Main memory that is cached in the constant cache, which is optimised for broadcasting to threads in a half-warp.

Each of the memory spaces described is controllable to some extent, either by using a memory space specifier such as `__shared__` or even through the use of inline PTX. The different spaces

are exposed to account for different use cases and patterns, and this can be particularly powerful when optimising applications. The ability to program the `shared` memory space, in particular, can overcome the limitations of the GPU cache structure when compared to the large and complex caches in modern CPUs.

3.2.2.3 Compilation and PTX

The CUDA programming model is part of the CUDA platform, which is now a mature parallel programming software ecosystem providing an extensive tool chain [31]. CUDA code is first compiled into an assembly form called PTX (Parallel Thread Execution), which can then be compiled into SASS (Streaming Assembler) code that can be assembled into an executable CUBIN binary format.

SASS is entirely proprietary and NVIDIA does not release details or documentation for it; further, NVIDIA does not provide an open SASS assembler. So while there are some open source attempts to provide the functionality, it is generally impossible to directly write SASS code [54]. The level above SASS code, PTX, has extensive open documentation and the compiler, *nvcc*, is able to directly compile PTX code into an executable. An important difference between SASS and PTX is that PTX is generation-independent, whereas SASS can vary between architecture generations. It is particularly important to the development of performance portable programming models that it is possible to directly output and compile PTX code, as this greatly increases the opportunities for the compiler to generate optimal code.

3.2.3 Exception to the Rule

There are few occasions where verifiable results demonstrate some performance portable model is able to achieve better performance than one of the low-level languages like CUDA or OpenMP. Typically, the CUDA or OpenMP implementation could simply be adjusted to manually apply any optimisations, as the performance portable models will generally output device-specific code like OpenMP or CUDA to the compiler. As discussed in Section 3.2.2.3, CUDA code is a level above PTX, which is the lowest programmable level in the compilation chain. As a consequence, there are rare opportunities to improve upon the performance of CUDA code if better PTX could have been output.

During this project some contribution was made in collaboration with IBM research to the campaign developing OpenMP 4.5 support in the Clang compiler. As part of the joint research, it was discovered that the CUDA compiler cost model for generating non-coherent loads was not optimal for all synthetic benchmarks investigated. It was possible to achieve marginal improvements in performance over the generated CUDA code by emitting non-coherent loads more sparingly [99]. If a user wanted to apply this optimisation to a CUDA code it is still possible, but requires inline PTX. This is the only example of this particular issue observed during the entire thesis, and in all other cases hand optimised CUDA could be tuned to achieve equal or better performance than the other models.

3.3 Directive-based Models

Several directive-based models exist, but two options allow programming Intel CPUs and NVIDIA GPUs, at the time of writing, which are OpenMP 4.5 and OpenACC. The directive-based mod-

els allow parallel programming through the use of compiler directives (`#pragma` in C/C++, and `$!` in Fortran), alongside a complementary runtime API.

3.3.1 Background

It was in 2013 that the OpenMP specification saw an explosion of features, designed to support the new trend of accelerator style processing. The features included a robust offloading model that supported hierarchical parallelism and data movement [135]. OpenMP was not the first directive-based models to successfully enable parallel processing on GPUs, the OpenACC specification was. The history of OpenMP and OpenACC is turbulent, originating from the decision of several members of the OpenMP ARB that the specification was moving too slowly towards full accelerator support. This bore the initial OpenACC standard, which closely approximated the work already discussed for introduction into the OpenMP standard [170]. The original intention was to utilise OpenACC as a research stepping stone to ultimately feed back into OpenMP; however, this re-integration never occurred, partially due to the different paths taken by each implementation.

An interesting comparison of the OpenACC and OpenMP specification was conducted by Wienke et al. and speculated that OpenMP adoption might be higher in the long term as OpenACC implementations did not support multi-cores [171]. Since that research, the PGI compiler has successfully implemented OpenACC for multi-cores, including, more recently, AVX512 support. They also suggested that OpenACC was ahead of OpenMP in terms of functionality, but as OpenMP heads towards OpenMP 5.0, the standards are becoming increasingly homogenised, which means that translation is becoming easier between the two models. Active support for OpenACC was discontinued in the Cray compilers around 2015, in favour of supporting the OpenMP standard. The given justification for this decision was that OpenMP was an open standard, a preference for supercomputer vendors who need flexibility in their choice of processing technologies. In spite of this, OpenACC is now embedded in many production scientific applications, including ANSYS Fluent, Gaussian, VASP, ACME, COSMO, and FLASH [124].

3.3.2 Models and Syntax

The increasing prevalence of GPU computing led to the consideration of directive-based models as a solution for reducing complexity, thereby increasing productivity, and improving portability through standardised interfaces. The focus of this thesis is on the multi-threading and offloading capabilities of the parallel programming models, and their impact on performance portability [102] [99] [98]. A selection of the most pertinent directives and concepts are introduced, and interested readers are referred to the OpenMP and OpenACC specifications for further details [18, 125].

3.3.2.1 Execution Model

OpenMP provides the `target` directive for prescriptive offloading, while OpenACC offers two directives: `parallel`, which supports a prescriptive parallelisation, and `kernels`, which supports a descriptive approach. In all cases, the host will offload a region enclosed by a block structured scope for execution on a target device.

Code Sample 3.2 shows three examples containing the least directives required to execute parallel code on the GPU using OpenMP (Lines 1-4) and OpenACC (Lines 6-14).

- Line 1 is the OpenMP `target` directive, and will result in the 100 iterations of the loop being executed on a single execution unit of the target processor.
- Line 6 is the OpenACC `parallel` directive, which leads to one or more `gang`s being initialised that will redundantly execute the region.
- Line 11 is the OpenACC `kernels` directive, where the code generation depends upon the target architecture, and ability of the compiler to determine the independence of the loop iterations. In the case that no loop carried dependencies are discovered, the N iterations of the loop might be broken into `gang`s of `worker`s and issued to the target processor in parallel.

Code Sample 3.2: Parallel offload directives OpenMP / OpenACC.

```
1  #pragma omp target
2   for(int i = 0; i < N; ++i) {
3     // Some work
4   }
5
6   #pragma acc parallel
7   for(int i = 0; i < N; ++i) {
8     // Some work
9   }
10
11  #pragma acc kernels
12  for(int i = 0; i < N; ++i) {
13    // Some work
14  }
```

The `kernels` directive also instructs the compiler that it can balance parallelism amongst loop nests as it deems most efficient, and even perform optimisations such as loop reordering. This is important because it is the basis for a more descriptive approach to parallelisation, which is somewhat akin to automatic parallelisation on a per loop basis, reducing the amount of code that would need to change between target architectures, potentially benefiting performance portability.

There is an argument that the descriptive approach, as offered by the `kernels` directive, can lead to inconsistent outputs, as different compilers or compiler versions compile with different parallel schemes depending upon their internal cost models. This places the burden of performance optimisation on the compiler, which is a popular approach, as many scientific developers would rather that the compiler was responsible for as much optimisation as possible. This does, however, increase the complexity of implementing the standard, and make the results more likely to be inconsistent between compilers. From the developer's perspective, the descriptive approach can significantly reduce the effort required to port a large application, but relies on compiler intelligence and maturity for more complex kernels. Recognising the simplicity offered by the `kernels` directive in OpenACC, a descriptive directive is planned for the OpenMP 5.0 specification.

3.3.2.2 Parallel Hierarchy

The OpenMP 4.0 specification introduced the concept of leagues of *teams* of threads with vector lanes, providing multiple distinct and controllable layers of parallelism. Instead of *teams* of *threads*, OpenACC uses *gangs* of *workers*. A depiction of the parallel hierarchy can be seen in Figure 3.3.2.2. With respect to NVIDIA GPUs, both *teams* and *gangs* can map directly to CUDA blocks.

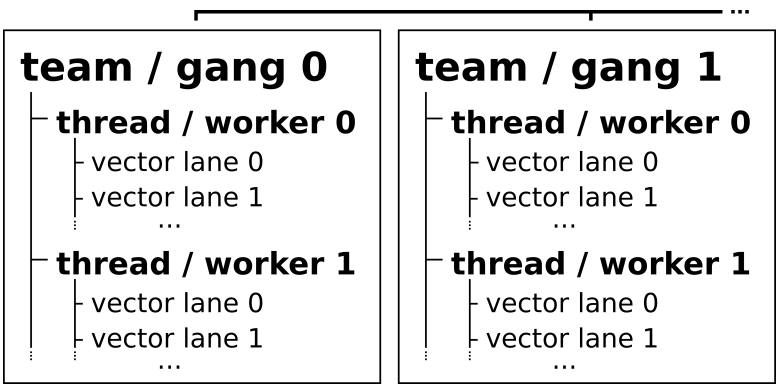

Figure 3.2: The models of parallel hierarchies provided by OpenMP and OpenACC.

In both models the parallel hierarchy can be explicitly controlled, using a range of different directives and clauses. OpenMP provides a number of combined constructs, where multiple directives are provided on the same line to provide some combined functionality. Importantly, the combined constructs may have additional restrictions or characteristics that are not present with the individual directives, and different compilers may implement the code generation differently when using combined constructs.

Code Sample 3.3: Parallel offload directives in OpenMP.

```
1 #pragma omp target teams distribute parallel for simd
2 for(int i = 0; i < N; ++i) {
3   // Some work
4 }
```

Code Sample 3.3 shows the most expressive and important example of a combined construct. In order to schedule the iterations of the loop onto the parallel hierarchy model, OpenMP requires additional syntax beyond the `target` directive. The `teams` directive prescribes that multiple teams should be created, and the master threads of each should execute the region redundantly, while `distribute` schedules the loop iterations to teams.

A major issue uncovered as part of this thesis is that there are two justifiable approaches that can be taken when implementing this offloading in a compiler [104]. Specifically, the final step of worksharing the iterations amongst the `team` can be initiated by the `parallel for`, which considers the OpenMP threads as mapping to the device threads, or the final work sharing can map the vector lanes within the teams to the device threads. In fact, the Clang compiler adopts the first approach, while the Cray compiler adopts the second approach.

The result is that the OpenMP code in Code Sample 3.3 will have a single thread per team, with multiple vectors lanes for the Cray compiler, and multiple threads per team with a single vector lane for the Clang compiler. This decision is problematic from a performance portability

perspective; either the developer has to provide different directives depending upon the compiler, or the compilers must correct the alternative approach and select sane defaults. In the case of Cray and Clang, it is necessary that the developer provides the combined construct `#pragma omp target teams distribute parallel for simd`. Failing to include the `simd` directive might lead to poor performance on the Cray compiler if the loop analysis does not deem the loop to be parallelisable.

Code Sample 3.4: Parallel offload directives in OpenACC.

```
1 #pragma acc parallel loop
2 for(int i = 0; i < N; ++i) {
3   // Some work
4 }
```

The equivalent loop in OpenACC is comparatively simple, as the same behaviour achieved by the OpenMP combined construct `target teams distribute parallel for simd` should theoretically be achieved by the OpenACC `parallel loop` directive. This approach is more prescriptive than the `kernels` directive but more descriptive than the OpenMP combined construct. In the case of nested loops for instance, the `loop` directive can be placed on each of the loops and the compiler will make a determination about how the loop can be best mapped to the architecture. For the PGI and Cray compilers, a `gang` maps directly to a CUDA block, whilst the threads of a CUDA block map to lanes of the `vector`, as per the Cray mapping of SIMD lanes to CUDA threads for OpenMP.

3.3.2.3 Memory Models

The offloading memory models in both OpenMP and OpenACC are designed to abstract the host and device memory spaces, with some productivity enhancing syntax. The specifications outline that implementations must manage the allocation of data on the target device, and then maintain some mapping between a host pointer and the device pointer, such that a user sees only a single pointer that can be used transparently in both the host and device code. The compiler is responsible for determining the relevant pointer to use based on the contextual location of the pointer dereference, i.e. a pointer used inside an `OpenMP` target region will lead to an access to device memory. This overcomes the issue of managing multiple pointers as discussed in Section 3.2.2 regarding CUDA.

Code Sample 3.5: Unstructured data movement directives.

```
1 #pragma omp target enter data map(to: arr[:N])
2
3 #pragma acc enter data copyin(arr[:N])
```

Code Sample 3.5 shows the unstructured data movement directives for both OpenMP (Line 1) and OpenACC (Line 3). Both of the directives are equivalent and direct the compiler to allocate some device memory for the pointer `arr` and copy the host data for `arr` to the device copy of `arr`. It is now possible to use the pointer `arr` to access the device data if it is referred to within within an OpenMP `target` region, or OpenACC `parallel` or `kernels` region. The same pointer will reference the host data if referred to outside of those offload regions.

The structured data mapping regions provided by OpenMP 4.0 were found to be insufficient for the purposes of porting larger software applications [102]. Using the structured data regions

can lead to unnecessary data movement as they are limited to a block structured lexical scope. Members of the OpenMP ARB raised the concern that the use of the unstructured regions might lead to the introduction of memory leaks within an application. Although this is a legitimate concern, scientific applications should develop robust memory management routines to avoid such issues, and the unstructured data directives provide that flexibility for all supported programming languages, without limiting the scoping. If memory management can be abstracted within an application, it makes the entire structure cleaner and much easier to change models in the future. It is a general recommendation of this thesis that the unstructured data mapping directives are used in favour of the structured directives.

3.4 Abstraction Layers

Abstraction layers are an alternative parallel programming paradigm that rely upon an API of routines, without requiring direct development within the compiler stack. C++ is a particularly popular language for the development of abstraction layers because the semantics for generic programming, via templates, allows compile time optimisation of routines using only standard language features. Template specialisation can be used to provide device-specific implementations based on the template parameter, which are then handled at compile time [47].

3.4.1 RAJA

In this thesis, the C++ abstraction layer *RAJA* will be used to port a number of exemplar applications alongside the other parallel programming models. RAJA is developed by Lawrence Livermore National Laboratories and is specifically designed for porting large complex scientific applications, allowing different back ends to target diverse architectures using the same code. RAJA offers portability through C++ template abstractions, and the syntax means that productivity can be greatly enhanced over writing individual architecture-specific codes.

RAJA has been chosen over Kokkos for this thesis as it makes it easier to maintain the existing data structures for an application. This has benefits and limitations, but for the applications ported in this thesis it was straightforward to introduce RAJA with only minor adjustments to the computational kernels and specialisation of the data allocation features in the `arch` project. In a real application it might be preferable to use the programming model's own memory abstractions, but it does not affect the findings presented in this thesis.

3.4.1.1 Execution Model

As previously discussed, the RAJA programming model relies upon C++ templates to abstract the underlying implementation of a parallel loop. To make the API more accessible, the C++ lambda notation is used, which is a significantly more terse syntax than the alternative approach of functors [103]. A fundamental principle in the development of RAJA was allowing the loop body and loop traversal to be separated [67]. RAJA was designed to use abstractions on the iteration space of a loop, enabling descriptions of complex access patterns such as blocking to be encapsulated. Theoretically the abstraction can also be applied to the data structure initialisation, allowing for flexible reordering of loops depending upon the particular specialisation determined by the policy.

Code Sample 3.6: RAJA simple loop example.

```
1  RAJA::RangeSegment r(0, N);
2  RAJA::forall<exec_policy>(r, N, [=] RAJA_DEVICE (int i) {
3    // Some work
4  });
```

Line 1 of Code Sample 3.6 shows the iteration space abstraction `RAJA::RangeSegment`, which describes a contiguous iteration space between the provided bounds, `0` and `N`. There are alternative abstractions to handle different patterns but `RangeSegment` is the most predominantly used set when porting the exemplar applications in this thesis.

Line 2 commences the lambda function, which is a `RAJA::forall`, that will execute the loop body in parallel using the provided segment. The execution policy is the key to specialising the target of execution. If the provided execution policy is `RAJA::omp_parallel_for_exec`, then the back end specialisation for the `forall` routine will be OpenMP targeting multicore CPUs. If the provided execution policy is `RAJA::cuda_exec<128>`, then the back end will produce a CUDA kernel that launches blocks of size 128 threads, targeting NVIDIA GPUs.

The lambda capture means that all variables in the enclosing scope can be referenced from within the RAJA loop, and no explicit data movement syntax is required. The macro `RAJA_DEVICE` will inject the `__device__` specifier necessary for lambda routines targeting NVIDIA GPUs, if the execution policy is targeting the CUDA back end.

Clearly, from the simple example presented in Code Sample 3.6, the concept of template specialisation within C++ is extremely powerful for this purpose. Portability can be enabled in an application without having to change individual characteristics of the parallel expression at the loop body. This does not account for cases where specialisation of algorithms is required, but it is quite conceivable that this could also be abstracted.

3.4.1.2 Memory Model

RAJA did not originally provide memory management functionality and so the user assumed full responsibility for memory management. When using the CUDA back end, this would require the developer to directly program the memory management in CUDA, and that is the approach taken in the `arch` project. The subtle problem that comes about from directly managing memory when programming with RAJA is that the pointers will be handled in the same fashion as CUDA when the CUDA back end is compiled to, and OpenMP for the OpenMP back end. It is undesirable to manage both host and device pointers within a production application, as previously discussed with respect to CUDA in Section 3.2.2.

3.5 Message Passing

The message passing interface (MPI) is one of the best adopted parallel programming models for scientific applications. MPI is efficiently implemented for modern supercomputers, and enables distributed communications for the majority of applications. The characteristics and limitations of MPI for scientific applications is well understood, and many large applications hosted by US and UK national labs were originally written purely in MPI [59].

While it does provide some shared-memory facilities, MPI will not be strictly considered as a programming paradigm and model in this thesis. The features provided by the MPI standard do not currently support targeting accelerator or GPU style devices, meaning that shared-memory

programming models are required to fill the gap. Pure MPI performance will provide a point of comparison for the shared-memory models, and the instances where MPI poses limitations on the performance portability of applications will be discussed throughout. MPI can be combined with all of the parallel programming models discussed in this thesis, allowing shared memory execution inside a domain with message passing between domains.

3.6 Domain Specific Languages

Domain Specific Languages (DSLs) could be considered an extension of an abstraction layer, but that rather than simply abstracting the target hardware, the library also abstracts domain specific concepts. The languages are not designed for general purpose parallelisation of applications, rather they are tailored to certain domains, encapsulating the key compute and communication patterns required to optimised the codes for a specific type of application. Some of the most well know examples of DSLs are stencil and mesh-based libraries, as the concept of a stencil can be well abstracted and covers a great many scientific domains. The OPS and OP2 libraries, for instance, essentially encapsulate domain concepts of structured and unstructured grids, enabling compile time optimisations and portability through C++ abstractions [134].

Some weather and climate codes have adopted DSLs, for instance, PSyclone is designed specifically for use within the LFRic model. Lawrence et al. suggested that the DSL concept allows a strong separation of concerns between the HPC experts who optimise for particular architectures, and the domain scientists who want to express the algorithms necessary to solve scientific problems [85]. The Exascale Computing Project[1] includes DSLs in the PARSEC project, the primary focus of which is distributed tasking for exascale [41]. DSLs are not directly considered in this thesis, as the use of more general approaches is preferred when considering diverse application types, as is the case in this thesis.

3.7 Performance, Portability and Productivity

Parallel programming models are often perpetually evolving to improve portability, productivity, and performance through new features, while maintaining correctness. There is some disagreement on how to reasonably quantify success in any one area [131]. The following section will outline the most important issues relating to performance, portability and productivity, with the focus being on those matters affecting scientific application development. Throughout Chapters 5 to 7, concrete empirical data will be presented to support the discussion, and Chapter 8 will outline those significant issues that arise with production applications, beyond those issues observed with research and benchmark codes.

3.7.1 Performance

Performance is an important focus in this thesis, but it is essential to motivate exactly why performance needs consideration at all. There is a strong focus in computer science on the adaptation of algorithms to improve their theoretical performance bounds. Some scientific domains are at a level of maturity in terms of simulation that truly revolutionary algorithmic changes are rare. Hydrodynamics, for instance, has been studied from a mathematical and computational perspective for many years, and it is unlikely that vastly superior algorithms will

[1] https://www.exascaleproject.org

be invented to solve existing hydrodynamical problems, especially given the elegance of current solutions [136].

In some cases it is necessary to improve or develop algorithms to include new physics or increase fidelity for novel purposes, which is likely accompanied by a period of high innovation. Outside of those sudden advances, however, there are long periods of maintenance, where many scientific institutions are simply concentrating on keeping the existing algorithms working efficiently on the ever changing landscape of high performance architectures. Leaving applications for long periods of time without considering the changes to new architectures can mean that the codes perform poorly with respect to the improvements in the architecture. To take advantage of the improvements in performance offered by each architectural generation, it is likely necessary to make significant and far reaching adjustments to legacy codes.

The perception of performance of an application is generally relative to the particular problem, and possibly domain. Poor performance in time-sensitive weather codes, that have strict restrictions on time to solution, might be considered reasonable performance in the domain of astrophysics. It is not in the interest of this thesis to attempt to suggest what constitutes a suitable absolute performance, but many of the discussions will consider the performance relative to some hardware limitation, particularly memory bandwidth and compute throughput. It is also possible to compare performance between different implementations, and the results of the `arch` implementations written using performance portable programming models will be compared to the performance of the architecture-specific programming model implementations.

There are other measurements of performance that are important, but not directly considered throughout this thesis. For instance, power is an important point of comparison between processors architectures, given that modern supercomputers are approaching the current limits of feasible total power consumption.

3.7.2 Functional Portability

Functional portability is introduced into a language by building abstract models and spanning the necessary features for targeting each architecture, but, without concrete implementations of the specification, this portability will not be realised. As such, the observed portability of each parallel programming language is often judged by compiler support rather than language features. Language maturity can restrict the features enough that portability may not be possible for particular applications, but there are few portability issues that cannot be solved by extending a specification to include new abstract models and syntax. The main problem with this approach is that it is challenging to develop a cohesive language when multiple independent actors contribute patch after patch to the original models. It is also undesirable to write new programming models every time an architectural change occurs, as it requires significant effort on the part of the language designers and the developers who have to learn a new model, although this appears to be a popular strategy.

Portability was much less of an issue before the widespread use of accelerators, as, for some time, the majority of high performance processors were CPUs. There is a wide array of compilers supporting parallel programming models such as OpenMP targeting CPUs. Adding a diverse architecture such as GPUs introduces many problems from the software environment down to the application. Functional portability is a relatively binary issue; either an application will successfully execute a test problem on a target platform or it will not. For the applications considered in this thesis it was possible to run all test problems with the chosen combinations

of programming models and processors with an acceptable time to solution, and so functional portability itself was not a significant issue.

3.7.3 Productivity

Productivity is a more nebulous and qualitative subject than performance and portability, as it is dependent upon the individual experience. It still remains an important characteristic of parallel programming models, as scientific developers are keen to solve scientific problems rather than concerning themselves with the details of complex models. As such, the language design must offer a simple enough model that it is accessible, with enough functionality to ensure that it is relatively complete and can express all of the patterns and algorithms that would be required by the user. This is a challenging prospect, and can be influenced greatly by the model's choice of syntax and abstractions.

The primary challenge in the analysis of productivity is that it is highly subjective. The metric of *lines of code* (LOC) is sometimes cited, but is reductionistic and often provides little introspection, and is easily abused. Lopez et al. suggested that OpenACC is "usually simpler than OpenMP" citing that the descriptive nature of the model reduced the programmer burden [94]. Herdman et al. utilised the *words of code* (WOC) metric to compare the productivity of porting a code to CUDA and OpenCL versus OpenACC. They found that an order of magnitude more WOC needed changing for the low level languages than the directive-based OpenACC [62]. Hammond et al. used a slightly different metric, the number of sites changed, to measure the impact of porting proxy applications to parallel programming models [57].

Although productivity is often discussed in terms of the features of the parallel programming models, and the ease with which applications can be parallelised, it has been observed that this is not necessarily the most important factor when porting a large legacy application. Retrospectively considering the challenges faced preparing science codes for the Titan supercomputer, Wells et al. suggested that as much as 70-80% of programmer effort would be in the restructuring of code to be made suitable for heterogeneous architectures [169].

3.7.4 Performance Portability

The topic of performance portability spans many challenges relating to the interaction between the scientific applications and the target architectures. This section will introduce and consider the key issues relating to performance portability, and the discussion will be continued in the context of each of the scientific applications discussed in Chapters 5 to 7.

The issue of performance portability has become increasingly important as supercomputing resources have diversified. Many scientific software developers are having to consider targeting CPUs alongside different types of accelerators, for instance Intel Xeon CPUs, Arm CPUs, Intel Xeon Phi Knights Landing CPUs and NVIDIA GPUs. This greatly complicates the programming task, requiring developers to be familiar with multiple architectures, and possibly multiple low-level programming models. Further, scientific codebases can be monolithic in structure with millions of lines of code, and so supporting multiple architecture-specific versions of codebases is likely intractable.

3.7.4.1 Definition

Defining performance portability was revealed to be a challenging and contentious task [131]. Over the last couple of years, performance portability has been shown to be a major issue facing the future of scientific progress. Groups of scientists have begun meeting annually for the Performance Portability Workshop[2] and related workshops at SC and ISC, in order to tackle the challenges introduced by increasing hardware complexity. At the workshop, prominent domain scientists, computer scientists and industry experts struggled to clearly agree on the exact definition, although there was good consensus on some of the key characteristics that a performance portable solution would need to present.

This thesis recommends that the definition, while aspirational, is not strictly important, as different computing centres can maintain definitions relative to their own expectations. The throughput requirements of weather centres are far more restricted than those of other domains, for instance. This suggests those developing weather codes might prefer a stricter definition of performance for a particular code, where 25% increased runtime might mean the difference between a prediction being on time or not. A typical scientific lab is unlikely to be burdened by such deadlines, meaning 25% has been cited as an acceptable trade-off [80].

It is possible to construct artificially performance portable solutions that simply maintain individual algorithms per architecture, but this would greatly increase the code size and maintenance costs. Key papers relating to performance portability within scientific institutions cite *working towards single source portability* [115] [75] [129]. The rigid restriction of a single source approach might be too limiting, however, as it is essential that specialisation is possible when algorithmic choices will lead to significant differences in performance. As such, productivity must also be considered, meaning that when the term *performance portability* is used, it is somehow tacit that productivity is a firm requirement also.

It is understood that improving one of the three characteristics in a problem might lead to another characteristics suffering, for instance, performance is improved at the expense of productivity with specialisation of algorithms to particular architectures. Achieving an academic expectation of performance portability is unlikely in a real world environment, and sacrifices and comprimises are necessary.

> *"In this thesis, performance portability is interpreted as a characteristic of programming environments and applications requiring that neither performance, portability, nor productivity are neglected in favour of another in a manner that will prove detrimental to an application's long term capability."*

This could have been further strengthened to suggest that performance portability should even account for future changes to architectures, but, given the radical differences between CPUs and GPUs, it is clear that this would be challenging to fulfil. Some expect it to be impossible to find a solution that allows perfect performance, portability, and productivity, however, it should be possible to find reasonable solutions for most applications. To achieve this goal, each of the scientific institutions need to compromise based on the most important qualities that are required of a particular selection of programming environments.

[2] http://performanceportability.org/

3.7.4.2 Inter-compiler Performance Portability

Depending upon the complexity of an application, there might be issues with the performance portability of a particular model between different compilers.

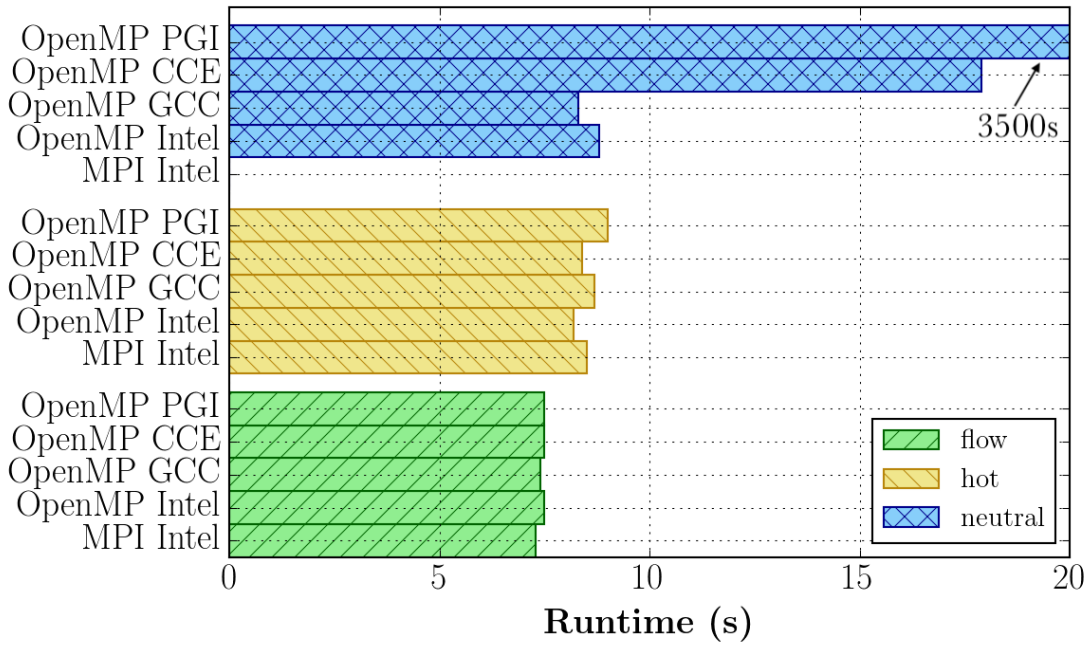

Figure 3.3: Inter-compiler performance portability of `arch` applications on a Skylake CPU.

Figure 3.3 demonstrates the differences in performance of compiler implementations for OpenMP code. The results are collected on dual sockets of 28-core Intel Xeon Skylake (8176) CPU. The compiler versions used are: PGI 18.5, CCE 8.7.4, GCC 7.3, and Intel 18.3. Both the heat diffusion and fluid dynamics codes demonstrate small (less than 10%) differences in the performance of the compilers, which would not affect the scientific throughput enough to warrant any additional effort. In the case of `neutral`, however, the performance between the compilers is surprisingly varied. Note that `neutral` does not have an optimal MPI implementation so the results are omitted. The PGI results for `neutral`, in particular, show an inefficient handling of `atomic` instructions which leads to the routine being treated as if it were a critical region. It is possible to instead use the LLVM back-end for the PGI compiler, and the correct atomic instruction is generated and the performance is within reasonable limits. This does, however, serve as an example that certain compilers might offer different levels of performance portability simply because of differences in implementation.

3.7.4.3 Performance Portability for Programming Models

As part of this thesis it was determined that a number of practices could be followed for performance portability, specifically for OpenMP targeting accelerators [104]. The first key point was to use the combined construct `#pragma omp target teams distribute parallel for simd` wherever possible. It is hopefully true that a large proportion of the kernels in a scientific application simply have one or more, potentially collapsible loops, that can be considered independent and scheduled onto the target accelerator. Of course, there may be kernels where this logic does

not apply, and more specific fine-tuning may be required to achieve good performance. The combined construct is the best way to prescribe that those loops are independent, in a manner that will typically be consistently handled by a majority of compilers.

The second key point was that when implementing an application using OpenMP, it might be tempting to tune particular parameters for the intended target architecture. This is a significant risk in terms of performance portability for OpenMP, as there are no robust mechanisms to manage this using the model. It is often best from a performance portability perspective to allow implementation defined parameters to be chosen by the compiler unless performance is significantly harmed. This point was later corroborated by researchers at Oak Ridge National Laboratory [94].

OpenACC has some additional syntax to aid performance portability, the `device_type` clause, which limits configurations to specific targets, for instance number of gangs or workers. This can improve performance portability, although it does increase the complexity at the loop body [27]. Sidelnik et al. suggested that a major issue for performance portability within a programming model was the need for the developer to directly manage data transfers [148]. Newer versions of the CUDA platform support managed memory, where data is moved implicitly between the device and host. This change to the CUDA platform immediately enabled the same functionality in RAJA without any changes to the source. As such, higher level languages might benefit from improvements in the lower-level platforms with minimal changes, and potentially no changes to the application, which greatly improves the productivity of maintenance of an application.

3.7.4.4 Algorithmic Performance Portability

As stated by Edwards et al. many legacy codes were simply not designed for thread parallelism, meaning that they are unlikely to perform well until they have been optimised for threading [47]. This is a consequence of the majority of early scientific codes being developed with MPI and Fortran, which were unburdened from the issues of shared memory parallelism. It is shown later in this thesis that the most efficient algorithms for particular architectures can be different, for instance, the case shown in Chapter 5 improves as much as 5x through the use of a specific algorithm when executed on a CPU. As previously eluded to in Section 3.7.4.1, this introduces a requirement for specialisation within an application, tailoring key algorithms for certain target architecture.

This is a powerful technique when a small number of kernels within an application require specialisation, but can result in the codebase bloating and inadvertently becoming multiple codebases, directly contravening the purpose of performance portability. Where possible, an alternative approach is to find common ground between algorithms and attempt to write a single algorithm that limits the impact of homogenisation on either architecture. This approach must generally rely on some parameterisation in order to achieve reasonable performance on both targets, which introduces a search space problem that can be solved with auto-tuning [77].

3.7.4.5 Achieving Performance Portability

Many programming models purport to provide performance portability, where they realistically only guarantee portability, given compiler support, and correctness, with some also offering significant benefits to productivity or tuning for performance. Some studies have already shown good results; for instance, the University of Bristol HPC group has been able to achieve good

performance portability for a number of application types across multiple modern processors with the majority of the most used parallel programming languages [111] [103] [35]. Hohnerbach et al. achieved performance portability for a molecular dynamics simulation through designing portable algorithmic optimisations [66].

In this thesis, the analysis of performance portability will be performed on a case by case basis with individual exemplar applications. It must be understood that the task of proving performance portability for all architectural configurations, application domains, and programming environments is beyond the scope of this thesis, but an attempt is made to cover key combinations. In Chapter 8 it will be shown how real production applications begin to introduce complexities rarely seen in benchmark codes and research applications, that can have a significant impact on performance portability. This makes it hard to generalise results and findings to real production applications, but it is possible to account for some of the features that might have the greatest impact.

3.8 Summary

In this chapter, a number of key parallel programming models have been introduced: OpenMP, OpenACC, RAJA, and CUDA, all of which will be used to port proxy codes developed as part of the `arch` suite of applications. Some of the models are marketed as performance portable, providing multiple back ends to target modern heterogeneous parallel processors, including Intel Xeon and Xeon Phi CPUs, and NVIDIA GPUs. Developing effective parallel programming models is a complex art, requiring that the specification writers introduce enough features for completeness, whilst making the model accessible enough that productivity is high. The success of the programming model is often judged on the quality of the implementations, as it is expected that they can achieve a high fraction of peak performance on the supported architectures. Further, the usability and intuitiveness of the syntax are important points of comparison, and cannot be neglected from the consideration of a parallel programming model's efficacy.

The topic of performance portability has been considered, constructing a definition that will be used throughout this thesis, and exposing the key characteristics required for a performance portable solution. It is an important consequence of the definition that the strictest single-source requirements are relaxed, and specialisation of algorithms is permitted as long as it does not create undue support burden. This discussion has shown that there is no single answer to the performance portability issue, but that some research has shown good performance portability is possible with specific codes. It is increasingly important that progress is made towards improving the current state of programming environments and development practices to account for future architectural changes.

Chapter 4

HPC Architectures

Key Publications

M. Martineau, P. Atkinson, and S. McIntosh-Smith. *Benchmarking the NVIDIA V100 GPU and Tensor Cores.* In International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms, 2018.

M. Martineau, and S. McIntosh-Smith. *Exploring on-node parallelism with neutral, a Monte Carlo neutral particle transport mini-app.* In Cluster Computing (CLUSTER), 2017 IEEE International Conference on, 2017.

T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. *GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models.* In International Conference on High Performance Computing, 2016.

4.1 Introduction

Throughout this thesis the performance of applications will be considered on a range of modern parallel processors. To clearly demonstrate the bounds of particular algorithms and applications, it will be important to know many details of the target architectures. This information is in some cases readily available, but different sources disagree on particular aspects of the processor performance characteristics, and finding accurate data about specific processor bins is not always possible. As such, this chapter focuses upon collecting accurate performance data about each of the processors used in this thesis, alongside a discussion of the impact of each processor's design and performance characteristics. It will be shown in Chapters 5 to 7 that the collected data is essential for describing and understanding the performance of the exemplar applications in `arch`.

4.2 Processor Configurations

Attempts to optimise HPC software typically require an intimate understanding of the target architectures. There are many choices in processor architecture available, but in Europe and the U.S., the supercomputing market is currently dominated by Intel, IBM, and NVIDIA [154]. Many parallel processors will be considered throughout this thesis, but the three most recent processors from Intel and the two most recent processors from NVIDIA will be the primary focus.

4.2.1 Intel CPUs

The Intel CPUs considered in this thesis span the last two generations of Intel Xeon processor and the most recent Intel Xeon Phi processor.

CPU	Cores/Socket	Sockets	CPU Clock
Intel Xeon Skylake (Platinum 8176)	28	2	2.1 GHz
Intel Xeon Broadwell (E5-2699v4)	22	2	2.2 GHz
Intel Xeon Phi Knights Landing (7210)	64	1	1.3 GHz

Table 4.1: Details of the key Intel processors used in this thesis.

Unless otherwise stated, all results in this thesis collected on the CPUs are for code compiled using the Intel compilers version 18.3, and the Skylake and Broadwell results are for dual-socket configuration.

Table 4.1 shows that modern Intel processors have a high core count and relatively low base clock speed. The Intel Xeon Phi architecture takes the concept further with a reduced clock speed in comparison to other models, but higher core count and high bandwidth memory. Further details about the processors can be gound in Section 4.3.

4.2.2 NVIDIA GPUs

In this thesis, a number of generations of GPU will be used to investigate the performance of the `arch` applications.

GPU	SMs	Architecture	GPU Boost Clock
NVIDIA V100 GPU	80	Volta	1.53 GHz
NVIDIA P100 GPU	56	Pascal	1.48 GHz

Table 4.2: Details of the key NVIDIA GPUs used in this thesis, including streaming multiprocessor (SM) count.

All results in this thesis collected on the P100 and V100 GPUs are for code compiled using CUDA 9 and GCC.

Table 4.2 presents key statistics for each of the GPU models, and shows that key parameters have increased from generation to generation. The number of streaming multiprocessors (SMs) have been increased between the architectural generations, and there have been changes to the warp scheduler structure, which will be discussed in Section 4.4.

4.3 Intel CPU Background

The Intel processors considered in this thesis are at different stages of maturity and have different levels of adoption. The Intel Xeon Skylake CPU is, at the time of writing, the newest server-grade processor available from Intel and features in 15 of the Top500 fastest supercomputers in June 2018, although it has no entries in the top 10 yet. The Intel Xeon Broadwell CPU is the previous generation and features in 246 of the Top500 list. The Intel Xeon Phi Knights Landing CPU is the newest Xeon Phi product from Intel and features in 27 of the Top500 fastest supercomputers, of which 2 are in the top 10. Those supercomputers are Cori at NERSC (9688 KNLs) and Trinity at Los Alomos National Laboratory (9984 KNLs). As such, the Intel server-grade processors discussed in this thesis are important targets for optimisation efforts.

Device	Broadwell	Skylake	KNL
Cores / Socket	22	28	64
L1 Data Cache Size	32 KiB / core	32 KiB / core	32 KiB / core
L2 Cache Size	256 KiB / core	1 MiB / core	1 MiB / tile
L3 Cache Size	2.5 MiB / core	1.375 MiB / core	n/a
SIMD ISA	AVX (256b)	AVX512 (512b)	AVX512 (512b)
DP Compute Throughput	0.99 GFLOP/s	1.70 TFLOP/s	2.66 TFLOP/s
Memory Bandwidth	76 GB/s	127 GB/s	460 GB/s

Table 4.3: Details of the key Intel CPUs used in this thesis. Note that a tile refers to a pair of cores on a KNL.

The data in Table 4.3 shows that there has been a significant increase in the number of cores and vector lanes between generations of the Xeon CPUs, with a large increase in computational throughput as a result. There has also been a significant increase in the memory bandwidth from DRAM between the Broadwell and Skylake CPUs, which is due to increase in the number of channels on the Skylake memory controller. The KNL is a different architecture to the other two CPUs, and offers high bandwidth memory (MCDRAM), which is rated to have 80% more bandwidth than the dual-socket Skylake configuration. The KNL has lower peak compute throughput than dual-socketed Skylake CPUs, primarily due to the low frequency of the CPUs. The cache architecture has changed between the Broadwell and Skylake CPUs, as the size of the L2 cache is greatly increased in the latter, and the L3 cache has been reduced and made non-inclusive.

