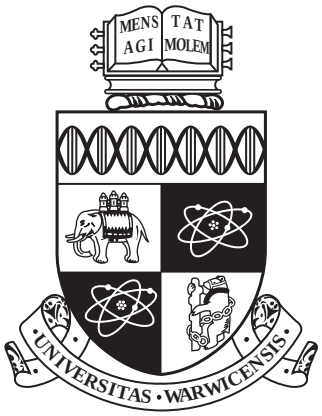
Student ID: 1218779

This student has been formally diagnosed with Specific Learning Differences. Please make appropriate allowance when marking. Guidance is available at: **http://www2.warwick.ac.uk/services/tutors/disability/guidance**

Disability Services



Data Structure Abstraction and Parallelisation of Multi-Material Hydrodynamic Applications

by

Richard Oliver Kirk

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy in Computer Science

Department of Computer Science

The University of Warwick

September 2020

Copyright

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

I would like to thank Prof. Steven Jarvis for giving me the opportunity to do this PhD. I would also like to thank both him and Dr. Gihan Mudalige for their supervision and support throughout the last four years. They have influenced me greatly, and without them this work would not have been what it is today.

I would also like to thank my colleagues and friends in the High Performance Scientific Computing Group: Dean Chester, Andrew Lamzed-Short, Alex Cooper, Dr. Andrew Owenson and David Truby. Without you, the days spent trying to get code to work would have been twice as long and only half as interesting. I would like to thank Dr. Dominic Brown in particular. Having known him for most of my student life, both as an undergraduate and as a PhD student, I can say there is nobody I would have rather worked alongside. I would also like to thank all of the lab supervisors: Dr. Steven Wright, Dr. James Davis and Dr. Timothy Law. The assistance and time they gave me was invaluable and helped me to work more efficiently and effectively.

From the Department of Computer Science, I would like to thank Dr. Claire Rocks, Prof. Jane Sinclair, Sharon Howard and Dr. Arshad Jhumka for all their guidance and support throughout the years. I would also like to thank Dr. Liam Steadman, James Van Hinsbergh, Helen McKay, Dr. Matthew Bradbury and Dr. David Purser for listening to me rant on about why my code wasn't working on any given day.

Last, but by no means least, I would like to thank all my friends from the University of Warwick and from Sittingbourne, for making the bad days brighter, and the good days excellent. I would like to thank my family: Mum, Dad, Liz, Nan, Grandad and Jean. Without your love and support, I wouldn't have been able to do any of this. Finally, I would also like to thank my partner Kirstie for being able to put up with me, even at my most annoying.

Declarations

This thesis is submitted to the University of Warwick in support of my application for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work presented (including data generated and data analysis) was carried out by the author except in the cases outlined below:

- Figures 7.4 and 7.5 in Chapter 7 were created by Dr. Timothy Law.

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

[49] R. O. Kirk, T. R. Law, S. Maheswaran, and S. A. Jarvis. Warwick Data Store: A HPC Library for Flexible Data Storage in Multi-Physics Applications. In *Super Computing 19 (SC19)*, Denver, CO, November 2019. Association for Computing Machinery, New York, NY

[50] 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 *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 834–841, Honolulu, HI, September 2017. IEEE Computer Society, Los Alamitos, CA

[51] R. O. Kirk, M. Nolten, R. Kevis, T. R. Law, S. Maheswaran, S. A. Wright, S. Powell, G. R. Mudalige, and S. A. Jarvis. Warwick Data Store: A Data Structure Abstraction Library. In *11th IEEE International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS20)*, pages 71–85, Atlanta, GA, November 2020. IEEE Computer Society, Los Alamitos, CA

Each author contributed to the aforementioned publications in the following ways:

- *Prof. Stephen Jarvis* was the lead supervisor for the project.
- *Dr. Gihan Mudalige* was the secondary supervisor for the project, taking over from *Prof. Stephen Jarvis* as the lead supervisor when *Prof. Stephen Jarvis* left the University of Warwick. He also developed and provided insight into Oxford Parallel Library for Structured mesh solvers (OPS) discussed in Chapter 4, alongside *Dr. Istvan Reguly*.
- *Dr. Steven Wright* helped oversee the research, and assisted in checking the work.
- *Dr. Timothy Law* helped oversee the research, and was a point of contact with the sponsors AWE plc.
- *Dr. Satheesh Maheswaran* was the lead contact with the sponsors AWE plc.
- *Dr. Martin Nolten*, *Dr. Robert Kevis* and *Dr. Seimon Powell* were key contacts at the sponsors AWE plc.
- *Dr. Matt Martineau* was a key contact for information regarding TeaLeaf, a mini-application referred to in Chapters 4 and 6.

Abstract

The aim for High Performance Computing (HPC) is to achieve the best performance for an application, in order to execute it as quickly as possible. This is often achieved through iterative improvements in Central Processing Unit (CPU) technology such as: including more circuitry by shrinking or making processors larger; making the processor run faster by increasing the clock speed; or increasing the amount of parallelism. Recently, there has been increasing diversity in how HPC systems achieve these performance improvements. The use of Graphics Processing Unit (GPU) processors has become more common, and there has been a growing interest in high bandwidth memory. This has lead to a need for performance portable code, so programs may be written once but compiled and ran on a range of differing systems, with minimal impact on the performance.

As memory becomes a major focus, so too should the data structure used by an application. Without a well designed data structure, the performance of a program can be affected. However, it is key that this is done in a performance portable way, where the data structure can be altered and optimised without the need for the application to be rewritten. As such, a data structure abstraction library was developed, called Warwick Data Store (WDS). This library is able to provide objects, which allow for access to data, without the application needing to know the detail of the data structure. The library also provides additional functionality that would otherwise be difficult and time consuming to implement, such as the ability to convert a variable or a collection of variables from one data structure to another. The performance impact of the library is shown to be minimal, especially in larger problem sizes. Because of the flexibility of the library, data structures for specialised cases can be implemented into WDS without impacting the performance of other data structures. The performance of these specialised data structures is also presented as being minimal.