All of the results presented in this section are for the values that are determined based on the configuration of the processors and memory controllers, but empirical data will be provided in this section to demonstrate the achievable throughputs.

4.4 NVIDIA GPU Background

NVIDIA GPUs were originally designed to offload graphical tasks from the host processor, for video graphics applications such as video games and film rendering. In recent years, NVIDIA demonstrated the applicability of such processing technologies to a number of diverse compute domains. They now provide an extensive software infrastructure, and HPC processors, offering double precision compute units and large register files, that feature in 97 of the fastest supercomputers in the world. The NVIDIA V100 GPU is, at the time of writing, the newest Tesla

GPU available from NVIDIA, and features in 11 of the Top 500 fastest supercomputers, 3 of which are in the top 10. Those supercomputers are the 3rd fastest in the world Sierra (17280 V100 GPUs) at Lawrence Livermore National Laboratory, and the fastest supercomputer in the world, Summit (27600 V100 GPUs) at Oak Ridge National Laboratory [154]. The P100 GPU was the generation before the V100 GPU, and features in the 6th largest supercomputer in the world, the Swiss national supercomputer Piz Daint, which contains over 5000 hybrid compute nodes with an Intel Xeon CPU and NVIDIA P100 GPU. NVIDIA GPUs also feature in 7 of the top 10 Green500 supercomputers, and the Summit supercomputer is the fastest supercomputer in the world, but also the 5th most energy efficient supercomputer in the world.

Device	**Tesla P100**	**Tesla V100**
SMs	56	80
FP32 cores / SM	64	64
FP64 cores / SM	32	32
Tensor cores / SM	-	8
Boosted GPU clock	1.48GHz	1.53GHz
Shared Memory / SM	64KB	96KB
L2 Cache Size	4096KB	6144KB
Global Memory Size	16GiB	16GiB
Peak DP Throughput	5.3 TFLOP/s	7.8 TFLOP/s
Peak Memory Bandwidth	732GB/s	900 GB/s

Table 4.4: Details of the key NVIDIA GPUs used in this thesis.

Table 4.4 shows that the performance has improved significantly between the two generations of GPU, even though they were both released within a short period of time. The key differences are that the number of streaming multiprocessors has increased, new Tensor cores have been added for deep learning workloads, and the memory bandwidth has been improved. Extensive details about the improvements made in the V100 GPU can be found in the V100 whitepaper and CUDA Programming Guide [123] [31].

In most cases, it should be straightforward to move applications from the P100 to the V100 and observe significant increases in performance due to the improvements to compute throughput and memory bandwidth. In spite of this, increases in shared memory capacity, and streaming multiprocessor count might require retuning of parameters or adaptations to algorithms. Also, the new design for intra-warp synchronisation invalidates some early optimisations that relied on warps acting in lockstep and thereby not requiring synchronisation [31].

4.4.0.1 Scheduling in Streaming Multiprocessors

A key component of the GPU architecture is the use of warp schedulers, which encapsulate the resources that work can be scheduled to in a streaming multiprocessor (SM). Figure 4.1 shows the adjustment to the warp scheduler layout, where the number of floating point units is maintained but the number of warp schedulers is doubled in the V100 and each SM is only capable of processing half the number of threads of a warp per cycle. Much of the scheduling process of the SMs is left as implementation defined or unspecified in the NVIDIA documentation. Some of the concepts are relatively transparent, as the scheduler must be capable of performing known scheduling tasks, such as issuing blocks to SMs, and warps to warp schedulers within an SM,

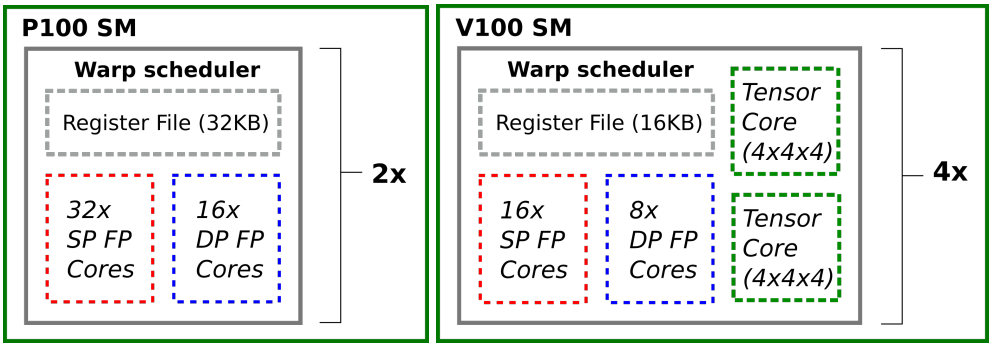

Figure 4.1: The layout of streaming multiprocessors in the P100 and V100 GPUs.

but the exact details are not well published on.

To test the scheduling of blocks to SMs it is possible to fetch the streaming multiprocessor ID for a particular block using the `%smid` register in inline PTX. On the V100, given a kernel that can fit 8 blocks per SM, the scheduler appears to issue the first 40 blocks (block IDs 0-39) to even SMs and the next 40 blocks (block IDs 40-79) to odd SMs. This behaviour continues until there are 8 blocks per SM, at which point the scheduler can no longer issue blocks to SMs until blocks have been retired. The same behaviour is seen with the P100. It has also been shown by Jia et al. that the scheduling of warps to warp schedulers is as `warp_id % 4` on the V100 GPU [72].

4.4.0.2 Kernel Launch Overhead

Using the approach discussed by Volkov et al., the kernel launch overhead was measured for a number of generations of NVIDIA Tesla GPUs [168]. A large number of kernels were launched asynchronously, but in a slight adaptation to the concept presented by Volkov et al. the kernel was left empty.

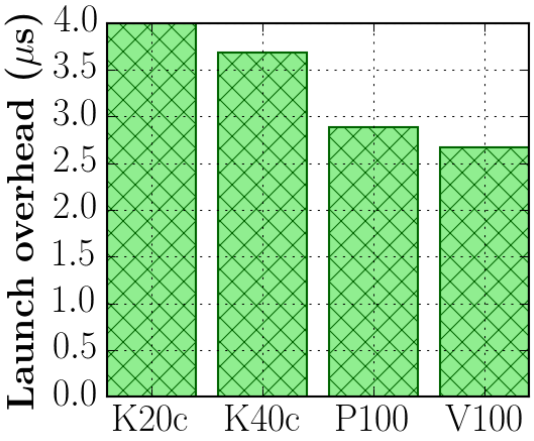

Figure 4.2: Overhead of individual kernel launch per generation.

It can be seen in Figure 4.2 that the kernel launch overhead has steadily improved over successive generations, but only by a small percentage each time. The kernel launch overhead is still significant and it is essential that the amount of work in each computational kernel is

sufficient to amortise this cost. In most of the applications considered in this thesis it is possible to construct large kernels that have individual runtimes significantly greater than the kernel enqueue overhead. There will, however, be an application where the problem of kernel launch overhead needs to be considered; this case will be discussed in Chapter 5.

4.5 Memory Bandwidth

The rate at which compute performance has grown has not been matched by equivalent increases in memory performance, and the problem has continued over the last two decades [110] [107]. As will be shown throughout this thesis, memory bandwidth is an important factor in the performance of many scientific applications, and is often the limiting factor to single node runtimes. Memory bandwidth can be easily measured on modern processors using micro-benchmarks, such as STREAM[1] and BabelStream[2] [106] [35]. The four kernels used to measure bandwidth in this thesis measure: streaming reads (*read*), streaming writes (*write*), streaming read then writes (*read write*), and the traditional triad kernel of STREAM (*triad*).

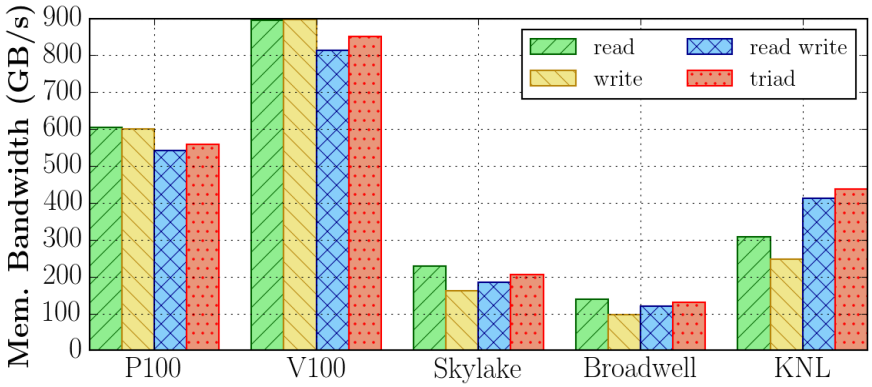

Figure 4.3: Memory bandwidth of four streaming kernels on parallel processors (see Section 4.2).

The memory bandwidth results in Figure 4.3 show that the NVIDIA P100 and V100 GPUs offer the highest memory bandwidth for all kernels. As previously discussed, both of the processors are packaged with high bandwidth memory, specifically HBM and HBM2, respectively. The KNL also offers a high bandwidth memory, MCDRAM, but does not achieve the same level of memory bandwidth, although it is significantly faster than the Skylake and Broadwell. The Skylake still uses DRAM but improved the memory bandwidth over the previous generation, the Broadwell, by increasing the number of channels on both memory controllers from 2 to 3.

Interestingly, for both the GPUs, the best memory bandwidth performance can be achieved using either pure reads or pure writes, with no substantial difference between them. For the CPU technologies, on the other hand, the write bandwidth is significantly lower than read, read/write or triad. In the case of the KNL, both the pure read and pure write tests score significantly lower than the triad benchmark. This is important when evaluating results for particular kernels, as the expected upper limit will be different based on the balance of reads and writes within the kernel for the CPU code.

[1] https://www.cs.virginia.edu/stream
[2] https://github.com/uob-hpc/babelstream

4.6 Memory Latency

Instruction latencies are measured in Appendix A, but in this section the memory latency of different levels of cache for each processor are considered. Using a pointer chasing approach inspired by 'lat_mem_rd' from *lmbench*, a new latency testing benchmark[3] was developed for the purposes of this thesis [152]. Accurately measuring latency requires care due to hardware and software mechanisms that attempt to optimise away latency overheads. The pointer chasing method greatly improves the accuracy of measurement of accurate CPU cycles by avoiding compile time optimisations and inhibiting prefetching. The benchmark measures the latency from cache or main memory to registers by stepping through a large array that acts as a ring of points with strides at the granularity of a full cache line.

The choice of stride is important; by choosing an array length that is a power of 2 and stride of 5, the ring of pointers is defined such that some prefetching mechanisms are avoided on the CPU and KNL. For instance, the hardware prefetching of up to 4 contiguous cache lines can be avoided without having to disable prefetching in the BIOS. Further to this, a choice of a prime stride will ensure that the entire array can be read by a single core in a strided sequence without collisions.

The primary reasons for developing a new benchmark was that there was no open source option for the NVIDIA GPUs, and it was preferable to apply a consistent approach for the CPU and GPU. At the time of writing, a thorough literature review could not uncover any prior attempts to plot the latency of Intel CPUs and NVIDIA GPUs alongside eachother. The results are quite relevant to the optimisation of the Monte Carlo neutral particle transport application `neutral`, discussed in Chapter 5.

4.6.1 Implementation

As discussed, the CPU implementation uses the known method of pointer chasing, and only a single core participates in the memory accesses. On the GPU there are a number of other considerations that mean that it is necessary to tailor the benchmark to the architecture. The L1 data cache is invalidated between kernel calls, and so it is no longer possible to perform a single warmup cycle before the test cycle. Instead the single thread launched on the GPU performs an un-timed warmup cycle through the entire array to ensure that data is resident in the highest possible level of cache prior to executing the pointer chase. The pointer chasing routine fetches the clock using the CUDA intrinsic routine before and after the operation has been performed. A large enough number accesses are performed to amortise the timing overheads introduced by the intrinsic calls. This is similar to the approach of Mei et al. except that the timing is placed outside of the iterative loop and the pointer chase accesses are performed at the cache line granularity [113].

4.6.2 Results

In this section the results of the benchmarks will be presented, with all results plotted together to make it easier to compare the results between different architectures.

Figure 4.4 plots the latency in CPU cycles for memory requests as the working array size is doubled. The Intel Xeon Skylake results exposes a four tiered memory hierarchy, where the first plateau is L1 (4 cycles), then L2 (14 cycles), then L3 (70 cycles), and finally DRAM (300 cycles).

[3] https://github.com/uob-hpc/lats

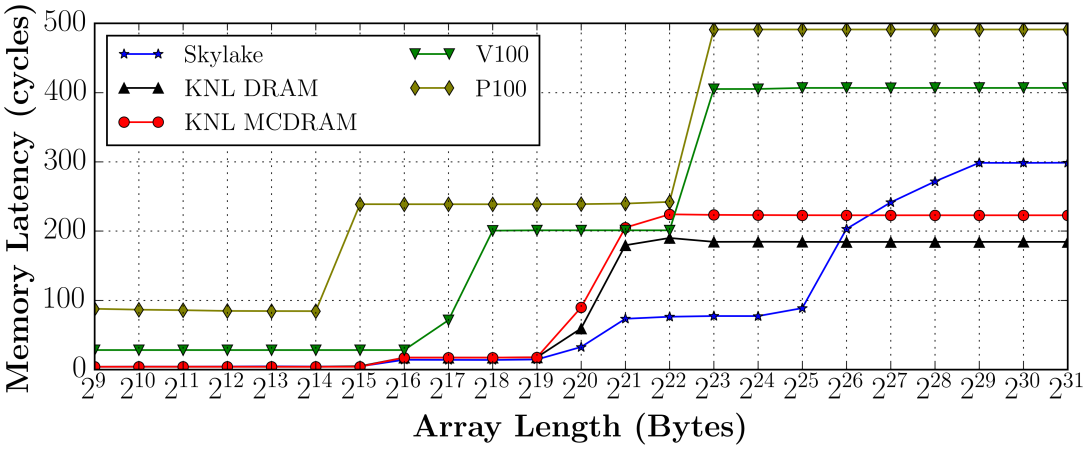

Figure 4.4: Memory latency in cycles for all considered HPC processors.

The KNL results show the latency for the two different types of main memory available to the CPU: DRAM and MCDRAM. There are again clear plateaus in the plot; however, compared to the Skylake latency plot there is one fewer plateau, due to the KNL having L2 as the last level of the memory hierarchy. The levels are L1 (4 cycles), L2 (17 cycles), and then DRAM (184 cycles) or MCDRAM (222 cycles), as the KNL does not have a level 3 cache. It is interesting to note that the latency of MCDRAM is 21% higher than the latency for accessing DRAM, an important trade-off for the high bandwidth. Each level of the CPU memory hierarchy is at least 3.5x slower to access than the previous level. The longest latency is 300 cycles to DRAM for the CPU and 222 cycles to MCDRAM for the KNL, meaning it is necessary to expose a significant amount of concurrency to amortise those latencies. The KNL results corroborate those collected by McCalpin in 2016 [109].

The data in Figure 4.4 shows that the V100 significantly improved the latency of each level of the memory hierarchy, when compared to the P100 GPU. The improvement was roughly 490 cycles for the P100 to 406 cycles for the V100 in HBM2, 240 cycles for the P100 to 200 cycles for the V100 in L2, and 84 cycles for the P100 to 28 cycles for V100 in L1. An NVIDIA white paper regarding the V100 GPU details the improvements to the L1 cache, which is merged with the shared memory subsystem, improving bandwidth and reducing latency [123].

Figure 4.5 transforms the same data presented in Figure 4.4, scaling the results by the CPU clock speed, in order to demonstrate the extent of the latency optimisation. The memory latency disparity is even larger between the V100 GPU and Intel Xeon CPU than Figure 4.4 suggests. There is an important reason for this architectural decision, and that is that NVIDIA GPUs utilise high concurrency to hide latencies, rather than optimising for latencies. NVIDIA GPUs allow essentially free context switching as each warp scheduler maintains a finite number of active `warps`, such that the warp states do not need to be written back to DRAM when switching between them [167]. Given enough independent instructions, the warp schedulers can amortise the high latency costs to memory. In contrast, Intel Xeon CPUs leverage deep latency-optimised memory hierarchies to achieve the same purpose.

Another possible observation in the latency plots (Figures 4.4 and 4.5) is the point at which memory accesses fall from one level of the hierarchy to a subsequent level. For the L1 cache the Intel Xeon and Xeon Phi processors spill at 32KiB, while the NVIDIA P100 GPU spills at 16KiB

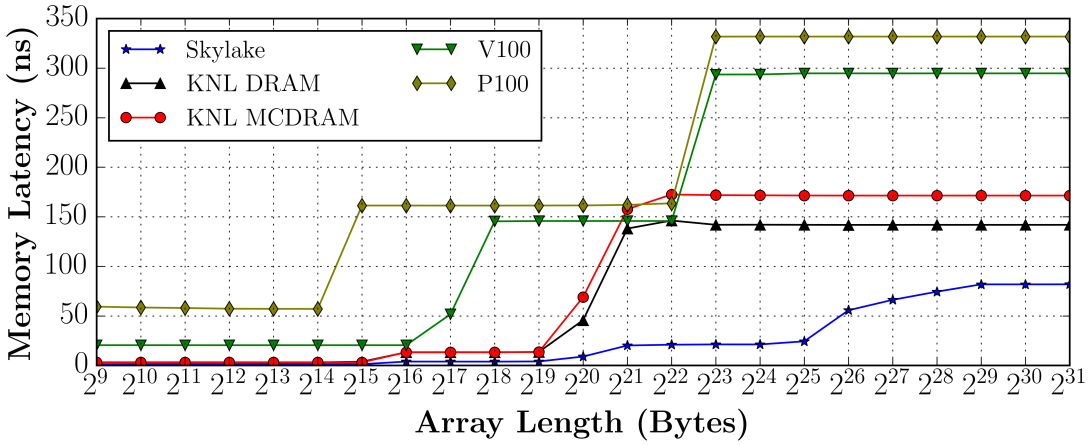

Figure 4.5: Memory latency in nanoseconds for all considered HPC processors.

and the V100 spills at 64KiB; in fact the P100 L1 cache is actually 24KiB but the granularity of the benchmark does not capture this. A similar pattern is exposed for the L2 cache, where the Intel Xeon and Xeon Phi CPUs begin to spill at 512KiB, although the apparent transition to L3 cache is not immediate and, for the Skylake, the majority of the performance is maintained until 1MiB, which is the L2 capacity for a single core. The L2 cache is shown to be equal in size between the P100 and V100 GPUs at 4MiB; however, the size of the V100 cache is actually slightly larger at 6MiB [123]. It is also possible to note from the results that, different to the Intel CPU L2 cache, the full cache is shared between all multiprocessors meaning that a single thread can access the entire capacity with a consistent latency.

4.7 In Flight Memory Requests

The latency benchmark `lats` showed that the memory bandwidth achieved when performing a chain of dependent loads is much lower than the maximum possible bandwidth of the system, as would be expected. For example, the Skylake CPU achieved:

$$\frac{64\text{B} \times 3.69\text{GHz}}{298 \text{ cycles}} = 800\text{MB/s} \tag{4.1}$$

If it were possible to issue a greater number of independent loads that could be serviced with some concurrency then the expected memory bandwidth could approach the true limit. For instance, the bandwidth benchmark showed a peak memory bandwidth for reads of 229 GB/s for the dual-socketed Skylake CPU. Assuming that all cores of both sockets would be required to maximise the loads in flight, the clock would turbo to 2.8Ghz, and so:

$$\frac{229\text{GB/s} \times 298 \text{ cycles}}{64\text{B} \times 2.8\text{GHz}} = 380 \text{ loads in flight} \tag{4.2}$$

This means that the CPU would be required to place around 7 loads in flight per core to saturate memory bandwidth. For the NVIDIA V100 GPU, the peak memory bandwidth was 894GB/s for reads, the latency for a single load from HBM was 400 cycles, and the turbo clock speed is 1.53GHz:

$$\frac{894\text{GB/s} \times 400 \text{ cycles}}{64\text{B} \times 1.56\text{GHz}} = 3582 \text{ loads in flight} \tag{4.3}$$

This shows that the NVIDIA V100 GPU requires many more loads to be maintained in flight than the CPU, at around 45 per SM. The NVIDIA GPU contains 80 streaming multiprocessors, which can each maintain a maximum of 64 warps (of 32 threads), meaning $80 \times 64 \times 32$ threads could be available to launch memory requests.

4.8 Random Memory Access

Typically, memory bandwidth is measured as effective memory bandwidth, where the number of bytes read and written is divided by the runtime. The analysis of the number of bytes read and written can be ad-hoc or measured directly, but the result is that locality is handled implicitly. For random accesses, the assumption is that locality cannot be expressed and therefore there will never be a possibility to improve the effective part of the bandwidth. Each individual read is actually transfered at the granularity of a cache line, even though the memory access generally results in only a single element of that cache line being utilised for the function of the application. Considering the memory accesses at the cache line granularity exposes the true bandwidth of memory accesses.

4.8.1 Random Memory Access Benchmark

Some prior work on measuring random memory access performance focused on the Cell processor, and the efforts were added to the HPC Challenge benchmarks [2] [144]. The approaches and benchmark do not fulfil the requirements of this thesis for multiple reasons. Benchmarks are needed that expose random memory access performance using in-loop random memory access, as well as showing the performance on modern GPUs. Two possible approaches that can be taken to measure random memory accesses, each with benefits and limitations:

- Random memory accesses can be performed using random number generation within the loop body. This has the potential to introduce overheads due to inefficiencies in the random number generation process, but might be more representative of real world cases.
- Random numbers can be pre-computed to construct a random walk through memory and accessed as an indirection. This increases the memory footprint, affecting the observed memory bandwidth results.

A more powerful approach is to use the pointer-chasing method, as discussed in the latency benchmarks (Section 4.6), so that the pre-computation of random walks does not require additional memory costs. Using this approach, a new random memory access benchmark, `random`[4], was developed for the purposes of this thesis in order to explore the performance of random memory accesses on modern HPC processors. The benchmark is multi-threaded for both CPU and GPU, vectorisable, and can be run with the arithmetic-free approach of pre-computed random walks, or the traditional approach of in-loop random number generation.

[4] https://github.com/uob-hpc/random

4.8.1.1 Implementation

The benchmark initialises memory that is large enough to provide `NBLOCKS` unique cache lines for each thread to read. Each block contains `NTHREADS` × `NLANES` cache lines, where `NTHREADS` is the number of threads and `NLANES` is the number of vector lanes, specific to the target processor. After the first initialisation step, each cache line contains a pointer to a unique cache line in the subsequent block, offset by some random distance, with the last block pointing to the first block in order to construct a ring. Figure 4.6 (left) shows a ring of pointers for thread 0 for cache lines $0 \rightarrow 12 \rightarrow 19 \rightarrow 30 \rightarrow 0$.

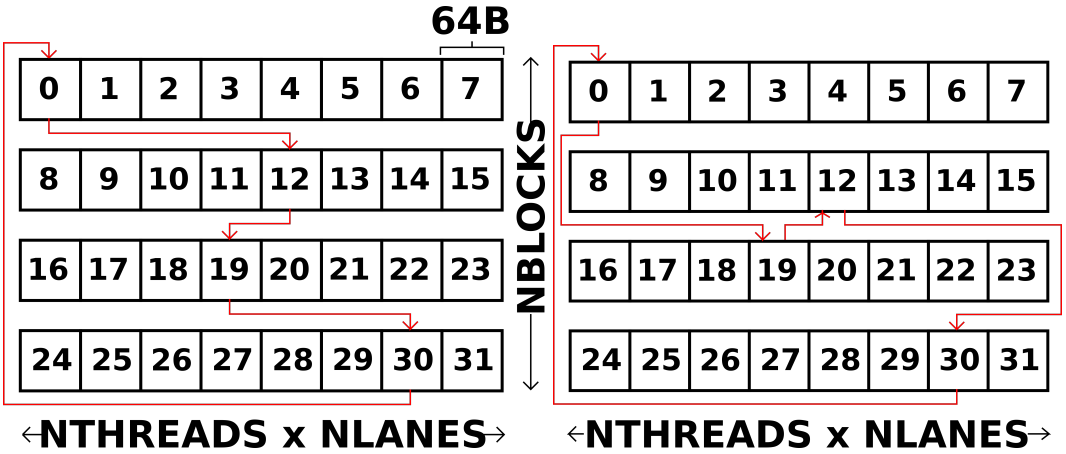

Figure 4.6: Block layout for the initialisation (left) and block shuffling (right) of memory in the random memory access benchmark. Each numbered square represents a unique cache line.

Subsequently, a random shuffling is performed on the pointers by removing each pointer and reinserting it into a random location in the same walk, as seen in Figure 4.6 (right). As such, each thread accesses random entries of random blocks, ensuring there is no bias or patterns introduced in the memory accesses. Further, every cache line is accessed exactly once, avoiding any incidental locality.

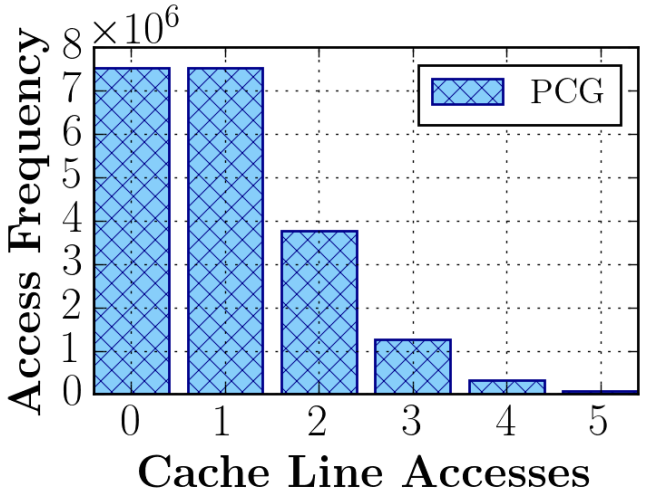

Figure 4.7: Frequency of cache lines accesses.

The second mode in the benchmark generates random numbers using the PCG random number generation library on each iteration of the loop [127] . A particular vectorised version of the library, provided by Intel, is used to enable vectorisation within the benchmark. The benchmark is parameterised so that the same number of memory accesses is performed as with the pointer chasing benchmark; however, the truly random selection of cache lines from the large one-dimensional array will result in collisions. Figure 4.7 shows the distribution of random accesses to cache lines for a large set of blocks. Given that the number of random accesses N is equal to N_c the number of cache lines, and the random number generation is unbiased, most cache lines are either accessed once or not at all. This means that the number of independent cache lines accessed during the benchmark using RNG is 30% less than the pointer chasing benchmark. In spite of this the total memory access in terms of number of cache lines accessed is equivalent for both benchmarks.

Another key difference to consider is that the random number generation is performed in the loop and so the instructions might have some influence on the overall performance, which will be discussed and considered. As such, this mode of execution in the benchmark gives a less accurate representation of the true limits of the hardware, while giving results more representative of the possible real world use cases.

4.8.2 Validation

Cache miss frequency and uncore counters were used to verify that the random memory access benchmark resulted in continuous accesses to DRAM, rather than inadvertently performing regular accesses to the fast caches. It was possible with the uncore counters to demonstrate that benchmark fetched at least 99.7% of all memory requests directly from DRAM, meaning that an insignificant number of cache lines were served from cache for both approaches. The access frequencies were validated for both of the benchmarks, and all random number generation was performed using the well tested PCG library. As such, the benchmark was validated as an accurate representation of random access memory pattens served primarily from main memory.

4.8.3 Results

The benchmark is initialised such that the array is distributed onto both sockets, even though the later accesses are random, in order to best represent the memory layout that might be seen in a mesh-based application. Also, all of the CPU results presented use huge pages of 2MiB, in order to avoid issues with the TLB; however, note that this setting does not affect the GPU's internal paging.

4.8.3.1 Unvectorised Results

The unvectorised results are particularly important to the analysis of the `neutral` application in Chapter 5, and so the results for the pointer-chasing benchmark are presented with and without vectorisation.

The results in Figure 4.8 at unroll factor 1 show the performance of each core performing a single memory access, and the memory bandwidth is less than 10% of the achievable memory bandwidth. Even without vectorising the problem, it is possible to allocate multiple random walks to each thread and unroll the independent accesses. This allows more loads to be added to the load buffer on a single thread, and the performance improves, although it is limited to

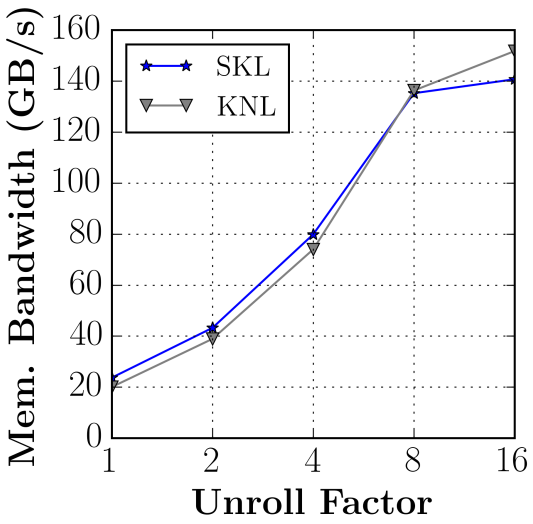

Figure 4.8: Unvectorised results for the *pchase* benchmark, by unroll factor.

65% of the achievable memory bandwidth. The perceived loss in memory bandwidth is due to the fact that the memory access patterns can cross NUMA domains, and if the same experiment is conducted on a single socket, it is possible to achieve full memory bandwidth on the Skylake using unrolling. It is important to note that this unrolling might not be as successful in a real-world scenario, unless the independent loads are close to each other in the application, as the instructions must be able to fit in the reorder buffer. In the pointer-chasing benchmark the loads can be immediate neighbours.

4.8.3.2 Vectorised and GPU Results

In order to enable vectorisation, the number of independent memory access streams through the one-dimensional array was increased from 28 *cores* to 28 *cores* $\times$ 16 *vector lanes* per socket on the Skylake CPU, and the same approach was applied to the KNL. The results are shown for a large working set of around 500 MiB.

Device	Mem. Bandwidth (*pchase*)	Mem. Bandwidth (*simple*)
Skylake	106 GB/s, 98% peak	80 GB/s, 74% peak
KNL	169 GB/s, 54% peak	152 GB/s, 49% peak
V100	413 GB/s, 46% peak	391 GB/s, 43% peak

Table 4.5: Best observed random memory access performance (Skylake results are for a single socket).

In Table 4.5, results are shown for both benchmark variations. It was hypothesised that the *simple* benchmark might be faster than the *pchase* benchmark, as it exposes some locality in the problem, but this is not the case for any of the processors. The Skylake results for the pointer chasing benchmark show the memory bandwidth is around 106GB/s, which is 98% of the achievable read bandwidth. While full bandwidth is achievable in *pchase* mode, for the *simple* case the use of random number generation within the loop reduces the performance to

74% of the read bandwidth. In contrast, the KNL is only able to achieve around 54% of read bandwidth, which appears to be due to the fact that prefetching is required to achieve maximum bandwidth on the KNL. Prefetching is not necessarily possible with random memory accesses, and so this is a significant limitation. The results on the GPU are surprisingly high at around 46% of maximum read bandwidth, which is more than was expected given that the random memory accesses are not able to be coalesced.

The results in this section will be useful when considering the performance achieved by the `neutral` proxy application in Chapter 5.

4.9 Summary

This chapter discussed the key details of the HPC-oriented parallel processors that will be used throughout this thesis. The Intel and NVIDIA processors were shown to be the most popular processors in the world, featuring in many of the worlds fastest supercomputers. Architectural improvements were considered for the most recent generations of Intel Xeon CPUs and NVIDIA GPUs, demonstrating an increasing focus on improving memory bandwidth, alongside the consistent increases in compute throughput. Empirical data was collected for the parallel processors, exposing key architectural details of their memory performance characteristics.

The NVIDIA GPU kernel launch overheads were shown to have only marginally improved over the last 4 generations, which limits the potential for optimisations involving many small kernel launches. Instruction and memory latency have dramatically improved over the last few generations of GPUs, and the results clearly demonstrated the difference in the latency optimised nature of the Intel Xeon CPUs, and the latency hiding requirements of the NVIDIA GPUs. The peak memory bandwidth of the GPUs is currently vastly superior to the CPU offerings, including the KNL, although it is expected that those differences will diminish in the future, as high bandwidth memory is introduced into modern CPUs. The processor designed by Fujitsu for their exascale machine, the A64FX, is set to include 32 GiB of HBM2 memory, making it the first traditional CPU with plans to include high bandwidth memory [172].

Random memory access performance was dissected using a new benchmark, and the results show that achieving full bandwidth is problematic for all of the architectures. To achieve maximum true bandwidth it was necessary to place enough memory requests in-flight using loop unrolling or vectorisation. It was also shown that random memory accesses across NUMA domains generally leads to poor performance due to the increased latency of the memory requests. Further, random memory access performance was shown to be lower than expected on the KNL, and higher than expected on the NVIDIA GPU.

Chapter 5

Monte Carlo Neutral Particle Transport

Key Publications

M. Martineau, and S. McIntosh-Smith. *Exploring on-node parallelism with neutral, a Monte Carlo neutral particle transport mini-app.* In Cluster Computing (CLUSTER), 2017 IEEE International Conference on, 2017.

M. Martineau, P. Atkinson, and S. McIntosh-Smith. *Benchmarking the NVIDIA V100 GPU and Tensor Cores.* In International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms, 2018.

S. Fogerty, M. Martineau, R. Garimella, and R. Robey. *A comparative study of multi-material data structures for computational physics applications.* Computers and Mathematics with Applications, 2018.

5.1 Introduction

The class of problems solved with Monte Carlo methods employ probabilistic approximations to the solution of some equation, relying upon the central limit theorem to infer the average behaviour of a system based on observations of simulated data [76] [155]. Typically, it is possible to reduce those solvers into independent tasks that can be parallelised with no dependencies. Asanovic et al. described the Monte Carlo Dwarf of parallel computation as relying on repeated trials to gather statistical results, and that the class is considered "embarrassingly parallel" [6]. They further stated that the communication patterns of Monte Carlo algorithms typically mean that communication is not a dominant factor in the performance. This has been shown to be an incorrect classification for some Monte Carlo cases in a distributed sense, and this chapter will show the extent to which this is not true when attempting to thread Monte Carlo neutral particle transport codes [133].

The specific case of Monte Carlo neutral particle transport relies upon apriori knowledge of the probability distributions describing the physics of a particle transporting through some mix

of materials. Using those distributions to describe the interactions of a single particle and scaling the simulation to a large number of particles, it is possible to accurately describe the average behaviour of a particle population [140]. However, there are multiple reasons why this particular method cannot be considered "embarrassingly parallel". For instance, the entire geometry for a target problem is typically too large to store for every individual processing element, meaning that some amount of domain decomposition is required [163] [24]. The necessity to decompose information about materials and geometries introduces the issues of load balance, and managing random communication patterns [132]. Even in the case where complete replication of the domain is possible, it might be necessary to perform all-to-all communications to synchronise tallying data.

5.2 Problems in Monte Carlo Neutral Particle Transport

The use of Monte Carlo methods for particle transport is known to date back to around the 1940s [96]. The method itself was well understood and described by early research, and many of the statistical techniques used today are similar to those described in the 1950s [76]. Diverse areas of science have found uses for Monte Carlo neutral particle transport, including radiation dosimetry and other medical purposes, the simulation of neutrinos in supernovae, and the design and safety of nuclear reactors [70] [5] [145]. Each of the diverse areas introduces different physics and data, but the fundamental methods often lead to the same limiting characteristics. A survey is presented of those issues that have been shown to make optimising Monte Carlo neutral particle transport codes challenging.

Much research has been devoted to the large scale communication of Monte Carlo neutral particle transport applications, and great success has been achieved in this area [25]. There are now a number of well established algorithms for the communication of particles within a domain decomposed Monte Carlo neutral particle transport simulation, and as such that aspect will not be the focus of the investigations in this thesis. Generally, the natural formulation of Monte Carlo neutral particle transport algorithms employs a loop *over histories*. Vector processors introduced a previously unseen problem with this particular formulation, as the individual particles in a candidate vector would be performing diverse computations [5]. The '*over events*' approach, originally suggested by Brown et al. resolved this problem, but at the expense of introducing a sort into the algorithm [23].

Siegel et al. more recently investigated the shared memory performance of the OpenMC code using several threading strategies with OpenMP [149]. They recognised that threading Monte Carlo methods is an important area for research due to the increase in core counts and requisite decomposition, and much investigation is required to prepare the production codes for the current architectural trends. They found that coarse-grained parallelism was the best performing, although this approach does not appear to vectorise and might not be suitable for GPU architectures, potentially leaving significant performance to be gained. As they increased the core counts for their parallel implementation of OpenMC, they observed a drop off in performance beyond some number of cores. Given the complexity of the application and the architectures, there was no clear answer as to why the performance did not scale linearly, demonstrating more work is required to understand the limitations of threaded Monte Carlo applications on existing architectures.

Brantley et al. outlined the need for future Monte Carlo codes to be able to target diverse parallel architectures [20]. Their future targets include the Trinity supercomputer, comprised

of Intel Xeon and Xeon Phi CPUs, and Sierra, containing IBM POWER9 CPUs and NVIDIA Volta GPUs. There was additional recognition that the '*over events*' algorithm might offer an important option for targeting highly parallel architectures. Tramm et al. recognised the need for extensive research into the performance of cross sectional calculations from large lookup tables [158]. They found that for some applications and problems, determining cross sections from lookup tables represents a large proportion of the overall solve time. For the purposes of investigating this particular issue, the `XSBench` proxy application was developed to isolate the performance characteristics of this step.

Long et al. considered a cache size timestep limiter, that would attempt to overcome the issues observed with random walks through a mesh seen in a photonics code [93]. In principal the approach should improve problems with locality in the Monte Carlo formulation, caused by the random natures of the particle movement, that limit the performance on modern cache based architectures.

As such, it has been found that a number of areas require further research for Monte Carlo neutral particle transport:

- Shared memory parallelism.
- Vectorisation for existing SIMD architectures.
- The performance of tallies.
- Lookup table performance.
- Random memory access performance with respect to streaming particles.
- Performance portability of the Monte Carlo method, including offloading to GPUs.

To be able to investigate those issues it will be necessary to perform algorithmic and data structure research, as well as conducting experiments porting to diverse architectures using the parallel programming models discussed in Chapter 3.

5.3 Monte Carlo Particle Transport Applications

Given the long history of Monte Carlo neutral particle transport codes, there are many open source Monte Carlo neutral particle transport applications. Each of the applications has been developed with diverse expectations and purposes, but typically using similar underlying methods.

5.3.1 MCNP

The Monte Carlo N-Particle Transport Code (MCNP), developed by Los Alomos National Laboratory, is a Monte Carlo neutral particle transport code that has been in development since at least 1957 [29]. It is a fully featured application that can simulate neutron, photon and electron transport, as well as couplings of those particles, within complex geometries [155]. MCNP is comprised of nearly 500k LOC, and is relied upon by a large international user base for many different purposes including nuclear reactor safety. The application is export controlled, which means that the source code is not freely available, and it is not possible to discuss the specific design decisions made in the application.

5.3.2 OpenMC

The OpenMC code was originally developed for the purposes of investigating scalable algorithms targeting exascale computing [140]. OpenMC uses constructive solid geometry and can handle all nuclear reactions producing secondary neutrons, but does not yet handle photons. At the time of writing, the application contains around 50,000 LOC. According to Tramm et al. a large proportion of the runtime of OpenMC is devoted to the calculation of macroscopic neutron cross sections [158]. The application has been particularly optimised for parallel fission bank site handling, using an algorithm designed to support parallel scaling [141].

5.3.3 Quicksilver

Quicksilver, developed at Lawrence Livermore National Laboratories, is a proxy application for the code Mercury [20]. The proxy application is intended to match the performance profile of Mercury in terms of memory access and communications patterns, and solves time-dependent neutron transport. The application is, at the time of writing, around 13000 LOC, with OpenMP and MPI used for parallelism. Quicksilver can handle cells with different materials, multiple different types of tallies, and has been designed in such a manner that the particle tracking is threaded over particles. The application has been designed to work with energy groups, which significantly diminishes the impact of cross sectional lookup tables, as a smaller number of energy bins are required.