Abbreviations

AI	Artificial Intelligence
ALE	Arbitrary Lagrangian-Eulerian
AoS	Array of Structures
AoSoA	Array of Structures of Arrays
AVX	Advanced Vector Extensions
BW	Bandwidth
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DoE	United States Department of Energy
DSL	Domain Specific Language
EAM	Embedded Atom Model
ECMWF	European Centre for Medium-Range Weather Forecasts
ECP	Exascale Computing Project
EOS	Equation of State
FLOP/s	Floating Point Operations per Second
GB/s	Gigabytes (10^9 bytes) per Second
GFLOP/s	Giga (10^9) Floating Point Operations per Second

GPU	Graphics Processing Unit
HDD	Hard Disk Drive
HPC	High Performance Computing
JIT	Just-In-Time
KNL	Intel's Xeon Phi Knights Landing
L1	Level 1 Cache Memory
L2	Level 2 Cache Memory
L3	Level 3 Cache Memory
LANL	Los Alamos National Laboratory
LLNL	Lawrence Livermore National Laboratory
MB/s	Megabytes (10^6 bytes) per Second
MCDRAM	Multi-Channel Dynamic Random Access Memory
MD	Molecular Dynamics
MFLOP/s	Mega (10^6) Floating Point Operations per Second
MIMD	Multiple Instruction - Multiple Data
MISD	Multiple Instruction - Single Data
ML	Machine Learning
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
OP2	Oxford Parallel Library for Unstructured mesh solvers
OPS	Oxford Parallel Library for Structured mesh solvers
OS	Operating System
PARSEC	Princeton Application Repository for Shared-Memory Computers
PPCG	Polynomially Preconditioned Conjugate Gradient (CG)

SDLT	Intel's Single Instruction - Multiple Data (SIMD) Data Layout Templates
SIMD	Single Instruction - Multiple Data
SISD	Single Instruction - Single Data
SNL	Sandia National Laboratory
SoA	Structure of Arrays
SSD	Solid State Drive
SSE	Streaming SIMD Extension
UKMAC	UK Mini-App Consortium
WDS	Warwick Data Store

Sponsorship and Grants

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

- UK Atomic Weapons Establishment (AWE plc):
  "AWE Technical Outreach Programme" (CDK0724, 2016 - 2020)

CHAPTER 1

Introduction

One of biggest hurdles within High Performance Computing (HPC) is the exascale barrier, which refers to the challenge of creating a supercomputer capable of performing 10^{18} Floating Point Operations per Second (FLOP/s), otherwise known as a exaFLOP/s. Exceeding this has been the goal since the petascale (10^{15} FLOP/s) barrier was broken by the Roadrunner system at Los Alamos National Laboratory (LANL) in 2008. [33] In order to pass the petascale barrier, faster Central Processing Unit (CPU) processors with high levels of parallelism were connected together through large networks, designed to act as one large system. Such techniques have allowed HPC systems to get closer to an exascale system. However, improvements through these mechanisms is no longer sustainable. The lower limit of transistor sizes is being reached. Increasing the clock speed is not sustainable as the power required would generate a large amount of heat that might be difficult to dissipate. [36] Thus, to continue to improve performance, the inherent nature of parallelism within many scientific algorithms needs to be exploited.

Whilst the increasing performance of CPUs has contributed towards achieving exascale, more radical ways have been explored to increase the performance. The most prevalent idea has been the use of accelerators such as Graphics Processing Unit (GPU) processors. These have shown that specialised hardware can also greatly improve the performance of a system, especially when used alongside fast CPUs. In fact, over the last decade, the number of systems with accelerators has grown dramatically. Figure 1.1 shows this trend of the largest supercomputers with accelerators over the last decade, as measured by the Top500 [113].

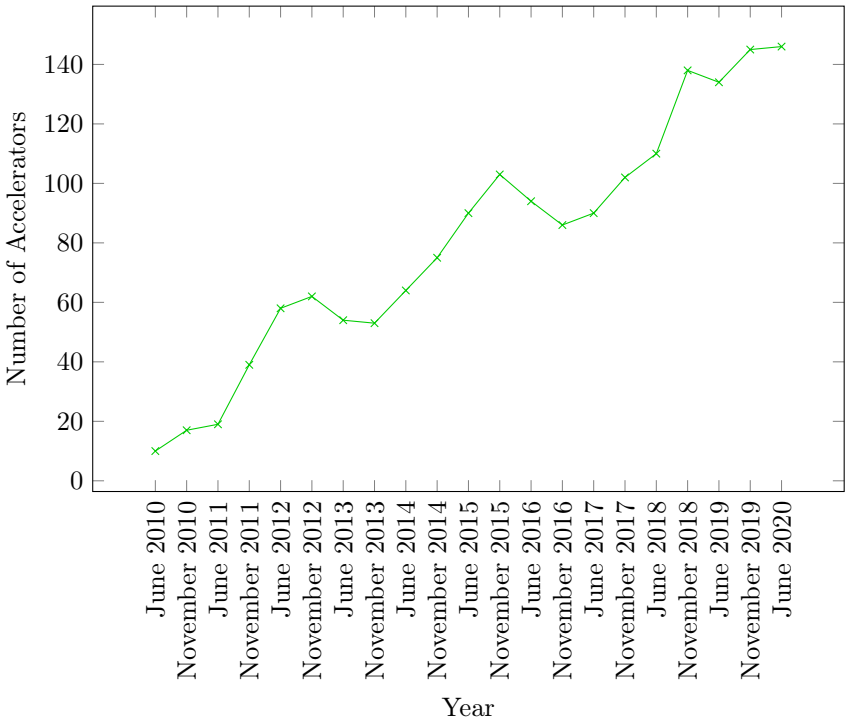


Figure 1.1: Trend of accelerators for the Top500 supercomputers [113] over the last decade

Even with improvements to CPUs and the use of accelerators, HPC systems have not yet broken the exascale barrier. One of the key innovations in the last few years is the development of hardware with higher bandwidths, which allow for more data to be passed between the processor and the systems memory. In fact, the current fastest machine in the world, Fugaku, has achieved 0.4155 exaflops through the use of ARM Fujitsu A64FX [25] processors which contain high bandwidth memory. [114] The use of high bandwidth memory can improve the performance of both CPUs and GPUs for many real-world applications.

Through the use of diverse hardware and new techniques such as high bandwidth memory, the exascale barrier will be broken soon. Whilst some are looking beyond exascale already [48, 85], there will be a massive undertaking to ensure applications can maximise their performance on an ever-changing HPC landscape.

1.1 Motivation

In recent years, the diversity and heterogeneity of systems within HPC has dramatically increased. From high bandwidth memory being implemented into GPUs (such as NVIDIA A100 [78]) and CPUs (such as ARM Marvell Thunder X2 [65] and ARM Fujitsu A64FX [25]), to more novel CPUs (for example, Intel's Xeon Phi Knights Landing (KNL) [103]), there is no longer a single approach to achieving better performance. This thesis explores performance portability through the utilisation of modern HPC hardware and software. In doing so, differences in both performance and performance portability can be examined across a range of hardware, using a memory-bound heat conduction proxy application as the base program. To effectively measure the performance portability, both CPUs and GPUs need to be examined.

One of the key areas of development, especially with regard to newer CPU design, is the growing importance attached to the performance of memory. This has led to an increased emphasis on the structure of the data within an application. This thesis demonstrates the idea of a data structure abstraction library with a priority on minimising the impact on the performance of a program. By implementing this approach, the data structure can be optimised to cater for different architectures and applications without additional developer input. Thus, the data structure and memory can become performance portable. Another benefit of using such a library is the ability to change the data structure between sections of code, in order to allow potential optimisations through the rearrangement of data. Listing 1.1 shows a C++ styled pseudo-code example of how this should look. This example demonstrates that the kernel(s) do not know, or need to know, that the data structure has been altered.

The creation, development, implementation and optimisation of the library is presented in detail, along with details of the library's impact on performance. The performance study of the library is tested against a collection of benchmark kernels and proxy applications, across a wide range of CPUs, each utilising mul-

```
//Get the required data in a view-like object
auto a = library.getView("a");
auto b = library.getView("b");

//Use the data to perform some calculations
kernel(a, b);

//Alter the data structure of the required data
library.alter("a", "b");

//Use the newly rearranged data to perform more calculations
kernel2(a, b);
```

Listing 1.1: Psuedo-code example of what is expected from the data structure abstraction library.

tiple compilers. The development and performance of specialised data structures within the library is also presented. In order to simplify the problem, and to show that there is a need for libraries such as the one presented in this thesis, the library solely focuses on a wide range of different CPU processors, each with varying designs and memory architectures. Potential future expansions are discussed, including the possibility of extending the library to work natively with GPUs

1.2 Thesis Contributions

The research presented in this thesis makes the following contributions:

- Measure the performance of a heat-conduction proxy application utilising different performance portable libraries, as well as manually parallelised versions, across multiple different architectures. Alongside the application efficiency, the architectural efficiency is also measured in order to gain a more complete picture of the performance portability of each library, across the given architectures.
- Development and implementation of a data structure abstraction library, Warwick Data Store (WDS). The library should be flexible enough that

additional data structures and features can be added with ease, as small as possible, extensible and have a minimal impact on an application's performance as possible. Alongside this, show that the performance impact of WDS is minimal across many different kernels, proxy applications and when used at scale.

- Demonstrate how specialised data structures can be implemented into WDS, using multi-material data structures as a basis. The performance impact of the library on different multi-material kernels is shown to be minimal, and altering the data structure can improve the performance of particular kernels.

1.3 Thesis Overview

The remainder of the thesis is structured as follows:

- **Chapter 2** explores how performance of applications and hardware can be measured through different techniques. It also looks into how the changes in performance can be quantified, and why this is necessary.

- **Chapter 3** discusses how increases in performance has been achieved, whether through the use of smaller components in processors, vectorisation of computation, paralleisation of processors through the use of multithreading, multiprocessing or distributed computing. Alongside this, the importance of memory is discussed, as well as the way in which it is utilised. In particular, different data structures are explored, including there benefits and drawbacks.

- **Chapter 4** presents and analyses the performance of a heat-conduction mini-application across a multitude of different parallelisation models using three different hardware architectures. After this, the efficiency of each model on each architecture is then measured to analyse the performance portability of these models.

- **Chapter 5** introduces a data structure abstraction library, WDS. Details on the development and the data structures are presented, as well as the additional functionality the library can provide through the use of data structure abstraction.

- **Chapter 6** explores the performance of the data structure abstraction library WDS. To do this, the library is tested using different benchmark kernels and a variety of mini-applications. The scaling performance is also measured and presented.

- **Chapter 7** discusses how WDS can use specialised data structures for given applications. The data structures are presented, along with the implementation details and the performance utilising different kernels and meshes.

- **Chapter 8** concludes the thesis by exploring the limitations of the work, the implications of the research, and where future work could be done within the area.

CHAPTER 2

Analyse and Performance of Applications and Architectures

The main objective of High Performance Computing (HPC) is to extract the most performance from a given application. This can be achieved through two methodologies, redesign the software to maximise the utilisation of a given system, or improve the system so that the application can execute faster. In either case, without quantification of the improvements gained, it is impossible to know how much of a gain has been achieved, and how much could theoretically be possible. Thus, reliable methods are required to measure the performance of both hardware and software, and tools are needed to analyse where further improvements could be gained.

In this chapter, some of these different techniques are discussed:

- Sections 2.1 and 2.2 show how different techniques can be used to measure the performance of hardware, in a way in which different aspects of a system can be compared.
- Section 2.3 discusses how software can be analysed in order to see where performance improvements could be made, both from a software and from a hardware utilisation perspective.
- Sections 2.4 and 2.5 describe two different formulae for measuring the performance improvements of an application.
- Section 2.6 outlines how the affect of adding a library affects the performance of an application.
- Section 2.7 describes the importance of performance portability, the principle of a single code base running on multiple systems with a high level

of performance. A formulae is also presented, allowing for multiple performance portable libraries to be compared.

2.1 Benchmarking

Benchmarking is the use of kernels to analyse the performance of a given aspect of a system. By doing this, it is possible to quantify performance and make comparisons with other systems, allowing for better informed decisions when procuring a large system. The use of benchmarking also enhances the predictability of how an application may perform on a given architecture. For example, say an algorithm is heavily compute bound (i.e. the speed of the algorithm is limited by how fast the computation can be done), then a benchmark which can measure the amount of computation that can be done in a given time frame, (usually in Mega (10^6) Floating Point Operations per Second (MFLOP/s) or Giga (10^9) Floating Point Operations per Second (GFLOP/s)), is useful. Whereas, for a memory bound problem (i.e. the speed of the algorithm is limited by how fast data can be retrieved from memory), a benchmark which can measure the memory bandwidth (usually in Megabytes (10^6 bytes) per Second (MB/s) or Gigabytes (10^9 bytes) per Second (GB/s)) is more useful. Examples of these include Livermore Loops (also known as Livermore FORTRAN Kernels) [20], the LINPACK Benchmark (which is used to categorise the most powerful machines in the world) [17, 115], and the STREAM benchmark [67].

Because each benchmarking software measures a different aspect of the machine, it is very common to use multiple benchmarks to compare machines, especially for procurement. Examples of these include the NAS Parallel Benchmarks [5, 72] developed by NASA, Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmarking suite [10] developed by Christian Beinia from Princeton University, and the ACS benchmarking suite [26] developed by Los Alamos National Laboratory (LANL) and Lawrence Livermore National Laboratory (LLNL).

However, a major issue with benchmarks is that while they provide a good indication of the hardware's overall performance, they are often quoted as optimal numbers, and as such they may not properly convey the complexity of a real-world application due to their simplicity. Therefore, when reporting benchmarks, it is good practice to: use a geometric or harmonic mean; clearly state confidence intervals; and specify the parameters used. [12, 21].

2.2 Representative Applications

As discussed in the previous section, whilst benchmarks are useful when comparing systems, they may not give an accurate picture of how a real-world application may perform. One way to get around this is to actually use the real-world application on a testbed system. However, this is not feasible due to their size, complexity, and the fact that they often contain sensitive code, commercially or otherwise. Therefore, there is a need for a representation of the production real-world application. These codes are often referred to as proxy applications, mini-applications, or mini-apps.

Mini-apps are much smaller than their production counterparts, consisting of key computational kernels and enough code to load a given state. There are many benefits to mini-apps, such as the fact that it is easy to implement parallelisation methodologies and optimisations. Additionally, they do not contain commercially sensitive information, and therefore can be shared with multiple parties. [31]

When used in the analysis of systems, proxy applications are grouped together to form suites. Proxy application suites can be used alongside benchmarking suites to gain a better picture of the system performance. These suites include the Mantevo suite [60], the UK Mini-App Consortium (UKMAC) suite [118], and the newly formed Exascale Computing Project (ECP) proxy app suite [19]. ECP also manage a large catalogue of mini-applications, together with their location.