5.3.4 Branson

Branson is an Implicit Monte Carlo (IMC) mini-app solving gray thermal radiative transfer (TRT), that was developed by Alex Long at Los Alomos National Laboratories [92]. The mini-app was developed for the purpose of conducting algorithmic experimentation, and is currently around 10000 LOC. The application contains many different algorithms for parallel transport, including the domain decomposition methods of particle passing and mesh passing. Branson is not yet designed for thread-based parallelism, and the parallelisation is currently handled with MPI only.

5.4 `neutral`: Monte Carlo Neutral Particle Transport

The available Monte Carlo applications were shown to be often large, offering many features, with complex geometries, and in some cases codes that do no lend themselves to thread parallelism. Dosanjh et al. outlined the strategy for exascale co-design, particularly in relation to the Mantevo project, and suggested that mini-apps should typically be constrained to O(1K) lines to allow rapid exploration of key performance issues [45]. The two best candidates for use in this thesis were Quicksilver and Branson, however, both options were open sourced after `neutral` had already been developed. Further, both of the applications have characteristics that would have limited the scope of the research:

- *Quicksilver* is large for regular porting exercises, large enough that it would take considerable effort to change algorithms and data structures to the extent that is performed with `neutral`. The application has been developed to use energy groups, which makes it more challenging to represent the performance of continuous energy applications.

- *Branson* is marginally smaller, but focuses purely on Implicit Monte Carlo, which would have excluded investigations into the complex area of managing particles produced due to fission, for instance. Further, the algorithms included primarily focus on MPI-based parallelism, and significant changes would be required to introduce thread parallelism.

As such, in order to effectively investigate the open areas, it was necessary to develop a streamlined application that would allow for a focus on shared memory parallelism, including SIMD and GPU algorithms. The application needed to be small and amenable to porting and algorithmic experiments, while offering enough functionality to be representative of the performance profile of feature subsets of the larger Monte Carlo neutral particle transport codes. The research areas discovered in Section 5.2 form the basis for the feature set required in a new application.

The `neutral` application has been written from scratch for use in this thesis to compare cutting edge and novel algorithms on architectures including NVIDIA GPUs. The computational code and physics were derived from a number of open research articles and books, with a focus on capturing the structure and computational profile of the features without extensive emphasis placed on scientific accuracy [138] [96] [88]. The computational code is encapsulated and expressed within around 1000 LOC, an order of magnitude fewer than Branson and Quicksilver, and nearly 3 orders fewer than OpenMC, making it much easier to perform porting and other code-affecting studies. In particular, the restricted size of `neutral` has made it possible to develop many variations of the application that adopt different algorithms, data structures, and programming models, without introducing an untenable maintenance overhead. The limitation of this approach is that it is more challenging to capture all of the characteristics of a large Monte Carlo neutral particle transport code. This issue will be discussed in detail and additional experiments performed to ensure that key issues with representativeness have been addressed.

The `neutral` application forms the basis of this chapter, and it is used to: develop new algorithms, investigate random memory access performance, demonstrate GPU parallelisation strategies, and show methods for vectorisation of Monte Carlo neutral particle transport codes on modern SIMD architectures. In some cases the results even defy conventional wisdom regarding the performance of code with deep branching on the GPU.

5.4.1 Particle Tracking

A convenient feature of the Monte Carlo formulation of particle transport is that the major elements of the algorithm are expressed with clear physical intuition. Particle tracking is the core of Monte Carlo transport codes, and encapsulates the majority of the physics that happens within a timestep. The particle tracking loop of the `neutral` application changes depending upon the particular choice of algorithm and depends upon whether the domain is decomposed in a distributed fashion. In spite of this, the concepts of the particle tracking are consistent between all approaches and the characteristics of the particle tracking loop play a major role in the overall performance of the application, and so it is described here. The particle tracking approach is typical, and was primarily derived from the descriptions given by Gentile et al. [53]. At a high level it is possible to consider three independent types of events within the particle tracking loop:

- **Facet events**: Individual particles can be transported through the mesh in a continuous fashion; however, dependencies upon the computational mesh mean that data must be

stored as particles move between mesh cells. In a distributed environment it is possible that the particles can transition between processing elements upon encountering the facet of a cell.

- **Collision events**: Throughout its lifetime a particle can encounter nuclei along its trajectory, either being absorbed or scattering, and potentially producing new particles.
- **Census events**: In a time-dependent application this event occurs once a particle has reached the end of the current timestep.

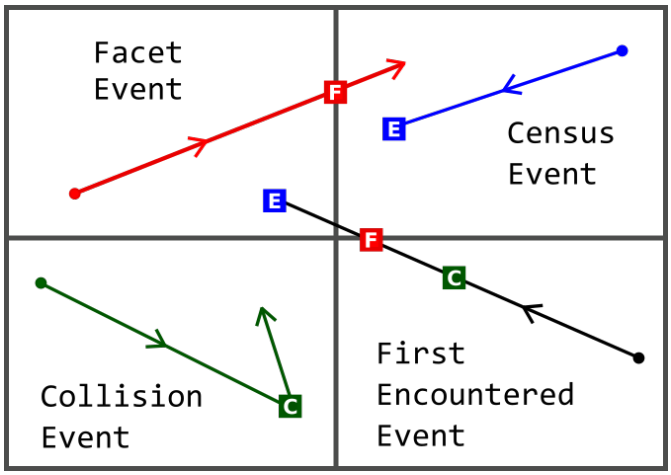

Figure 5.1: The particle tracking concept of Monte Carlo neutral particle transport, depicting the three events, and the determination of the first encountered event.

As seen in Figure 5.1 the distance to those events are calculated for a particle, before the first encountered event is handled and the particle moved appropriately. As previously mentioned, the Monte Carlo method is popularly considered to be an embarrassingly parallel approach, due to the fact that the particles are, in theory, completely independent. In reality, for this particular application the particles are dependent upon the shared computational mesh. An obvious solution to this problem is to entirely replicate the computational domain between processing elements. While this would resolve the interprocess scaling due to communication, the resulting algorithm grows greatly in terms of capacity, and the performance can suffer due to poor cache utilisation. As modern supercomputers move towards massively parallel on-node processing, domain replication is becoming increasingly challenging to support [139].

An important consequence of the particular particle transport approach is that load imbalance can be introduced locally and remotely. Given reflective boundary conditions, as in `neutral`, it is possible that a particle can stream from one side of the computational mesh to the other side in a single timestep. Particle densities within a particular domain might start extremely imbalanced, say if the particle source was a single cell of a full mesh, but then even out as particles transport randomly across the space. Alternatively the converse is possible, where the source is evenly distributed across the entire computational domain but, due to the problem specification, a majority of particles become 'stuck' in a single high density location. It is therefore challenging to select an optimal decomposition using a general approach, as the optimal decomposition changes at the sub-timestep granularity on a per-problem basis.

From another perspective, a single particle can transport across the whole mesh, depositing energy, while on the same mesh another particle could be sourced into a single cell, collide many times and then reach census in that same cell. As such, the application includes many branches, and is highly sensitive to the problem specification, a problem corroborated in the existing research [20].

5.4.2 Tallying

Tallying is the process by which observations are captured for particular quantities within a simulation. There are many different possible types of tallies, and the relevant choices will change based on the domain and particular purpose of the simulation. Tallies can capture different quantities within a simulation, for instance fluxes, reaction rates, and secondary particle production. The domain of a tally could be a single cell or geometric object within the spatial domain, or it could span the entire spatial domain [155].

Van Veen et al. considered the performance of full reactor simulation, and found that the cost of fission energy tallies significantly impacts the performance of the solve for large numbers of tallying bins [39]. Romano et al. stated that care must be taken to ensure that the scaling of the tallies is not limited by the number of tally bins [140]. The `neutral` application has been developed with two different types of tally: (1) balance tallies, that keep track of event counts from particle histories; and (2) a continuous energy deposition tally, where the energy deposited on average by particles transporting is stored in a tallying mesh.

The continuous energy deposition tally is an important feature of the application because it requires a large number of tally bins, one per cell in the computational domain, and introduces the potential for a race condition. The tallying does not require a search, as the tally bin is determined directly from the spatial location of the particle. If each particle is handled independently, it is possible that the independent histories require tallying to the same cell, which introduces a data race that can be solved using atomic instructions. As the energy deposition does not have to be resolved until a particle has left a cell or reached census, facet events are always required to perform an expensive tallying operation. The collision events do not, increasing the potential for load imbalance, depending upon the algorithm.

5.4.2.1 Random Number Generation

In the `neutral` application, random number generation is used extensively to handle sourcing, particle population control, and sampling of random events for each particle history. A robust parallel random number generation facility is required to avoid bias in the results. Different random number generators offer particular characteristics and trade-offs, for instance, some offer reproducibility to enhance testing and debugging, while others might use secure cryptographic schemes. It would be possible to take advantage of one of the cryptographically secure random number generation facilities and techniques; however, their robustness often leads to inefficiencies in the computations and difficulties with parallelisation due to shared state. An alternative approach is counter-based random number generation (CBRNG), which resolves the reproducibility issue and allows for easy parallelisation due to the scalar state [143].

Both OpenMC and the Quicksilver mini-app use a custom linear congruential generator (LCG) class, while Branson uses Random123 [20] [140]. The Random123 library offers a high period, and the underlying approach means that it is possible to skip-ahead and even rewind through the number stream in O(1) time [143]. Although CBRNG approaches are highly paral-

lelisable, it is not necessarily true that they are vectorisable, and this becomes important when attempting to vectorise the Monte Carlo neutral particle transport algorithm, as discussed in Section 5.7.

5.4.3 Nuclear Cross-Sections

The probability that a particle will encounter an event on its current trajectory is dependent upon the properties of the material that it is presently transporting through. In a mesh-based approach, probability tables are required for all combinations of simulated materials and simulated reactions within those materials, across the whole problem domain.

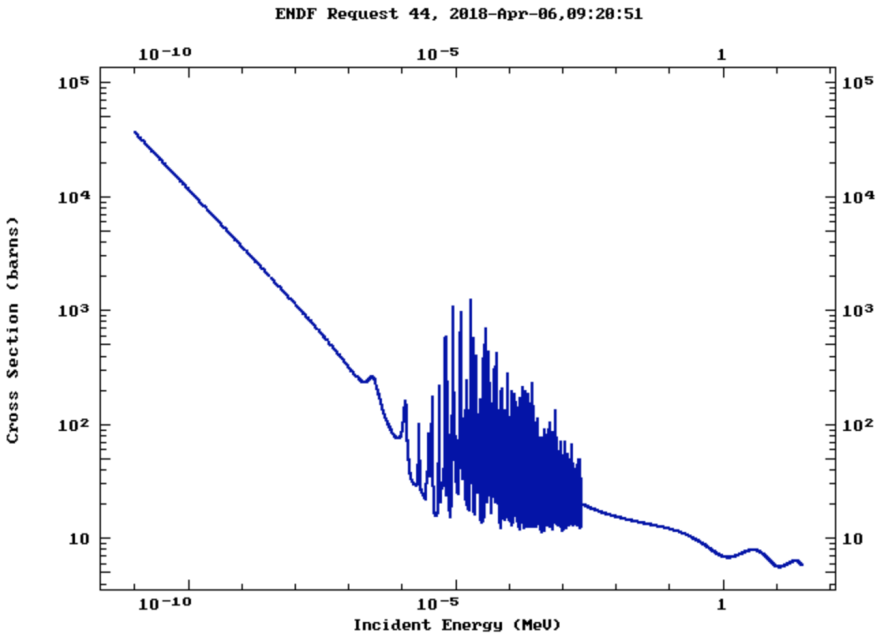

Figure 5.2: The nuclear cross section of U-235 in log-log scale.

The computer science response to the threat of large data tables is to recommend the employment of compression techniques or approximations, to polynomials for instance. Unfortunately, the complex nature of the resonances do not lend themselves to approximation. Figure 5.2 is an example of an evaluated nuclear cross section taken from the European Evaluated Nuclear Data File (ENDF) database [21]. Clearly, compression or approximation by polynomial is intractable for such a complex function. A more suitable approach is to select a finite number data points for interpolation, storing them in a lookup table.

The `neutral` application is designed such that the nuclear data tables can be easily changed, allowing for the addition of new materials and events. Each of the test data sets is representative in size, around 30K double precision energy/cross-section pairs, for 468KiB each, but loosely approximates real data in order to present a more average case. Some nuclides require a total lookup table capacity that is larger than described, and some applications will require numerous tables that combined spill out to DRAM [158]. The cross sections in `neutral` are for absorption and elastic scattering. The probabilities are affected by resonances between the particle and the material, and they are given as a function of the energy of the particle. This means that the lookups must be updated whenever the particle energy changes during its history, or moves into a new material with new lookup tables [88].

Readers familiar with the published research regarding `neutral` should be aware that results are presented using only a binary search rather than using the linear search optimisation. This means that the performance of the `scattering` problem is significantly different in some cases, due to the change in memory access patterns.

5.4.4 Core Algorithm

In this thesis, a number of algorithms are explored in order to discover the best performing approach relative to each of the target HPC processors. The most obvious and well known approach is to parallelise over the list of independent particle histories, named *'over histories'*. This approach is highly parallel, except for the dependencies on the computational mesh, but it makes vectorisation challenging, and the branching appears too complex for parallelisation on GPUs. Liu et al. found that vectorisation was prohibited by the complex nature of the loops in the *over histories* approach, and those small loops that could be vectorised in their Monte Carlo code observed insignificant performance improvements [90]. They went on to suggest that the *over events* approach described by Brown et al. might improve the vectorisability of their code and outlined it as future work [23].

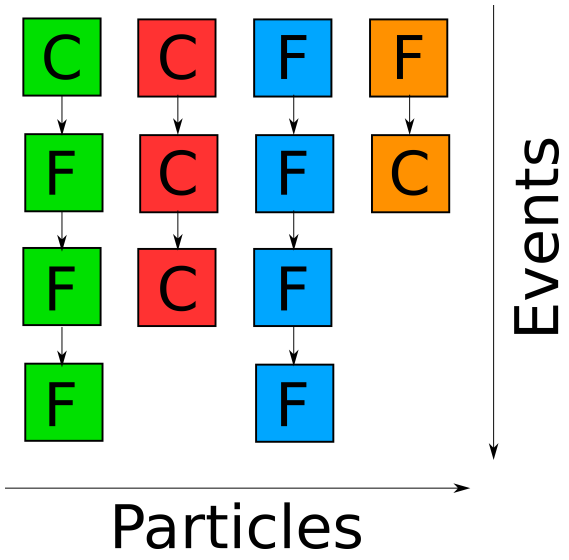

Figure 5.3: Matrix depicting the organisation of events and particles throughout time.

The events that occur during a `neutral` test problem are effectively described as in Figure 5.3. Each column is an individual particle history, where 'C' represents a collision, 'F' represents a facet event, and the arrows represent the transition between events towards census. Every history ends with an implicit census event, which for most problems will not impact the performance due to being executed only once per N events in a particular history. There are two important features of the representation: (1) the particle history lengths are not necessarily equal, and (2) the dependencies are only between events for a particle, not between particles, so concurrency can be introduced across particles. To complement the *over histories* approach, the *over events* approach will also be considered, and a novel algorithm will later be presented that implements a hybridisation of the two approaches.

5.4.4.1 Over Histories

The *over histories* algorithm follows each particle history independently from birth to census, before the next particle is processed. The outer particle loop is a parallel loop that distributes the particle population to threads. The inner loop moves a particle through continuous space until it encounters the edge of a computational domain or reaches census, named the history loop.

Code Sample 5.1: The *over histories* algorithm for `neutral`.

```
1  foreach particle:                    // particle loop
2    loop:                              // history loop
3      distance_to_events(particle)
4
5      if colliding:
6        handle_collision(particle)
7      else if encounter_facet:
8        handle_facet(particle)
9      else:
10       handle_census(particle)
11       exit loop
```

Although Code Sample 5.1 does not show the implementation of each of the routines, they contain a mix of computational code, branching, random number generation etc., which will be discussed throughout the subsequent sections. The organisation of the code into this parallel structure has a number of important consequences in terms of performance:

- **Particle data is generally maintained in registers or high levels of cache**

  As the particles are being processed in parallel, with one particle per processing element, the particle data can generally be stored in registers or L1 cache. This means that the particle data does not adversely impact the performance of the application; however, the memory requests for the cross sectional data and mesh variables, which reside in lower levels of cache or main memory, are still required.

- **Load imbalance is possible**

  As the particles loop over independent events, and each event has a different computational cost and profile, it is possible that a load imbalance is present between threads. For instance, it is possible that some subset of particles perform significantly more collision events than facet events; if the collision events take significantly longer to process then it is possible that threads would have to wait for others to complete the full history.

- **Minimal thread synchronisation is required**

  The synchronisation of threads is only required after all particles have been processed, meaning that there is negligible synchronisation overhead.

- **Deep branching is present in the parallel loop**

  As the nested routines also contain branching, the depth of the branching is considerable, particularly in comparison to the other exemplar applications discussed in this thesis. An important consequence is that vectorisation of this particular loop structure is not possible with current compiler technologies targeting modern CPUs.

The *over histories* approach is essentially a base case with which to compare other algorithmic techniques. If possible it is essential that the approach is extended or replaced with some vectorisable method.

5.4.4.2 Over Events

The *over events* formulation has been included in this thesis as an alternative to the *over histories* approach that appears to be more amenable to vectorisation and GPU programming. The approach was first described by Troubetzkoy et al. as a solution to porting Monte Carlo neutral particle transport codes to vector processors, and later refined by Brown et al. [161] [23]. Considering the physics from a breadth-first rather than depth-first perspective allows for a reformulation of the particle Monte Carlo algorithm to process individual events in batches of particles.

The algorithm as described by Brown et al. shuffles particles into queues that are to encounter the same event [23]. Although code for this particular algorithm was not available, the description given in the paper provides many implementation details.

Code Sample 5.2: The *over events* algorithm based on Brown et al.

```
1  shuffle(particles) -> facet_particles, collision_particles, census_particles
2
3  collision_loop:
4    tracking_loop:
5      handle_facet(facet_particles)
6      shuffle(particles) -> facet_particles, collision_particles, census_particles
7
8      if empty(facet_particles):
9        exit tracking_loop
10
11   handle_collision(collision_particles)
12   shuffle(particles) -> facet_particles, collision_particles, census_particles
13
14   if full(census_particles):
15     exit collision_loop
16
17 handle_census(census_particles)
```

The algorithm in Code Sample 5.2 shuffles particles into groups, handling all facets before handling collisions and then starting the cycle again until census is reached for all particles. It is not entirely clear from the original paper how the shuffling procedure is actually implemented, nor its connection to the calculation of future events. Necessarily after each event a particle encounters, the next event for that particle must be determined. It is therefore assumed that the shuffling operation has to perform the distance calculations for each of the possible reactions and then shuffles based on the outcomes.

The *over events* approach has several diverse characteristics compared to the *over histories* approach:

- **The event handling routines may be vectorisable**

  It will be possible to vectorise the event handling routines over particles without needing to mask out particles.

- **Sorting is required in the history loop**

  If the sorting cannot be completed in high levels of cache, the cost is expected to be considerable even on a CPU. The implications for GPUs are much more significant as sorting can be an expensive operation that might have to be performed in the high latency global memory.

- **Branch depth is decreased**

  The depth of branching is decreased because the first set of tests is completely replaced with parallel loops. This, coupled with the fact that the particles would be sorted into streams that are undergoing the same event, means that the divergence should be reduced. This is theoretically an attractive property for targeting GPUs.

- **Particle data can no longer be cached**

  With the *over events* approach particle data is no longer guaranteed to reside in high levels of cache as, in the worst case, the *entire* particle history might be accessed during the process of following the individual particle histories.

The algorithm did not explicitly handle the vectorisation of tallying, which will be investigated in this thesis. Unfortunately, the use of sorting is expected to perform poorly for modern processors, and so it is necessary to consider alternative approaches to the *over events* algorithm. Ideally, this approach would provide the benefits of the *over events* algorithm but without the need to perform the expensive sorting operations.

5.4.4.3 Sort-free Over Events

Modern SIMD processors and GPUs are able to avoid the need for sorting vectors through the use of masking, and this technique can be used for this particular algorithm. The *over events* algorithm will be adapted to take advantage of vector masking.

Code Sample 5.3: The *over events* algorithm for `neutral` using predication.

```
1  loop:
2    distance_to_events(particles)
3
4    foreach particle:
5      determine_next_event(particles)
6
7    foreach particle:
8      if particle.colliding:
9        handle_collision(particle)
10
11   foreach particle:
12     if particle.encounter_facet:
13       handle_facet(particle)
14
15   if all_particles_at_census(particles):
16     exit loop
17
18 foreach particle:
19   handle_census(particles)
```

Code Sample 5.3 shows the new algorithm that represents the *over events* concept, but implemented in a sort-free manner. All of the particles are processed regardless of their next event, but masking is used to ensure that the particles not encountering the presently processed event do not have their state updated. The result is that the algorithm is able to work on large vectorisable streams of particles without having to move the particle data around in memory. This approach introduces some additional challenges compared to the original *over events* approach:

- **Synchronisation is increased**

  Due to the fact that the events are performed in parallel there will be synchronisation for every particle and every iteration of the loop. This means that the synchronisation cost increases from N in the *over histories* approach to roughly $3LN$, where 3 is the number of independent parallel loops, and L is the number of loop iterations required to bring all particles to census. The parameter L depends upon the problem specification, but in `neutral` L is typically many orders of magnitude smaller than N.

- **Sorting is replaced with predication**

  The potentially expensive sorting operations are removed, but depending upon the problem specification the predication might limit performance.

An immediate limitation of this approach is that the particles cannot all be moved through the tracking phase before the collisions are handled, as is possible with the original *over events* approach. Regardless, this formulation is expected to be more compatible with highly parallel processors, whether CPUs or KNLs which require vectorisation, or GPUs which require long parallel streams.

5.4.5 Parameters

Even though `neutral` includes only a minimal set of features, the number of parameters for each problem is still large. The choice of parameters and problem specification can have a significant impact on the performance.

5.4.5.1 Particle Population

The particle count determines the precision of the solution, as increasing the number of particles will converge the solution towards an increasingly precise value describing average behaviour of the system, as dictated by the central limit theorem [155]. The runtime of the application is proportional to the number of particles, and for each of the constructed test problems a linear increase in the number of particles is honoured by an equivalent increase in runtime.

The particle population plots in Figure 5.4 demonstrate a smoothing of the solution as the population is increased. The population is increased from 1e6 particles to 1e7 particles, and a diffusive solution is observed, as would be expected of particles targeted at dense homogeneous regions. There is no expectation that `neutral` would be used to determine accurate results to test problems, but it is important that the correct behaviour on parameter change is observed, and this has been extensively validated.

5.4.5.2 Particle Sourcing

Each of the particles is sourced at some spatial location with a particular initial energy. The location of sourcing is problem dependent, and will be described alongside the discussion of the

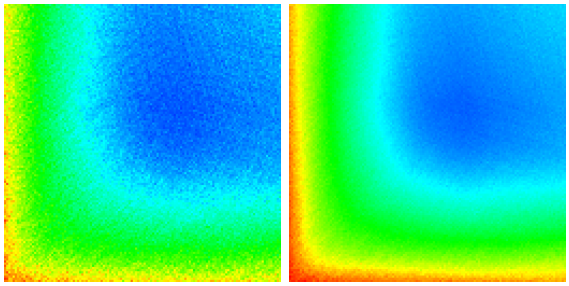

Figure 5.4: Tuning the number of particles towards convergence, 1e6 particles (left) and 1e7 particles (right).

particular problem. When initialising the energy of the particles, it is possible to either set all particles to the same initial energy or sample each particle's energy from a distribution. There are several potential impacts regarding the distribution of energies:

- Particles with different initial energies will access different entries from the cross sectional lookup tables. This is highly problem dependent but particles with similar energies might perform many accesses to proximate locations within the lookup tables, expressing unexpected locality. It must be noted, however, that in `neutral`, the lookup tables are small enough that they will fit in L2 cache, making locality less of an issue.
- The frequency of specific events performed by each particle could be significantly different depending upon the starting energy of that particle. It is possible to construct problems containing a mixture of high energy particles streaming across facets and rarely colliding, alongside particles of low energy regularly colliding and rarely crossing facets.

Particle sourcing is modeled as a singular instantaneous occurrence in `neutral`, and the initial population of particles is maintained until a number of time steps have been completed. In spite of this, it is possible to adapt `neutral` to perform continuous sourcing, perhaps by timestep.

5.4.5.3 Timestep

The timestep in `neutral` is relatively inconsequential, as the time-dependence primarily serves as a point of synchronisation and storage of the tallying data. As such, the timestep is chosen to be long enough to observe a large number of events, but the interpretation of the results is independent of this choice.

5.4.5.4 Mesh Dimensions

The dimensions of the computational mesh influence several aspects of the application. For instance, finer mesh cells in the domain are expected to lead to more frequent facet events, and might increase the distance between random memory accesses. In a multi-physics environment, the mesh management is typically dictated by other packages, rather than the Monte Carlo simulation itself. If the mesh is controlled by, for instance, a hydrodynamics package, a level of refinement will be necessary to reach the necessary fidelity of the simulated flow. For each of the test problems considered for the `neutral` application, the mesh dimensions 4000^2 are chosen, and this is consistent with the other applications considered.

5.4.6 Problems

The problem dependence of the Monte Carlo neutral particle transport method makes it challenging to develop general optimisations. In order to make claims about the application performance, it is necessary to consider a set of test problems that exercise different conditional code paths, and also to represent more realistic test problems. As the problems are introduced below, the characteristics of the routines will be discussed. Each of the problems specified are entirely synthetic, but have been designed to expose the performance characteristics of the application, while being easily validated.

5.4.6.1 The `streaming` problem

The `streaming` problem sources particles in a small region in the center of the spatial domain and allows them to stream freely across the space without collision (Figure 5.5). The specification means that each particle travels across roughly 7000 facets per timestep, so the particles will interact with at least one boundary of the domain within each timestep if the local problem size is 4000^2. This test problem is particularly interesting as it exposes an important issue with the Monte Carlo codes: that particles can move randomly and independently from each other, exposing little to no spatial or temporal locality.

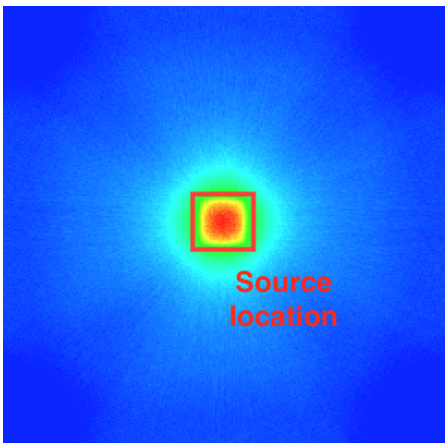

Figure 5.5: Example plot of energy deposition for the `streaming` problem.

In this test problem, the branching overhead is reduced as only the facet event will occur in each timestep, isolating out the performance of the facet events for individual analysis. When a collision or facet event occurs, it is necessary to calculate and store the energy deposition for that portion of the particle history. In a facet event it is also necessary to store this energy deposition tally into the tally mesh, which means continual random access read-modify-write operations to update a mesh the size of the local computational domain. As multiple independent particles interact with the same tally mesh, it is necessary to perform the tally atomically in order to avoid multiple threads updating the same tally mesh location at the same time.

The facet event has to handle the movement of the particle through the computational domain and resolve the reflective boundary conditions, which results in simple code with several levels of nested branches. It is also necessary to fetch the local density of each cell that a particle travels through by accessing the cell centered density mesh. This results in a random read from a mesh the size of the local computational domain. Constructing a test problem that performs

only facet events makes it easy to calculate the figure of merit *facets per second*. This particular metric is invariant to many variables in the problem parameter space, often allowing fairer evaluation of performance than absolute runtime. For this problem 1e6 particles are simulated for a single timestep.

5.4.6.2 The `scattering` problem

The `scattering` problem, as seen in Figure 5.6, sources particles in a high density material that results in only collisions. The particles will typically reach census without leaving the cell they were sourced into.

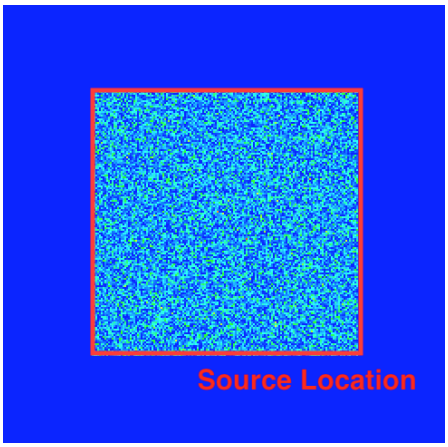

Figure 5.6: Example plot of energy deposition for the `scattering` problem.

During a collision, the particular reaction is randomly sampled based on the macroscopic cross sections for absorption and scattering. Handling an absorption reaction involves a simple update of the particle weight, while the scattering reaction results in a new direction and energy being calculated for the particle. Energy deposition for the trajectory up until the present event will be tallied locally, given that there will be a change in energy or particle weight, but the value need not be flushed into the global tallying mesh. Further, a particle can *'die'*, which is the point at which its energy has fallen low enough that it is not longer of relevance to the simulation. The local energy deposition is tallied globally in the particle's terminal cell, and it is marked to be ignored from that point forward.

Once the collision event has completed, new microscopic cross sections are fetched from the lookup tables, and a new mean free path to collision is sampled. Due to there being several random processes within the collision event, it is necessary to generate up to three random numbers per collision. The simple single-nuclide single-material case will be used as a starting point, but Chapter 8 will consider the implications of including multiple materials and complex materials comprised of multiple nuclides in the solve. As with the facet events, it is possible to use a figure or merit, *collisions per second*, for this particular problem, observing the same benefits of parameter invariance for some parameters. For this problem 1e7 particles are simulated for a single timestep.

5.4.6.3 The csp problem

The csp problem, as seen in Figure 5.7, has been constructed to be more representative of a standard test problem, where particles are sourced within a low density material and stream unless they collide with a square region of high density material in the center of the domain. The problem does not perfectly balance the number of collisions and facet events, but the proportion is more natural than the other synthetic problems.

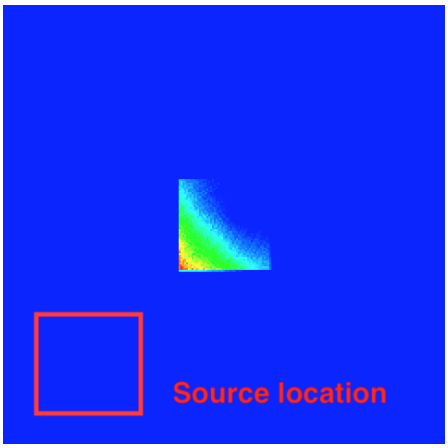

Figure 5.7: Example plot of energy deposition for the csp problem.

Note that, although the problem is more balanced, the number of facet events is significantly higher than the number of collision events. As particles both collide and cross facets in this test problem, all of the branches will be active in the simulation. This has consequences for both vectorisation and GPU programming that will be discussed throughout this chapter. For this problem 1e6 particles are simulated for a single timestep.

5.5 Implementation on CPU

Having understood the performance characteristics of the application, it is possible to perform specific optimisations to the code and algorithms. The targets for this section are Intel Xeon and Intel Xeon Phi CPUs, as they offer diverse architectures but allow a common programming approach.

5.5.1 Over Histories

As previously discussed, the *over histories* approach is the most well known approach to Monte Carlo neutral particle transport and follows the individual histories of particles. The consequence of following a single particle at a time is that the particle data can be cached in registers or cache through the entire history and only written back once the particle reaches census. The memory access of the particles is therefore negligible compared to the overall cost of the full timestep. The processors used in this section are discussed in Section 4.2, and the code is compiled with Intel 18.3.

The results for each processor in Figure 5.8 are not informative in isolation, however, in comparison to each other it is observed that the KNL is significantly slower to execute the same problems as the Skylake CPU. The achievable memory bandwidth is around 2.2x larger

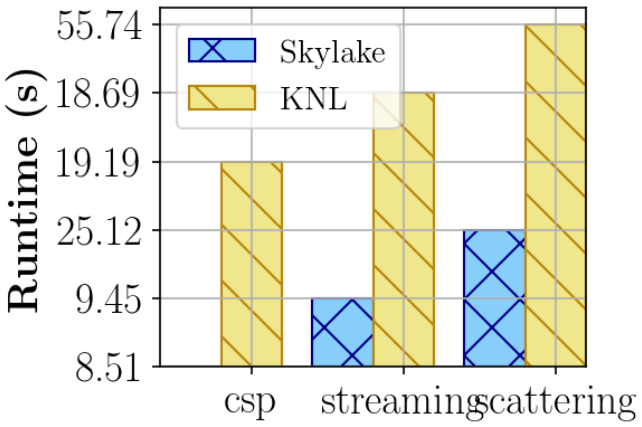

Figure 5.8: Performance of the *over histories* approach for the Skylake and KNL.

for KNL's MCDRAM than Skylake's DRAM, while the dual-socketed Skylake CPUs achieve around 1.3x the compute throughput of the KNL. Given that the KNL requires vectorisation to achieve full memory bandwidth, the lack of vectorisation in the *over histories* approach precludes maximum performance in the KNL. In spite of this, the difference between the Skylake and KNL is so significant that it is expected that there are additional issues affecting the performance, beyond just vectorisation.

5.5.2 Performance Analysis of Over Histories

In order to have some basis for future discussion, an extensive profiling of the *over histories* algorithm running on an Intel Xeon Skylake CPU is performed for each of the test problems. Although fine-grained profiling of the *over histories* approach is possible, the accuracy is typically poor, as each kernel amounts to a small number of operations, where only a single particle is processed at a time per thread and the events are small. For the `scattering` problem, for instance, the solver handles 7 billion distance calculations and collisions within around 25s on an Intel Xeon Skylake CPU. Given 56 cores, then per core this is roughly one event every 200 nanoseconds. Profiling the average time per history or wall clock time is useful, but due to the high problem dependence there is little comparability of results between problems.

5.5.2.1 Profiling the `scattering` Problem

During the `scattering` problem each particle performs roughly 700 events, comprised of: (1) evaluating the distance to facet, and (2) performing the *collision event*. The *collision events* either lead to absorption of the particle, which reduces the particle weight, or particle scattering, where the energy and direction of the particle changes. At the end of the simulation each particle will complete a *census* event, but the performance impact is negligible.

The performance affecting characteristics of this kernel are the two calls to generate random numbers and the two binary searches to find entries in the cross-sectional lookup tables. As seen in Figure 5.9, each particle's energy diminishes over hundreds of events until the particles fall below the energy level of interest. To conform to the approaches taken by other applications, the lookup table is accessed using a binary search, which typically accounts for around 90% of the cost of a *collision event*. For the *over histories* method on an Intel Xeon Skylake CPU, the

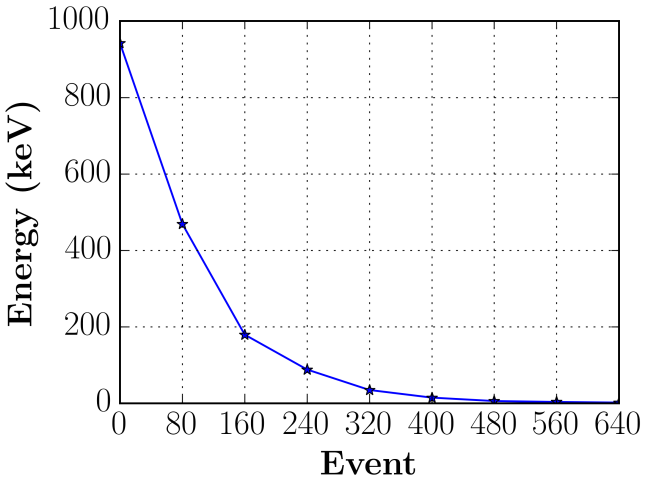

Figure 5.9: The energy profile of a particle throughout the `scattering` problem.

best performance observed for the `scattering` problem was a rate of 2.8e8 *collisions per second*.

5.5.2.2 Memory Bandwidth of the `scattering` Problem

During each `scattering` event, a *collision* event occurs that searches through two cross-sectional lookup tables. Each search is a binary search and so up to 15 individual memory accesses are performed, with at least 11 and at most 12 of those accesses touching distinct cache lines on an Intel CPU. Given the random distribution of particles and trajectories leading to random energy changes, an average of 11.5 individual cache line accesses per binary search is a reasonable approximation. Taking the best results observed on the Skylake CPU for this problem, 2.8e8 *collisions per second*, this would then suggest a memory bandwidth of 404 GB/s for the two table lookups per event. In fact, the cross-section data tables are only 468KiB each, while the size of the L2 cache is 1MiB per core, and so the tables will be cached in L2 and L3 on the Skylake CPU. Given that the bandwidth to L2 is around 5 TB/s (see Appendix B), the bandwidth to cache is significantly underutilised.

As the lookup tables are cached, the majority of DRAM accesses during the solve are to the density and energy deposition meshes. Running on a KNL and collecting the uncore counters, it was observed that a `scattering` test run with 1e6 particles (note that this is an order of magnitude fewer simulated particles than the typical test problem handles) resulted in 430MiB of reads to MCDRAM and 205MiB of writes to MCDRAM. The particle data is around 70MiB, and each particle fetches the material density for a single cell of the mesh, for around 60MiB of additional reads, and each particle performs a final tally leading to another 60MiB of reads and writes. This data confirms that a majority of the lookup table data persists in L2 cache, as two instances of 11.5 accesses per *collision event* would lead to nearly 1 TiB of read accesses.

As with many features of probabilistic transport, this issue is problem dependent, as once the size and quantity of lookup tables changes, data may begin to spill out of cache leading to different performance characteristics. Artificially increasing the lookup tables to 128MiB each, for instance, the throughput dropped to 6.1e7 *collisions per second* from 2.8e8 *collisions per second* on the Skylake CPU. This equates to around 83.6GB/s memory bandwidth, which is around 38% of peak observable DRAM bandwidth for the dual-socketed Skylake CPUs. The

issue of varying lookup table sizes will be discussed in detail in Section 5.8.

5.5.2.3 Computational Throughput of `scattering` Problem

Irrespective of the differences observable as the lookup tables increase, the optimisation of table lookups is deferred. VTune shows that over 90% of all memory accesses hit L1 or L2 cache and that the VPU utilisation is low, which needs to be resolved by vectorisation. In this particular regime, the computations might become the dominant factor, so it is necessary to measure the computational throughput of the `scattering` events on the CPU. Intel CPU hardware counters over-count the number of floating point instructions, as instructions issued where the operands are not yet available might be counted multiple times. A more exact method for counting the number of floating point operations is to use the Intel Software Development Emulator[1] (SDE).

SDE counts around 1e5 floating point instructions for every particle history in the `scattering` problem. The Skylake can execute the `streaming` problem in roughly 25s, and the KNL takes around 55s, which leads to compute throughputs of 40 GFLOP/s and 18 GFLOP/s respectively. Given that each core of the Skylake can turbo to 2.8GHz if not executing AVX instructions, and each core can dual issue FMAs per cycle, the peak performance is 627 GFLOP/s for a dual-socket configuration. This shows that the un-vectorised version of the application is reaching around 6% of peak compute throughput for the non-vectorised code on the Skylake. The same analysis applied to the KNL shows that around 5% of peak non-vectorised performance is achieved. As the vast majority of accesses are serviced from L1 or L2, it appears that neither the memory bandwidth nor the compute throughput are the limiting factor on the CPU. The profiler suggests that the application is back end bound, which is caused by long latency operations. Until the application is successfully vectorised, accurate analysis of the routine is more challenging, and so further profiling is delayed until the Section 5.9.1.

5.5.3 Profiling the `streaming` Problem

The `streaming` problem is particularly interesting, as it focuses on one of the most critical issues with solving particle transport problems using the Monte Carlo method, regardless of the type of simulated particle. When particles stream continuously and independently through the computational domain, accesses to domain variables are randomly distributed, and this is well known to be a challenge for existing processors. The results in Section 4.8 show that achieving full bandwidth for random memory accesses is challenging on the processors considered in this study. Further to this, each facet event needs to atomically write to the global tally mesh. In `neutral` this is the energy deposition tally, and the tally updates are protected against data races through the use of atomic instructions.