2.3 Profiling

Profiling is defined as the examination of a program to better understand the operations performed. The aim of profiling is to find performance issues and potential bottlenecks. In this examination, the software being used to inspect and application, (often referred to as the profiler), can measure a large range of different factors, depending on what is required. These factors can include: the instructions called by each processing unit; the amount of memory used; the length of time spent in a function; and the number of calls to different hardware elements such as: memory, hard drives, and network cards. Depending on the data collected, the profiler may collect data at regular intervals (such as the usage of memory), or may look for given signals built into the program at compile time (such as when the program moves into or out of a function).

Due to the scope of profilers, they can identify many different issues with an application. These issues may have come from poorly written code, the compiler incorrectly generating code or generating code that is not performant, or utilising hardware incorrectly. As such, there are many uses for profilers, including: understanding why a large number of unnecessary instructions are being used (i.e. the run-time of a given function is higher than expected); large amounts of data are being written to the hard disk or to memory; or communication with another machine is taking longer than expected. By identifying these issues, an application developer can take steps to alter the code in order to maximise machine utilisation and program performance, or find issues that would have been difficult to detect otherwise.

Profiling tools have been built into UNIX since 1974, with the use of the `prof`, `profil` and `monitor` commands. These commands allowed for the kernel to sample the program counter and record the execution status of the program. This data was then stored in a separate file, which could later be read and interpreted by a separate program. [47] Later, improvements were realised through the `gprof` command, which generated a call graph of the given application.

The introduction of `gprof` allowed for an amelioration in the analysis of function runtimes, and made it easier for the developer to read and understand the software program. [29]

More detail can be obtained by using instrumentation to profile an application. This involves integrating additional information into the program, either through compilation or implementation into the code. In this way, a more accurate profile can be achieved, and can be used to debug the application. However, only functions which run will be tested (compared to analytical profiling which inspects all possible branching paths for a program) The drawback to instumented profiling is that it can increase the time it takes to run the code, and requires the program to be modified in order to include specialised flags the profiler can use. Examples of this include ATOM [104] and Caliper [11].

The profilers discussed thus far measure different aspects of an application, but don't measure the hardware these programs are running on. As such, these profilers provide a good indication on where potential optimisations could be made, but cannot tell why the application might be running slower than is possible though better utilisation of the system. To overcome this issue, a profiler with the ability to measure the hardware usage is required. For memory, hard drives, and network interfaces, the profiler would have to poll the usages for a given frequency. However, for most processors, profiling the hardware can be achieved through the monitoring of processor counters. These profilers are used to determine how many times a given operation has been carried out. This can include the movement of data to and from the processor's cache and how many times a request has been made to a hard drive. Due to the nature of these profilers, they are often more specialised, and developed for a particular architecture. Examples include Intel VTune [40], Nvprof [79] Arm Forge [3], and PAPI [107]. Other hardware aspects can be profiled more explicitly, such as the memory through the use of Valgrind [121], and I/O interactions through the use of RIOT [127].

2.4 Speedup

When optimising a program or making it run in parallel, it is useful to quantify the effect of these changes. The most common way to create this is to measure the new time, and compare this to the original, creating a ratio. This method is often used to show how an application scales when parallelised. Equation 2.1 shows the equation, where T_s is the serial/original time and T_p is the parallel/optimised time.

$$S = \frac{T_s}{T_p} \tag{2.1}$$

Because the equation relies on only two times, it can be used as a quick metric to show how well a program scales, or how much of an effect optimisations have on the program. This metric can also be used to examine how well a program scales with larger systems, by looking at how the score increases as more resources are given to the program. If speedup increases at the same rate as the resources increase, then the program scales well.

2.5 Amdahl's Law

Amdahl's Law was proposed in 1967 by Gene Amdahl and shows the performance differences for a program across different amounts of parallelism. [2] The equation (shown in Equation 2.2, where p is the fraction of the code parallelised and n is the number of processors used when running the parallel code) shows that the amount of speedup is dependant on two sections, the serial portion of the program and the parallel portion of the program.

$$S = \frac{1}{(1-p) + \frac{p}{n}} \tag{2.2}$$

The first section (relating to $(1-p)$) determines the amount of time spent in serial parts of the program. In order to increase the performance of this section, the performance of the core itself needs to be improved. The most common ways to achieve this are to increase the core complexity, or to increase the speed

of the core.

The second section (relating to $\frac{p}{n}$) determines the amount of time spent in parallel parts, and how a change in the number of processors will affect this. Due to the nature of this portion of the program, the easiest way to increase the performance is to increase the number of cores that can operate on the application. Recently, the performance of an individual core has slowed, along with the proportion of programs that can be run with parallelisation. So in order to increase the performance of an average application, the number of processors has had to be increased. This in turn has driven the need for larger systems.

2.6 Overhead

To achieve additional functionality or parallelisation in an application, a library or framework is often utilised. In doing so, the requirement for specialised code within the programs development is negated. This result is that, should the application need to be run on different hardware, it is not necessary to make large modifications to the code. However, this carries a cost in the form of additional computation (also known as overhead), which may slow down the program in like-for-like comparisons. To measure the amount of impact, Equation 2.3 can be used, where n is the time taken for the new implementation, and r is the time taken for the reference implementation.

$$O = \frac{n}{r} - 1 \tag{2.3}$$

The equation produces a percentage overhead cost for the library. As such, the aim is to achieve an overhead value as close to 0% as possible. In order to effectively use this equation, it is imperative that both versions of the program should be running with the same configurations, compiler, hardware, problem sizes and parallelisation strategies.

2.7 Performance Portability

Within HPC, the majority of computations have traditionally been performed on a Central Processing Unit (CPU) with multiple cores. This has meant that, until recently, there has been a large focus on optimisations and increasing the parallelism of these processors. However, with the mainstream adoption of Graphics Processing Unit (GPU) within HPC, new programming paradigms are necessary. In order for this new hardware to be effectively utilised, a multitude of different technologies have been developed. Some of these, such as Compute Unified Device Architecture (CUDA) [77], allowed code to be specifically written for particular GPUs. Whilst these allowed for better performance when compared to other techniques, a full redevelopment of the program is required in order to utilise this. Other technologies, such as OpenCL [109], allowed for the application to be written in such a way that the code could be transformed when compiled for a given architecture. As a wider range of different processors and memory configurations are adopted, these paradigms became more important. Time, energy and money did not need to be spent writing multiple versions of the same program. However, this often came at the cost of additional computation and runtime. This cost often outweighed the additional development time required, spawning an area of research called Performance Portability. [8, 9] Whilst the exact definition has been debated, it is generally agreed that the principle of a Performance Portable program is one where the code could be run across multiple different architectures, with minimal change to the application and minimal cost to the performance compared with a similar application written with manual optimisations. [55]

One of the first frameworks to embrace this idea was OpenCL [68, 109]. Using this method, the code to be parallelised was compiled at runtime for the required architecture. This meant that some of runtime would be spent compiling code, but utilising this framework ensured that the code was compiled for the correct architecture. SYCL [110] has now become the main way to utilise

OpenCL, rather than writing an application with the OpenCL library. A different approach has been taken by both OpenACC [80, 105] and OpenMP 4+ [82]. These frameworks use `pragma` statements within code. `pragma` statements are sections of code written as specialised comments before blocks of code that are to be parallelised or optimised in some way. Thus, if the compiler recognises the statement, it can perform the required actions to improve the code. However, if the compiler does not support the framework, the statements are ignored without error. Template C++ libraries have also been built to enable performance portability within applications. Kokkos [18] and RAJA [32] are the most used examples of this. In these frameworks, C++ templates and lambda functions are used to optimise as much code as possible at compile time for the required architecture.

A different methodology is to use a Domain Specific Language (DSL) such as Oxford Parallel Library for Structured mesh solvers (OPS) [93] and Oxford Parallel Library for Unstructured mesh solvers (OP2) [27]. These are specialised frameworks that allow for more specialised optimisations, but are limited in the scope of problems they can be used to solve. OPS and OP2 work by the user implementing specific functions into the application, then utilising a source-to-source translator to transform the application, depending on what is required. In particular, this technique has been used in these DSLs to allow for a variety of different parallelism methodologies (such as the ones mentioned above) to be implemented into the same code base, thus allowing an application to be performance portable.

With a wide range of different frameworks designed to work on a large range of different architectures, it is difficult to measure their overarching performance and to make comparisons. A library might perform better on one architecture, but might perform worse on another. In order to overcome this issue, Pennycook et al. developed a metric which allowed for comparisons between different frameworks across multiple architectures. [87] Equation 2.4 shows how this can be calculated, where H is the set of architectures, a is a given application, p

is the set of parameters for the given application, and e is the efficiency of the given application with the given parameters.

$$P(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

From the equation, it can be seen that if the library is not supported by a system tested on, then the library is given a performance portability of 0%. This is due to the fact that if the library cannot run on a tested architecture, then it is not performance portable. The term *efficiency* is also loosely defined. This is so that it can apply to both the application efficiency (how fast a given version of the application runs compared to other versions) and the architectural efficiency (how well the version of the application utilised a given hardware). Even then, the architecture efficiency can be measured in different ways, such as the computational efficiency and the memory efficiency. The metrics used to calculate the performance portability should be dependant on what is the most appropriate for the application tested.

2.8 Summary

In this chapter, the use of benchmark suites and proxy applications in assessing the effectiveness of a system has been discussed. The benefits and disadvantages of benchmarks were discussed, and how mini-applications can help resolve this. Analysis of a program can be achieved through profiling, examining both break-down of functions and utilisation of hardware. As well as this, the performance of applications, and the effect of a library on an application, can be measured in a multitude of different ways. Finally, the principle of performance portability was discussed, along with its importance and how it can be calculated for a given library.

CHAPTER 3

Achieving Performance through Hardware Optimisations

As discussed at the beginning of Chapter 2, one way to achieve better performance for an application is to improve the system which is being used. These system improvements have taken many different forms since the creation of the silicon Central Processing Unit (CPU). Two key ways in which this has been achieved is through improvements to the processor itself (whether through the use of smaller transistors or the utilisation of parallelism) and the use of larger, faster memory hardware. However, there is still a need for software advancements that can make use of these new hardware improvements.

This following chapter is broken down as follows:

- Section 3.1 discusses Moore's Law, and its impact in both High Performance Computing (HPC) and beyond.
- Section 3.2 discusses Flynn's Taxonomy and the categorisation of systems by the way in which data and instruction streams are handled.
- Section 3.3 explores how parallelism within computing allows for better performance within applications, through the utilisation of vectorisation, multithreading, multiprocessing and distributed computing.
- Section 3.4 explores the importance of the memory hierarchy and data structures in relation to an applications performance.

3.1 Moore's Law

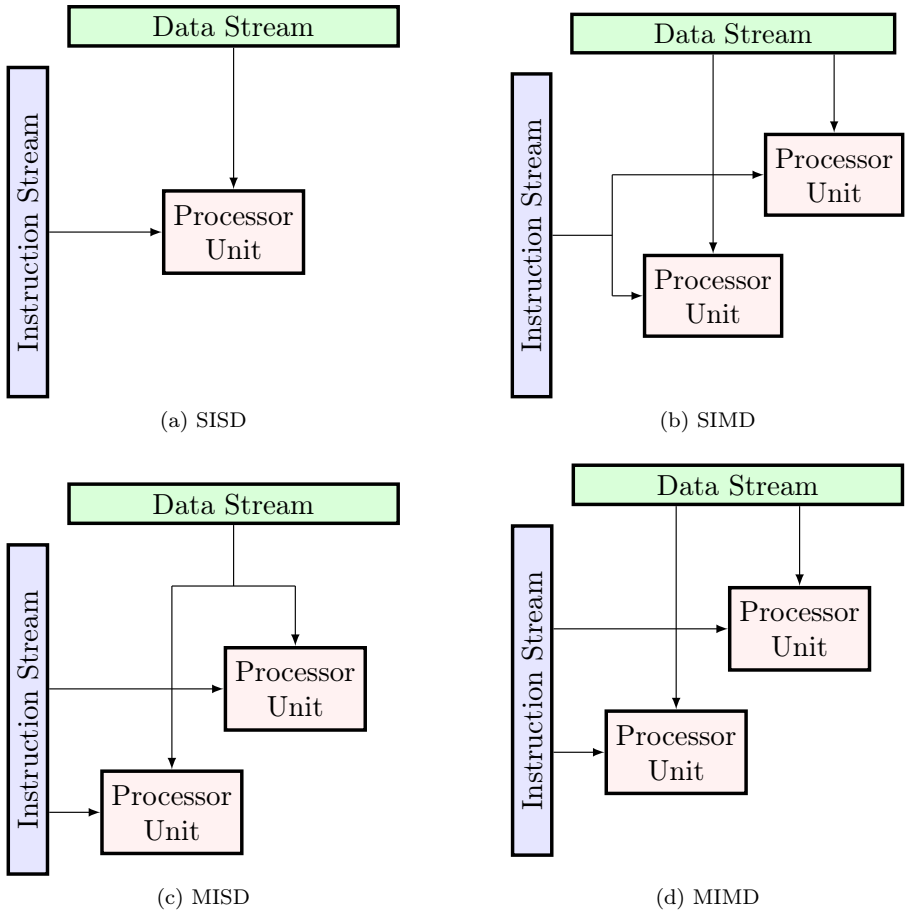
Gordon E. Moore, born in 1929, is one of the most important names in Computer Science. Having gained his PhD in Physical Chemistry at the California Institute of Technology in 1954, Moore worked at Fairchild Semiconductors. He worked on the manufacturing process for transistor contacts, and eventually worked his way up to manage the research and development department. In 1965, Moore wrote *Cramming More Components onto Integrated Circuits* [69]. The paper stated that: for a given size of chip, the number of components would double every year. This would result in: more powerful computers, lower manufacturing expense and, as a consequence, reduced purchasing costs. This concept would later be known as *Moore's Law*. In 1968, Moore and his colleague Robert Noyce founded the Intel Corporation, which is now one of the largest chip manufacturers in the world. Moore's Law would later be revised to 'a doubling every two years' [70]. [99]

In general, Moore's law has proven to be true. However, the physical limits of transistor sizes are now starting to reached. The limits of silicon-based integrated circuits mean that transistors can be no smaller than 5nm, because a single nanometer can accommodate only two silicon atoms. [100] There are also physical constraints associated with voltages, heat dissipation, and clock frequencies, all of which are limiting factors for computing power. [112, 124] Subsequently: algorithmic changes to programmes, the use of larger multi-core systems and the introduction of accelerator cards have allowed computer performance to increase, and have enabled better utilisation of hardware.