5.5.3.1 Memory Bandwidth of the `streaming` Problem

If random sourcing is performed across the entire computational mesh, it is possible to perform comparisons of different problem sizes by adjusting the size of the computational mesh. Figure 5.10 shows the performance change as the size of the computational mesh increases from 512 KiB to 16 GiB. As the memory footprint reaches 32 MiB (2048^2), the performance begins to decrease, to around 2x fewer facets per second by the time the working set reaches 16 GiB. The results show an inverse correlation between the performance and the number of L3 cache

[1] https://software.intel.com/en-us/articles/intel-software-development-emulator

misses, a strong indication that memory bandwidth is the limiting factor in this case. There are a number of other memory accesses beyond those to the mesh data, for instance the particle data and the `x` and `y` location on the mesh. In spite of this, these structures are small enough to fit within L2, where the bandwidth was shown to be large for all of the processors in Appendix B. In this section, the throughput will be measured with respect to the local density and tallying memory accesses only.

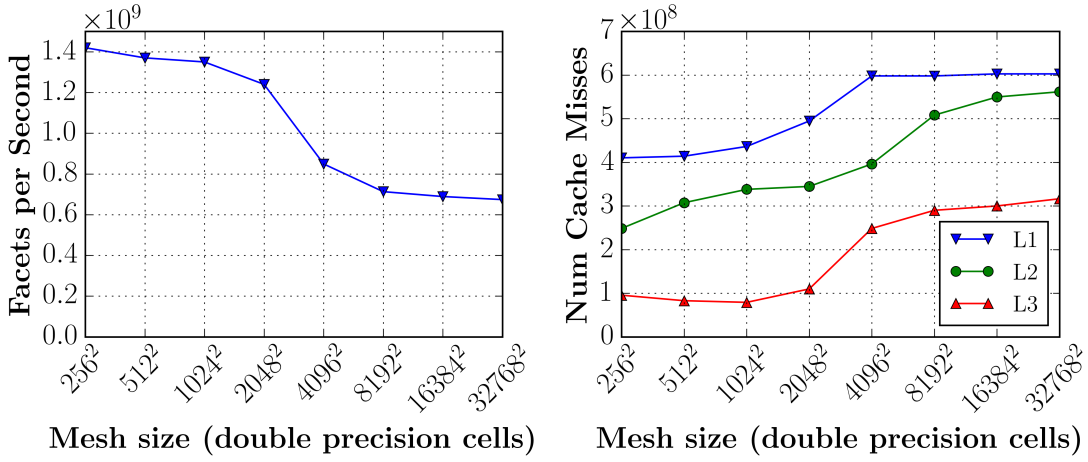

Figure 5.10: Scaling the `streaming` problem and plotting cache misses.

The best throughput observed for the `streaming` problem was a rate of 7.5e8 *facets per second* on the Skylake CPU, or roughly one facet event per core every 75 nanoseconds. Given that each of the facet events fetches the local density and updates the tallying mesh once, there is a single random read and a single random atomic read-modify-write. As such, each facet event reads 16 bytes and writes 8 bytes, and so the memory bandwidth of the application can be perceived as 18GB/s, which is around 9% of the peak achievable DRAM bandwidth. There is an important distinction to be made between effective and true bandwidth, which plays a major role in modeling the performance of `neutral`. Each double precision memory access is actually eight times larger than the previous metric suggests, as the access is at the granularity of a cache line, and given that the solver expresses negligible locality for the facet events, this must be accounted for in the model. As such, the bandwidth calculation can be scaled to 144 GB/s, which is 69% of the peak observable DRAM bandwidth. Although this brings the result much closer to the observable DRAM bandwidth, the analysis does ignore the inherent locality in the problem, which is discussed in Section 5.5.5.

Each of the tally updates is performed atomically, and so it is possible that the memory bandwidth is influenced by the performance of handling the atomics. Changing the energy deposition routine so that it uses a regular read-modify-write, rather than performing the tallying atomically, improves the throughput of the `scattering` problem by 1.2x to 9.0e8 *facets per second* on a Skylake CPU. This suggests that the atomic instruction itself is not limiting the performance by a significant amount, and that the hardware atomics are quite efficient for this particular use case.

5.5.4 Profiling the csp Problem

The csp problem includes a mix of both *facet events* and *collision events*, so it is expected that new issues will arise in the performance as predication is required to enable vectorisation. Due to the dynamic mix of events occurring in every timestep, it is more challenging to reason about the csp problem on a per event basis.

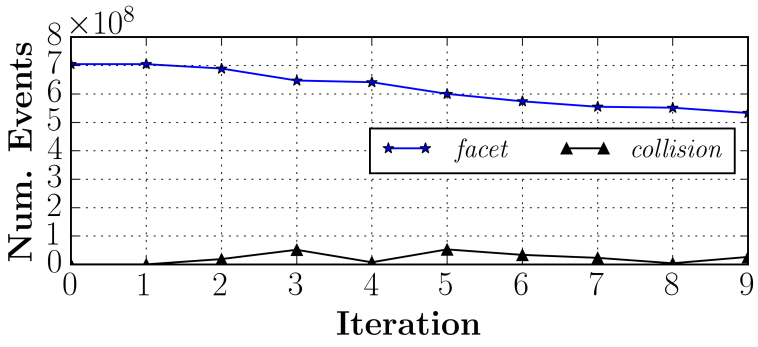

Figure 5.11: The balance of events in the csp problem.

As can be seen in Figure 5.11, the number of collisions is relatively low as particles entering the high density material and colliding multiple times will fall below the energy threshold. It is possible to construct problems where the event profile is more even between the two events, but this is not necessarily more representative of a true problem. There are many problems where the majority of the time the particles will be streaming through materials and only colliding infrequently.

5.5.5 Incidental Locality

Although the trajectories of each individual particle can be considered random with respect to one another, there are some important points of locality that need to be addressed. Due to the structured mesh, there is a chance that the next cell in a particle's trajectory will be on the same cache line as the current cell. This means that some of the memory accesses do in fact exhibit locality, as can be seen by the coloured entries in Figure 5.12. This is less pronounced in the three-dimensional case, where only two of six faces result in the possibility of entering the same cache line.

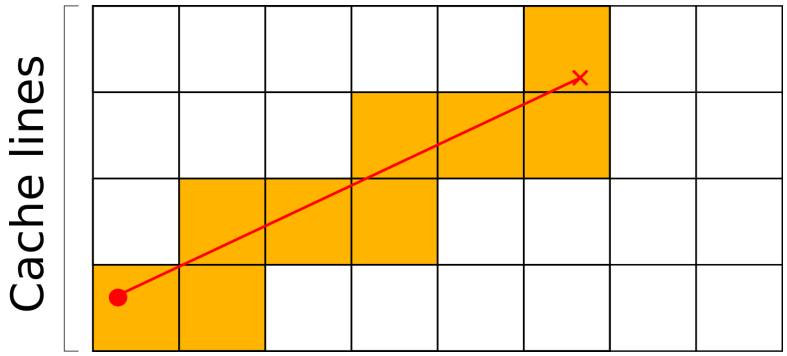

Figure 5.12: The incidental locality of a random particle trajectory in neutral.

On average, the chance for random trajectory to stay in the same cache line is 37.5%, which can be accounted for in any modeling of the performance. The random memory access performance experiments performed in Section 4.8 showed that TLB misses could be almost entirely mitigated with huge pages. The working set for `neutral` is at most 244MiB, and huge pages are enabled for all experiments, minimising the potential impact of TLB misses on the application.

Another interesting issue that can come about from the random locality exhibited by the problem is that the access patterns might invoke the hardware prefetcher in a manner that actually harms performance. Depending upon the angle of trajectory, a streaming particle might cause the prefetcher to continually bring in more cache lines than necessary, as it detects accesses to multiple cache lines, thereby reducing the available memory bandwidth from DRAM due to irrelevant accesses.

5.5.6 Thread Scheduling

It was hypothesised that, given the different frequencies of events encountered by each particle, and the corresponding cost of each event, there might be some load imbalance between threads.

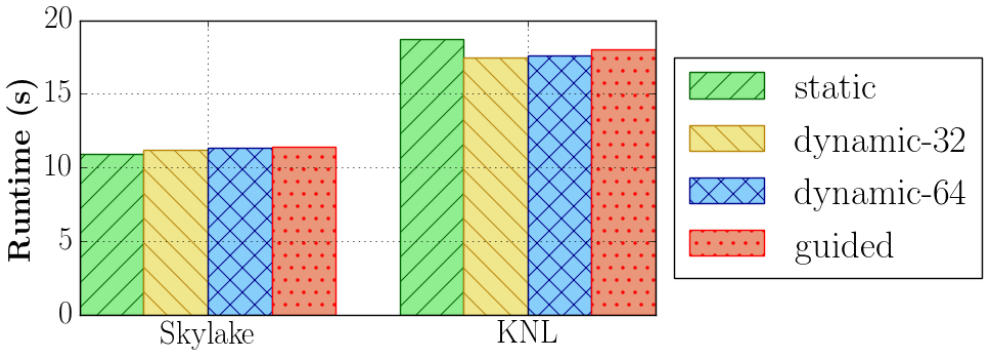

Figure 5.13: Adjusting the OpenMP thread scheduling for the `csp` problem.

Figure 5.13 shows the results of using thread scheduling to mitigate the load imbalance for the *over histories* approach. The performance is marginally improved on the KNL, however, the Skylake performance worsens due to the scheduling overheads if a form of dynamic scheduling is enabled. The load balance of `neutral` is problem dependent, and the `csp` problem does not lead to a significant load imbalance on the CPU architectures. It is nevertheless important to rule this out as a cause of performance differences, and tracing is used to ensure that the optimisation discussed throughout the chapter do not introduce unexpected load imbalances.

5.5.7 Hyperthreading

For many scientific applications, hyperthreading has little impact on performance as memory bandwidth or compute performance dominate the core kernels. In spite of this, Tramm et al. found that hyperthreading directly improved the performance of a stripped back kernel representing macroscopic cross sectional lookups [157]. Their observation was that the hyperthreading improved the performance of XSBench, as it allowed more read requests to be in flight. The macroscopic cross section lookup used in their study requires a binary search, which has a random memory access pattern, the same as the cross sectional lookup in `neutral`. Further to this,

`neutral` requires random memory accesses for particle tracking and tallying, which might also benefit from hyperthreading.

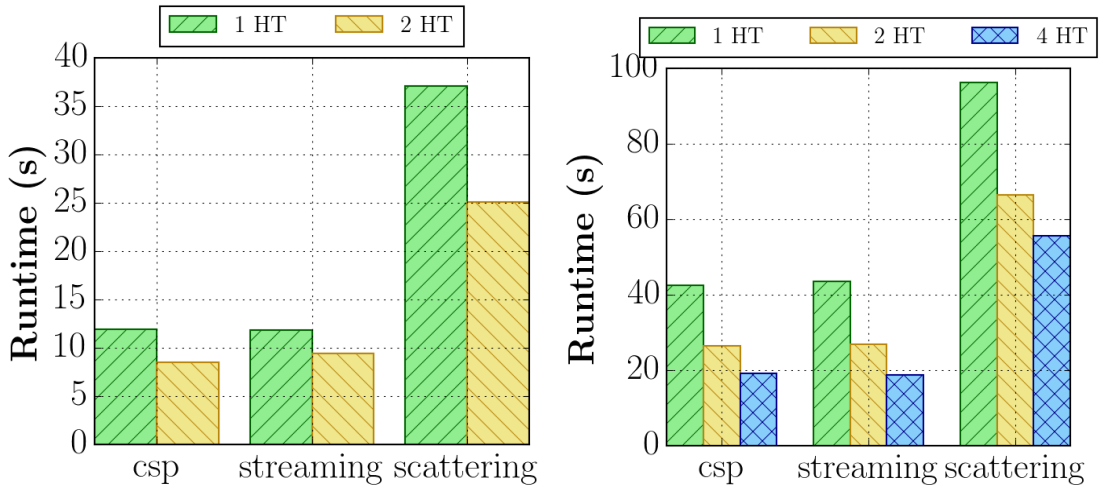

Figure 5.14: Adjusting the number of hardware threads for the problems in `neutral`. The results are for Skylake (left) and KNL (right).

The results in Figure 5.14 demonstrate that a significant increase in performance is observed as the number of hyperthreads is increased on both the Intel Xeon Skylake and Knights Landing processors. On the CPU, increasing the number of threads per core from 1 to 2 improves the runtime of each test problem by 1.3x to 1.5x, while on the KNL the performance improves by 1.8x to 2.5x. It is relatively common for applications to improve as the number of hyperthreads is increased on the KNL, as more than one hardware thread is generally required to saturate memory bandwidth [128]. In contrast, it is rare that scientific applications benefit from additional hyperthreads on CPUs, and some HPC clusters even disable hyperthreading. The results here suggest that there is a significant difference in the performance profile between the `neutral` application and other more typical scientific applications.

The `streaming` case, which does not have to update macroscopic cross sections in this single-material problem, also observes a significant improvement in performance. It is therefore hypothesised that the improved performance through hyperthreading observed by Tramm et al. related to random memory access performance. The results of Section 4.8 showed that the *simple* mode of the random memory access benchmark benefited significantly from an increase in hyperthreads, corroborating this hypothesis. When applied to `neutral` it is shown that even the random memory accesses for particle tracking could be improved through the use of hyperthreading.

5.5.8 Over Events

Using the *over events* approach described in Section 5.4.4.2, it is possible to expose vectorisable streams of work, which might enable improved performance compared to the *over histories* approach. Vectorisable loop structures mean it will be possible to place a greater number of loads in flight, hopefully overcoming the issues with random memory access performance observed for the *over histories* approach, and discussed in Section 4.8.

5.5.9 Sort-free Over Events

Taking advantage of vector masking and predication rather than sorting particles into streams is expected to be vastly superior on modern architectures, where data movement is so expensive.

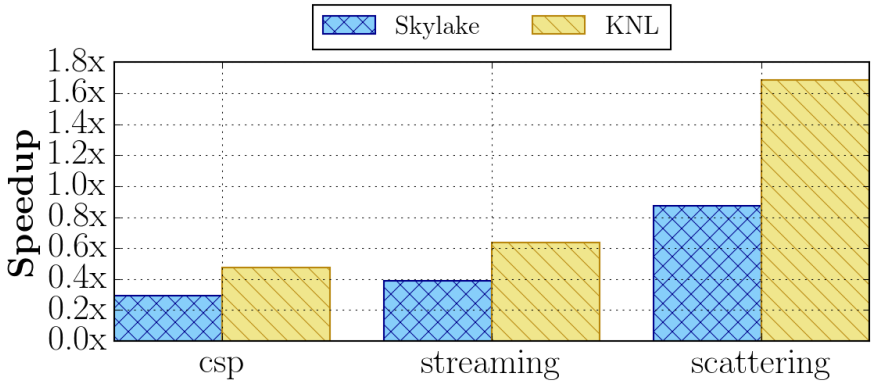

Figure 5.15: The performance of the *over events* approach with respect to the *over histories* approach.

Figure 5.15 shows that the *over events* approach actually leads to worse performance for the majority of the test cases, and in many cases the slow down is significant. The single case where the *over events* approach is shown to be superior is the `scattering` case on the KNL, and the result for the Skylake is closer to parity as compared to the other test problems. There are multiple contributing factors to the poor performance for the *over events* model. The particle data structure is increased in size as fewer elements of the particle history can be stored in registers, but must instead be stored against the particle to avoid expensive re-computations. Depending upon the problem and whether some subset of the particle population is operated on, it might be necessary to fetch the particle data from memory, resulting in greatly increased memory access costs.

In the `scattering` case for *over histories*, the amount of memory access was strictly limited by the size of the lookup tables, as discussed in Section 5.5.2.1. Enabling vectorisation allows the application to exploit the full computational throughput of the architectures, potentially allowing significant performance improvements for the small lookup table case. In the *over events* case, the particle data can no longer be cached and so the `scattering` case becomes bottlenecked on the Skylake's memory bandwidth. On the contrary, the MCDRAM on the KNL offers high enough memory bandwidth for the application to benefit from the improvements in computational throughput.

For the CPU technologies, the *over events* approach was not successful in improving performance compared to the *over histories* approach in the majority of cases. In Section 5.7 a novel approach to improving the issue will be presented for the CPU and KNL.

5.6 Implementation on GPU

Conventional wisdom suggests that the *over histories* algorithm would not be well suited for acceleration on a GPU. In this thesis all of the different algorithmic variants were considered on the GPU and the results demonstrate important characteristics of GPU programming that contradict popular belief about their capabilities. It was suggested by van Heerden et al. that

traditional *over histories* parallelisation on the GPU would only be able to observe moderate speedups due to the level of divergence between threads [164]. They outlined an approach for avoiding warp divergence, by assigning individual particles to warps, allowing a sharing of data between the particles. The cooperation between threads within each warp appears to rely upon a number of factors, for instance, the geometric data being a reasonable size to be fetched by the full warp. The main drawback for use with `neutral` is that the computations are not amenable to cooperation within a warp, and the geometric data is small enough that it would not be profitable for coalesced access. In `neutral`, allocating particles to warps vastly underutilises the GPU compute and memory resources.

Liu et al. investigated the difference between the *over histories* and *over events* approaches, but found that the simulation speed was an order of magnitude slower for the *over events* approach [90]. Their investigation suggested that the large number of global memory transactions was the determinant factor in this. The original research was conducted with an NVIDIA Tesla M2090 GPU, and so re-consideration of this approach with new hardware and modern insight might yield improved results. The details of the V100 GPU used in this chapter are discussed in Section 4.2, and all code is compiled with CUDA 9.0.

5.6.1 Over Histories

The reason that the *over histories* approach appears to be poorly constructed for the GPU is that it leads to code with many deep branches. An implementation of the *over histories* approach was developed in `neutral` using CUDA, which essentially comprised of a single large computational loop parallelised over the particle population.

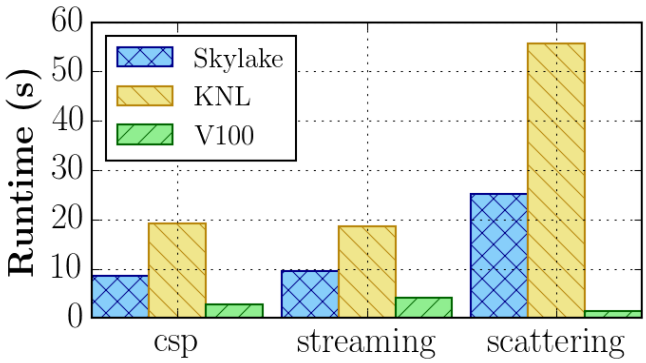

Figure 5.16: Performance of the V100 compared to the Skylake for the *over histories* approach.

Figure 5.16 shows the performance of the *over histories* algorithm on a V100 GPU compared to the Skylake, where the performance of the `streaming` problem is 2.3x faster, while the `scattering` problem was around 20.5x faster. The stark difference between the CPU and GPU in this case is caused by a number of coinciding issues. The CPU implementation is not vectorised, while the particles are distributed evenly across warps on the GPU, allowing better use of the GPU compute and memory resources. The `streaming` case on the GPU achieves a true bandwidth of 327 GB/s, which is around 41% of read-write bandwidth, as shown in Section 4.5. This proportion of peak is less than observed for the Skylake CPU, which achieved around 69% of peak achievable DRAM bandwidth.

As previously mentioned, this particular problem specification only accesses a small quantity

of lookup table data, more representative of energy group codes, but the performance can change as the lookup table is increased in Section 5.8. The results defy the expectation that the branching within the kernels make the algorithm untenable on the GPU, as even the `csp` problem is able to improve upon the CPU performance.

Problem	Read B/W	Write B/W	% of Achievable
`csp`	179GB/s	48GB/s	29%
`streaming`	187GB/s	48GB/s	30%
`scattering`	190GB/s	6GB/s	25%

Table 5.1: Bandwidth results collected with `nvprof` on a V100 GPU.

Table 5.1 shows the memory bandwidth as reported by `nvprof` for the *over histories* approach on a V100 GPU. Importantly, the bandwidth measured by `nvprof` is a true bandwidth measurement, derived by measuring the individual accesses across the bus. The reason for the disparity between the expected true bandwidth and the measured true bandwidth relates to the previously discussed issue of incidental locality (Section 5.5.5). Locality is present in the 2D problem that means that the bandwidth measured as 2 cache line reads and 1 cache line writes over-estimates the volume of main memory accesses if caching is not taken into account.

The 'stall_memory_dependency' counter shows the percentage of stalls that occur because resources are not available for a memory request or the maximum number of memory requests possible are in flight. For `streaming` the result is 90% and for `scattering` it is 32%. The high fraction of stalls due to memory requests is important as it shows that, for the `streaming` problem, memory performance is being limited due to the number of outstanding memory requests being saturated. Considering that the achieved bandwidth was only 30% of observable peak, there is a disparity between the memory requests being in flight and the successful memory accesses. Further to this, the `scatter` problem, reaches around 1.3 TFLOP/s, which is around 19% of peak compute performance, alongside 25% of peak memory bandwidth.

5.6.1.1 Sort-free Over Events

Having successfully implemented the *over histories* approach on the CPU, it was possible to implement the *predicated over events* approach on the GPU. As the approach finds long streams of work, it was expected that the performance could improve upon the *over histories* approach on the GPU. This was especially expected to be true for the `streaming` and `scattering` problems, as the events performed by the problems would be the same for all particles in the stream, minimising thread divergence.

The results in Figure 5.17 show that the performance is significantly worse than expected, with the *over events* approach executing 2-3x slower than the *over histories* approach. There are several reasons why the performance of the *predicated over events* approach did not reach the expected performance. It is useful to first consider the single event problems, `scattering` and `streaming`, ignoring the top level of branching in the application.

For both of the single event problems, the GPU achieves around 80% of observed peak memory bandwidth for the event handling routines. The reason that the memory bandwidth utilisation has been greatly increased is that the total amount of memory accessed has been increased, as the particle data needs to be re-fetched for each event loop. The traditional *over events* approach does not suffer from this issue as a queue of particles is maintained, reducing the

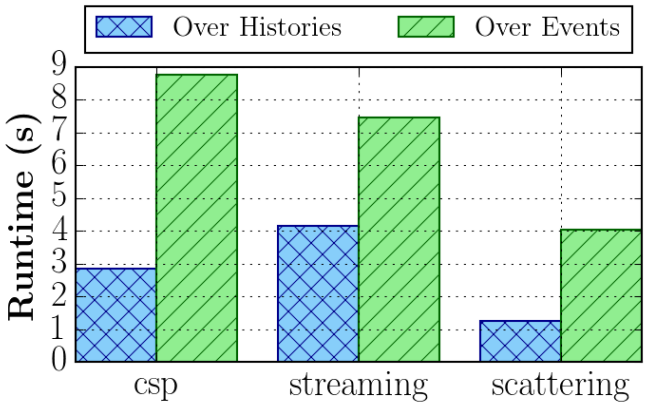

Figure 5.17: Performance of the *predicated over events* approach on a V100 GPU.

amount of particle data that is re-fetched. In spite of this, the traditional *over events* approach was prohibitively slow due to the necessity to sort, meaning that the *over events* approach appears to be less suited to the GPU than originally expected.

In fact, the *over histories* approach avoids any performance overhead for accessing the particle data, meaning that the performance is instead limited by the random memory accesses for lookup tables and streaming accesses. The *over events* approach reduces branching but potentially at the expense of increased global memory access, unless the working set is constrained.

5.7 Enabling Vectorisation via Blocked Over Events

It has been shown that the *over histories* algorithm in `neutral` does not vectorise with the optimising compilers available at the time of writing. The *over events* approach enables vectorisation with sorting or masking, but the performance was generally significantly worse than the *over histories* version regardless of which approach was taken. It is clear from the performance analysis that the CPU architectures could benefit from vectorisation, particularly in the streaming case where the compute performance is an influential factor. In collaboration with Intel, a new algorithm (Code Sample 5.4) was developed that attempts to capture the benefits of both the *over histories* and predicated *over events* approaches at the same time, which will be referred to as the *blocked over events* approach throughout.

The approach initialises a number of threads, which act as coarse grained tasks that independently work on batches of particles. Each thread decomposes its batch of particles into blocks, and works on each block in turn, taking all particles in the block to census before moving on to the next block in the batch. Inside the `event_loop` of Line 7 the loop structure is essentially the same as the *over events* approach, except that the loop bounds are now compile time constants, that can be set to less than or equal to the number of particles N. As the loop structure is the same as the *over events* approach it is possible to vectorise the loops by masking out vector lanes based on a particle's participation in the current event.

The approach introduces the block size as a parameter. An initial default block size of 32 was chosen, as this would span four AVX512 registers, allowing for four way unrolling, ensuring that the dual-issuing of the Skylake CPU is taken advantage of, alongside any potential instruction-level parallelism. This approach is similar to the predicated *over events*, except that the block

size can be changed to ensure that the particle data is cached in a high level of cache, removing the issue of particle data re-fetching. Further, the problem of synchronisation is solved as the SPMD pattern is applied over blocks, while allowing vectorisation of the inner loops, meaning that the synchronisation costs are reduced to the same as the *over histories* approach.

Code Sample 5.4: The *blocked over events* algorithm for `neutral`.

```
1  block_loop:
2    choose_block(particles) -> block_particles
3
4    for particle in block_particles:
5      cache_particle_data(particle)
6
7    event_loop:
8      for particle in block_particles:
9        calculate_time_to_events(particle)
10
11     for particle in block_particles:
12       if colliding(particle):
13         handle_collision(particle)
14
15     for particle in block_particles:
16       if encounter_facet(particle):
17         handle_facet(particle)
18
19     if all_particles_at_census(block_particles):
20       exit event_loop
21
22   for particle in block_particles:
23     handle_census(particle)
```

The approach introduces the possibility for executing on a sliding scale between the *over histories* and *over events* approaches. The block size can be changed between 1 particle per block for the *over histories* approach and `N/nthreads` particles per block for the *over events* approach. Executing the optimised implementation in the two extreme cases added a minor overhead of around 10% compared to the *over histories* implementation but added no overhead compared to the *over events* implementation. As such, the algorithm could be tuned to take advantage of the benefits of either approach when targeting a specific problem.

5.7.1 Vectorising the Collision Event Routine

The facet and census events, and distance calculations, all vectorised well with only minor adaptations to the event code. It was necessary to perform some additional work to make sure that the collision events were vectorised with the Intel compiler.

5.7.1.1 Restructuring of the Binary Search

The binary search that handled the discovery of the energy index for the current nuclide was originally implemented using a simple `while` loop. This implementation inhibited vectorisation using the Intel compiler targeting both the Skylake and KNL, due to the complex loop constraint.

In order to resolve this issue a `for`-loop based binary search was developed that calculates the initial trip count from the base-2 logarithm of the number of entries in the table, while using the same constructs in the loop body.

5.7.1.2 Intrinsic Atomic Call

Under certain circumstances, each of the events are responsible for atomically writing back to the tallying mesh for energy deposition. Using OpenMP as the parallel programming model for parallelising the code introduced an issue in this regard as the use of the `atomic` directive from within a vectorised region was expressly prohibited. The resolution to this issue was to develop a simple intrinsic routine that could be called within the vectorised routine, which overcame the restriction of OpenMP. Having fixed the binary search and removed the `atomic` directive the routines now successfully vectorised. In spite of this there were still performance issues that needed to be resolved.

5.7.2 Tunable Block Size

Being able to tune the block size means that it is possible to control the amount of particle data caching that occurs while ensuring that there is enough data to fill some number of vector registers. This additional parameter had to be tuned for both the CPU and KNL, as it was not feasible to determine the best possible choice analytically.

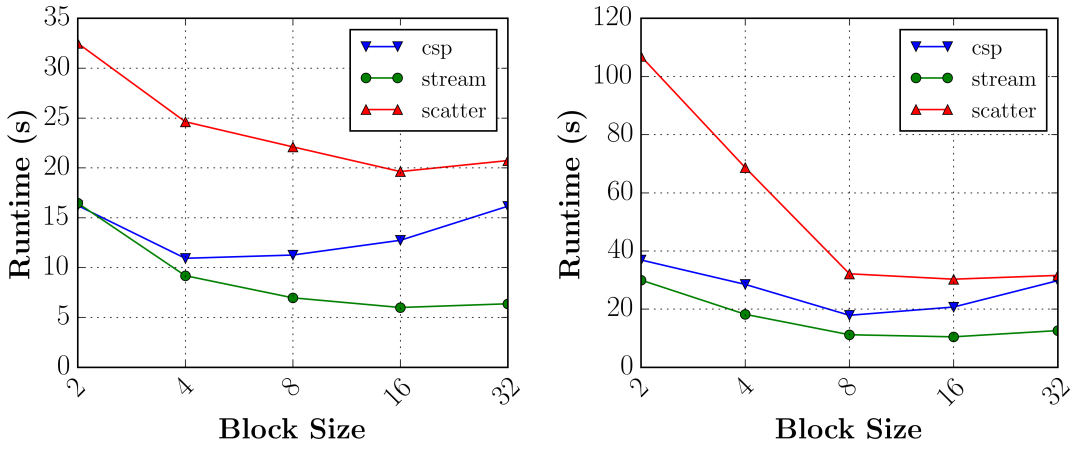

Figure 5.18: Tuning the block size for the *blocked over events* algorithm, for the Skylake (left) and KNL (right) CPUs.

The results in Figure 5.18 are for the Skylake and KNL CPUs, executed with the *blocked over events* algorithm with all routines successfully vectorised according to the Intel compiler reports. On the KNL, the block size of 8 is shown to provide the best performance across the three test cases, while on the Skylake the different events observe significantly different performance results as the block size is increased. The `csp` problem in particular worsens as the block size is increased beyond 4, which is indicative of an overhead introduced by the vectorisation of the problem. The block size of 4 is presumably the point where the vector masking incurs the least overhead.

5.7.3 Particle Data Structures

The Array of Structures (AoS) particle data layout means that the `x` and `y` coordinates are packed next to each other in memory, and so both can be brought into L1 at the same time. This is the optimal choice of data structure for the *over histories* approach, which works on individual particles, but the same is not necessarily true for the *blocked over events* approach, which works on batches of particles. Although the access to particle data represents a low proportion of memory accesses, as seen with the *over histories* approach, the *blocked over events* approach is vectorised. As such, the data structure affects the vector instructions output by the compiler.

Code Sample 5.5: Access to particle data within the SIMD region for AoS.

```
1 #pragma omp simd
2 for (int ip = 0; ip < np; ++ip) {
3   if (particles[ip].dead) {
4     ...
5   }
6 }
```

Code Sample 5.5 shows the beginning of a vector loop where the particle data needs to be accessed. The condition stops the loop from updating the state of particles that have fallen below the energy level of interest, and are now considered 'dead'. In order for each lane to fill a register that can test this branch, it is necessary to gather the strided variable 'dead' from the particle data structure. This example extends to all other particle data accesses within the vector loops, causing frequent gathering and scattering. Changing the data structures to Structure of Arrays (SoA), or Array of Structures of Arrays (AoSoA) will resolve this issue, making it so that gathers and scatters are not required. Bareford et al. improved the performance of miniEPOCH, and Shulenberger et al. improved the performance of QMCPACK by changing structures to an AoSoA layout [147] [11].

5.7.3.1 Performance

The results are now plotted for the change in data structure for both the Skylake and KNL CPUs, where the block size for all tests was set to 8 (see Section 5.7.2).

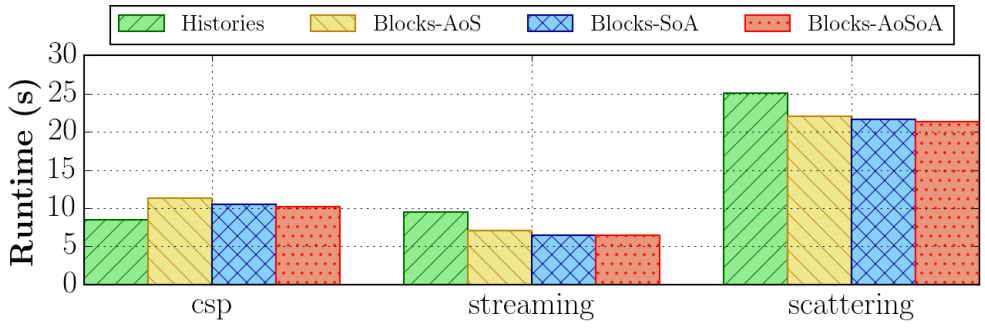

Figure 5.19: Altering the data structure for the *blocked over events* approach on the Skylake CPU.

The performance results shown in Figure 5.19 demonstrate that a reasonable performance improvement is observed through the vectorisation of the `streaming` and `scattering` test problems. The `csp` problem, on the other hand, performed significantly worse than the *over histories*

approach. Again, this is expected to be caused by overheads associated with the masking in the `csp` problem. Regardless of the comparison to the *over histories* approach, all of the versions performed best with the AoSoA data structure, although the performance difference was minimal between SoA and AoSoA.

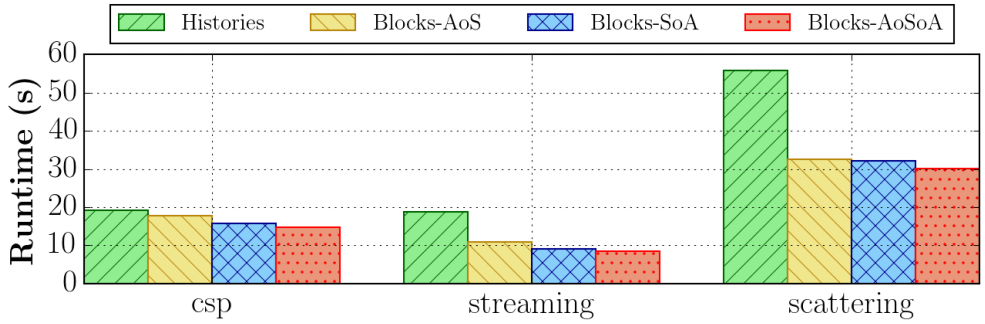

Figure 5.20: Altering the data structure for the *blocked over events* approach on the KNL.

Figure 5.20 presents the results for the KNL, which show a performance improvement across all of the test problems. The `csp` problem was significantly faster to run with the *blocked over events* approach with the AoSoA data structure, and both the `streaming` and `scattering` problems sped up significantly as a result of the vectorisation. As with the Skylake CPU, the AoSoA data structure was the best performing for all problems, by a slightly larger but still small margin.

Random123, while theoretically vectorisable, lead to inefficient instructions that reduced the vectorised performance. Although a vectorisable version of Random123 has been developed, the implementation requires calculation of batches of random numbers, which increases the memory footprint and was not useful for this particular application. The Random123 library was replaced with the vectorisable PCG RNG library provided by Intel [127].

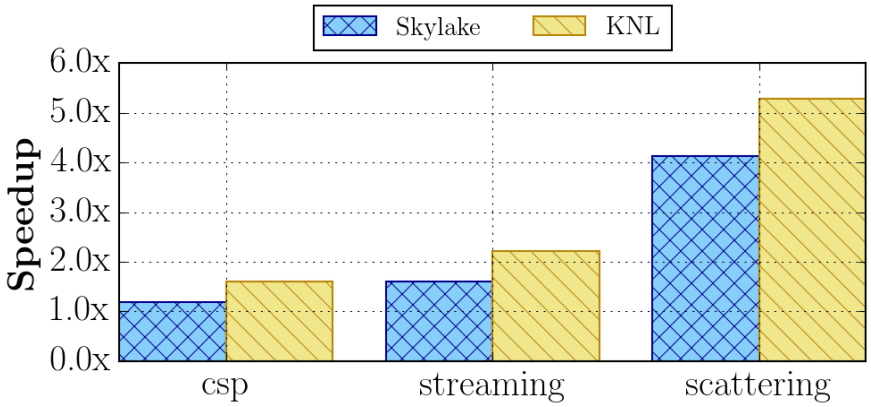

Figure 5.21: The speedup of the *over blocks* approach compared to the *over histories* approach for the Skylake and KNL.

The final speedups for the *blocked over events* approach using the PCG RNG library can be seen in Figure 5.21. The results show that a vectorised implementation is important for applications with small lookup tables, and that the KNL greatly benefited from vectorisation in all of the kernels, although it was not able to best the performance of the dual-socketed Skylake

CPUs. While the improvements to the `streaming` and `scattering` case were significant on the Skylake CPU, the `csp` problem appears to benefit less from the optimisations due to the loss of efficiency from masking vector operations with a mix of events.

5.8 Increasing the Lookup Table Size

One of the main points of problem dependence for Monte Carlo neutral particle transport is the capacity of lookup tables accessed during the solve. To demonstrate the change in performance that occurs for large lookup tables, the unionised grid approach is implemented in `neutral`. The unionised grid approach means that only a single binary search needs to be performed for each cross sectional lookup, regardless of the number of different nuclides in a particular material [22]. A large number of nuclides, 300, is chosen to approximately match the number of nuclides used in benchmarks of reactor simulations [158]. The size of the lookup tables was also increased to 1e5 entries, to make the overall pool of lookup entries larger.

The data touched by the particles in the simulation depends upon the energy profile that the particles transition through during the histories. This does not significantly affect the binary search, which still has to step through the whole energy grid, but does affect the number of cross sectional entries that will be accessed by a particle population, potentially limiting the memory footprint. In a regime where the accesses to the lookup tables are likely to be more expensive than the binary search, it was important that the initial energy profile of the particles was randomly distributed, to avoid inadvertently introducing locality.

Device	**Skylake**	**KNL**	**V100**
Mem. Bandwidth	180 GB/s	126 GB/s	130 GB/s

Table 5.2: The memory bandwidth achieved by the different processors when executing the `scattering` problem for 300 nuclides.

The large lookup table regime significantly alters the performance profile of the collision events. Using the uncore counters it is possible to determine the amount of memory that is moved during the large lookup table version of the `scattering` problem, around 2 TiB. The results in Table 5.2, for the *over histories* approach, show that the Skylake CPU achieves a high fraction of peak memory bandwidth for this problem, around 180 GB/s or 83%. The KNL achieves a lower fraction of peak, at around 126 GB/s, or 28%. Introducing this functionality into the optimised *blocked over events* implementation marginally improves the Skylake performance to around 195 GB/s, but the KNL achieves a lower fraction of memory bandwidth. It is hypothesised that the reason that the KNL performance worsens is because the memory requests are now served as gathers from MCDRAM, saturating the finite gather units. The performance on the GPU is around 130 GB/s, which is a low fraction of peak performance, and is caused by the fact that the memory accesses are not coalesced. In order to coalesce the read accesses, it is necessary to make blocks co-operate on the fetch of the nuclide data, which requires a significant restructuring of the *over histories* algorithm.

It is observed that when simulating problems with many nuclides, it is likely that the entire simulation will become bound by memory bandwidth. Given that the GPU and KNL offer high bandwidth memory, it should be possible to achieve even higher peak memory bandwidth by tailoring the algorithms to account for this.

5.9 Performance Portability

In this section, the performance portability of the `neutral` mini-app will be considered with respect to different programming models and the optimal algorithms discovered during the chapter.

5.9.1 Best Cases Across Architectures

In this section, the *blocked over events* implementation for the CPU, and *over histories* implementation for the GPU will be considered as the optimal implementations.

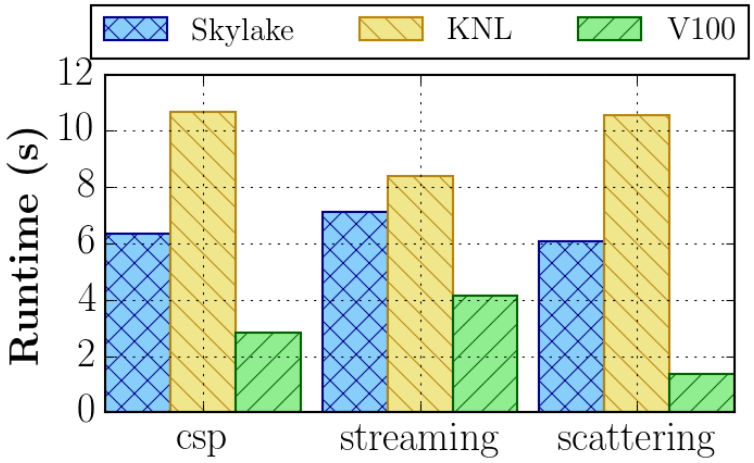

Figure 5.22: The performance of the best performing versions of `neutral` on the 3 parallel processors.

Figure 5.22 shows that the best achieved memory bandwidth on the Skylake CPU for the `streaming` case is around 144 GB/s, while the KNL achieves around 120 GB/s. The fraction of peak memory bandwidth can be improved on the Skylake by splitting MPI ranks per NUMA domain, reducing the costs of random communication across the domains. The V100 GPU achieves around 248 GB/s, or 30% of peak memory bandwidth, which is limited because the random memory accesses to the density and energy deposition tally meshes cannot be coalesced. The results on the KNL are close to the results that are observed with the *simple* mode of the random memory access benchmark in Section 4.8. It is understood, through consultation with Intel, that the limitation in this case is that prefetching is not possible, and this restricts the achievable bandwidth. The KNL and V100 achieve 71% and 59% of the best observed random memory access performance, respectively, when executing the `streaming` case.

The `scattering` case achieves 260 TFLOP/s on the Skylake CPU, around 8% of peak compute throughput, with negligible memory bandwidth used as the majority of the data is resident within cache. From profiling, and considering the instruction mix, it is hypothesised that the reason the performance is limited on the CPU is due to the latency of long chains of dependent L2 cache requests required to complete the binary search. Another issue that increases the latency in the solution is that each collision event can perform up to 4 double precision `sqrt` operations, and up to 8 double precision divisions, and a double precision `log` operation. The latency hiding nature of the GPU helps the latency of the memory accesses

and long-latency operations to be hidden in the `scattering` case, enabling a greater fraction of compute performance, 1.3 TLOP/s, or 17% of peak throughput, to be achieved.

The `csp` problem does not improve as much as the single events on the CPU, suggesting that the cost of masking becomes a limiting factor to the performance. Importantly, the best performance was achieved on the three architectures using two radically different approaches. This introduces a challenge in terms of achieving performance portability, that is made even more difficult by the differences in performance observed in Section 5.8, for the different lookup table sizes.

5.9.2 Programming Model Performance

The other Monte Carlo codes discussed in this thesis, for instance Quicksilver, and OpenMC, use an *over histories* style algorithm, and so it is the most interesting algorithm from the perspective of performance portability for programming models [20] [140]. The *over histories* implementation of the `neutral` application has been ported to run on Intel Xeon CPUs, Intel Xeon Phi CPUs, and NVIDIA GPUs, using a number of different parallel programming models, and the results will be discussed in the following section. Note that each of the independent code ports were algorithmically and compute-workload identical, ensuring that the results can be consistently compared between the different implementations.

The processors used in this section are discussed in Section 4.2, and the compilers used are: Intel 18.3 for OpenMP and RAJA on the Skylake and KNL, PGI 18.5 for OpenACC on all targets, CCE 8.7.4 for OpenMP 4.5 on the GPU, and CUDA 9.0 for all GPU implementations. First, the performance is compared on the Skylake CPU.

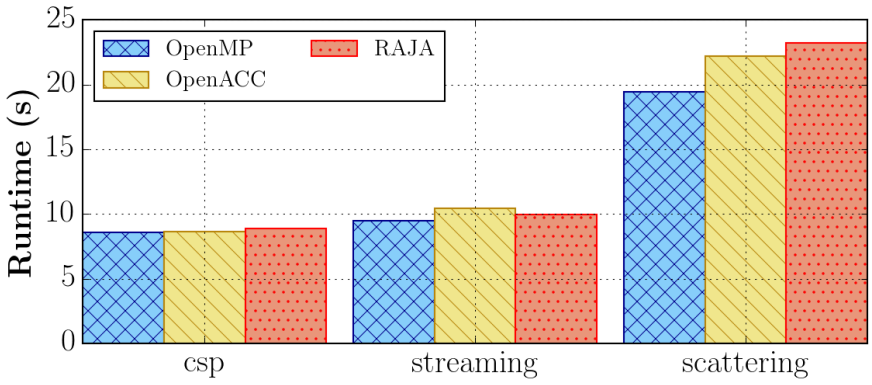

Figure 5.23: The performance on `neutral` executed on a Skylake CPU with varying programming models.

In Figure 5.23 the results are presented for executing each of the test problems using OpenMP, OpenACC, and RAJA, all targeting the Skylake CPU. As can be seen from the plot, the performance portable implementations are close to the performance of the best performing OpenMP implementations for all of the test problems (within 5% for `csp` and `streaming`, and 15% for `scattering`). In fact, the code generation for each of the models will lead to similar outputs, as RAJA internally uses OpenMP and OpenACC uses a similar threading model for its multicore-targeting capabilities.

Figure 5.24 shows the results of executing the *over histories* version of `neutral` using OpenMP, OpenACC and RAJA targeting a KNL. The results show that there is a larger

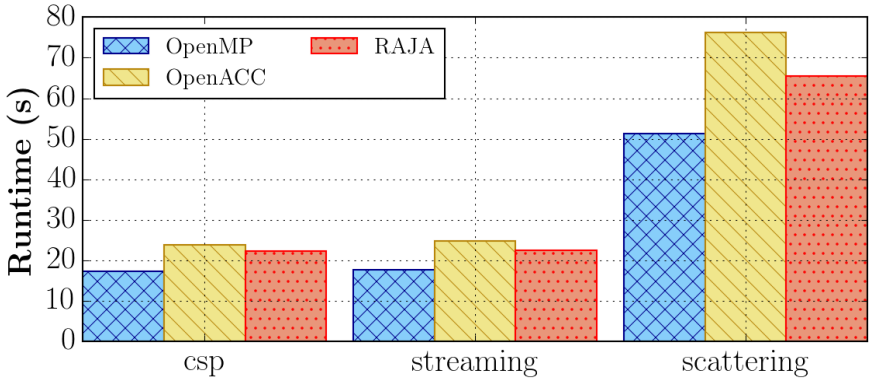

Figure 5.24: The performance on `neutral` executed on a KNL with varying programming models.

overhead of around 40-50% for OpenACC, while RAJA executes with a relatively consistent overhead of around 28%. The OpenACC support for KNLs is relatively recent, and the primary focus of optimisation efforts has been for the Skylake CPU, which is obvious from the results in Figure 5.23. Given that the Intel Xeon Phi line of processors has been discontinued, further optimisation of the PGI OpenACC code generation for the KNL is unlikely.

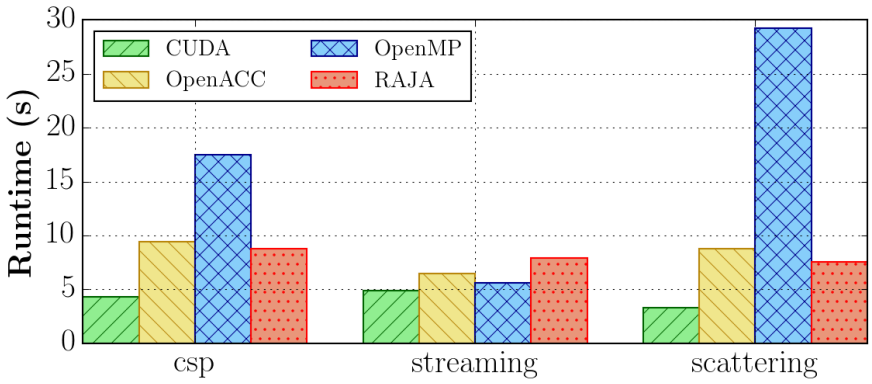

Figure 5.25: The performance on `neutral` executed on an NVIDIA P100 GPU with varying programming models.

The results in Figure 5.25 show a significant difference in the performance of the parallel programming models, which was not seen with the Skylake. CUDA being the low-level architecture-specific programming model achieves the best performance, by a significant margin in some cases. None of the performance portable programming models perform well for the `csp` problem, due to the poor performance of the `scattering` case, although the results for the `streaming` case are good for OpenACC and OpenMP, and tolerable for RAJA.

It was not possible to compile `neutral` using OpenACC, OpenMP, or RAJA, using the Random123 library, which has a GPU-optimised implementation. Instead, PCG was used, and it is possible that this implementation introduces some overhead for the performance portable solutions for the `scattering` problem. The OpenMP `scattering` is the worst case, and even this result required the number of teams to be increased so that there is only one iteration of the particle loop per thread. This fine-tuning improved the performance by 20%, but no other

optimisations had an impact.

One observation is that the register count is significantly higher for OpenMP at 128 registers per thread, but that this appears to be a hard cap imposed upon compilation by the Cray compilers, which suggests that the register count could be even higher. The CUDA implementation uses 76 registers, while the OpenACC implementation uses 136, which leads to a low occupancy of around 15%. Limiting the number of registers to 128 improves the performance of the OpenACC implementation by around 10%. The OpenACC implementation only adds an overhead of around 1.25x floating point operations, but the utilisation of memory bandwidth is reported as significantly higher by `nvprof`, suggesting more data is being stored in global memory.

5.10 Summary

Monte Carlo neutral particle transport is a useful exemplar case that has given insight into not just a general class of algorithms and scientific applications, but has been useful for exposing characteristics of a broad range of target architectures and parallel programming models. The algorithms for solving Monte Carlo neutral particle transport include random memory accesses, varying sizes of lookup tables, atomic updates, and load imbalance, for millions of independent continuously transporting particles. Depending upon the input problem, the application can have multiple performance affecting bounds, emphasising that it is not always possible to determine a specific bound for an application, rather the bound is potentially tied to the problem specification. The performance impact seen when increasing the lookup table size shows that the problem dependence of the Monte Carlo neutral particle transport application means that different algorithms might be necessary for different problem specifications. This makes it particularly challenging to implement a single optimal solver that can cover different problem cases on modern architectures.

It was possible to characterise the performance of a number of asymptotic problem specifications using `neutral`. The `scattering` problem was shown to be limited by latency for the small lookup tables, and the GPU was able to overcome those latencies to some extent to achieve a higher fraction peak throughput. For the large lookup tables, the adjusted `scattering` problem becomes memory bandwidth bound on all of the architectures, and more work is needed to optimise the KNL and GPU implementations for this particular case. The `streaming` case is memory bandwidth bound, although the modeling of the memory bandwidth is challenging. Accurate analysis must take into account the true memory bandwidth and the incidental locality in the problem. The `csp` problem most closely tracks the `streaming` case, but in some cases the performance is limited by the masking required to handle both event types.

Until `neutral` was optimised and vectorised, memory and instruction latency dominated the performance on the CPU and KNL. Through the development of a novel sort-free blocked particle tracking algorithm, and a number of optimisations to the random number generation and energy deposition tallying, it was possible to vectorise the entire particle tracking process efficiently. The results were a significant improvement in the single event performance, particularly the `scattering` events. The final results show that a high fraction of peak performance for `streaming` events is achievable on the Skylake CPU, and considerable improvements were observed on the KNL, although there is still room for improvement. Although the `scattering` case improved most from vectorisation, the performance is still sensitive to latencies.

Chapter 6

Heat Diffusion via a Conjugate Gradient Solver

Key Publications

M. Martineau, S. McIntosh-Smith, T. Deakin, G. Pawelczak, W. Gaudin, P. Garrett, W. Liu, R. Smedley-Stevenson, and D. Beckingsale. *Tealeaf: a mini-application to enable design-space explorations for iterative sparse linear solvers.* In Cluster Computing (CLUSTER), 2017.

M. Martineau, P. Atkinson, and S. McIntosh-Smith. *Benchmarking the NVIDIA V100 GPU and Tensor Cores.* In International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms, 2018.

M. Martineau and S. McIntosh-Smith. *The arch project: Physics mini-apps for algorithmic exploration and evaluating programming environments on HPC architectures.* In 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017.

M. Martineau, S. McIntosh-Smith, C. Bertolli, A. C. Jacob, S. F. Antao, A. Eichenberger, G.-T. Bercea, T. Chen, T. Jin, K. OBrien, et al. *Performance analysis and optimization of Clangs OpenMP 4.5 GPU support.* In Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), International Workshop on, 2016.

M. Martineau, S. McIntosh-Smith, and W. Gaudin. *Assessing the performance portability of modern parallel programming models using TeaLeaf.* Concurrency and Computation: Practice and Experience, 2017.

R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis. *Achieving performance portability for a heat conduction solver mini-application on modern multi-core systems.* In Cluster Computing (CLUSTER), 2017 IEEE International Conference on, 2017.

6.1 Introduction

The problem of heat diffusion on a two-dimensional structured grid has been chosen because it is simple to implement and is quite representative of large memory bandwidth bound scientific codes [112]. Each of the kernels is essentially a sparse linear algebra routine, and there are not many opportunities for node-level optimisation. The focus of this chapter will instead be on characterising and modeling the problem, as well as considering its performance portability.

6.1.1 Associated Research

The use of representative applications for the purposes of investigating performance portability and developing new algorithms has been used extensively on the path to exascale. Bird et al. developed the miniEPOCH mini-app, which is a proxy for the EPOCH particle-in-cell code, and used it to uncover a number of optimisations for modern processors, including loop fission and data structure alterations [14]. Using the miniMD mini-app, Pennycook et al. investigated the effect of gather-scatter performance on the vectorisation of molecular dynamics simulation, in particular considering the performance on the then recently released Intel Xeon Phi coprocessors [130].

Much research has been conducted into the performance of the parallel programming models with respect to a number of scientific applications. The development of robust parallel programming models that can support CPUs and offloading to accelerators is an important area of research that has a number of open questions. Karlin et al. ported three applications, LULESH, Kripke and Cardiod to OpenMP 4.5 in support of the CORAL project, and found that it was possible to achieve a reasonable level of performance for the applications using the directive based model [80]. Lopez et al. also ported representative kernels to OpenMP 4.5, particularly observing differences between the compilers resulting in inconsistent performance [94]. They further noted that OpenACC was "simpler" to use than OpenMP, as the compiler assumes more responsibility for the parallel configuration. Kirk et al. considered the performance of a number of parallel programming models using TeaLeaf [81].

Lin et al. ported a number of stencil codes belonging to the DOE, and found good performance once all data movement between the host and device was minimised [89]. They noted that support for managing the memory spaces on an NVIDIA GPU was not available in OpenMP 4.0, but the features will be introduced in OpenMP 5.0 as custom memory spaces and handling of unified memory [116].

A synopsis of the Advanced Simulation and Computing (ASC) strategy for Lawrence Livermore National Laboratory was described by Neely et al. In this document, OpenMP 4.5, OpenACC, RAJA, and Kokkos are all cited as planned parallel programming models to be used in the porting of DOE applications [120]. The strong traction of C++ abstraction layers motivated the inclusion of RAJA in the investigations in this thesis. Results demonstrated in research related to other applications, including TeaLeaf and CloverLeaf, conducted as part of this thesis, have shown that RAJA and Kokkos are similar and offer consistent performance [103] [100].

The Tpetra sparse linear algebra library was ported by Hoemenn et al. using Kokkos to enable performance portability between CPUs and GPUs [65]. This was important research as it showed the relevance of the C++ abstraction layers to provide performance portability to the high performance libraries.

6.1.1.1 Libraries

Scientific simulations often directly or indirectly solve a system of partial differential equations [13]. When implementing simulations that result in linear systems of equations, there are many choices of libraries that can support the development process. The libraries offer different functionality and levels of abstraction, for instance, high performance linear solver libraries, such as Trilinos and PETSc provide many high and low level routines [9]. The methods are typically constructed from multiple BLAS style linear algebra primitives, and so it is generally possible to manually implement solvers using lower-level linear algebra libraries such as BLAS, cuBLAS and MKL [44] [122] [69].

Although there are many successful high performance libraries, they can introduce difficulties in terms of performance portability. The vendor-tuned low-level libraries like MKL and cuBLAS are of course architecture-specific, while the higher level libraries like PETSc and Trilinos do not all provide performance portable interfaces. In the case where specialisation is provided to diverse architectures, such as the GPU support in PETSc, it can be challenging to integrate those features with custom kernels developed within an application. The problem is amplified when considering multi-physics codes that require interoperation between multiple high performance libraries, especially when offloading computation to an accelerator.

6.2 Implementation

In this thesis the Conjugate Gradient (CG) method is employed to solve heat diffusion, as it is a fast-converging linear solver that is easily implemented.

6.2.1 The Conjugate Gradient Method

A full mathematical treatment of the Conjugate Gradient method is outside of the scope this thesis so only the key details are presented, however, detailed introductions to the method are available from Shewchuk et al. and Nocedal et al. [146] [121]. The Conjugate Gradient method is a member of the Krylov subspace solvers and can be interpreted as an iterative approximation to the solution of a linear system $\mathbf{Ax} = \mathbf{b}$ that uses the fact that the conjugate vectors $\mathbf{p_i}$ of an $n \times n$ matrix $\mathbf{A}$ can be used to form a basis. Using that basis it is possible to describe the solution vector x as a linear combination of those vectors $\mathbf{x} = \sum_i \alpha_i \mathbf{p_i}$. The coefficients α_i can be determined using the closed equation $\alpha_i = \frac{\mathbf{p_i^T r_i}}{\mathbf{p_i^T A p_i}}$, where $\mathbf{r_i} = \mathbf{b} - \mathbf{Ax_i}$ is the residual of the i-th guess for the solution vector.

These characteristics make it possible to iteratively determine the solution by successively calculating conjugate vectors and coefficients and updating the solution vector $\mathbf{x}$. In practice, this iterative approach is expensive and incurs large storage overheads. A further optimisation of this algorithm is introduced by interpreting the problem as a projection into Krylov subspace, allowing each residual and conjugate vector to be calculated directly from their values in the previous iteration, greatly reducing storage overheads.

6.2.2 The Conjugate Gradient Algorithm

Given the previous mathematical derivation, the approximate iterative algorithm can be constructed. Pseudo-code for the algorithm can be seen in Code Sample 6.1.

Code Sample 6.1: The local CG algorithm.

```
x = 0          // Initial guess for solution
r = b - A * x  // Calculate initial residual
p = r

loop:
  alpha = (r * r) / (p * A * p)
  x_new = x + alpha * p
  r_new = r - alpha * Ap
  if((r_new * r_new) < EPS) break    // Convergence check
  beta = (r_new * r_new) / (r * r)
  p_new = r_new + beta * p
```

When applied to the heat diffusion equation, the sparse matrix **A** can be explicitly constructed or a matrix free approach can be employed. The entries of the matrix are the coefficients of the implicit formulation and, depending upon the problem specification, might be static or dynamic per time step. Noting that both `alpha` and `beta` are scalar values, a manual inspection of the local algorithm shows the following operation counts: 1 sparse matrix-vector multiplication, 3 N-length dot products, 3 N-length vector additions, and 3 N-length scalar-vector multiplications. In a dense calculation, the matrix-vector multiplication would likely be the limiting factor; however, given that the matrix-vector operation is sparse, it does not necessarily dominate the performance of the algorithm [112].

For the heat diffusion problem, the matrix **A** is a collection of coefficients that describe a stencil of neighbours for each cell in the computational domain. The sparsity of the matrix **A** is dependent upon the number of dimensions and the depth of the stencil. In the three-dimensional case with a stencil depth of one, there will be at most seven non-zero entries in each of the rows of the matrix. As such, the matrix-vector multiplication performs 7 multiplications for each entry of the solution vector, with several of the accesses to the solution vector being non-contiguous.

A convenient aspect of the described algorithm is that it is perfectly composed of simple linear algebra kernels, which makes it straightforward to predict the performance of the application based on the well-known characteristics of the kernels. In fact, the sparse matrix-vector, dot product, vector addition and vector scaling are memory bandwidth bound on all of the modern processors discussed in this thesis.

6.2.3 The `hot` Application

The `hot` application is a two-dimensional heat diffusion solver written for this thesis. There exist several established applications solving heat diffusion, such as TeaLeaf[1] and HPCG[2], which have been used throughout this thesis [103] [112] [101] [42]. The `hot` application was developed as a minimal example of a heat diffusion solver in the `arch` framework, to enable rapid exploration of parallel programming models. Those results observed for `hot` are directly applicable to other applications such as TeaLeaf, and even larger applications employing sparse linear algebra solvers.

A thorough discussion of the reasoning behind developing the `arch` project is provided in Chapter 1, but for `hot` the principle purpose was reducing the code size, and to demonstrate integration into the `arch` infrastructural layer. The entire solver is expressed in less than 200 lines

[1] https://github.com/uob-hpc/tealeaf
[2] http://www.hpcg-benchmark.org/

of computational code, meaning that porting to new parallel programming models is straightforward.

6.3 Performance Analysis

The performance of the Conjugate Gradient method is well understood on modern architectures. The key details will be briefly presented in relation to `hot` before a performance portability analysis is presented. The processors used in this section are discussed in Section 4.2, and the compilers used are: Intel 18.3 for the Skylake and KNL, and CUDA 9.0 for the V100.

6.3.1 Default Test Case

Due to the fact that `hot` performs an implicit solve, there is little parameter dependence except for the iteration count to convergence, which can change depending upon properties of the solution, parameters, and the initial conditions or guess. The iteration count does not matter for the purposes of this performance evaluation as that will generally be considered by the computational scientist developing the particular test problems. It is instead the goal of this project to understand the performance of individual iterations of individual time steps, considering them to be constant. In spite of this, in order to be easily related to other research and easily presented, the absolute runtime will be presented for a simple default test case.

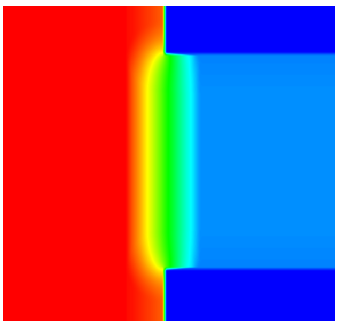

Figure 6.1: Solution of a heat diffusion problem solved by the `hot` application.

The default test case solves a single time step of length 1e-2s for a mesh of size 4096×4096, requiring 4030 iterations to minimise the error to below 1e-10.

6.3.2 Kernels

The `hot` application contains multiple kernels that all contribute to the overall runtime.

Kernel	Calls	Runtime
calculate alpha	4030	11.88s
calculate new r2	4030	13.57s
update conjugate	4029	5.98s

Table 6.1: Performance by kernel for `hot` on a Skylake CPU.

As shown in Table 6.1, the three major computational kernels in `hot` are relatively balanced in terms of their contribution to the runtime, meaning that all three of the kernels would need

to be optimised to achieve the best outcome. When the kernels are executed in a distributed fashion, there is also a communication cost to perform the all-to-all reductions of the scalar variables required by the kernels.

6.3.3 Performance on CPU and KNL

It will be subsequently demonstrated that the algorithm is predictably memory bandwidth bound. As such, considering the performance on a modern CPU when executing on all cores of a socket guarantees that the results are representative. Using fewer than the maximum number of available cores on a CPU might lead to incorrect representation of the performance.

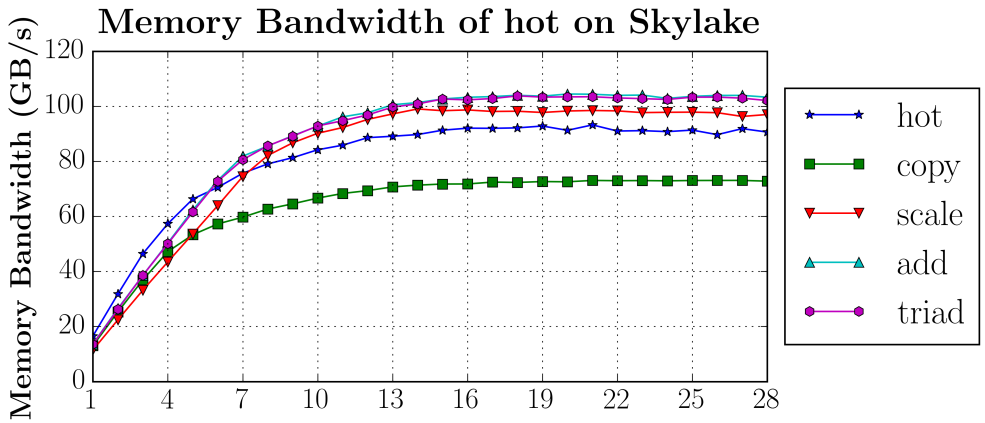

Figure 6.2: The memory bandwidth achieved by `hot` relative to STREAM kernels on a single socket of Skylake CPU.

On a modern Xeon server grade processor like the Skylake, for instance, it is necessary to execute on many cores to generate enough L3 cache misses to saturate DRAM bandwidth. Further to this, the distribution of cache is more challenging to reason about when under-utilising the cores on a socket. In spite of this, it is still informative to observe the achieved scaling of the algorithm given additional cores of the CPU.

The application is known to be memory bandwidth bound, and Figure 6.2 presents the memory bandwidth of `hot` scaled up to 28 cores of a Skylake CPU, alongside the results of the same test performed on STREAM. The memory bandwidth achievable with the Intel Xeon Skylake is significantly higher than previous Intel CPUs and, when dual-socketed, can rival previous generations of GDRAM-based NVIDIA GPUs. For a single socket, the Skylake processors achieves roughly 100 GB/s for the STREAM triad benchmark. The results show that memory bandwidth is saturated on the Skylake when `hot` is executed on 12 cores, and the behaviour of the application almost perfectly matches that of the STREAM benchmark.

6.3.4 Vectorisation

Vectorisation is not expected to have a significant influence on the performance of a memory bandwidth bound code running on a modern Intel Xeon CPU. When successfully vectorised and running a 4096×4096 problem, `hot` runs in around 7.9s on a Skylake CPU. If vectorisation is completely inhibited then the performance decreases to 8.7s. Performing the same analysis using the STREAM benchmark, the improvement in performance for vectorisation is 199GB/s

to 215GB/s, or around 8%, showing that there may only be a minor benefit for vectorising highly memory bandwidth bound kernels.

The same analysis applied on the KNL resulted in a 60% increase in runtime when vectorisation was disabled. This increased reliance on successful vectorisation was observed with `neutral` in Chapter 5 and will be seen in later discussions (Sections 6.7 and 7.2.2).

6.4 Performance on GPU

The linear algebra operations performed in `hot` can be easily translated to GPUs and similar architectures.

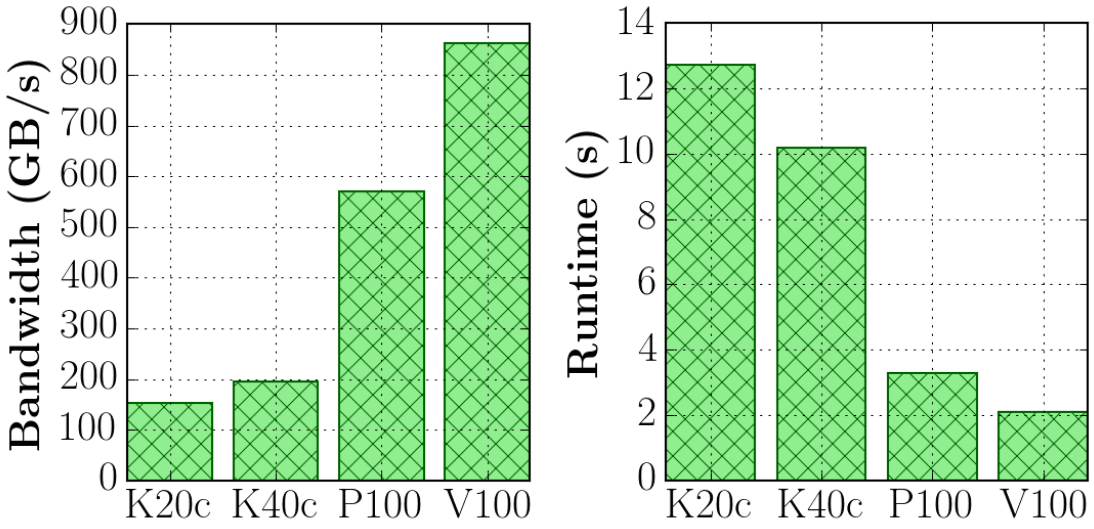

Figure 6.3: Performance of `hot` executing on NVIDIA GPUs, bandwidth (left) and runtime (right).

As the memory bandwidth on subsequent GPUs has increased, the performance achieved by the `hot` application has increased significantly, a trend which has also been observed by many scientific institutions including Sandia National Laboratories according to Trott et al [160]. The results in Figure 6.3 demonstrate that the runtime of the application for a representative problem has improved more than 6x across the multiple generations of GPU. The newest generation of GPUs, the V100, achieves around 860 GB/s which is roughly 4.2x more attainable bandwidth than dual-socketed Skylake CPUs. The results of the codes on the two architectures matches this trend closely as the runtime is around 4x faster between Skylake CPU and V100 GPU.

6.5 Balance

It is useful to consider the performance of the `hot` application using a simple performance model to ensure that the algorithm behaves as expected. Both the lower and upper bounds of memory access patterns will be modeled by considering perfect caching and zero caching.

Approximation 1 *Considering execution on a single node of a large test problem, more than 98% of the runtime is spent in three routines, which will be the only routines modeled.*

In order to seed the performance models, a number of statistics were calculated for the application. The assembly code and PTX for each of the routines was analysed to determine the double-precision FLOP count O_{dp}. The number of independent memory accesses is calculated by considering a perfect model of caching where each unique array is accessed once per read and/or write.

			O_{dp}			I_A		
Routine	B_R	B_W	SKL	KNL	V100	SKL	KNL	V100
`calculate_alpha`	32B	8B	10	11	14	0.25	0.28	0.35
`calculate_new_r2`	32B	16B	4	4	4	0.08	0.08	0.08
`update_conjugate`	16B	8B	1	1	1	0.04	0.04	0.04

Table 6.2: Statically analysed arithmetic intensities for routines in `hot`.

The results in Table 6.2 suggest that the performance of `hot` would be dominated by the memory accesses. Modern HPC architectures require high arithmetic intensities and/or extensive locality in order to overcome the large memory access latencies and limited bandwidths, as discussed in Chapter 4.

Routine	B_R	B_W	O_{dp}	I_A
`calculate_alpha`	38.0B	8.3B	10	0.22
`calculate_new_r2`	32.8B	16.2B	3	0.06
`update_conjugate`	16.1B	8.1B	1	0.04

Table 6.3: Empirical derived calculation of the arithmetic intensity for routines in `hot` for KNL.

It was possible to empirically determine an arithmetic intensity for the 4096×4096 on the KNL using the precise instruction mix emulated by the Intel Software Development Emulator Toolkit (SDE) and the total bytes accessed as measured by the uncore counters on the processor.

Table 6.3 shows a close fit between the analysed arithmetic intensity and result observed at runtime. The arithmetic intensity is so low that working sets large enough to spill out of cache will be memory bandwidth bound.

Approximation 2 *Given such a low arithmetic intensity, it is possible to approximate that the cost of any FLOPs are amortised by the memory accesses.*

Therefore, the runtime of an individual routine could be described by the following simple model when executed on a single processor:

$$T = \sum_{t=0}^{I_t} I_c^{(t)} \times MN \sum_r \frac{(B_R^{(r)} + B_W^{(r)})}{D} \tag{6.1}$$

$$r \in \{\text{calculate\_pAp, calculate\_new\_r2, update\_conjugate}\} \tag{6.2}$$

where:

$B_R^{(r)}$ = number of bytes written per mesh cell for routine r
$B_W^{(r)}$ = number of bytes read per mesh cell for routine r
D = aggregate memory bandwidth achievable for memory level (in $bytes/sec$)
$I_c^{(t)}$ = number of iterations to convergence for time step t
I_t = number of time step iterations
M, N = dimensions of 2D mesh
T = predicted wall clock runtime in seconds

The parameter D is an aggregate value accounting for the total memory bandwidth available on a single node or accelerator; this metric could therefore refer to the bandwidth of dual-socketed CPUs or a single GPU.

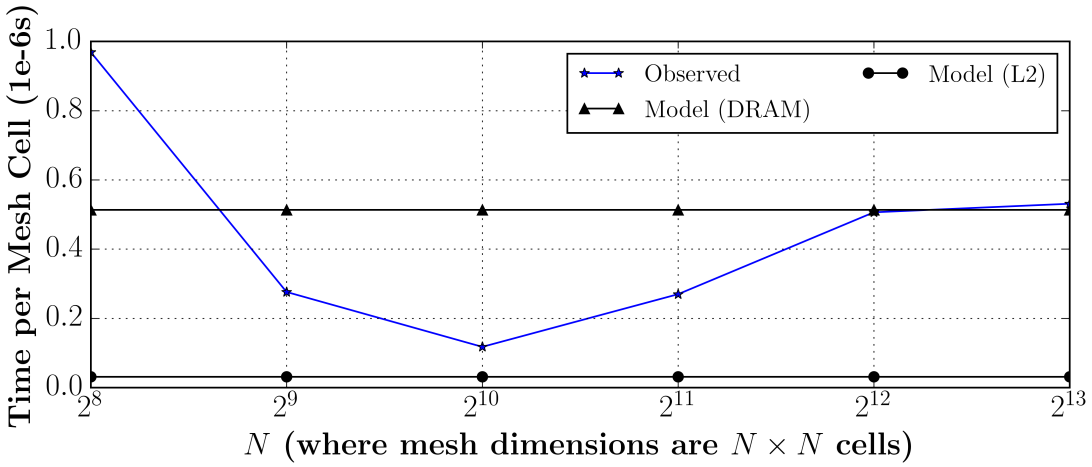

Figure 6.4: Varying mesh dimensions for `hot` with modeled and observed runtime results.

In Figure 6.5 the modeled data is shown for DRAM and L2, and the results show that the model accurately describes the performance from DRAM. Going from the right side of Figure 6.5 to the left side, the time per cell decreases between the problem sizes $N = 2^{12}$ and $N = 2^{10}$, as the working set begins to fit in the 1 MiB per core L2 cache ($\frac{(2^{10})^2 \times 8B \times 6 \text{ arrays}}{56 \text{ cores}} =$ 877KiB per core). Interestingly, the performance never achieves the L2 cache bandwidth, and a considerable overhead is observed as the mesh size reduces to $N = 256$.

The results of this modeling exercise further prove that large working sets in the solver will only benefit from improvements in DRAM. When the problem small enough that it is resident in high levels of cache, the runtime approaches the cache bandwidth for the processor but small problems sizes do not saturate memory bandwidth. The cause of the overheads is an interesting issue, as the optimisation for small problems requires a more subtle approach than for the large memory bandwidth bound problem. The overheads will be discussed relative to the parallel programming model ports in Section 6.7.

6.6 Distributed Performance

The previous sections demonstrated optimal parallel execution on CPUs, GPUs and KNLs, regardless of whether the parallelisation uses OpenMP to take advantage of shared memory

or uses MPI to execute isolated parallel processes. It is subsequently possible to observe the performance as the application is scaled across multiple nodes within a cluster. An exploration of different communication avoiding strategies for the TeaLeaf mini-app is presented in Martineau et al. [112]. The research was a multi-organisational effort to understand how the scaling of heat conduction could be improved using novel algorithms.

6.7 Performance Portability

Having proven that `hot` exhibits expected performance characteristics, and understood the subtleties of implementation on a Skylake CPU, it is now possible to consider the wider problem of achieving performance portability with the application. It is reasonable to expect that the parallel programming models achieve a high fraction of peak performance for `hot`, given that the structure of the application is simple and the kernels are comprised of linear algebra primitives. The processors used in this section are discussed in Section 4.2, and the compilers used are: Intel 18.3 for OpenMP and RAJA on the Skylake and KNL, PGI 18.5 for OpenACC on all targets, CCE 8.7.4 for OpenMP 4.5 on the GPU, and CUDA 9.0 for all GPU implementations.

6.7.1 Preliminary Performance for Default Test Case

The application `hot` is ported to a number of different parallel programming models and the performance is measured for the default test case, as described in Section 6.3.1. The results presented are for the best performing implementations, while still allowing the compilers to choose implementation defined parameters based on internal cost models.

6.7.1.1 CPU Performance

The results in Figure 6.5 are similar to those found in published work performed as part of this thesis with the mini-app TeaLeaf [100]. The difference between the best and worst implementations on the Skylake processor is less than 12% of the best achieved memory bandwidth, which will be a tolerable for scientific institutions unless they have strict timeliness requirements.

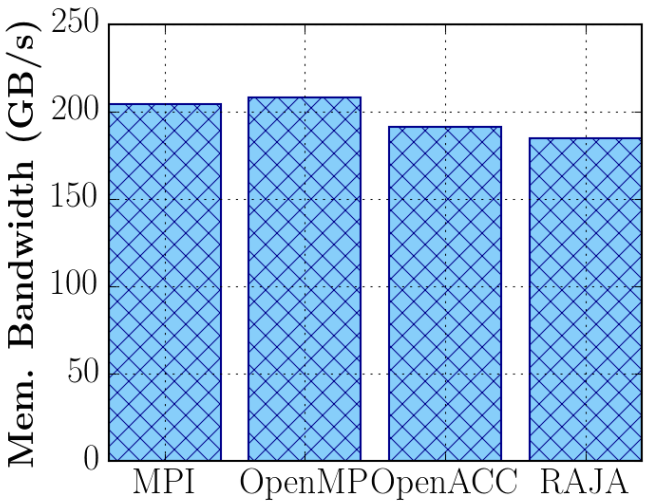

Figure 6.5: The performance of `hot` on a Skylake CPU.

The results are so consistent between the models that the benefits to portability offered by the performance portable parallel programming models are particularly pronounced. Even if scientific developers only expect to target CPUs in the near future, then the performance portable parallel programming models offer future proofing and quality of life improvements with little impact on the CPU performance.

6.7.1.2 KNL Performance

Figure 6.6 presents the results for the KNL, including the three sensible configurations of hyperthreads. It can be noted that each of the different implementations performs optimally with different numbers of hyperthreads. This highlights that the difference in performance between the configurations is challenging to analytically determine.

The OpenMP implementation serves as the base case with which the other implementations can be compared, and achieves 75% of the achievable peak performance, shown by the *triad* results from Section 4.5. The MPI performance is slightly lower than OpenMP, and this is most pronounced for 4 hyperthreads. This issue can be explained by a large increase in the cost of communication as the number of hyperthreads doubles from 2 to 4.

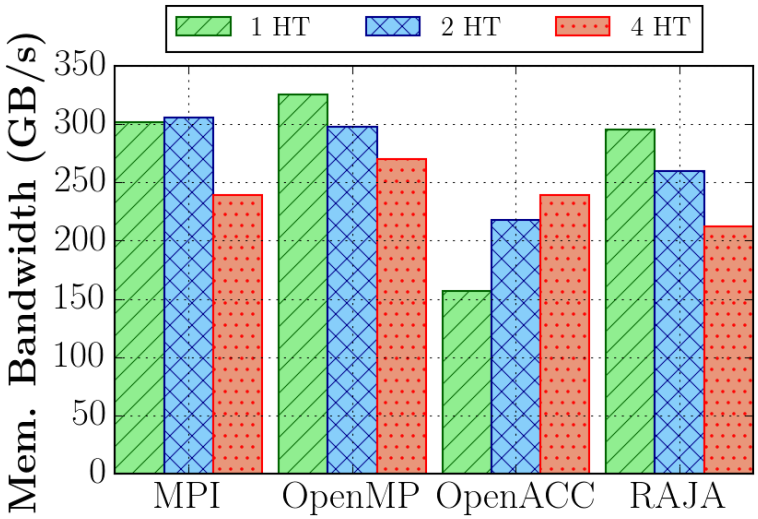

Figure 6.6: The performance of `hot` on a KNL.

The initial RAJA port simply collapsed the nested loops over space into a single RAJA `forall`, but using this scheme only achieved around 25% of the memory bandwidth of OpenMP. The performance overhead was only observed for those routines that were using the RAJA reduction templates. To resolve this issue it was first necessary to more closely follow the parallelisation performed with OpenMP, where the outer loop was parallelised with the RAJA `forall`, and the inner loop was a sequential loop with `omp simd` above it. Subsequently it was necessary to create an additional temporary variable to perform the SIMD reduction within the inner loop, and then accumulate this into the RAJA reduction template in the outer loop. Structuring the code in this manner enabled the performance shown in Figure 6.6. The problem with this particular fix is that it has quite a significant impact on the performance portability of the solver, where the configuration is now not suited to the GPU.

The best performing implementation using OpenACC introduces around a 26% overhead,

while the best performing RAJA implementation introduces a roughly 10% overhead, when compared to the optimal OpenMP. As discussed in Section 6.3.4, achieving maximum memory bandwidth on the KNL requires successful vectorisation; however, both OpenACC and RAJA have been compiled with AVX512 support, and RAJA is successfully vectorising. The loop-body code generation for RAJA is handled by the Intel compiler, which is the same as the OpenMP implementation, making it likely that, as long as vectorisation has been enabled, the output vector code is similar between the implementations. In the case of OpenACC, the code generation is unique to the PGI compilers, and so it is possible that the vector code being generated is not as well optimised for the KNL.

6.7.1.3 GPU Performance

The final architecture considered is the GPU, where the particular processor is the NVIDIA P100 GPU. The code is compiled with the compilers listed in Section 4.2.