3.2 Flynn's Taxonomy

In 1966, Michael Flynn set out to categorise the different ways in which a computer could theoretically perform operations on data. He generalised that a computer processor comprises of two elements: a data stream and an instruction stream (which provides operations for the computer to apply to the data

stream). From this idea, he proposed that a computer could utilise each of these streams in either a serial or parallel fashion. He developed the theory further to state that each machine could be classified as one of four categories. [22] This is colloquially known as *Flynn's Taxonomy*. For simplicity, the term *stream* is often omitted when referring to both the data and instruction streams. Figure 3.1 shows a graphical representation of these different classifications.



(a) SISD (b) SIMD

(c) MISD (d) MIMD

Figure 3.1: Graphical representation of the different categories in Flynn's Taxonomy

3.2.1 Single Instruction - Single Data

Single Instruction - Single Data (SISD) (Figure 3.1a) is the simplest classification, as the computer in this category will run in a serial manner. Each single piece of data is acted upon by a single instruction. Once this is complete, the next piece of data will undergo its operation. In order to achieve an improved performance from a machine using SISD, the clock speed of the machine would have to be increased. However, this often leads to higher voltages and increased heat generation. Therefore, cheaper, parallel-based systems are often favoured over this category.

3.2.2 Single Instruction - Multiple Data

Single Instruction - Multiple Data (SIMD) (Figure 3.1b) offers a degree of parallelism by performing the same instruction across multiple items of data simultaneously. The amount of data that can be coincidentally operated upon is dependant on the hardware used. Thus, a simple way to improve the performance of a SIMD computer is to increase the amount of data that can be processed by a single instruction. However, because only a single instruction can be used, some algorithms may not be expressed in a way that makes full use of the hardware. Examples of this include Streaming SIMD Extension (SSE) [91] and Advanced Vector Extensions (AVX) [37], which allow for programmers to make full use of SIMD hardware. Listings 3.1 and 3.2 show a side-by-side comparison of a simple loop before and after it has been vectorised using SSE. Both of the code examples multiply each corresponding element from arrays `a` and `b`, and then adds the result to the corresponding element in the array `c`. However, in Listing 3.2, the loop contains far fewer iterations. This can be achieved through packing data into vector registers (performed by `_mm_load_ps`), performing the calculations on all variables stored in the registers at the same time (in this case, `_mm_add_ps` and `_mm_mul_ps`), then storing the resulting values back into the correct area in main memory (performed by `_mm_store_ps`). As

the number of iterations may not be exactly divisible by the number of elements that can fit into a vector register, a "cleanup loop" is required. This performs the required calculations on any remaining elements, in the same manner as the original code.

```
/*
 * Initialisation of array of
 * floats a, b and c,
 * and variables i and N
 */
for (i = 0; i < N; i++) {
    c[i] += a[i] * b[i];
}
```

Listing 3.1: Example C code of loop without vectorisation

```
/*
 * Initialisation of array of
 * floats a, b and c,
 * and variables i and N
 */
int loopN = (N/4)*4;
for (i = 0; i < loopN; i+=4) {
    __m128 aVec=_mm_load_ps(a+i);
    __m128 bVec=_mm_load_ps(b+i);
    __m128 cVec=_mm_load_ps(c+i);
    cVec=_mm_add_ps(cVec,
         _mm_mul_ps(aVec, bVec));
    _mm_store_ps(cVec, c+i);
}

//Cleanup loop
//(needed if (N%4) != 0)
for (; i < N; i++) {
    c[i] += a[i] * b[i];
}
```

Listing 3.2: Example C code of loop with vectorisation (utilising SSE)

SIMD is also utilised within modern Graphics Processing Unit (GPU) hardware, owing to the fact that the processor will have to apply the same function to multiple pixels in graphical computation. As such, GPUs contain a much larger number of threads which are grouped together to form Streaming Multiprocessors. Each thread within a Streaming Multiprocessor will run the same instruction, but will operate on separate pieces of data, with multiple Streaming Processors will being contained within a single GPU. It follows, therefore, that GPUs are large SIMD processors. [126]

3.2.3 Multiple Instruction - Single Data

Multiple Instruction - Single Data (MISD) (Figure 3.1c) is the least popular category, as it relies on a single piece of data being operated on in different ways, at the same time. This is because algorithms which fit the remit of a shared data store being used in differing calculations, are rare. As such, machines in this category are not often found in HPC.

3.2.4 Multiple Instruction - Multiple Data

Multiple Instruction - Multiple Data (MIMD) (Figure 3.1d) is one of the most common parallelisation methodologies in modern HPC. This category allows for multiple processors to perform instructions on different pieces of data simultaneously; it is therefore analogous to the multi-core processor arrangements used in many computation devices and HPC systems. Listings 3.3 and 3.4 show a side-by-side comparison of a simple loop before and after it has been parallelised through the framework called OpenMP. In the example shown in Listing 3.4, OpenMP creates multiple threads, then partitions the iterations into the threads, and finally destroys the threads once complete. This process is carried out at compile time, and is denoted by the tag **#pragma** omp.

```
/*
 * Initialisation of arrays a, b
 * and c, and variables i and N
 */
for (i = 0; i < N; i++) {
    c[i] += a[i] * b[i];
}
```

Listing 3.3: Example C code of loop without parallelisation

```
/*
 * Initialisation of arrays a, b
 * and c, and variables i and N
 */
#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] += a[i] * b[i];
}
```

Listing 3.4: Example C code of loop with parallelisation (utilising OpenMP)

3.3 Parallelism

When the modern silicon processor was first introduced, the easiest way to increase the performance of a program was to either: increase the clock speed, which allowed more computational cycles in a unit of time; to use more specialised circuitry; or to make the components smaller; or a combination of these options. However, the physical limits of this approach were quickly reached. Increasing the clock speed required more power which, in turn, generated more heat, making the chip more unreliable and prone to breaking down. [61] The inclusion of more specialised circuitry increased the cost of the processor, as it meant that it was harder to create code for. A reduction in the size of the components allowed for more transistors to be contained in the same area and, if too much power was supplied, issues such as electron tunnelling arose, meaning that the data within the processor would become corrupted. Therefore, a different technique was required in order to continue improving performance.