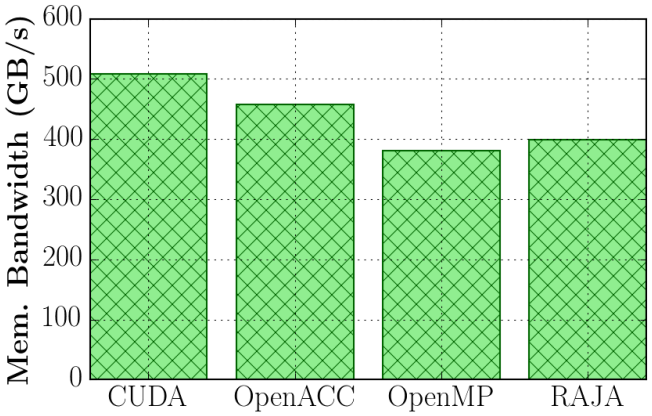

Figure 6.7: The performance of `hot` on a P100 GPU.

The results in Figure 6.7 are for the P100 GPU, where OpenACC performs well but both OpenMP and RAJA exhibit overheads of 24% and 20%. When the research was originally performed with TeaLeaf, this result was quite impressive, as OpenMP 4.0 and RAJA had only just implemented GPU support [101]. Having had many years to mature, it is now expected that the programming models could achieve better proportions of peak for this particular application.

As with the CPU, a 10-20% overhead is likely permissible for the benefits of only having to maintain a single code base. Karlin et al. suggested that 25% was permissible for porting LULESH at Lawrence Livermore National Laboratories (LLNL), for instance [80]. Given the simplicity of the application, the results for the performance portable models likely should have been better than those observed, but only marginally. It is possible that the poor results stem from decisions in the compilers that are based on cost models that simply did not determine the optimal selection of parameters, and it might be fruitful to perform fine tuning.

6.7.2 Performance for Small Problems

It was noted in Section 6.5 that the performance of the OpenMP implementation was sub-optimal for small problems. In order to explore this issue further, an experiment is conducted where each of the parallel programming models executes a small test problem of 1000 iterations

on a mesh of dimensions 256^2. Identical parallel configurations and code are used for the small problems as were used for the large test problem.

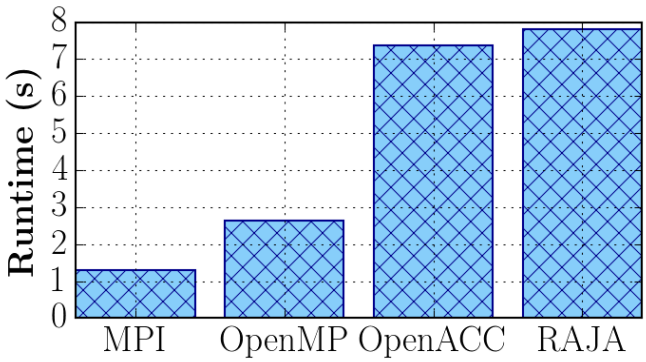

Figure 6.8: The performance of `hot` for a small test problem on a Skylake CPU.

The results in Figure 6.8 show the performance of this small test problem on the Skylake CPU. Further to the issue observed initially with OpenMP, the overheads introduced by the performance portable programming models are far greater than observed when the test problem was larger than the CPU cache.

Kernel	**MPI**	**OpenMP**	**OpenACC**	**RAJA**
communication	0.40s	n/a	n/a	n/a
`boundary`	0.66s	1.58s	2.42s	3.26s
`calculate_alpha`	0.10s	0.41s	2.31s	2.09s
`calculate_new_r2`	0.08s	0.43s	2.33s	2.00s
`update_conjugate`	0.05s	0.22s	0.32s	0.47s

Table 6.4: Performance of key routines in `hot` for small test problem on Skylake CPU.

The problem size is small enough that the communication costs in MPI are the dominant performance factor, as can be seen in Table 6.4. The computation accounts for only around 10% of the total runtime, with the rest consumed by halo updating and communication. In comparison, the OpenMP implementation results present significantly increased costs for the `boundary` communication and computation. The reason for this increase is that each of the individual parallel regions incurs an overhead, firstly to fetch the threads from the thread pools and then finally to synchronise all of the threads. For many problems this cost is completely amortised, but as can be seen in the results, the performance impact can be quite significant for small problems.

The RAJA results in Table 6.4 are for the RAJA `forall` loop, which does not explicitly provide any hint to the compiler to vectorise the loop. The compiler did not auto-vectorise the kernels in `hot`, but fixing vectorisation did not improve the performance. As the loop-level overheads cannot be explained by poor vectorisation or resolvable differences between the implementations, it is expected that there are intrinsic overheads in the APIs, and the implementation defined parameters are not suited to small problems. Those overheads likely support the generality required by the programming models; however, they might make small kernels prohibitively slow, likely requiring a coarsening of parallelism in some applications.

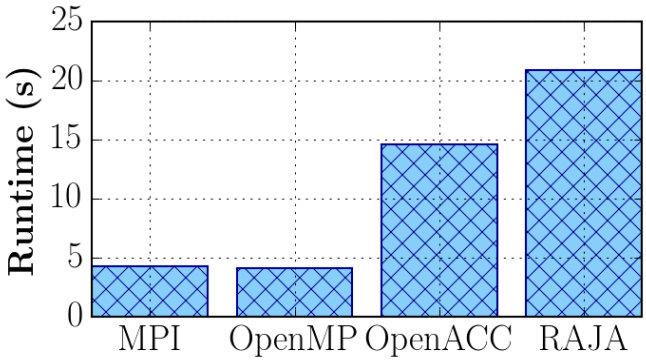

Figure 6.9: The performance of `hot` for a small test problem on a KNL.

The results presented in Figure 6.9 are for the KNL with 1 hyperthread per core, as this was considerably faster than other configurations for all of the implementations. The OpenMP implementation was the fastest in this case, as the cost of the MPI communication becomes larger in proportion to cost of synchronising in OpenMP. Overall, all of the implementations perform worse for small problems on the KNL than the Skylake. This is partly due to the increased cost of calculating halo boundaries and communication for the increased core count.

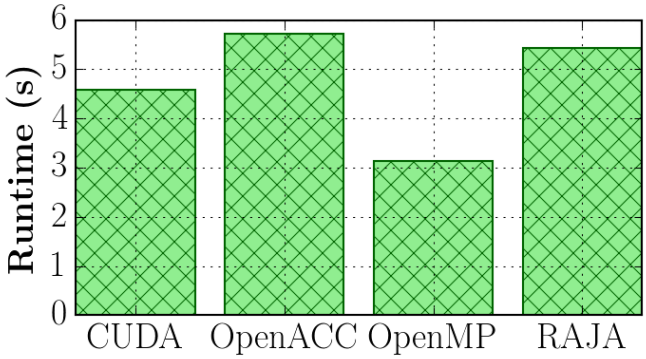

Figure 6.10: The performance of `hot` for a small test problem on a P100 GPU.

The results in Figure 6.10 are for the P100 GPU, and demonstrate some interesting features about running the small problem on a GPU. The performance is worse overall than the CPU, although it can be seen in Table 6.5 that the performance is marginally faster at the kernel level than for the CPU, as seen in Table 6.4. The extra runtime is caused by kernel launch and synchronisation overheads, as well as the slightly inflated costs of halo exchanges.

Kernel	CUDA	OpenACC	OpenMP	RAJA
`calculate_alpha`	324ms	338ms	297ms	375ms
`calculate_new_r2`	285ms	293ms	258ms	330ms
`update_conjugate`	107ms	106ms	99ms	108ms

Table 6.5: Performance of key routines in `hot` measured by `nvprof` on P100 GPU.

In the case of OpenMP, the results are actually superior to the CUDA implementation. It

is important to recognise that this is not suggesting that the performance portable model is inherently faster than CUDA, just the implementation. The approach taken in the OpenMP implementation is to chunk the iteration space and give multiple iterations to each of the threads, rather than the approach taken in the CUDA implementation of greatly over subscribing the GPU with a single iteration per thread. Also, the reduction implementation in OpenMP does not require additional kernel calls, which means that the performance of the reduction is optimised for small problems where kernel launch overheads are not amortised by the size of the individual computations. In contrast, the CUDA port, the OpenACC implementation in the PGI compiler, and the RAJA library have been optimised for large reductions and perform an additional post-kernel step to finalise the reduction. It is of course possible to reimplement the reduction in CUDA, but the reductions in OpenACC and RAJA are likely to require extensive work to account for this issue.

6.8 Summary

Although `hot` is small, it is representative of a large class of applications that solve sparse linear systems, and while the performance of large problems is well understood, new issues arise as the problem size is reduced. It has been possible to demonstrate that the CG method is memory bandwidth bound and communication bound, depending upon the size of the input problem and target processors [112].

In spite of the algorithmic simplicity of the kernels included in the solver, the performance portable programming models did not achieve optimal performance in all cases observed. The performance portable models were able to achieve within 30% of the best achieved performance for test problems that were large enough to saturate DRAM, but did not perform as well for the small problems on the CPUs due to overheads introduced for generality. It must be noted, however, that the chosen small problems are unlikely to be useful in real-world scenarios for the CG method, but the overheads might be important to other problem domains.

Problem dependence does not just affect performance portable models, however, as exposed by the OpenMP results beating CUDA on the P100 GPU due to the reduction implementation. Writing low-level code that is optimal for the full range of problems in the domain of a particular solver can be challenging, and often needs to be handled on a case by case basis. This particular issue was discussed further in Chapter 5. Surprisingly, the performance portable models performed particularly well for the small problem on the NVIDIA GPU, which shows some resilience to the issue of problem dependence, and overheads seen with the CPU implementations.

For `hot` there is little tuning opportunity other than the parallel configuration, which only resulted in a small difference between the applications. As parameter tuning generally harms performance portability this was not a major issue, but it would have been preferable if it were possible to completely tune away any differences between the architecture specific codes and the performance portable implementations. Given an application running large scientific test problems and a memory bandwidth bound code, which is quite typical, the performance portable parallel programming models are likely to be attractive options.

Chapter 7

Hydrodynamics

Key Publications

M. Martineau and S. McIntosh-Smith. *The arch project: Physics mini-apps for algorithmic exploration and evaluating programming environments on HPC architectures.* In 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017.

7.1 Introduction

Another important class of applications is hydrodynamical solvers, which features in all production scientific domains that require fluid motion. Some diverse examples are astrophysics, and simulating laser ignition [119] [83]. The hydrodynamical step is often supported by other physics in a multi-physics environment, and generally drives mesh motion within an application [55]. Although a structured hydrodynamics application can be highly optimised for modern architecture, the necessary approximations lead to complex algorithms designed to overcome numerical inaccuracies. In this thesis, full consideration is given to the inviscid compressible Eulerian equations of hydrodynamics, with an expectation that many of the insights will be applicable to formulations based on alternative governing equations.

It is possible to create formulations up to three dimensions, with structured or unstructured grids, complex dynamic meshing constructions using Adaptive Mesh Refinement (AMR) or Arbitrary Lagrangian-Eulerian (ALE), and different frames of reference that fix or follow the compressible volume [83] [78]. Each of the different formulations of hydrodynamics introduces differences in performance characteristics. Although it is not possible to address the full spectrum within this thesis, an attempt will be made to address two common formulations in modern HPC applications: Eulerian and Lagrangian. Alongside those applications the differences between structured and unstructured mesh solvers will also be considered.

Prior research has shown that explicit hydrocodes should typically achieve a good level of performance on modern architectures, so the purpose of this chapter is to: (1) show that the performance portable models can achieve good performance with different variations of hydrocode, and (2) highlight those characteristics of hydrocodes that are different from the Monte Carlo and sparse linear algebra methods considered in previous chapters.

7.2 Structured Eulerian Hydrodynamics

Eulerian hydrodynamics makes the assumption that all volumes on the mesh are fixed, and the hydrodynamical quantities flow through those fixed volumes over time. At the most fundamental level this results in the change in internal quantities for each cell using upwind schemes for the flux through the cell faces.

The `flow` application[1] is a two-dimensional structured Eulerian hydrodynamics solver written from scratch for this thesis. The application uses an ideal gas equation of state, quadratic artificial viscosity, and a Van Leer flux limiter [19] [165]. The solver is staggered in both space and time, with dimensional splitting to handle the multiple dimensions, alternating the first dimension in each timestep to improve numerical symmetry throughout the solve. The application supports distributed execution and has been optimised for CPUs, GPUs, and other accelerator devices and has been validated on a range of scientific test problems.

The application is similar in construction to CloverLeaf, although it does not use a predictor-corrector scheme, and the results for `flow` are directly applicable to CloverLeaf [61]. The reasons for developing `flow` rather than using an application like CloverLeaf are:

- The programming language is C throughout, which is the easiest language to interoperate with low-level, directive-based, and C++ abstraction programming models.
- As with all other applications in `arch`, `flow` is designed to support multiple parallel programming models within a single source framework, where only the core parallel loops are duplicated between models.
- The application is lightweight, with only 1000 lines of computational code, making it easy to conduct fast performance investigations with. In contrast, the CloverLeaf mini-app contains 11000 F90 LOC, and 3000 C LOC.
- As the application is hosted by the `arch` project, it is possible to provide insights into managing hydrodynamics as part of a suite of applications.

There are similarities between `flow` and `hot`, and so it is possible to exclude some of the analyses for `flow` as key interesting features of structured grid and memory bandwidth bound codes have been shown in Chapter 6. In spite of this, the `flow` application is a new proxy application developed specifically for this thesis, and so it is necessary to consider the performance profile, and validate that the application is exhibiting the expected memory bandwidth bound shown with other explicit hydrocodes [62].

7.2.1 Performance Analysis

The processors used in this section are discussed in Section 4.2, and the compilers used are: Intel 18.3 for the Skylake and KNL, and CUDA 9.0 for the V100 GPU. Although the `flow` application is a structured grid code, similar to `hot`, the application solves a different problem using an explicit solver. The resulting code is comprised of 18 separate kernels that perform the advection and numerical fixups required to reach an accurate solution.

[1] https://github.com/uob-hpc/flow

7.2.1.1 Default Test Case

The `flow` application is quite resilient to changes in the problem specification. In order to demonstrate the performance and other key details about the application, a test problem has been chosen that is a representative size that can fit on a single node.

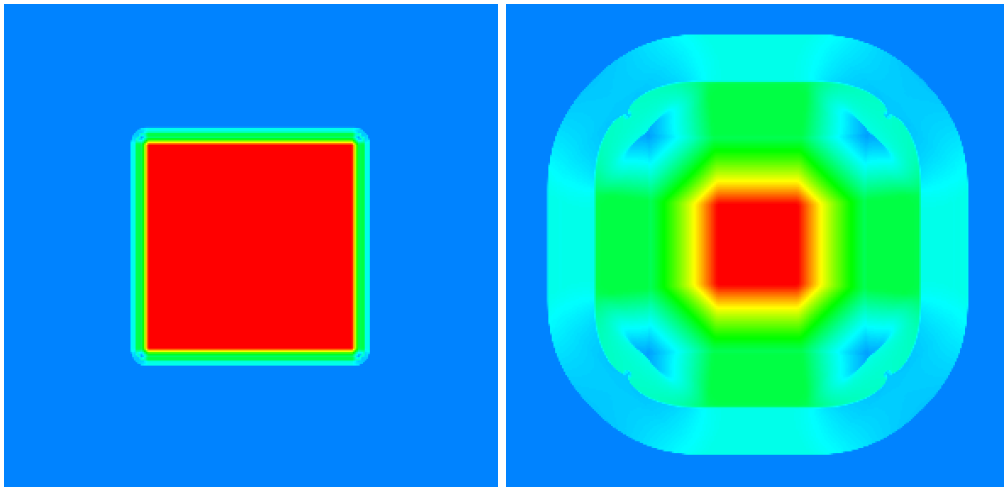

Figure 7.1: Default test problem (left), and the same problem after 1000 timesteps (right).

The default test case solves for 4096^2 cells and runs for 100 iterations, with reflective boundary conditions applied. A square region of high density gas is constructed in the center of a region of low density gas and the initial condition is that the entire mesh is static. This particular problem is sufficiently large and long running to begin to see details emerge in the flow, a segment of the final solution is shown in Figure 7.1 (left). Further details are seen as the solver progresses through subsequent timesteps, as shown in Figure 7.1 (right); however, the performance characteristics do not change. It is also important to note that, although some mesh cells change dramatically between timesteps, the particular hydrodynamical formulation is quite balanced, and each of the cells will require roughly the same amount of computation regardless of whether a significant proportion of the cell was advected or not. It will be discussed later in Chapter 8, that production hydrodynamical solvers are likely to encounter issues of load balance not represented in this scheme.

7.2.1.2 Kernels

The `flow` application is significantly different from the other applications observed so far as it is comprised of a large number of small kernels, that generally run in a similar length of time.

Table 7.1 shows the performance of 100 runs of each kernel, for a total solve time of 6.87s on the dual-socketed Skylake CPUs. In some cases the kernels are grouped: `artificial_viscosity` encompasses 2 kernels, `advect_mass_and_energy` encompasses 4 kernels, and `advect_momentum` encompasses 8 kernels. Each of the kernels executes in around 2-6ms, meaning that there are no distinct 'hot-spots' that can immediately benefit from optimisation.

For each of the kernel groups, a value has been given for the minimum amount of data touched in a single iteration. This particular metric accounts for accessing each mesh element once (twice if read and written) for the dependent variables in a function, with subsequent accesses assumed free. This models a scenario of perfect caching, which would be observed in

Kernel	Num. Kernels	Runtime	Data Moved	Mem. BW
set_timestep	1	0.24s	500 MB	210 GB/s
equation_of_state	1	0.23s	402 MB	175 GB/s
pressure_acceleration	1	0.62s	1024 MB	165 GB/s
artificial_viscosity	2	1.13s	1792 MB	159 GB/s
shock_heating_work	1	0.51s	977 MB	196 GB/s
storing_old_density	1	0.16s	256 MB	160 GB/s
advect_mass_energy	4	1.73s	3072 MB	178 GB/s
advect_momentum	8	2.25s	3584 MB	159 GB/s

Table 7.1: Performance by kernel for `flow` on Skylake CPU.

the case that locality had been perfectly expressed within the problem.

Studies considering Eulerian hydrodynamics applications, like CloverLeaf, have shown that the solver is strongly memory bandwidth bound [62]. The results in Table 7.1 clearly show that `flow` application also achieves a high fraction of the achievable memory bandwidth, as discussed in Section 4.5. The lowest achieved result is 159 GB/s, which is 73% of achievable memory bandwidth, and the other kernels improve upon this up to 97% of maximum achievable memory bandwidth. The key to achieving this level of performance is in ensuring that all data accesses are stride 1 wherever possible, with simple Structure of Arrays data structures.

7.2.1.3 Vectorisation

The memory bandwidth results are for the well vectorised version of the code. On a Skylake CPU, vectorisation has a minimal impact on the performance of the application, improving the runtime of the default test case from 7.4s to 7.2s. Vectorisation does not have a significant impact because, similar to `hot`, the majority of the cycles are spent waiting for requests to be served from DRAM.

7.2.1.4 GPU Performance

Taking the same analysis performed for the CPU it is possible to determined the memory bandwidth achieved for the individual kernels when executed on a V100 GPU.

Kernel	Calls	Runtime	Data Moved	Mem BW
set_timestep	101	0.07s	500 MB	721 GB/s
equation_of_state	100	0.05s	402 MB	804 GB/s
pressure_acceleration	100	0.14s	1024 MB	731 GB/s
artificial_viscosity	100	0.24s	1792 MB	747 GB/s
shock_heating_work	100	0.13s	977 MB	769 GB/s
storing_old_density	100	0.03s	256 MB	853 GB/s
advect_mass_energy	100	0.41s	3072 MB	749 GB/s
advect_momentum	100	0.49s	3584 MB	731 GB/s

Table 7.2: Performance by kernel for `flow` on V100 GPU.

Table 7.2 presents the results of 100 timesteps of the GPU kernels, for a total runtime of

1.56s. The results show that the kernels ported well to the GPU, achieving a large fraction of the maximum achievable memory bandwidth, at least 84%. In comparison to the Skylake CPU, the runtime has improved by 4.4x on the GPU, which is around 10% more than the 4x difference in memory bandwidth observed through benchmarking.

7.2.2 Performance Portability

In this section, the performance portability of the `flow` application will be considered with respect to the parallel programming models discussed in Chapter 3. The processors used are discussed in Section 4.2, and the compilers used are: Intel 18.3 for OpenMP and RAJA on the Skylake and KNL, PGI 18.5 for OpenACC on all targets, CCE 8.7.4 for OpenMP 4.5 on the GPU, and CUDA 9.0 for all GPU implementations.

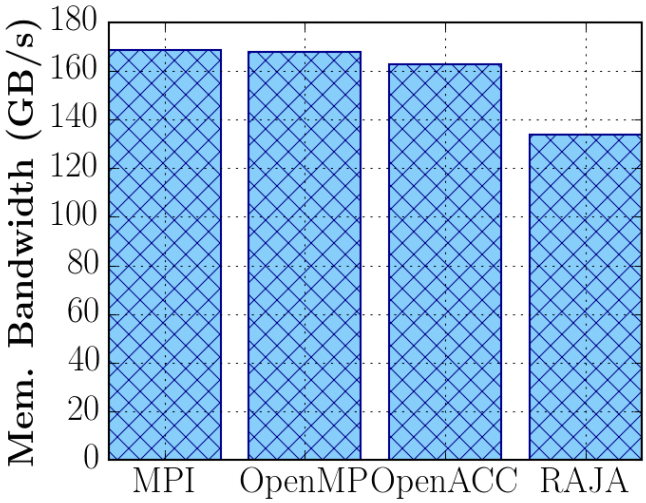

Figure 7.2: The memory bandwidth achieved by ports of `flow` executing on a Skylake CPU.

Figure 7.2 shows the average overall memory bandwidth achieved by each of the ports of `flow` executing on the Skylake CPU. All of the ports are able to achieve a high fraction of achievable memory bandwidth, but that RAJA is slightly lagging behind the other models. MPI and OpenMP achieve the best performance, as expected, at around 79% of achievable memory bandwidth, while OpenACC achieves 75% and RAJA achieves 63%. The fraction of peak performance achieved by the performance portable models, and particularly OpenACC, is within a tolerable limit of the best achieved performance for the application.

Figure 7.3 shows the memory bandwidth achieved by each of the ports of `flow` executing on the KNL. The best fraction of peak performance recorded for the KNL was 57% for OpenMP, compared to the 440 GB/s achieved by the *triad* kernel. Section 4.5 demonstrated that there were large differences in the best achievable memory bandwidth on the KNL, depending upon the balance of reads and writes within the kernel. The majority of kernels in `flow` are read-access dominant, and so a more reasonable estimation of the best achievable memory bandwidth is 300 GB/s, as achieved by the *read* kernel. In this case, the OpenMP performance is roughly 77% of the achievable memory bandwidth, which is a more reasonable result of peak performance.

The OpenACC performance is significantly lower than the other programming models, and there are several reasons that this occurs. The timestep calculation uses the OpenACC

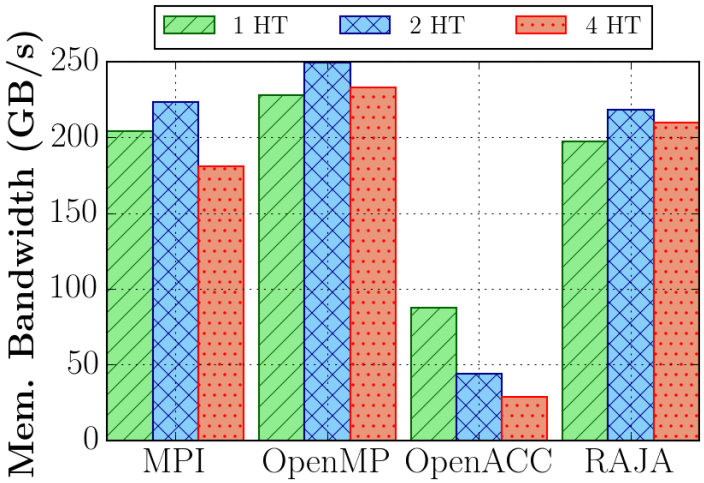

Figure 7.3: The memory bandwidth achieved by ports of `flow` executing on a KNL.

`reduction` directive with a *min* operator, which appears to perform quite poorly on the KNL, and likely does not represent expected performance.

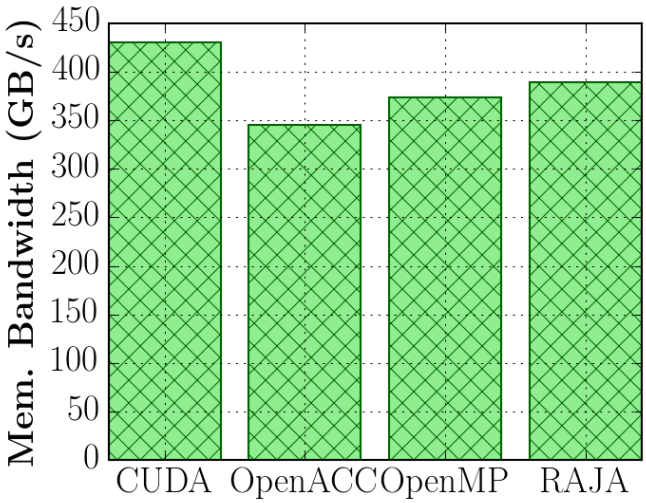

Figure 7.4: The memory bandwidth achieved by ports of `flow` executing on a P100 GPU.

Figure 7.4 shows the memory bandwidth achieved by each of the ports of `flow` executing on the P100 GPU. The CUDA port achieves a good fraction of achievable memory bandwidth, around 77%, while OpenACC achieves 63%, OpenMP achieves 68%, and RAJA achieves 71%. Note that in this case, the OpenMP code is using the target offload features introduced in the OpenMP 4.5 specification, so the source code is different from the CPU implementation. The results demonstrate that the models are able to achieve a high fraction of peak performance, although the performance portable implementations incur an overhead of around 10%-20% compared to the best case CUDA code.

7.2.3 Productivity

Out of the application classes considered within this thesis, hydrodynamics is the most insightful in terms of productivity. The applications are larger with more complex code structures than the other applications considered; however, they are not necessarily representative of the complexity exposed in typical production applications. A common feature of the proxy applications considered in this thesis is that the code has been specifically developed with threading in mind. The codes require little to no refactoring or cleansing between porting exercises, which means that the observations below account only for changes due to the programming model. In a real application, the effort required to port a legacy code to a GPU, for instance, might be overwhelmingly dominated by the cost of refactoring existing code constructs to be suitable for highly parallel execution, which is not easily captured by proxy applications [159].

All of the applications are part of the `arch` suite, and all of the shared infrastructural code is contained within the `arch` project. The most significant shared feature in the `arch` project is the data management routines that encapsulate the allocation and movement of data using the features provided by each of the parallel programming models. For a discussion about making the `arch` project performance portable, refer to Section 8.1.1.

7.2.3.1 OpenMP

The OpenMP porting exercise was fast and straightforward, where the entire port only required 23 code sites to be changed in order to achieve the performance shown above. In most change sites the combined construct `target teams distribute parallel for simd` was applied (Code Sample 7.1), but some sites required the use of the `reduction` directive.

Code Sample 7.1: Example kernel ported with OpenMP.

```
1 #pragma omp target teams distribute parallel for
2   for (int i = pad; i < (ny + 1) - pad; ++i) {
3     for (int j = pad; j < (nx + 1) - pad; ++j) {
```

The main issue with respect to productivity is that OpenMP is not easily made performance portable, and in the `arch` suite of applications, the OpenMP 3.0 CPU-targeting code is considered a distinct port from the OpenMP 4.5 GPU-targeting code. The consequence is that it is necessary to maintain duplicate versions of the computational code, which from a production perspective would not be acceptable. Although the directives are verbose, the productivity enhancements of developing an OpenMP 4.5 port of a hydrocode compared to a CUDA port are significant. Relying upon implementation-defined parameters for the balance of teams and threads, for instance, greatly reduced the number of considerations when porting each of the loops. As the kernels were already threaded for multi-core execution using OpenMP targeting the CPU, with stride 1 memory accesses where possible, the main point of consideration for each loop was that enough parallel work was exposed for the GPU.

7.2.3.2 OpenACC

For the most part, the OpenACC port was as straightforward as the OpenMP port, and the number of code sites changed was the same, although the number of LOCs changed was higher at 71 LOC. In many cases it was possible to use the `kernels` directive enclosing the loop nest, with the `loop independent` construct before each of the loops (Code Sample 7.2).

Code Sample 7.2: Example kernel ported with OpenACC.

```
#pragma acc kernels
#pragma acc loop independent
  for (int i = pad; i < (ny + 1) - pad; ++i) {
#pragma acc loop independent
    for (int j = pad; j < (nx + 1) - pad; ++j) {
```

The use of the `kernels` directive allowed all of the loops to be parallelised without considering the underlying implementation. As with the OpenMP port, there were not many decisions to make about each of the loop nests, as the compiler can choose parallel decompositions, detect memory movement, data sharing, and reductions automatically. In one case it was necessary to use the `present` directive to inform the compiler of data movement relating to a particular `parallel` region, where the `parallel` directive was necessary to generate the correct *min* reduction.

7.2.3.3 RAJA

When porting to RAJA, it was necessary to replace all of the loops with lambda functions. The majority of the loops in `flow` are comprised of two nested loops, due to the solver being two-dimensional. There are several different approaches to parallelise loop nests containing a pair of loops using RAJA, for instance: (1) collapse the nest and use a single RAJA *forall* over the whole iteration space; (2) use the RAJA nested loops approach. The element of choice when porting a particular loop nest can introduce immediate problems for productivity, as significant time and attention might need to be devoted to analyse or prototype the best approach.

Code Sample 7.3: Example kernel ported with RAJA.

```
RAJA::forall<exec_policy>(
  RAJA::RangeSegment(0, (nx+1)*(ny+1)), [=] RAJA_DEVICE (int i) {
    const int ii = i / (nx+1);
    const int jj = i % (nx+1);
    if(ii >= pad && ii < (ny+1)-pad && jj >= pad && jj < (nx+1)-pad) {
```

Code Sample 7.3 depicts a kernel ported to RAJA using a style particularly well-adapted to execution on a GPU. The style ensures that the maximum possible parallel workload is passed to the GPU, and the data accesses can be more easily organised for coalesced memory accesses, assuming a standard SoA data structure layout.

Code Sample 7.4: Kernel with RAJA outer loop and inner `for` loop.

```
RAJA::forall<exec_policy>(
  RAJA::RangeSegment(0, (ny+1)-2*pad), [=] RAJA_DEVICE (int i) {
    for(int j = pad; j < (nx+1)-pad); ++j) {
```

Code Sample 7.4 shows the porting approach that achieved the best performance for `hot` and `flow`, which is not optimal on the GPU. The solution would be the use of the RAJA nested loops, and it will be useful future work to investigate whether optimal implementations of the `arch` suite can be developed using the nested loop functionality.

7.3 Unstructured Lagrangian Hydrodynamics

Lagrangian hydrodynamics assumes that the mass within a fluid volume is fixed, but that pressure translates, compresses, and expands that cell over time. This deformation of the cell means that the Lagrangian approach leads to dynamic mesh motion throughout the solve, whereas the Eulerian approach leads to a static or fixed mesh.

The `lags` application has been written from scratch as a solver for Lagrangian hydrodynamics that can solve for arbitrary polyhedra using a predictor-corrector scheme. The solver uses a compatible discretisation, where the gradient and divergence operators have the same properties in their approximate discrete forms. Further to this, the solver uses an edge-based artificial viscosity, and a subcell discretisation so that additional pressures can be introduced to avoid hourglassing. The solver is a significant departure from the Eulerian approach discussed in Section 7.2 as the geometry is exposed within the solve. Also, `lags` has been written to support fully unstructured meshes with static connectivity throughout the duration of the solve.

The `lags` application has similarities to applications such as PENNANT[2], and to some extent LULESH[3] [48] [79]. The reason that `lags` was developed in isolation from those applications is similar to the reasoning for the other applications in `arch`:

- The programming language is C throughout, which is the easiest language to interoperate with low-level, directive-based, and C++ abstraction programming models.
- As with all other applications in `arch`, `flow` is designed to support multiple parallel programming models within a single source framework, where only the core parallel loops are duplicated between models.
- Again, the exposed computational code in `lags` is lightweight, at roughly 1000 LOC, while PENNANT is 5000 LOC, and LULESH is around 7000 LOC.
- As the application is hosted by the `arch` project, it is possible to provide insights into managing hydrodynamics as part of a suite of applications.
- The application is an integral component of the `hal3d` Arbitrary Lagrangian-Eulerian hydrocode, discussed in Section 8.2.6.

For consistency, a similar analysis will be provided for `lags` as was provided for `flow`, to enable easier comparisons between the performance profiles for the different numerical methods.

7.3.1 Performance Analysis

Although the hydrodynamics packages share a number of common characteristics, the Eulerian and Lagrangian frames of reference introduce significant differences in the numerical method. This is even more pronounced in `lags` as it supports fully unstructured meshes, composed of a single type of polyhedra. The processors used in this section are discussed in Section 4.2, and the compilers used are: Intel 18.3 for the Skylake and KNL, and CUDA 9.0 for the V100 GPU.

[2] https://github.com/lanl/pennant
[3] https://github.com/llnl/lulesh

7.3.1.1 Default Test Case

The dimensions for the default test case are 256^3, which is less cells than was considered in the Eulerian case. In fact, the total memory footprint of the solver is significantly higher than for `flow`, as there is a subcell discretisation, which means some variables are stored for multiple subcells within a cell. In the experiments discussed in the subsequent sections, the mesh is composed of hexahedrons and initialised as a regular Cartesian mesh. As the solution evolves, the mesh will deform but the connectivity will not change.

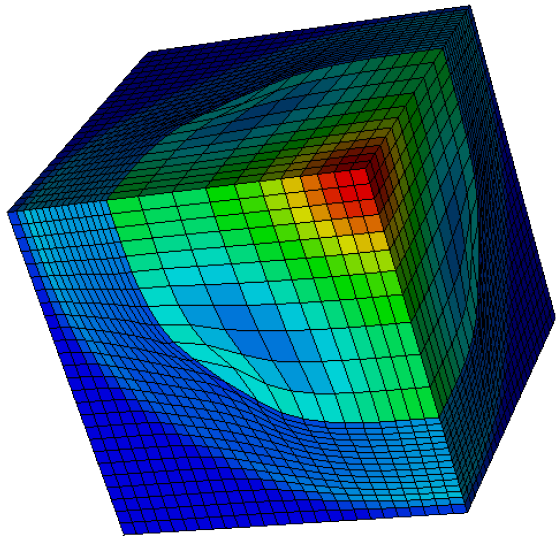

Figure 7.5: Problem solved by `lags`, note that the grid is structured but the algorithms assume an unstructured mesh.

7.3.1.2 CPU Performance

The `lags` application is quite different from `flow`, and introduces new characteristics and issues that must be resolved using numerical fix-ups.

Kernel	Calls	Runtime
calc_nodal_vol_and_c	20	2.9s
calc_subcell_force_from_pressure	20	2.7s
calc_artificial_viscosity	20	5.1s

Table 7.3: Performance by kernel for the Lagrangian solve in `lags` on a Skylake CPU.

Similar to `flow`, the application is composed of many independent kernels; however, the performance of those kernels is significantly less flat for `lags`. There are 19 core computational kernels in `lags`, and the 3 kernels shown in Table 7.3 account for around 73% of the total runtime of the default test problem. One of the reasons that those kernels dominate the performance is because they are involved in the processing of subcell forces, which requires stepping through the large subcell data structures. Different from the other applications discussed in prior sections of this thesis, the geometric data stored in `lags` has to be accessed via indirection. The nodal positions and mesh connectivity are all stored in indirection arrays, and the only fixed

information relates to the properties of the single type of polyhedra that the mesh is constructed from, for instance, the number of nodes and faces on the cell.

Kernel	Mem. Accessed	Mem. Bandwidth
calc_nodal_vol_and_c	6.0 GiB	43 GB/s
calc_subcell_force_from_pressure	8.1 GiB	62 GB/s
calc_artificial_viscosity	9.3 GiB	36 GB/s

Table 7.4: Memory bandwidth by kernel for the Lagrangian solve in `lags` on a Skylake CPU.

Table 7.4 shows the memory bandwidth results for the largest kernels in `lags`. The results demonstrate that the kernels are achieving between 17% and 29% of the achievable memory bandwidth on the Skylake CPU. The memory accessed is measured using a perfect cache model, which considers that a loop will touch each element in an array once upon reading and once upon writing. This is the same model that has been utilised for `hot` and `flow`, and serves as a best case memory bandwidth result, assuming maximum locality is expressed in the algorithm.

Locality is not as easily expressed in unstructured algorithms, as the indirections and unstructured data layouts lead to memory access patterns with potentially large and unpredictable strides. Unlike the `flow` application, where the simple implementation resulted in a high utilisation of peak memory bandwidth, the `lags` application only achieves a fraction of the memory bandwidth in the most expensive computational kernels on the Skylake CPU. It is hypothesised that a greater fraction of peak memory bandwidth could be achieved using blocking, to improve locality as threads traverse the mesh. In an unstructured mesh this might be achieved by re-ordering the cell list so that threads operate on more focused chunks of the mesh.

7.3.1.3 Preliminary GPU Performance

Implementing the solver on the GPU was relatively straightforward, and the initial implementation used the same traversal strategies and data structures as were used on the CPU.

Kernel	`nvprof` Mem. Bandwidth	Mem. Bandwidth
calc_nodal_vol_and_c	730 GB/s	54 GB/s
calc_subcell_force_from_pressure	688 GB/s	18 GB/s
calc_artificial_viscosity	639 GB/s	23.3 GB/s

Table 7.5: Memory bandwidth by kernel for the Lagrangian solve in `lags` on a NVIDIA V100 GPU.

Table 7.5 shows that while `nvprof` is reporting a large fraction of peak memory bandwidth is achieved, the analytic model, which considers perfect caching, is showing that in the worst case only 2% of bandwidth is being achieved. This demonstrates an important issue with the measurement of memory bandwidth, where `nvprof` is reporting the true memory bandwidth across the bus, but the effective bandwidth is significantly lower in fact, and a greater fraction of effective bandwidth can be achieved. The perfect caching model assumes that the locality has been expressed well in the algorithm, but this is not the case with the GPU implementation, and algorithmic changes are required. In order to optimise the application, it is necessary to think about the data structures that handle the unstructured mesh and the subcell data.

7.3.1.4 Implications of Supporting Unstructured Meshes

There are multiple challenges introduced when supporting an unstructured mesh within a hydrodynamics package, even when the mesh itself is initialised as a structured mesh as for `lags`. The loops now need to include indirections, in order to determine which parts of the mesh connect together. The indirections increase the memory footprint, and introduce overheads depending upon the mesh traversal strategies. Another challenge introduced by the nature of the unstructured mesh indirections is that it is sometimes necessary to perform a short search through the indirection arrays to match the connectivity between different elements of the mesh. Those operations potentially result in a traversal of unused data elements, which is undesirable.

7.3.1.5 Implications of Supporting Subcell Forces

The kernels that dominate performance in Table 7.4 are long and complex, making it difficult to perform a straightforward analysis. A more simplistic kernel also touching subcell data is considered to make the discussion clearer and easier to follow, but the findings are directly applicable to the most expensive kernels in the application. The techniques discovered using this test kernel will later be applied to optimised the GPU implementation.

Code Sample 7.5: Energy correction routine in `lags`.

```
1  for (int cc = 0; cc < ncells; ++cc) {
2    double cell_force = 0.0;
3    for (int nn = 0; nn < NNODES_BY_CELL; ++nn) {
4      const int ni = cells_to_nodes[cc * NNODES_BY_CELL + nn];
5      const int si = cc * NSUBCELLS_BY_CELL + nn;
6      cell_force += (velocity_x0[ni] * subcell_force_x[si] +
7                     velocity_y0[ni] * subcell_force_y[si] +
8                     velocity_z0[ni] * subcell_force_z[si]);
9    }
10
11   energy0[cc] -= dt * cell_force / cell_mass[cc];
12 }
```

An example kernel is presented in Code Sample 7.5, where the method first loops over the cells in the mesh, and then over the nodes attached to each cell. In this regime, every element of `subcell_force_{x,y,z}`, `energy0`, `cell_mass` and the indirection array `cells_to_nodes` will be accessed once, while every element of `velocity_{x0,y0,z0}` will be accessed up to `NNODES_BY_CELL` times. In the perfect caching scenario, the velocities are stored within cache and never need to be refetched after their first use in the kernel. In reality, when considering the default test case, the velocities in total are around 400 MiB of data and the caching is dependent upon the architecture and data layout. In total, the routine touches around 4.3 GiB of data, and processes it with a bandwidth of 184 GB/s on a Skylake CPU, which is around 85% of achievable memory bandwidth. Around 11% of the total memory footprint in this kernel is introduced by the indirection array for accessing the nodes surrounding each cell.

Considering this same kernel executed on the NVIDIA V100 GPU, the results are quite different, as the achieved memory bandwidth, based on a perfect caching model, is 130 GB/s, which is around 15% of achievable memory bandwidth. A major issue that can be observed on the GPU is that the `cells_to_nodes` indirection and the subcell data `subcell_force_{x,y,z}` are not accessed in a coalesced manner. A reorganisation of the subcell data, so that the access

is by nodes in the leading dimension, enables coalescence, and increases the achieved memory bandwidth by 5.5x to 716 GB/s.

7.3.1.6 GPU Data Structure Transposition

Having recognised the issues with the data structures, it is observed that many of the kernels are not performing coalesced memory accesses. In order to resolve this, it is possible to re-organise the data structures to attempt to enable coalesced access for the largest number of kernels possible.

Kernel	Mem. Bandwidth	Speedup
calc_nodal_vol_and_c	171 GB/s	3.1x
calc_subcell_force_from_pressure	151 GB/s	8.3x
calc_artificial_viscosity	93 GB/s	4.0x

Table 7.6: Memory bandwidth by kernel for the Lagrangian solve in `lags` on a NVIDIA V100 GPU, with transposed data structures.

The results in Table 7.6 demonstrate that a significant improvement in performance was possible by transposing those data structures. The solver is now achieving between 11% and 20% of the peak achievable memory bandwidth, which brings it closer to the fraction of peak performance achieved on the Skylake. Taking the routine `calc_subcell_force_from_pressure`, for instance, the parallel loop steps over the cells in the mesh. In the loop it is necessary to iterate over all of the faces attached to a cell, and then further fetch the nodes attached to each face. The structures `cells_to_faces`, `cells_to_nodes`, and `subcell_force_{x,y,z}` have all been transposed so that they are accessed in a coalesced manner by the GPU threads. The pressure array is cell-centered and the kernel traverses by cell, and so this array was already accessed optimally by the threads. The total capacity of the arrays that can be accessed in a coalesced manner is 7 GiB, while the remaining data structures, containing over 1.1 GiB worth of data, are not accessed in a coalesced manner. Coalescing accesses to the remaining data structures would require fundamental algorithmic changes, that could potentially lead to the performance of other kernels being affected. Further, coalescing accesses to those arrays might mean that it is not possible to coalesce accesses to the larger subcell data structures, which would defeat the purpose.

Although the re-ordering of data structures can have a positive impact on the performance, there are also cases where transposing the data structures results in worse performance. An example is the data structure `cells_to_nodes`, which is optimally accessed in node order if the iteration space is traversed in node order, with the converse also being true. There is a surprisingly simple resolution to this problem, which is to duplicate the indirections specialised to the particular traversal. This optimisation increases the overall memory footprint of the application, but does not increase the memory footprint at the kernel level, and enables coalesced memory accesses. A problem with this approach is that it is only useful for the data that is static during the solve, which is the indirection data. For the variable data, the same effect could only be achieved through transposition operations in between kernels, which is prohibitively expensive.

7.4 Summary

The problem of structured Eulerian hydrodynamics can be solved in a straightforward manner due to the regular organisation of the mesh allowing the kernels to perform simple geometric calculations. Although the kernels in `flow` are of varying length and complexity, and the performance profile is flat between the kernels, achieving good performance was straightforward through consistent organisation of data structures to enable stride one memory accesses, and minimising the amount of data touched within each kernel. The low arithmetic intensity of the application means that the performance on each different processor follows the available memory bandwidth, as was observed with `hot`.

It was shown that performance portable implementations of the application could be successfully developed with RAJA, OpenMP, and OpenACC, although there is currently a performance bug limiting the performance of *min* reductions for OpenACC on the KNL. It was possible to use `flow` to consider the level of productivity offered by each of the parallel programming models. The number of code sites that needed to be changed was low for all of the models, but there were some challenges exposed in RAJA in terms of organising the data traversal for optimal memory access, that might force the use of the RAJA nested loop syntax. The nested loop syntax was not directly explored with `flow`, but represents an interesting area for future work.

The problem of unstructured Lagrangian hydrodynamics was also considered, using the `lags` proxy application. The unstructured mesh with subcell data lead to challenges in terms of organising memory for the best performance on each of the architectures. It will be important future work to consider mechanisms by which the traversal of data structures can be encapsulated and accesses to specific arrays transposed, depending upon the architecture. RAJA offers some applicable functionality, but this is limited to the traversal of the iteration space, which might not be flexible enough to resolve the issues observed in `lags` or relevant production applications.

Chapter 8

Production Challenges

Key Publications

M. Martineau, P. Atkinson, and S. McIntosh-Smith. *Benchmarking the NVIDIA V100 GPU and Tensor Cores.* In International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms, 2018.

M. Martineau and S. McIntosh-Smith. *The arch project: Physics mini-apps for algorithmic exploration and evaluating programming environments on HPC architectures.* In 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017.

M. Martineau, and S. McIntosh-Smith. *Exploring on-node parallelism with neutral, a Monte Carlo neutral particle transport mini-app.* In Cluster Computing (CLUSTER), 2017 IEEE International Conference on, 2017.

S. Fogerty, M. Martineau, R. Garimella, and R. Robey. *A comparative study of multi-material data structures for computational physics applications.* Computers and Mathematics with Applications, 2018.

The focus of this thesis has been the performance, portability, and productivity of a number of small research codes, sometimes called proxy applications [63]. The concept of using small codes in place of larger applications has been used for a long time, for instance, in 1984 Brown et al. used a small representative Monte Carlo proxy application to prototype their *over events* version of the Monte Carlo algorithm [23]. In spite of their long and prolific use, there are a number of issues inherent with the use of proxy applications, particularly when they are intended to represent specific production applications.

As discussed by Hemmert et al., the applications might not capture all of the necessary features of a production application, and this is an important point of discussion and critical evaluation [58]. This section provides some critical analysis of the results of this thesis, by considering which aspects of performance are not represented in the proxy applications discussed

in the preceding chapters.

In some cases the decisions will only influence the performance by some small margin for a particular architecture. In other cases the performance difference can be so significant that codes optimised for a CPU, for instance, can have unacceptably high runtimes on other architectures. Further, it is possible that optimisations applied to an application do not work at all when introduced into a production application because the proxy application was not representative in its feature set.

8.1 Infrastructural Code

Real production scientific codes can include multiple physics packages that need to be co-ordinated, which means transferring, and potentially transforming, input data between the packages. For instance, the Integrated Forecasting System developed by the European Centre for Medium-Range Weather Forecasts includes physics packages that solve convection, radiation, cloud physics, etc. [55].

Such applications can be comprised of hundreds of thousands of lines of code, where a portion of the code has to be dedicated to common components that handle tasks such as reading inputs, managing data tables, performing visualisation, and other tasks indirectly related to the wider simulation [62]. It is important that there is a delineation between the two concerns, as the programming approaches and models that are best suited to high performance parallel computation are unlikely to be the same as those best suited to building a robust software application.

8.1.1 The `arch` project

Throughout this thesis, a number of the applications have been discussed that are part of `arch`. The `arch` suite is a collection of applications, while the `arch` project[1] is software in its own right that contains all of the cross-cutting infrastructural code for the applications contained within the suite [97]. The code has been developed in an attempt to demonstrate the feasibility of completely isolating the infrastructural concerns of a set of applications, while maintaining generality and completeness.

Figure 8.1 depicts the applications in the `arch` suite (bottom), and the shared features in the `arch` project (top). Each of the components has to work with the range of parallel programming models, including threaded and distributed models.

8.1.1.1 Communication

The communication component is straightforward in `arch`, and essentially manifests as a wrapper around MPI calls. This allows MPI to be disabled at compile time and the interfaces to be simplified. It is possible that certain applications might require different communicators, which is currently a limitation of the approach.

8.1.1.2 Meshes and Mesh Data

Meshes are constructed within `arch`, and any application-specific artefacts are managed directly by the application code. The project supports multiple mesh types, structured and unstructured,

[1] https://github.com/uob-hpc/arch/

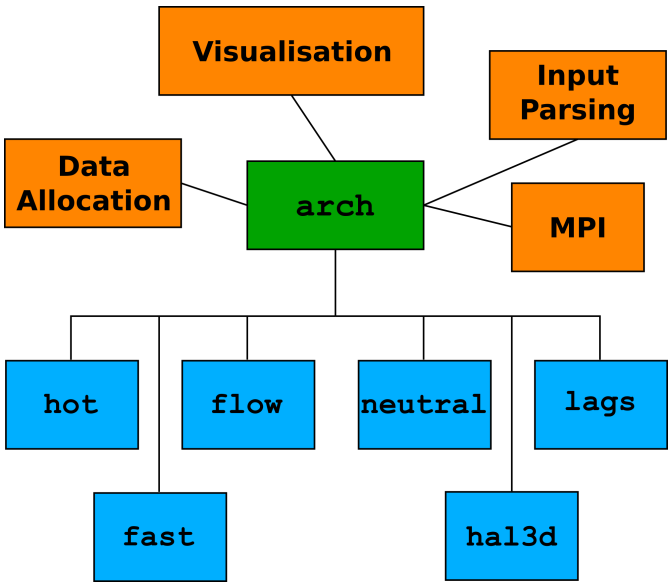

Figure 8.1: The `arch` infrastructure.

as well as some sub-cell discretisations, but the initialisation is typically to a regular 2D or 3D Cartesian mesh, for simplicity and ease of analysis. In the unstructured case the mesh is allowed to deform, where the mesh is stored as list of nodes (or vertices) continuously positioned in space, with static connectivity. No assumptions are made about the mesh connectivity that would inhibit simulation on any unstructured mesh, but it is possible to choose to restrict the mesh so that it is constructed of only a single type of polyhedra, reducing the number of required indirections.

Managing meshes as shared resources is relatively straightforward when considering an individual physics package; however, the management of meshes that span several packages within a multi-physics environment can be more challenging. In most applications, the choice of mesh is consistent between the different packages, though it is possible to convert the mesh between packages. The problem is that continually converting between two types of mesh comes at the cost of additional memory movement [73].

8.1.1.3 Performance and Portability

Although the individual features of the `arch` project are not novel, most applications will be required to develop some provision for those components, and there are some interesting issues that arise in terms of performance portability. There was the additional requirement with `arch` that it could support a range of parallel programming models (see Chapter 3), which demanded greater generality than might be necessary in a production infrastructural code.

The programming models that `arch` currently supports are MPI, CUDA, OpenMP, OpenACC, and RAJA, which span multi-threaded CPU execution, GPU execution, and distributed execution. The key findings from a performance portability perspective were:

- *It was possible to achieve functional portability with all of the programming models using an encapsulated infrastructural layer:* The experience strongly supports the idea that a independent infrastructural interface can be written in any chosen language, with the computational components developed in a high performance language such as Fortran

or C. Although `arch` is written purely in C, it is recommended that the infrastructural layer of new codes is written in a performant but modern programming language, particularly one that handles generic programming, such as C++. The reasoning behind this recommendation can be found in Section 8.1.2.

- *Data management routines were best encapsulated in generic wrappers:* This includes all allocation and data movement. By encapsulating the data management, the cost of changing parallel programming models is greatly reduced, as long as the encapsulation is written in a general manner. Key optimisations, such as ensuring NUMA first touch allocation is correctly handled, can be applied to all routines with relative ease. The main limitation of this approach, which is the repetitive nature of such APIs, can be alleviated using generic programming features of a language, such as C++.

- *Both OpenACC and RAJA easily integrated into `arch`:* When considering only the performance portable programming models, `arch` was easily made portable between CPU and GPU, for instance, using OpenACC and RAJA. OpenMP required two different versions of the core features, which contradicts the single source requirement imposed by many scientific institutions. With some additional effort, it is possible to develop a single source version of `arch` by using preprocessor macros to switch between the CPU and GPU implementations; however, this approach essentially fakes performance portability, and more works needs to be done within the language to support this issue.

With a relaxed interpretation of performance portability, where the only goal is to be able to achieve functional portability to existing parallel processors, within some tolerable expectation of performance, OpenACC and RAJA were successful in achieving performance portability with the `arch` project. It was possible to construct test problems where the overheads of performance portable programming models were prohibitively large, but it is unlikely that the test problem represent true scientific workloads, where the overheads are expected to be amortised (Chapter 6). Care needs to be taken to ensure that data management is handled in a robust and portable manner, and further investigation is required to understand how those programming models handle other, more complicated, production features.

8.1.2 Note on Programming Language Choice

Many scientific software applications are written in Fortran, and, while support for the earlier versions of the Fortran language is extensive in modern compilers, support for modern Fortran features and tooling is not [26]. In spite of this, modern Fortran includes a multitude of features that attempt to emulate the successful aspects of object-oriented programming languages, such as C++, which improve the prospects of Fortran as a suitable candidate for the development of the whole application stack.

As part of this thesis it has been found, using proxy applications such as TeaLeaf, that Fortran can be used at the loop level by domain scientists, with C++ used to manage the cross-cutting concerns and infrastructural code [103]. This maintains the benefits to the scientific programmers of Fortran for clean kernels with first class multi-dimensional array support, and improved chances of auto-vectorisation for those performance-sensitive computational components. The infrastructure programmers can leverage generic and object-oriented programming to make the performance-insensitive components easier to develop and maintain, while ensuring that components can be easily encapsulated to make the application more robust.

In many U.S. labs, core applications have been ported entirely to C++, with Sandia national laboratory using Kokkos, and Lawrence Livermore National Laboratory using RAJA for performance portability [47] [67]. This change from Fortran required scientific developers to become familiar with the C++ language and idioms, and does have some drawbacks. Extensive use of C++ might make it more challenging to port applications to achieve good performance on modern architectures, and sets a more challenging barrier-to-entry for new scientific software developers.

Computer scientists, particularly those with training in parallel programming and high performance computing, have an acute awareness of the implications to performance and portability of certain coding practices. Those computer scientists might be able to develop highly performant C++ codes that can interoperate with the available programming models and libraries. The same cannot necessarily be expected of scientific software developers whose focus is primarily on the pursuit of their particular scientific goals, especially when you consider the increasing complexity of targeting today's massively parallel processors. The use of C++ abstraction layers, such as RAJA and Kokkos, greatly reduces the risk that scientific developers become bogged down with the features provided by the language, as they are able to instead focus on the abstractions provided by the model.

8.2 Features Sometimes Ignored in Proxy Applications

Packages coupled in multi-physics applications are generally consistent in their handling of specific features, for instance, if the solution requires multi-material interfaces, then all of the packages are required to handle those interfaces. In this section, features that might be required in production codes, but are typically ignored in proxy applications, will be considered.

Proxy applications exist today that handle each of the features discussed in this section, for instance, Quicksilver handles multiple materials, XSBench manages large lookup tables, and SNAP handles a numerical error in the main computational loop [20] [158] [34]. In spite of this, proxy applications are regularly developed as isolated projects that focus on a single package, with reduced feature sets. Optimising a package without considering a particular feature might lead to poor performance once those optimisations are applied to a production application.

8.2.1 Multiple Materials

The introduction of multiple materials means that the mesh data structures become significantly more complicated, and some additional computation is required to manage material interface tracking. Figure 8.2 shows an example 3x3 grid containing 4 materials [49]. The materials have continuous interfaces independent from the mesh discretisation. Beyond the numerical issues of solving multi-material problems, the key computational challenge can be observed in the difference between cell 0 and cell 7. Cell 0 contains 1 material, while cell 7 contains 4 materials, which requires 4x the storage of relevant variables.

In many cases it is not possible to determine apriori a sensible upper limit on the number of materials, as different problems will require varying numbers of materials. For most problems, accessing the data structure now relies upon the use of indirection arrays, which increases the memory footprint, and, depending upon the algorithm, might lead to non-contiguous memory accesses. The requirements of simulating multiple materials on a structured grid mean that the choice of data structures is not easily determined. As part of this thesis, experiments were

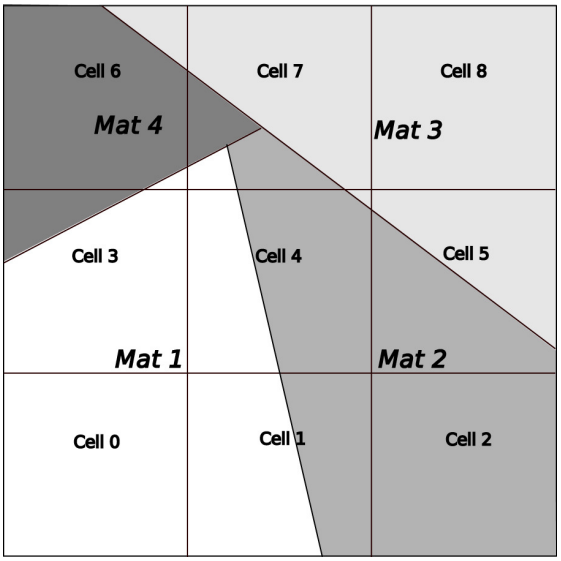

Figure 8.2: Multi-material layout for a structured mesh, showing material interfaces [49].

performed that looked at the performance of multi-material data structures on modern parallel processors [49].

Figure 8.3 and 8.4 present the memory bandwidth achieved with the particular data structures for three exemplar kernels. The kernels are memory bandwidth bound and represent workloads that could be found in a typical hydrodynamic application requiring multi-material interfaces. Details of the data structures are given in the paper [49]. The results demonstrate that, for even simple structured grid kernels, the performance could be dramatically altered by the selection of multi-material data structures.

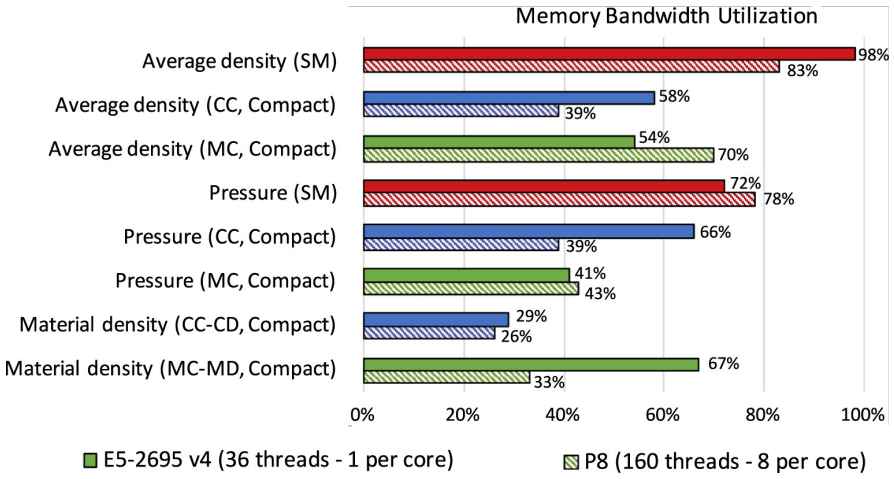

Figure 8.3: Performance of multi-material data structures ported to two CPUs, where P8 refers to the IBM POWER8 [49].

In the single material (SM) case, a reasonably high percentage of peak memory bandwidth is achieved for the two key kernels, this is because the kernels are simple and contain contiguous stride 1 memory accesses. As seen in Chapter 7, single material hydrodynamics applications are typically able to achieve a high fraction of peak memory bandwidth, and the optimisation of

hydrodynamics applications often relies upon this.

For the material-centric (MC) data compact structure, the worst observed performance is 18% of peak memory bandwidth, while the best performance observed is 61% of peak, over 3x higher. This is a significant reduction in the achieved peak memory bandwidth, and this thesis has found that optimising multi-material data structures is a challenging area that requires further investigation.

The combinations of kernel and processor influence the optimal choice of data structure. Further, the different choices of data structures have important consequences on the structure of the individual kernels, as some data structures might, for instance, lead to searches within the computational loops, which can introduce a significant performance overhead. The choice of data structure also imposes restrictions on the performance of all of the physics packages within a production application, where each package might perform optimally with a different data structure. As such, the search space for optimisation becomes larger and more challenging to explore, given that each of the data structure choices requires significant changes to the computational kernels.

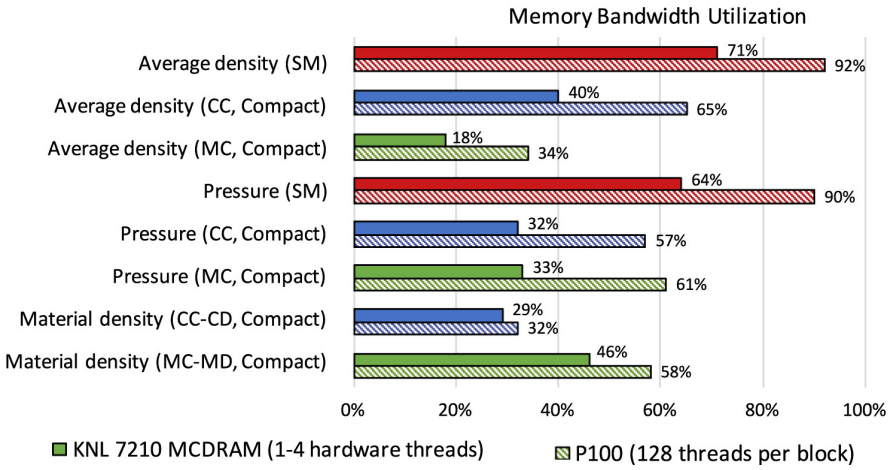

Figure 8.4: Performance of multi-material data structures ported to the KNL and P100 GPU [49].

This particular investigation into multi-material data structures focused on the case where the underlying material structure was static. The problem is even further complicated once particular forms of fluid motion are enabled.

8.2.1.1 Eulerian Flow Field

If the flow field is described in an Eulerian manner, then an even larger problem is introduced than the static multi-material data layouts. Those material interfaces will no longer be determined apriori and the interfaces must be maintained throughout the simulation as they advect with the flow.

The key issue with this formulation is that mesh cells can contain as few or as many materials as exist within the system, and the frequencies can change each timestep. The consequence is that it is challenging to perform data allocation, as allocating space for all possible materials within all cells will greatly increase the memory footprint as the number of materials grows. This might impact the performance in the best case, or make lead to prohibitive memory footprints

in the worse case.

Dynamic data structures are required that grow and shrink to accommodate the flow of materials through the system. In most cases, the use of dynamic data structures will lead to the potential for race conditions, that must be guarded using, for instance, atomic instructions. This can have a major influence on the performance depending upon how regularly the race conditions need to be handled throughout the simulation. The implementation of such data structures on modern parallel processors is an open area of research.

8.2.2 Large Lookup Tables

Tramm et al. observed that OpenMC requires lookup tables amounting to several gigabytes in total capacity [158]. The findings of the present study were that `neutral`'s performance profile was altered by the inclusion of large lookup tables, and writing an optimal version of the code that could handle the different lookup table sizes is challenging. As the size of the lookup table was increased, to the point where the lookup tables spilled out of the available caches, certain routines transitioned from compute bound to memory bandwidth bound. In more extreme cases, where the size of lookup tables are larger than the available DRAM capacity for a particular node, the problem could lead to an application or specific kernels having to page data to and from disk, or from the limited HBM on a GPU to the DRAM of a CPU.

Large lookup tables are used for other purposes than cross sectional lookups, for instance with Equation of State (EOS) calculations [137]. Given the significant performance impact that large lookup tables had on the performance of `neutral`, it might be useful to consider the performance of large lookup tables in proxy applications that have typically used small closed form equations instead [4].

8.2.3 Load Imbalance

Large lookup tables can be an issue from the perspective of managing the data capacity, and formulating efficient data structures to hold such information. Further, there is another issue that can arise from large EOS tables, the potential for load imbalances. It is possible that materials in a simulation require different equations of state, for instance, one material could be represented by a closed form ideal gas equation, while another material requires a lookup in a large EOS table. This difference in the time taken to calculate the equations of state could lead to a load imbalance, which has implications at the node-level and for distributed execution.

Karlin et al. introduced an optional parameter to emulate load imbalance for the EOS equations within LULESH [78]. This represents an interesting and potentially powerful approach, whereby the true computational features of a problem are essentially abstracted and parameterised. This reduces the requirement to maintain complex domain-specific code, and the flexibility to experiment with various parameters.

There are many other sources of load imbalance, for instance, Monte Carlo neutral particle transport solved across distributed domains will generally have to manage load balancing [132]. Geist et al. predicted that load balance will become a major issue moving towards exascale computing due to the number of distributed processors co-operating in the simulation [52].

8.2.4 Internal Error Handling and Diagnostics

Typically, internal error handling and diagnostics are left out of proxy applications, either as part of the initial refactoring exercise, or simply because the application was written from the ground up. Error handling and diagnostics are an important aspect of many production applications, and might have subtle consequences for the resulting applications.

Error handling within loop bodies can inhibit vectorisation, for instance, due to early loop exits or printing error messages. In most cases, error handling can be moved outside of the loop bodies, leaving the error checks inside the loop and potentially leveraging parallel reductions to capture the results of the error check. If more robust results are required by the error handling, then it may be necessary to store more extensive results from the error checking, for instance, on a per-cell basis of a computational mesh.

8.2.5 Dynamic Connectivity in Unstructured Meshes

Another issue that has not been addressed in this thesis is the problem of dynamic connectivity in unstructured meshes. Applying the constraint of static connectivity to unstructured meshes enables many optimisations that would otherwise be impossible. Koniges et al. developed a code called ALE-AMR as part of the validation process for the National Ignition Facility at Lawrence Livermore National Laboratory [83]. The code simulated dynamic mesh connectivity, including tearing of the mesh; faithfully representing this production code would greatly increase the complexity of the hydrodynamics applications considered in this thesis.

8.2.6 Mesh Quality Control

As part of this thesis, a 3D ALE hydro code, `hal3d`, was developed from scratch that uses a subcell discretisation and supported arbitrary polyhedral meshes [51]. This was an attempt to develop a proxy application that managed more challenging computational features than present in the other applications considered in the preceding chapters.

In hydrodynamics, mesh quality will affect the timestep, as the Courant-Friedrichs-Lewy (CFL) condition stipulates that the timestep must be bound by the speed of sound within the system [19]. This condition asserts that information cannot travel further than the bounds of an individual cell within a single timestep; as the timestep is a scalar value for the whole mesh this limitation means that a single small or narrow cell will reduce the timestep for the whole simulation, as seen in Figure 8.5. Another issue of mesh quality that can occur with Lagrangian hydrodynamics is the tangling of meshes due to rotation, or vorticity, in the problem. Once a mesh has tangled, the results are no longer correct, leaving few options for successful simulation.

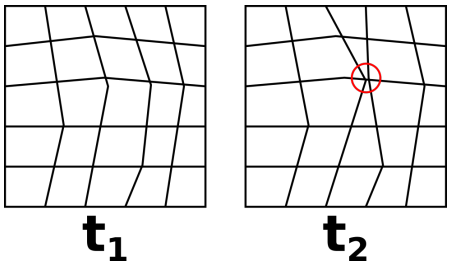

Figure 8.5: A mesh where compression will lead to a reduced timestep.

Arbitrary Lagrangian-Eulerian (ALE) hydrodynamics is an extension of Lagrangian hydrodynamics that adds a fix-up for the issues with mesh quality. The approach allows one or more Lagrangian hydrodynamical steps to occur before applying some relaxation algorithm to the mesh, ensuring that the mesh does not tangle or suffer from an imbalance of mesh cell compressions. The remap step has to operate in a conservative manner and so many algorithms need to

be applied to ensure that all conservation laws are strictly upheld. Regardless of performance, this remap step will allow the solver to continue the simulation, and improves the length of the timestep by improving cell quality based on the CFL condition.

Even implementing the bare minimum code necessary to handle ALE remapping for arbitrary polyhedra, the code is significantly larger (roughly 10000 LOC) than the other applications in this study, making it much more difficult to work with. The process of porting to new parallel programming models, for instance, took far longer than with the small proxy applications.

8.3 Problems for Proxy Applications

The majority of proxy applications in use today are isolated codes that have been stripped down from larger scientific applications or research codes, or written from the ground up to represent some class of applications [63]. The removal of features such as those discussed above can mean that coding efforts attempting to optimise the core solvers miss out on important characteristics of the performance profile of the real applications.

As suggested by Dosanjh et al. and observed in Section 8.2.6, it is challenging to perform agile experiments with codes that are much larger than 1000 LOC [45]. Introducing large features, such as multiple-materials, increases the code size and complexity, and greatly reduces the capacity for Computer Scientists to optimise the codes in a timely manner.

A recommendation of this work is that, where applicable, independent proxy applications are developed to pilot features, as performed in the multi-material investigation in Section 8.2.1. Separation of concerns is not without risk, as features such as multi-materials might have far-reaching influences on the proxy application, potentially even requiring different core algorithms. For more contained features, an alternative approach is that of emulation and parameterisation, proposed by Karlin et al., depending upon the particular feature and how pervasive the changes are to the code [78].

8.3.0.1 Applications in the `arch` Suite

Considering the applications that are discussed in this thesis, it would be possible to incorporate several of the discussed features. For instance, the `hot` application could become part of a multi-physics stack that requires an unstructured mesh, which would necessitate the use of a different algorithm, perhaps GMRES.

It is hypothesised that introducing multi-material cells into `flow` would be far more disruptive than the results of the research described in Section 8.2.1 suggest, as the application results in an Eulerian motion of the underlying fluid. The change would impact most of the routines in the application, which would have to access dynamic data structures through indirections. This could have a major influence on the performance of the application, and the impact will be variable depending upon the target architecture. It would be useful future work to add multiple materials or some load imbalance to the hydrodynamics applications, `flow`, `lags`, and `hal3d`, perhaps using the approach in LULESH.

With respect to the Monte Carlo neutral particle transport application, the relevant features can be influenced by the particle that is being transported. Photons and neutrons, for instance, behave differently and lead to different physical approximations in the simulation. The purpose of `neutral` was to capture some of the similar performance-affecting problems of the general class of neutral particle transport applications, but issues such as fission, which is specific to

neutronics, will introduce problems that are not captured by the work. Error handling can have a much more significant impact on an *over histories* Monte Carlo neutral particle code, as the majority of the computation is performed in a single large parallel computational loop. If the error handling was not parallelisable, then it would likely be necessary to store additional state and perform the error handling at the end of the timestep.

Even though the features discussed in the preceding sections exist in some production applications, the results collected for the proxy applications in this thesis are still pertinent to many applications. In fact, the results can be representative of classes of the applications, but the features discussed above introduce nuances that might need reconsideration for particular problems. Coupling this with the recurring issue of problem dependence, shown especially with `neutral`, the search space for optimisation is large and challenging to explore.

8.3.0.2 Validation of Proxy Applications

Proxy applications must be validated to ensure they return correct answers, to catch bugs and avoid inadvertently adjusting the amount of work processed. This is regularly accomplished using either pre-calculated or analytic solutions. Another interesting aspect of validating proxy applications considers quantification of their representativeness, which can be challenging due to the reduced feature-set of proxy applications.

Researchers at Lawrence Livermore National Laboratories developed the Veritas project, that collects performance counters and supports comparisons between real application and proxy applications [91]. In theory, this allows proxy application developers to tune their codes to more closely represent production applications. It would be important future work to analyse the `arch` proxy-apps using Veritas, in order to specialise them to particular production applications, potentially in different domains.

8.4 Summary

The `arch` project is one of the only examples of a common infrastructural code that supports multiple research codes, and represents a novel effort in extension to the general area of research relating to proxy applications [63]. Further, it is the only known example that supports multiple parallel programming models for those applications, and provides a useful tool for the investigation of performance portability in this regard. The project has shown that it is possible to develop such an infrastructural layer to support performance portability, but there are some issues that arise from attempting to construct the generic interfaces such that new applications do not require radical shifts in the code architecture.

The use of proxy applications is now popular amongst computer scientists for algorithmic exploration, but this chapter has shown that there are a number of key features of real applications that are not often considered in the proxies. Those features can have implications for the implementation on modern hardware, for instance, managing dynamic multi-material data structures on a GPU is a challenging open area for research.

Developing truly representative proxy applications is challenging, especially when considering multi-physics applications. In spite of this, it is generally intractable to perform experimentation with production applications, and so the use of proxy applications is the best option for many problem domains. Proxy developers must carefully consider the impact particular features have on the resulting implementation, and recreate the performance profiles as faithfully as possible.

Chapter 9

Conclusions and Future Work

This thesis has presented a thorough investigation into the porting and optimisation of four important types of physics application: Eulerian hydrodynamics, Lagrangian hydrodynamics, Heat diffusion via CG solve, and Monte Carlo neutral particle transport. Brand new exemplar applications have been developed for each, in order to allow extensive performance analysis, algorithmic development, and investigations into performance portability. There were many common themes between the applications, but there were also some differences that make porting and optimisation more difficult in some cases.

Each of the applications has been shown to exhibit some level of problem dependence, which affected the performance of the final solutions. In the case of the conjugate gradient (CG) solver, the performance is relatively normalised if the majority of the memory accesses are from DRAM, as the solver becomes heavily memory bandwidth bound in the sparse matrix-vector multiplications. If the main problem parameter, mesh dimensions, is changed then some performance issues can be observed depending upon the parallel programming methodology. When tuning the size of the computational mesh to a small cache-resident size, the performance results changed quite significantly, as the small workloads in each kernel exposed the varying overheads in the programming models. The small problems are only likely to be required in a strong scaling scenario, where a large test problem is parallelised over a large cluster. As shown in related research, the heat diffusion solver TeaLeaf is heavily communication bound when strong scaling, making the result less relevant to heat diffusion [112]. In spite of this, the overheads have been shown to be present, and there are many applications with small solves, or solves where the parallel workloads shrink over time, for instance, those using multi-grid approaches. It would be useful future work to consider the overheads of OpenMP, OpenACC, and RAJA with respect to diverse algorithms like multi-grid solvers.

The problem dependence in other applications was more far-reaching, for instance, the hydrodynamics package selection is strongly influenced by the desired types of meshes. Solving for structured grids in two or three dimensions was highly performant on all architectures, and the parallel programming models were successful in achieving a high level of performance portability. If the problem instead demands an unstructured mesh, the application is made more complex and the memory footprint is increased by the structures required to handle the mesh. In some of the kernels, the indirections necessary to support the unstructured mesh accounted for 50% of the memory footprint, which has a significant impact on the performance. The indirections potentially mean that data has to be gathered from memory, which can be expensive, depending upon the proximity of the accesses. One of the major issues explored was that it was challenging

to determine an optimal ordering for the data structures, as selecting the best strides for a kernel that traversed the mesh in cell order might not be the optimal selection for a kernel that stepped through in node order. This problem was even more noticeable when switching between architectures, as the CPU and GPU preferred a different ordering of the data structures to optimise for cache or enable coalescence. The memory access patterns of the unstructured routines mean that it was not possible to achieve the full bandwidth predicted by a perfect caching model for all kernels. The traversal approach means that data is touched with large strided accesses and not enough locality is expressed, such that the local memory footprint is too large to maintain the working set in cache, and data has to be regularly refetched. The suggested approach to overcome those issues, in the `lags` application, is to use blocking over the compuational domain, which is left as interesting future work.

The most problem-dependent application considered was the Monte Carlo neutral particle transport proxy, `neutral`. Prior research has discussed issues at the distributed level, considering load balancing and domain decomposition, while this thesis has explored exploiting on-node parallelism [132] [25]. As the solution of the transport equations are handled with explicit representations of the physics rather than through numerical solution, the divergence in physical behaviour translates into divergence in the algorithm at runtime. The different types of events in such a simulation all lead to different paths of execution with individual memory footprints and computational requirements, making it challenging to optimise for a "*general*" case. It was shown that, depending upon the size of lookup tables required for the solution of the problem, the problem could shift between being heavily memory bandwidth-bound to becoming latency-bound, achieving only a fraction of the peak computational throughput and memory bandwidth. The lookup tables become the limiting factor as they increase in size and spill out of cache, as the amount of data fetched per search is greatly increased, and the performance becomes more strongly limited by memory bandwidth.

In the `streaming` case, the performance limiting characteristic is the random memory access patterns of the particles randomly streaming across the computational mesh. The lack of locality means that the memory access patterns are generally unique between threads, and hardware prefetching can do more harm than good for most particle trajectories. The performance of random memory accesses was found to be quite poor on some of the architectures, even when carefully accounting for the true bandwidth. This inhibits good performance for codes like Monte Carlo neutral particle transport, where random memory access patterns are present with little to no predictable locality. The random memory access performance observed on the KNL, in particular, requires more research to understand if there are techniques that can be applied to Monte Carlo neutral particle transport to achieve a greater fraction of peak memory bandwidth.

When targeting a GPU, divergence can lead to an under-utilisation of the available GPU resources, which restricts the achievable throughput. This was not observed for the CG solve or the hydrodynamics solver, although it is possible that equation of state lookups and handling of artificial viscosities could lead to divergence. It is typically recommended that codes containing deep branching would not be suitable for GPU acceleration, as the branching reduces the number of threads co-operating on a stream of instructions, reducing the throughput. A number of algorithms were explored for the Monte Carlo neutral particle transport application, and the results contradict the conventional wisdom. The results showed that, while the divergence does incur some penalty, the best approach on the GPU, the *over histories* algorithm, contained many branches, and still achieved good performance with respect to the CPU. On an NVIDIA GPU, the architectural approach of latency hiding makes it easier to achieve good performance,

as the warp schedulers are able to place more memory requests in flight, and reduce the impact of stalls, when compared to the CPU.

A challenge for the future of Monte Carlo neutral particle transport applications on the GPU is finding ways to ensure that coalescence is enabled, which is not necessarily possible for the streaming requests without some innovation. One possible line of investigation is the biasing of the problem at the cell level, such that individual master particles are followed but that, for each new cell that the particle enters, a large number of imaginary particles are also simulated that have subtly perturbed trajectories, and can uniquely contribute to the statistics. If the biasing was such that the number of particles resident in the cell was large enough to fill a warp on a GPU, it is possible that the statistical accuracy of the solver could be increased, while increasing the utilisation of the GPU resources, resulting in a net gain. This is an important direction for future research, where the fidelity of an algorithm can be increased to take better advantage of the resources of the hardware.

On the CPU, it was possible to improve the memory access performance of the `streaming` case and achieve a high fraction of the peak true memory bandwidth, although this required the development of a novel sort-free algorithm, named the *blocked over events* approach. This approach is successful in improving the memory access performance of the `streaming` case on the Intel Xeon and Xeon Phi CPUs, and reducing the latency issues seen with small lookup tables for the `scattering` case. However, the implementation is significantly more complicated than the *over histories* approach, and more work is required to ensure that it would be a suitable method for a production-scale application.

A criteria for success when porting a scientific application might be to achieve a high fraction of peak performance on all target architectures. In particular, the case where this is achieved with a single-source solution can be considered a performance portable solution, which was in some cases shown to be achievable with a number of important modern parallel programming models. There were cases where the programming model support in the compilers precluded reasonable performance, for instance, the Cray OpenMP implementation performed poorly for the `scattering` problem of the `neutral` proxy application on the GPU, and OpenACC performed poorly for `flow` on the KNL.

This thesis has demonstrated that, in most cases, the considered performance portable parallel programming models, RAJA, and OpenACC, can currently support a performant single source solution for the scientific applications considered. The OpenMP implementation, on the other hand, requires the use of the preprocessor to achieve a performance portable single-source solution, which has limitations for support. This approach is more attractive than duplicating the computational code, which greatly increases the opportunity for bug replication, and increases maintenance costs, but some mechanisms need to be introduced into the specification to support performance portability. The next iteration of the OpenMP specification, version 5.0, will include a directive similar to `kernels` in OpenACC that will enable greater performance portability at the expense of control, although it is not necessarily a full solution to the performance portability problems in OpenMP. It will be important future work to consider if the new features in OpenMP 5.0 are sufficient to enable performance portability in the `arch` applications.

From a productivity perspective, the difference in syntax between the directive-based models and C++ abstraction layers is largely a matter of taste, as the models have been shown to offer numerous productivity enhancements, and all support rapid porting of already threaded loops. In spite of this, many scientific applications are written in Fortran, and so the transition to a

C++ abstraction layer will be a more expensive pursuit than using one of the directive-based models. It is hypothesised that the cost of refactoring codes for threading will represent a more significant cost for legacy applications [159]. In the case of PENNANT, for instance, it was found that the original description of the parallelism required considerable restructuring in order to target a GPU, and this is expected to be more pronounced in a production application.