Many algorithms contain loops where calculations can be done independently of each other. This is the ideal case when attempting to introduce parallelism as an algorithm, as each each loop iteration can be executed separately. However, there are also many algorithms where a given loop iteration is dependant on previous loop iterations, or where data may be altered out-of-sequence to its serial counterpart. When implementing algorithms and applications in HPC, these situations are avoided wherever possible through the use of approximations or refactoring the algorithm to avoid data dependencies. We can, therefore, explore the idea of performing multiple calculations at the same time in order to increase the performance of a program.

As highlighted in Section 3.2, a parallel processor can fit into three different categories: SIMD, MISD, and MIMD. Due to the fact that MISD computers only exist in very particular situations, SIMD and MIMD computers will be discussed. In Section 3.3.1, vectorisation as a form of SIMD processing is discussed. Two different MIMD processors are then discussed through the use of

multithreading and multiprocesors in Section 3.3.2, and distributed computing is discussed in Section 3.3.3.

3.3.1 Vectorising

Vectorising is one of the simplest ways in which an algorithm can be parallelised. This involves the placing of concurrent memory in larger registers, known as vector registers, followed by the application of specialised instructions to the entire register. As such, this follows the SIMD model discussed in Section 3.2.2. This approach is limited by the fact that all data accessed by the vectorised algorithm needs to be in the form of concurrent memory addresses, as the data needs to be loaded in and out of vector registers in full blocks.

One of the earliest frameworks for vectorisation was MMX. Initially introduced by Intel in 1996, it was one of the first frameworks with a unified set of instructions. Whilst Intel had produced chips with SIMD instructions, they were not for general purpose use, so code had to be specifically written to exploit a given chip. Furthermore, code had to be rewritten if moved to a different processor. With MMX, the aim was to provide a a generalised set of instructions for a fixed vector register of 64 bits. [86]

SSE was developed on the principles outlined by MMX, which extended the range of instructions. Specifically: SSE allowed for four floating-point operations to be calculated through SIMD, and allowed for larger registers. This was necessary for accelerating 3D graphics. [108, 91] The instruction set was extended multiple times, forming SSE2, SSE3/SSSE3, SSE4, SSE4.1, and SSE4.2.

AVX extended the functionality of SSE4.2, as well as the size of the vector registers to 256 bits. [37] This was later extended to AVX2 [38], to include more instructions; then, again, to AVX-512 [95], which included more instructions and increased the vector register sizes further to 512 bits.

Listings 3.5 and 3.6 show how the same simple example can be vectorised using SSE and AVX. Both vectorised examples take the same form as the example shown in Listing 3.1, which is found in Section 3.2.2. Specifically, these

examples show how calculations and branching statements can be achieved in vectorisation, both with SSE and AVX The examples demonstrate a kernel where if the value of `a[i]` is less than 0.5, it is set to 1, otherwise it is set to 2.

```
/* Creation of loopFactor = 4, N, i, and a float* a */
int loopN = (N/loopFactor)*loopFactor;
for (i = 0; i < loopN; i+=loopFactor) {
    /* Load vector aVec = (0.8, 0.2, 0.3, 0.7) */

    /* Setup mask */
    __m128 half = _mm_set1_ps(0.5); //half = (0.5, 0.5, 0.5, 0.5)
    __m128 mask = _mm_cmpgt_ps(aVec, half); //mask = (T, F, F, T)

    /* Set up vectors for if everything is True or False */
    __m128 branchT = _mm_set1_ps(1.0); //branchT=(1.0,1.0,1.0,1.0)
    __m128 branchF = _mm_set1_ps(2.0); //branchF=(2.0,2.0,2.0,2.0)

    /* Merge the branches together */
    __m128 resultT = _mm_and_ps(mask, branchT);
    //resultT = (1.0, 0.0, 0.0, 1.0)
    __m128 resultF = _mm_andnot_ps(mask, branchF);
    //resultF = (0.0, 2.0, 2.0, 0.0)
    __m128 result = _mm_or_ps(resultT, resultF);
    //result = (1.0, 2.0, 2.0, 1.0)

    /* If processor is able to use SSE4.1, blendv can be used *
     * __m128 result = _mm_blendv_ps(branchF, branchT, mask);
     * //result = (1.0, 2.0, 2.0, 1.0)
     */

    /* Store vector result in a */
}
//Cleanup loop
```

Listing 3.5: SSE example showing how branching statements can be vectorised

Whilst it is possible to write programs using these instruction sets, it is often not recommended as it limits the ability to compile and run the application on other platforms and CPUs. It can also make the code harder to read and update. Instead, nearly all modern compilers will detect which vectorisation frameworks can be applied on the hardware, and will automatically vectorise all applicable code. This method facilitates an increase in performance without the need to extensively change the code.

```
/* Creation of loopFactor = 8, N, i, and a float* a */
int loopN = (N/loopFactor)*loopFactor;
for (i = 0; i < loopN; i+=loopFactor) {
    /* Load vector aVec = (0.8, 0.2, 0.3, 0.7, 0.5, 0.6, 0.1, 0.4) */

    /* Setup mask */
    __m256 half = _mm256_set1_ps(0.5);
    //half = (0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5)
    __m256 mask = _mm256_cmp_ps(aVec, half, _CMP_GT_OQ);
    //mask = (T, F, F, T, T, T, F, F)

    /* Set up vectors for if everything is True or False */
    __m256 branchT = _mm256_set1_ps(1.0);
    //branchT = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
    __m256 branchF = _mm256_set1_ps(2.0);
    //branchF = (2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0)

    /* Apply mask to data */
    __m256 result = _mm256_blendv_ps(branchF, branchT, mask);
    //result = (1.0, 2.0, 2.0, 1.0, 1.0, 1.0, 2.0, 2.0)

    /* Store vector result in a */
}
//Cleanup loop
```

Listing 3.6: AVX example showing how branching statements can be vectorised

3.3.2 Multithreading and Multiprocessing

Although vectorisation of a program can provide a performance improvement, the impact may be limited by the fact that only a given number of elements can be processed in a clock cycle. Therefore, to improve the performance of a HPC application, parallelism can be used to further increase the performance of an application. One of the primary ways to do this is through *multithreading*. A thread is a section of code, which can be run on a CPU; utilising the resources available, until the execution has been completed. Once completed, the thread is destroyed. [53] By using multiple threads, it is possible to execute code, which utilises different parts of a variables data, or will execute different parts of the same algorithm.

Multithreading can be used on single CPU cores, and also on CPUs with multiple cores. However, when mutlithreading is used on a single-core CPU, concurrency is achieved, rather than parallelisation. This is because, in order for a single-core processor to compute each thread, the CPU has to evacuate the current thread and load in a new thread from memory. This takes up clock cycles which could be better utilised executing the already-running thread. In HPC, when using a multi-core processor, it is considered optimal to have a thread running on each core of the processor. Thus, the thread does not need to be switched out for other threads, thereby ensuring more parallelism and higher performance.