One of the key conclusions of this research is that it is often the compiler implementations that limit the performance, rather than the parallel programming model, as the models considered all provided features to achieve a high fraction of the best performance possible with tuned implementations. In spite of this, there are some cases that have been exposed as part of this thesis where features missing from a specification can limit the potential for performance, for instance, the lack of shared memory control in OpenMP limits the potential for optimising codes that benefit from shared memory use [99]. In terms of performance portability, the compiler support can introduce a burden to the programmer, as it might be necessary to maintain different versions of parallel descriptions that are tuned for particular compiler implementations.

As each of the applications has been shown to exhibit varying levels of problem dependence and performance portability, it is important to consider the implications of those issues on the future of those codes. Compute performance continues to grow faster than memory performance, and this will exacerbate the issues of problem dependence, as more specialised algorithms are required to account for particular problems. Proxy applications play an important role in preparing for those challenges, and it will be important future work to consider how algorithmic specialisation can be introduced into an application in a robust manner. It must be noted that the use of proxy applications is limited by the representativeness of the feature-set included in the application. It has been shown that modern physics applications contain a number of diverse features, such as multiple materials and large lookup tables, that can greatly alter the performance profile of an application. Applications including features such as complex multi-material interfaces exhibit diverse and challenging performance characteristics, that might be difficult to optimise for modern architectures. In spite of this, it is likely that, going forward, science will become more reliant upon such features to improve the accuracy of simulations, making the future task increasingly challenging.

Appendix A

Instruction Latency

Instructions such as addition, subtraction, multiplication, and FMA are generally optimised such that each execution unit within the processor is utilised for a cycle per simple floating point instruction. Complicated instructions such as division and transcendental functions typically require multiple clock cycles to process, although many architectures provide optimised paths for those instructions if there is some relaxed tolerance regarding accuracy.

Processor cores can contain multiple execution units, meaning that multiple simple floating point arithmetic instructions can be executed in a single cycle. This also applies to vector instructions that accept vector registers; for instance a Skylake 8176 can dual issue FMA instructions on 512-bit vector registers, processing 32 single precision FMAs per cycle per core.

For Intel Xeon and Xeon Phi CPUs the exact latencies are provided in their user documentation; whereas the same values are not extensively openly documented for NVIDIA GPUs. To measure the instruction latencies for all processors, a benchmark was developed that constructs a dependency chain on a variable and repeats instructions on that variable using a single thread. Capturing the clock cycles elapsed for a long chain asymptotes towards the latency in cycles.

Instruction	V100	V100 (FM)	P100 (FM)	K20X (FM)
Add	4 cycles	4 cycles	6 cycles	9 cycles
Mul	4 cycles	4 cycles	6 cycles	9 cycles
FMA	4 cycles	4 cycles	6 cycles	10 cycles
Sqrt	60 cycles	15 cycles	14 cycles	19 cycles
Div	132 cycles	4 cycles	6 cycles	9 cycles

Table A.1: Latencies observed when executing different instructions on NVIDIA GPUs; the 'FM' label indicates that the latency benchmark was compiled with the '--use_fast_math' flag passed to `nvcc`.

Table A.1 demonstrates that the number of cycles to perform dependent floating point operations, the latency for those instructions, has significantly improved across the three generations. Further to this, the number of cycles for a division was greatly increased when an over- or under-flow occurred; for instance, the latency of a division would increase from 132 cycles to 288 cycles for specific test problems if strict mathematical precision is required. In the case that fast math can be enabled the latencies are more stable.

Appendix B

Cache Bandwidth

Cache bandwidth has been measured in several prior studies for a range of processors [117] [36] [72]. The results will be collected and presented for all of the processors relevant to this thesis, with some minor amendments to the prior approaches.

B.0.1 Skylake and KNL Cache Bandwidth

In order to provide the most accurate performance data for each level of cache as well as DRAM, a maximum observed memory bandwidth was taken between two kernels that perform the same operation but one kernel uses non-temporal stores. In all cases the loops were vectorised to take advantage of the 512-bit vector registers and instructions.

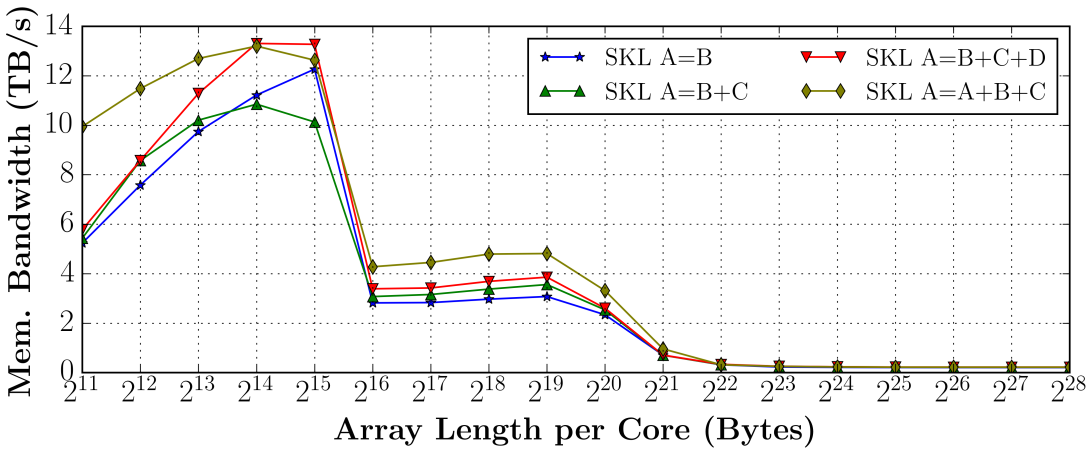

Figure B.1: Cache bandwidth measured for the Intel Xeon Skylake.

Figure B.1 shows the cache bandwidth as measured for the Skylake processor using a number of different kernels. The L1 aggregate bandwidth is shown to be 13.7TB/s, while the L2 aggregate cache bandwidth was around 4.8TB/s. For the L1 cache bandwidth this equates to 127B/cycle, around 66% of the theoretical peak, given that the Skylake supports 128B/cycle reads and 64B/cycle writes in L1.

The cache bandwidth on the KNL is lower than observed for the Skylake, with the maximum L1 bandwidth reaching around 6TB/s and the maximum L2 bandwidth reaching around 2TB/s. For both processors, different results were observed depending upon the chosen kernel.

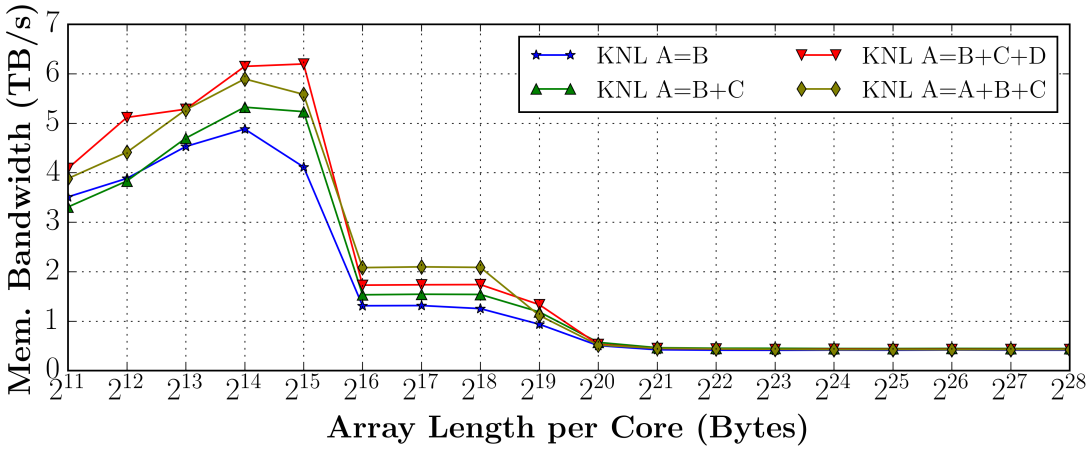

Figure B.2: Cache bandwidth measured for the Intel Xeon Phi Knights Landing.

B.0.2 NVIDIA GPU Cache Bandwidth

In order to measure cache bandwidth on the GPU it was necessary to carefully organise the working set on the GPU. The chosen approach was to maintain a single block of 32 warps per SM, ensuring that a single array was tied to a single multiprocessor, which made it easy to reason about the location of the working set.

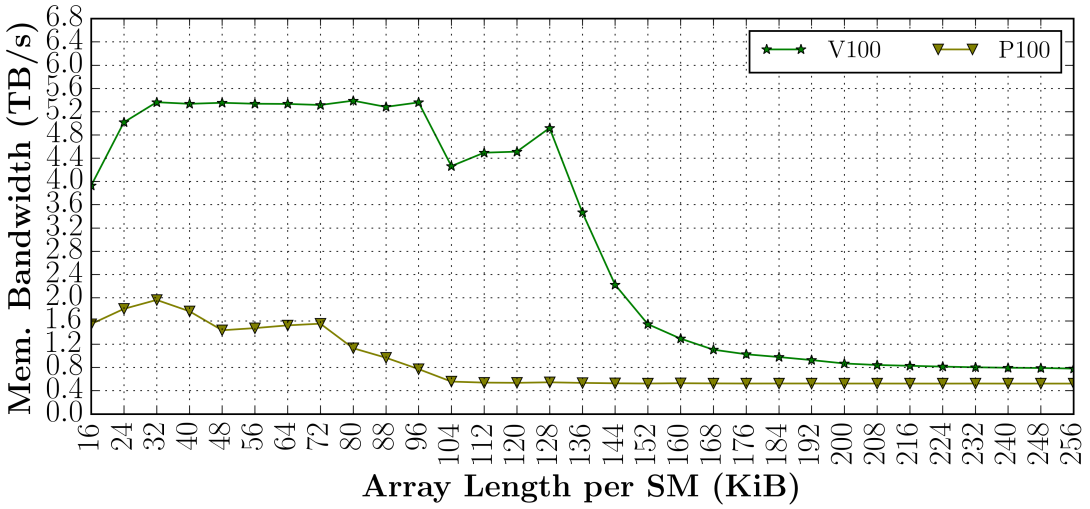

Figure B.3: Bandwidth targeting L1 cache for P100 and V100 GPUs.

The problem with this approach is that the GPU cannot saturate the memory bandwidth of each level of the memory hierarchy using scalar loads for a single 1024 thread wide block per SM. The solution was to adjust the scalar loads to vector loads, which ensures that sufficient memory transactions could be initiated by each of the threads to overcome the limited number of warps per SM.

The performance measured for the L1 cache, as seen in Figure B.3, peaks at nearly 2.0TB/s for the P100 and 5.3TB/s for the V100. This demonstrates a significant improvement in the aggregate performance of the L1 cache, which derives from the increased number of SMs and

the amalgamation of the L1 and shared memory caches, as previously discussed [123].

Code Sample B.1: Cache bandwidth benchmark.

```
asm volatile("{\n\t"
    ".reg .f32 t<13>;\n\t"
    "ld.global.cg.v4.f32 {t1, t2, t3, t4}, [%1];\n\t"
    "ld.global.cg.v4.f32 {t5, t6, t7, t8}, [%2];\n\t"
    "ld.global.cg.v4.f32 {t9, t10, t11, t12}, [%3];\n\t"
    "fma.rn.ftz.f32 t1, t5, 0f40000000, t1;\n\t"
    "fma.rn.ftz.f32 t1, t9, 0f40000000, t1;\n\t"
    "fma.rn.ftz.f32 t2, t6, 0f40000000, t2;\n\t"
    "fma.rn.ftz.f32 t2, t10, 0f40000000, t2;\n\t"
    "fma.rn.ftz.f32 t3, t7, 0f40000000, t3;\n\t"
    "fma.rn.ftz.f32 t3, t11, 0f40000000, t3;\n\t"
    "fma.rn.ftz.f32 t4, t8, 0f40000000, t3;\n\t"
    "fma.rn.ftz.f32 t4, t12, 0f40000000, t3;\n\t"
    "st.global.cg.v4.f32 [%0], {t1,t2,t3,t4};\n\t"
    "}" :: "l"(a) , "l"(b), "l"(c), "l"(d));
```

Another challenge on the GPU is the balance between L1 and L2 cache sizes. In total each V100 SM contains 128KB of unified L1 cache, for 10MiB total and 6MiB of L2; which means that measuring the bandwidth of both levels of cache requires more care than with the CPU where the L2 capacity greatly exceeds the L1 capacity. The L1 hit rate was measured as 100% for the kernel for problem sizes smaller than 128KiB per SM, showing that the observed performance is for the bandwidth in L1. Uncovering the L2 cache performance requires some manual adjustments to the caching policy of the load and store operations. This is achieved by manually writing the kernel in inline PTX, and setting the load cache policy to `.cg`, which represents a cache global policy that avoids L1 caching.

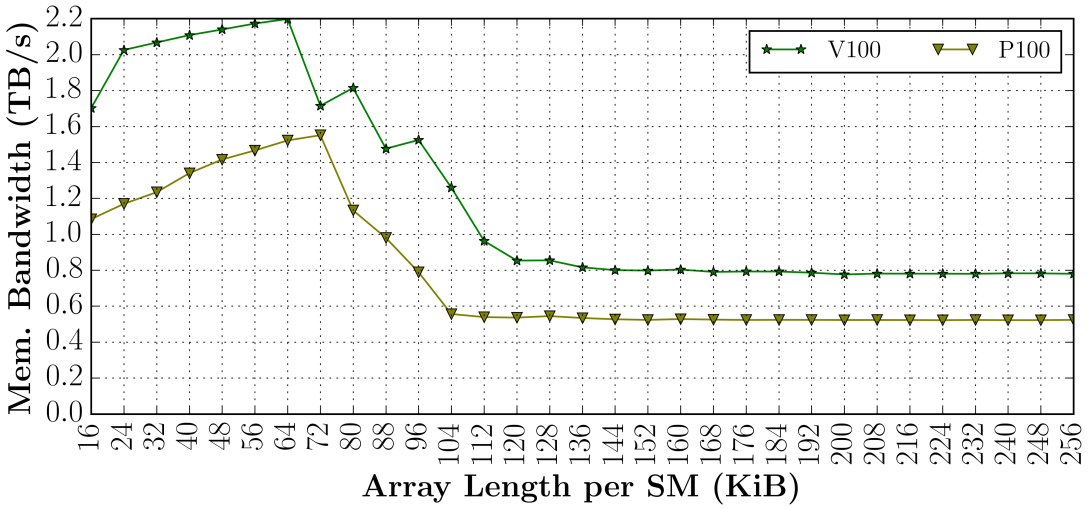

Figure B.4: Bandwidth targeting L2 cache for P100 and V100 GPUs.

Code Sample B.1 shows the cache bandwidth benchmark written for measuring cache bandwidth accurately on the NVIDIA GPUs. Inline assembly is used to support fine grained control of the caching policies for each memory access and inhibit the outer iteration loop from being

optimised away. The included FMA operations is a minimal set required to ensure that the vector loads are performed in full, without being optimised out.

The L2 bandwidth for the P100 is shown to peak at around 1.6TB/s, while the L2 bandwidth peaks at 2.2TB/s for the V100. This difference in performance can be mostly accounted for by the increased number of streaming multiprocessors and different clock frequencies. It can be noted in both Figure B.3 and Figure B.4 that for the largest problem sizes the performance is representative of the maximum attainable memory bandwidth on the V100 GPU, as seen in Section 4.5.

Bibliography

[1] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1:19–25, 2015.

[2] V. Aggarwal, Y. Sabharwal, R. Garg, and P. Heidelberger. HPCC RandomAccess benchmark for next generation supercomputers. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[3] 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.

[4] M. Anderson, M. Brodowicz, T. Sterling, H. Kaiser, and B. Adelstein-Lelbach. Tabulated equations of state with a many-tasking execution model. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1691–1699. IEEE, 2013.

[5] P. Andreo. Monte Carlo techniques in medical radiation physics. *Physics in Medicine & Biology*, 36(7):861, 1991.

[6] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[7] P. Atkinson and S. McIntosh-Smith. On the performance of parallel tasking runtimes for an irregular fast multipole method application. In *International Workshop on OpenMP*, pages 92–106. Springer, 2017.

[8] N. Attig, P. Gibbon, and T. Lippert. Trends in supercomputing: The European path to exascale. *Computer Physics Communications*, 182(9):2041–2046, 2011.

[9] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, et al. Petsc users manual revision 3.8. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[10] L. A. Barba and R. Yokota. How will the fast multipole method fare in the exascale era. *SIAM News*, 46(6):1–3, 2013.

[11] M. Bareford. minEPOCH3D Performance and Load Balancing on Cray XC30.

[12] A. Bertsch. The Sierra Supercomputer: A POWER Collaboration Success Story. OpenPOWER Summit 2018, 2018.

[13] S. Bhowmick, P. Raghavan, L. McInnes, and B. Norris. Faster PDE-based simulations using robust composite linear solvers. *Future Generation Computer Systems*, 20(3):373–387, 2004.

[14] R. F. Bird, P. Gillies, M. Bareford, J. Herdman, and S. A. Jarvis. Mini-App Driven Optimisation of Inertial Confinement Fusion Codes. In *2015 IEEE International Conference on Cluster Computing*, pages 768–776. IEEE, 2015.

[15] B. Bland. Titan-early experience with the titan system at oak ridge national laboratory. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 2189–2211. IEEE, 2012.

[16] O. A. R. Board. OpenMP Application Programming Interface, version 3.0, 2008.

[17] O. A. R. Board. OpenMP Application Program Interface Version 4.0, 2013.

[18] O. A. R. Board. OpenMP Application Program Interface v4.5, 2015.

[19] R. L. Bowers and J. R. Wilson. Numerical Modeling in Applied Physics and Astrophysics. *Boston: Jones and Bartlett, c1991.*, 1, 1991.

[20] P. Brantley, R. Bleile, S. Dawson, N. Gentile, M. McKinley, M. O'Brien, M. Pozulp, D. Richards, D. Stevens, J. Walsh, et al. LLNL monte carlo transport research efforts for advanced computing architectures. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

[21] D. Brown, M. Chadwick, R. Capote, A. Kahler, A. Trkov, M. Herman, A. Sonzogni, Y. Danon, A. Carlson, M. Dunn, et al. ENDF/B-VIII. 0: The 8 th Major Release of the Nuclear Reaction Data Library with CIELO-project Cross Sections, New Standards and Thermal Scattering Data. *Nuclear Data Sheets*, 148:1–142, 2018.

[22] F. B. Brown. New hash-based energy lookup algorithm for Monte Carlo codes. *Trans. Am. Nucl. Soc*, 111(1):659–662, 2014.

[23] F. B. Brown and W. R. Martin. Monte Carlo methods for radiation transport analysis on vector computers. *Progress in Nuclear Energy*, 14(3):269–299, 1984.

[24] T. A. Brunner and P. S. Brantley. An efficient, robust, domain-decomposition algorithm for particle Monte Carlo. *Journal of Computational Physics*, 228(10):3882–3890, 2009.

[25] T. A. Brunner, T. J. Urbatsch, T. M. Evans, and N. A. Gentile. Comparison of four parallel algorithms for domain decomposed implicit Monte Carlo. *Journal of Computational Physics*, 212(2):527–539, 2006.

[26] K. G. Budge and J. S. Peery. Experiences developing ALEGRA: A C++ coupled physics framework. In *Object oriented methods for interoperable scientific and engineering computing, proceedings of the 1998 SIAM workshop*, 1998.

[27] E. Calore, A. Gabbana, J. Kraus, S. F. Schifano, and R. Tripiccione. Performance and portability of accelerated lattice Boltzmann applications with OpenACC. *Concurrency and Computation: Practice and Experience*, 28(12):3485–3502, 2016.

[28] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[29] E. D. Cashwell and C. J. Everett. A practical manual on the Monte Carlo method for random walk problems. 1959.

[30] C. Chauliac, J.-M. Aragonés, D. Bestion, D. G. Cacuci, N. Crouzet, F.-P. Weiss, and M. A. Zimmermann. NURESIM–A European simulation platform for nuclear reactor safety: multi-scale and multi-physics calculations, sensitivity and uncertainty analysis. *Nuclear Engineering and Design*, 241(9):3416–3426, 2011.

[31] N. Corporation. CUDA C Programming Guide Version 9.1, 2018.

[32] M. Curtis-Maury, X. Ding, C. D. Antonopoulos, and D. S. Nikolopoulos. An evaluation of OpenMP on current and emerging multithreaded/multicore processors. In *OpenMP Shared Memory Parallel Programming*, pages 133–144. Springer, 2008.

[33] I. Cutress. The AMD Ryzen Threadripper 1950X and 1920X Review: CPUs on Steroids, 2018.

[34] T. Deakin, S. McIntosh-Smith, M. Martineau, and W. Gaudin. An improved parallelism scheme for deterministic discrete ordinates transport. *The International Journal of High Performance Computing Applications*, page 1094342016668978, 2016.

[35] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. In *International Conference on High Performance Computing*, pages 489–507. Springer, 2016.

[36] T. Deakin, J. Price, and S. McIntosh-Smith. Portable Methods for Measuring Cache Hierarchy Performance. 2017.

[37] J. Demmel and H. D. Nguyen. Numerical reproducibility and accuracy at exascale. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 235–237. IEEE, 2013.

[38] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[39] V. Derk and J. E. Hoogenboom. Efficiency improvement of local power estimation in the general purpose Monte Carlo code MCNP. 2011.

[40] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems*, 23(8):1369–1386, 2012.

[41] J. Dongarra, G. Bosilca, T. Herault, and A. Bouteiller. PARSEC: DISTRIBUTED TASKING FOR EXASCALE, 2018.

[42] J. Dongarra, M. A. Heroux, and P. Luszczek. HPCG benchmark: a new metric for ranking high performance computing systems. *Knoxville, Tennessee*, 2015.

[43] J. Dongarra, S. W. Otto, M. Snir, and D. Walker. An introduction to the MPI standard. *Communications of the ACM*, page 18, 1995.

[44] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[45] S. S. Dosanjh, R. F. Barrett, D. Doerfler, S. D. Hammond, K. S. Hemmert, M. A. Heroux, P. T. Lin, K. T. Pedretti, A. F. Rodrigues, T. Trucano, et al. Exascale design space exploration and co-design. *Future Generation Computer Systems*, 30:46–58, 2014.

[46] A. Dubey, K. Antypas, M. K. Ganapathy, L. B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide. Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10-11):512–522, 2009.

[47] H. C. Edwards, C. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[48] C. R. Ferenbaugh. The PENNANT Mini-App: Unstructured Mesh Hydrodynamics for Advanced Architectures (U). Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2013.

[49] S. Fogerty, M. Martineau, R. Garimella, and R. Robey. A comparative study of multi-material data structures for computational physics applications. *Computers & Mathematics with Applications*, 2018.

[50] D. Foley and J. Danskin. Ultra-performance Pascal GPU and NVLink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[51] R. Garimella, M. Kucharik, and M. Shashkov. An efficient linearity and bound preserving conservative interpolation (remapping) on polyhedral meshes. *Computers & fluids*, 36(2):224–237, 2007.

[52] A. Geist and R. Lucas. Major computer science challenges at exascale. *The International Journal of High Performance Computing Applications*, 23(4):427–436, 2009.

[53] N. Gentile, R. Procassini, and H. Scott. Monte Carlo Particle Transport: Algorithm and Performance Overview. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2005.

[54] X. Gong, R. Ubal, and D. Kaeli. Multi2Sim Kepler: A detailed architectural GPU simulator. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, pages 269–278. IEEE, 2017.

[55] D. Gregory, J.-J. Morcrette, C. Jakob, A. Beljaars, and T. Stockdale. Revision of convection, radiation and cloud schemes in the ECMWF Integrated Forecasting System. *Quarterly Journal of the Royal Meteorological Society*, 126(566):1685–1710, 2000.

[56] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[57] S. D. Hammond. Balancing Productivity Portability and Performance-The Challenge for Programming Models at Exascale?. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[58] K. S. Hemmert, M. Rajan, R. J. Hoekstra, S. L. Dawson, M. L. Vigil, D. L. Grunau, J. L. Lujan, D. L. Morton, H. A. L. Nam, P. Peltz Jr, et al. Trinity: Architecture and Early Experience. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[59] R. Hempel and D. W. Walker. The emergence of the MPI message passing standard for parallel computing. *Computer Standards & Interfaces*, 21(1):51–62, 1999.

[60] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[61] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. Jarvis. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 465–471. IEEE, 2012.

[62] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. A. Jarvis. Accelerating hydrocodes with OpenACC, OpenCL and CUDA. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 465–471. IEEE, 2012.

[63] M. Heroux, D. Doerfler, et al. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[64] J. Hines. Stepping up to Summit. *Computing in Science & Engineering*, 20(2):78–82, 2018.

[65] M. F. Hoemmen. Using Kokkos for Performance Portability of the Tpetra Sparse Linear Algebra Library on Intel KNL and NVIDIA GPUs. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2017.

[66] M. Höhnerbach, A. E. Ismail, and P. Bientinesi. The vectorization of the tersoff multi-body potential: an exercise in performance portability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 7. IEEE Press, 2016.

[67] R. Hornung, J. Keasler, et al. The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory, 2014.

[68] A. Ilic, F. Pratas, and L. Sousa. Cache-aware Roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2014.

[69] Intel. Intel Math Kernel Library, 2018.

[70] H.-T. Janka and W. Hillebrandt. Monte Carlo simulations of neutrino transport in type II supernovae. *Astronomy and Astrophysics Supplement Series*, 78:375–397, 1989.

[71] W. Jeong and J. Seong. Comparison of effects on technical variances of computational fluid dynamics (CFD) software based on finite element and finite volume methods. *International Journal of Mechanical Sciences*, 78:19–26, 2014.

[72] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. apr 2018.

[73] X. Jiao and M. T. Heath. Common-refinement-based data transfer between non-matching meshes in multiphysics simulations. *International Journal for Numerical Methods in Engineering*, 61(14):2402–2427, 2004.

[74] M. Johnson and M. Johnson. *Superscalar microprocessor design*, volume 77. prentice Hall Englewood Cliffs, New Jersey, 1991.

[75] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. White III, and J. Levesque. Practical performance portability in the Parallel Ocean Program (POP). *Concurrency and Computation: Practice and Experience*, 17(10):1317–1327, 2005.

[76] H. Kahn. Applications of Monte Carlo. Technical report, RAND Corp., Santa Monica, Calif., 1954.

[77] S. Kamil, C. Chan, et al. An Auto-tuning Framework for Parallel Multicore Stencil Computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium*, pages 1–12, 2010.

[78] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. *Tech. Rep. LLNL-TR-641973*, 2013.

[79] I. Karlin, J. McGraw, E. Gallarado, J. Keasler, E. Leon, and B. Still. Memory and parallelism tuning exploration using the LULESH proxy application. *2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC 2012)*, 2012.

[80] I. Karlin, T. Scogland, A. C. Jacob, S. F. Antao, G.-T. Bercea, C. Bertolli, B. R. de Supinski, E. W. Draeger, A. E. Eichenberger, J. Glosli, et al. Early Experiences Porting Three Applications to OpenMP 4.5. In *International Workshop on OpenMP*, pages 281–292. Springer, 2016.

[81] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis. Achieving Performance Portability for a Heat Conduction Solver Mini-Application on Modern Multi-core Systems. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 834–841. IEEE, 2017.

[82] P. Kogge and J. Shalf. Exascale Computing Trends: Adjusting to the "New Normal" for Computer Architecture. *Computing in Science & Engineering*, 15(6):16–26, 2013.

[83] A. Koniges, N. Masters, A. Fisher, R. Anderson, D. Eder, T. Kaiser, D. Bailey, B. Gunney, P. Wang, B. Brown, et al. Ale-amr: A new 3d multi-physics code for modeling laser/-target effects. In *Journal of Physics: Conference Series*, volume 244, page 032019. IOP Publishing, 2010.

[84] C. Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40, 2013.

[85] B. N. Lawrence, M. Rezny, R. G. Budich, P. Bauer, J. Behrens, M. Carter, W. Deconinck, R. Ford, C. Maynard, S. Mullerworth, et al. Crossing the chasm: how to develop weather and climate models for next generation computers? *Geoscientific Model Development*, 11:1799–1821, 2018.

[86] R. J. LeVeque. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

[87] R. J. LeVeque and H. C. Yee. A study of numerical methods for hyperbolic conservation laws with stiff source terms. *Journal of computational physics*, 86(1):187–210, 1990.

[88] E. Lewis and W. Miller. *Computational methods of neutron transport*. John Wiley and Sons, Inc.,New York, NY, Jan 1984.

[89] P. Lin, C. Liao, D. Quinlan, et al. Experiences of Using The OpenMP Accelerator Model to Port DOE Stencil Applications. In *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Proceedings*, pages 45–59, 2015.

[90] T. Liu, X. Du, W. Ji, X. G. Xu, and F. B. Brown. A comparative study of history-based versus vectorized Monte Carlo methods in the GPU/CUDA environment for a simple neutron eigenvalue problem. In *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*, page 04206. EDP Sciences, 2014.

[91] L. L. N. L. (LLNL). Veritas: Validating Proxy Apps. https://computation.llnl.gov/projects/veritas, 2019.

[92] A. Long. Branson: A Mini-App for Studying Parallel IMC, Version 1.0. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2017.

[93] A. R. Long. A cache size based timestep limiter for Implicit Monte Carlo. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2018.

[94] M. G. Lopez, V. V. Larrea, W. Joubert, O. Hernandez, A. Haidar, S. Tomov, and J. Dongarra. Towards achieving performance portability using directives for accelerators. In *Proceedings of the Third International Workshop on Accelerator Programming Using Directives, WACCPD*, volume 162016, 2016.

[95] E. A. Lufkin and J. F. Hawley. The piecewise-linear predictor-corrector code-A Lagrangian-remap method for astrophysical flows. *The Astrophysical Journal Supplement Series*, 88:569–588, 1993.

[96] I. Lux and L. Koblinger. *Monte Carlo particle transport methods: neutron and photon calculations*, volume 102. Citeseer, 1991.

[97] M. Martineau and S. McIntosh-Smith. The Arch Project: Physics Mini-Apps for Algorithmic Exploration and Evaluating Programming Environments on HPC Architectures. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 850–857, Sept 2017.

[98] M. Martineau and S. McIntosh-Smith. The Productivity, Portability and Performance of OpenMP 4.5 for Scientific Applications Targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs. In *Proceedings of the 13th International Workshop on OpenMP (IWOMP)*, 2017.

[99] M. Martineau, S. McIntosh-Smith, C. Bertolli, A. C. Jacob, S. F. Antao, A. Eichenberger, G.-T. Bercea, T. Chen, T. Jin, K. O'Brien, et al. Performance analysis and optimization of Clang's OpenMP 4.5 GPU support. In *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), International Workshop on*, pages 54–64. IEEE, 2016.

[100] 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*, PMAM'16, 2016.

[101] M. Martineau, S. McIntosh-Smith, M. Boulton, W. Gaudin, and D. Beckingsale. A performance evaluation of Kokkos & RAJA using the TeaLeaf mini-app. In *The International Conference for High Performance Computing, Networking, Storage and Analysis, SC15*, 2015.

[102] M. Martineau, S. McIntosh-Smith, and W. Gaudin. Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model. In *Proceedings of 21st International Workship on High-Level Parallel Programming Models and Supportive Environments*, HIPS'16, 2016.

[103] M. Martineau, S. McIntosh-Smith, and W. Gaudin. Assessing the performance portability of modern parallel programming models using TeaLeaf. *Concurrency and Computation: Practice and Experience*, 29(15):e4117, 2017.

[104] M. Martineau, J. Price, S. McIntosh-Smith, and W. Gaudin. Pragmatic performance portability with OpenMP 4.x. In *International Workshop on OpenMP*, pages 253–267. Springer, 2016.

[105] T. G. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Pearson Education, 2004.

[106] J. D. McCalpin. STREAM benchmark. *Link: www.cs.virginia.edu/stream/ref.html*, 22, 1995.

[107] J. D. McCalpin. Trends in system cost and performance balances and implications for the future of HPC. In *Proceedings of the 2nd International Workshop on Hardware-Software Co-Design for High Performance Computing*, page 2. ACM, 2015.

[108] J. D. McCalpin. Memory bandwidth and system balance in HPC systems. In *Invited Talk, International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2016.

[109] J. D. McCalpin. Memory Latency on the Intel Xeon Phi x200 Knights Landing processor, 2016.

[110] J. D. McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19–25), 1995.

[111] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price. On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures. In *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 53–75. Springer International Publishing, 2014.

[112] S. McIntosh-Smith, M. Martineau, T. Deakin, G. Pawelczak, W. Gaudin, P. Garrett, W. Liu, R. Smedley-Stevenson, and D. Beckingsale. TeaLeaf: a mini-application to enable design-space explorations for iterative sparse linear solvers. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 842–849. IEEE, 2017.

[113] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[114] P. Messina. The exascale computing project. *Computing in Science & Engineering*, 19(3):63–67, 2017.

[115] J. Michalakes, S. Chen, J. Dudhia, L. Hart, J. Klemp, J. Middlecoff, and W. Skamarock. Development of a next-generation regional weather research and forecast model. In *Developments in Teracomputing*, pages 269–276. World Scientific, 2001.

[116] A. Mishra, L. Li, M. Kong, H. Finkel, and B. Chapman. Benchmarking and Evaluating Unified Memory for OpenMP GPU Offloading. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, page 6. ACM, 2017.

[117] D. Molka, D. Hackenberg, and R. Schöne. Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, page 4. ACM, 2014.

[118] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[119] J. W. Murphy and A. Burrows. BETHE-HYDRO: An arbitrary Lagrangian-Eulerian multidimensional hydrodynamics code for astrophysical simulations. *The Astrophysical Journal Supplement Series*, 179(1):209, 2008.

[120] J. R. Neely and B. R. de Supinski. Application Modernization at LLNL and the Sierra Center of Excellence. *Computing in Science & Engineering*, 19(5):9–18, 2017.

[121] J. Nocedal and S. J. Wright. Conjugate gradient methods. *Numerical optimization*, pages 101–134, 2006.

[122] NVIDIA. cuBLAS library. *NVIDIA Corporation, Santa Clara, California*, 2008.

[123] NVIDIA. NVIDIA TESLA V100 GPU ARCHITECTURE: The World's Most Advanced Data Center GPU, 2017.

[124] OpenACC. OpenACC December 2017 Highlights. https://www.openacc.org/news/openacc-december-2017-highlights, 2017.

[125] OpenACC-standard.org. The OpenACC Application Programming Interface, version 2.6, 2017.

[126] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[127] M. E. ONeill. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Transactions on Mathematical Software*, 2014.

[128] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis. Exploring the performance benefit of hybrid memory system on HPC environments. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 683–692. IEEE, 2017.

[129] 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.

[130] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring simd for molecular dynamics, using intel® xeon® processors and intel® xeon phi coprocessors. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1085–1097. IEEE, 2013.

[131] S. J. Pennycook, J. D. Sewall, and V. Lee. A metric for performance portability. *arXiv preprint arXiv:1611.07409*, 2016.

[132] R. Procassini, M. O'Brien, and J. Taylor. Load balancing of parallel Monte Carlo transport calculations. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2005.

[133] R. Procassini, M. OBrien, and J. Taylor. Load balancing of parallel Monte Carlo transport calculations. *Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications, Palais des Papes, Avignon, Fra*, 2005.

[134] 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.

[135] A. P. Rendell, B. Ch Apm An, and M. S. Müller. OpenMP in the Era of Low Power Devices and Accelerators. In *9th International Workshop on OpenMP, IWOMP*. Springer, 2013.

[136] L. Richardson. Weather Prediction by Numerical Process. 1922.

[137] F. J. Rogers, F. J. Swenson, and C. A. Iglesias. OPAL equation-of-state tables for astrophysical applications. *The Astrophysical Journal*, 456:902, 1996.

[138] P. K. Romano. *Parallel algorithms for Monte Carlo particle transport simulation on exascale computing architectures*. PhD thesis, Massachusetts Institute of Technology, 2013.

[139] P. K. Romano, F. B. Brown, and B. Forget. Towards scalable parellelism in Monte Carlo particle transport codes using remote memory access. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2010.

[140] P. K. Romano and B. Forget. The OpenMC monte carlo particle transport code. *Annals of Nuclear Energy*, 51:274–281, 2013.

[141] P. K. Romano and B. Forget. The OpenMC Monte Carlo particle transport code. *Annals of Nuclear Energy*, 51:274 – 281, 2013.

[142] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[143] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[144] V. Saxena, Y. Sabharwal, and P. Bhatotia. Performance evaluation and optimization of random memory access on multicores with high productivity. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[145] V. Seker, J. W. Thomas, and T. J. Downar. Reactor simulation with coupled Monte Carlo and computational fluid dynamics. In *Proceedings of the Joint International Topical Meeting on Mathematics and Computations and Supercomputing in Nuclear Applications*. Citeseer, 2007.

[146] J. R. Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[147] L. Shulenburger. Preparing QMCPACK for the exascale. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[148] A. Sidelnik, S. Maleki, et al. Performance portability with the Chapel language. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 582–594. IEEE, 2012.

[149] A. R. Siegel, K. Smith, P. K. Romano, B. Forget, and K. G. Felker. Multi-core performance studies of a Monte Carlo neutron transport code. *The International Journal of High Performance Computing Applications*, 28(1):87–96, 2014.

[150] F. Silla, J. Prades, and C. Reaño. Leveraging rCUDA for Enhancing Low-Power Deployments in the Physics Domain. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, page 17. ACM, 2018.

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

[152] C. Staelin. lmbench: an extensible micro-benchmark suite. *Software: Practice and Experience*, 35(11):1079–1105, 2005.

[153] T. N. Team. TITAN: OAK RIDGE NATIONAL LABORATORY (No. 1 system in November 2012), 2012.

[154] T. N. Team. US Regains TOP500 Crown with Summit Supercomputer, Sierra Grabs Number Three Spot, 2018.

[155] X.-. M. C. Team. MCNPA General Monte Carlo N-Particle Transport Code, Version 5, 2003.

[156] J. M. Tendler, J. S. Dodson, J. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[157] J. R. Tramm and A. R. Siegel. Memory bottlenecks and memory contention in multi-core Monte Carlo transport codes. In *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*, page 04208. EDP Sciences, 2014.

[158] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[159] C. Trott, S. Hammond, D. Dinge, P. Lin, C. Vaughan, J. Cook, H. Edwards, M. Rajan, and R. Hoekstra. ASC Trilab L2 Codesign Milestone 2015–Sandia. Technical report, Technical Report SAND2015-7886, Sandia National Laboratories, 2015. 2.

[160] C. R. Trott. Early Experience with P100 on POWER8. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[161] E. Troubetzkoy, H. Steinberg, and M. Kalos. Monte Carlo radiation penetration calculations on a parallel computer. *Trans. Amer. Nucl. Soc*, 17, 1973.

[162] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 392–403. ACM, 1995.

[163] T. J. Urbatsch and T. M. Evans. Milagro version 2 an implicit Monte Carlo code for thermal radiative transfer: capabilities, development, and usage. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM, 2006.

[164] F. A. van Heerden. A coarse grained particle transport solver designed specifically for graphics processing units. *Transport Theory and Statistical Physics*, 41(1-2):80–100, 2012.

[165] B. Van Leer. Upwind and high-resolution methods for compressible flow: From donor cell to residual-distribution schemes. In *16th AIAA Computational Fluid Dynamics Conference*, page 3559, 2006.

[166] C. T. Vaughan, D. Dinge, P. Lin, S. D. Hammond, J. Cook, C. R. Trott, A. M. Agelastos, D. M. Pase, R. E. Benner, M. Rajan, et al. Early Experiences with Trinity-The First Advanced Technology Platform for the ASC Program. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[167] V. Volkov. *Understanding latency hiding on gpus.* PhD thesis, UC Berkeley, 2016.

[168] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[169] J. Wells. What does Titan tell us about preparing for exascale supercomputers? In *Programme du 32eme Forum ORAP, Maison de la Simulation, Saclay, France*, 2013.

[170] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACCfirst experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[171] S. Wienke, C. Terboven, J. C. Beyer, and M. Müller. A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing. In *Euro-Par 2014 Parallel Processing*, pages 812–823. Springer, 2014.

[172] C. Williams. Heads up: Fujitsu tips its hand to reveal exascale Arm supercomputer processor the A64FX. https://www.theregister.co.uk/2018/08/22/fujitsu_post_k_a64fx/, 2018.

[173] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[174] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664. ACM, 1989.