Whilst multithreading through the use of multiple cores increases the amount of parallelism, and therefore performance, the processor may not be fully utilised if threads have to wait for other operations to complete. These other operations may comprise of memory requests or waiting for other threads to finish due to load imbalances. In order to better utilise the hardware in these cases, *simultaneous multithreading* (also called *hyperthreading*) can assist in using up this available time. In simultaneous multithreading, each CPU core can have multiple threads. Whilst a thread is being executed, the core can run a separate thread which does not need the same hardware as the first thread. [117] In order to achieve this, specialised hardware is required. Most modern Intel processors have been designed to have two threads per core [96], with some processors such as ARM Marvell Thunder X2 being designed to have four threads per core [65].

Multithreading is achieved in much the same way as vectorisation, through the use of frameworks. One of the most well known implementations is POSIX Threads, commonly known as PThreads. This framework allows for threads to be created, and access gained to the shared memory on the CPU, before being joined back up to the main thread. The behaviour of these threads is defined by an IEEE Standard (IEEE POSIX 1003.1c). [35] This approach is also supported by multiple compilers. [7] However, it is not often used in HPC due to its complexity, as can be seen in Listing 3.7. This example creates a

collection of threads based on `THREAD_COUNT`, each of which runs the function `kernel` and passes the arguments to the function as specified in the structure element `arg_threads`.

```
#include <pthread.h>
/*
 * Initialisation of arrays, variables threadCounter, THREAD_COUNT
 * and N, an array of structure arg_threads, array of pthread_t
 * threads and a virtual function kernel
 */
for (threadCounter = 0; threadCounter < THREAD_COUNT;
    threadCounter++) {
        pthread_create(&(threads[threadCounter]), NULL,
            kernel, (void *) &(arg_threads[threadCounter]));
}
for (threadCounter = 0; threadCounter < THREAD_COUNT;
    threadCounter++) {
        pthread_join(threads[threadCounter], NULL);
}
```

Listing 3.7: PThread example of a parallelism loop

More often, a HPC application will use OpenMP to implement multithreading for particular sections of code, as OpenMP uses specialised statements in FORTRAN and C/C++ to inform the compiler on how to parallelise a given section. These statements are designed in such a way that they will be ignored by the compiler, should it not be compatible with OpenMP. [81] Listing 3.8 shows an example of OpenMP multithreading being utilised on the function `kernel`.

```
#include <omp.h>
/*
 * Initialisation of arrays, and variables i and N
 */
#pragma omp parallel
kernel( /* Kernel arguments */ );
```

Listing 3.8: OpenMP example of a parallelism loop

Multithreading, whether using PThreads or OpenMP, is limited by the fact that the threads can only access local memory. This means that, in order to increase the performance through the use of multithreading, larger processors with more cores are required. However, like vectorising and the reduction of component size, the size of the processor is limited by: the voltage requirements, heat dissipation, and larger production cost required to power a larger, more complex CPU. Therefore, a more efficient way to increase the level of parallelism is to locate multiple CPUs within the same computer, and design them so that they can act as one, larger, processor. This is known as multiprocessing, and is often implemented with HPC systems. The Operating System (OS) sees both processors as a single unit, with different Non-Uniform Memory Access (NUMA) regions. Therefore, when running multithreaded problems, it is often recommended that the threads are limited to a single NUMA region, to avoid conflicts and slow memory access.

3.3.3 Distributed computing

Achieving performance through parallelism has often lead to better utilisation of hardware by exploiting the nature of many computational algorithms. We have explored parallelism through vectorisation, multithreading and multiprocessing, but we can also achieve parallelism through the idea of splitting a given problem across multiple machines. This is how HPC solves its largest problems, and consequently how the largest supercomputers have been designed to date. The Top500 tracks the computational performance of these machines across the world. [113] Within a supercomputer, a user can request access to a collection of individual computers (referred to as nodes) for a period of time. To ensure that other users do not operate on the same nodes at the same time, a scheduling program such as Slurm [98] or OpenPBS [84] is used to manage access.

The issue with distributed computing in this fashion is that, if a given block of data is partitioned across multiple machines, computation of an algorithm on a given node may require data that resides on a different node. For example, an

algorithm may need to calculate the average value for a given position based on it's neighbours data. However, for data located on the edge of a node's partition, the results would be incorrect as a neighbour would be on a different machine. Thus, for the program to execute correctly, the nodes need to communicate between each other. The most common library used to achieve this is Message Passing Interface (MPI). First discussed in 1992 [122] and proposed a year later [111], MPI allows for communication of data between different processors regardless of whether they are on the same machine or on a machine on the same network. This methodology is widely used within HPC to allow for parallelism across a large number of nodes, and across both NUMA regions and cores within a single node. Listing 3.9 gives an example of how MPI can be utilised. In this example, the MPI environment is initialised using the requested parameters. This would include information such as the number of ranks available, and which cores/processors these ranks are attached to. The current rank number and total number of ranks is then obtained. This information can be used to split up (often referred to as decompose) the data. The code can then be ran, before the MPI environment is finalised. There are different implementations of MPI, including OpenMPI [83] and Intel MPI [39]. [6]

3.4 Memory Layouts and Data Structures

Optimising the amount of computation is not the only way to increase the performance of a program. The utilisation and speed of a particular system's memory is also critical to the execution speed. In an ideal situation, the memory would be as fast as possible, with all of the fastest memory being used all of the time. However, this is not always possible for a number of reasons. In order to achieve fast access to memory, the memory has to be physically close to the CPU, if not within it. A large amount of channels are also required in order to allow the data to move in as few clock cycles as possible. A consequence of this approach is that the economic cost of the memory becomes much more

```
#include <mpi.h>
/*
 * Initialisation of arrays.
 */

//Create the MPI environment
MPI_Init(NULL, NULL);

//Get the current rank ID of and the total number of ranks
int currentRank, totalRanks;
MPI_Comm_rank(MPI_COMM_WORLD, &currentRank);
MPI_Comm_size(MPI_COMM_WORLD, &totalRanks);

kernel( /* Kernel arguments */ );

//Finalise the MPI environment
MPI_Finalize();
```

Listing 3.9: MPI example of parallelism loop

expensive, and means that either the processor has to become physically larger (the disadvantages have been discussed in Chapter 3.1) or, conversely, less room is made available to the computational part of the processor.

Due to these constraints, a memory hierarchy is formed, with the fastest but smallest at the top, extending down to the slowest, but largest formats. The memory hierarchy can be seen in Figure 3.2. At the top, *registers* are small blocks of memory that are used in instructions, often containing the instructions themselves and the current piece of data being operated upon. Whilst it is important to the operation of the processor, it is not often that an application will be optimised specifically for registers, due to their size and the fact that they are usually completely controlled by the CPU's internal controller. Instead, a HPC application will often optimise for the next level of memory, *cache*. The cache is usually placed in blocks inside the chip, whereas registers would be interlaced with the circuitry. Cache often comes in multiple levels, with the closest being Level 1 Cache Memory (L1), followed by Level 2 Cache Memory (L2), and then Level 3 Cache Memory (L3). In much the same way as the overarching memory, these cache levels have a similar hierarchy, with L1 being

the closest to the chip and the fastest, but containing less memory than the other levels.

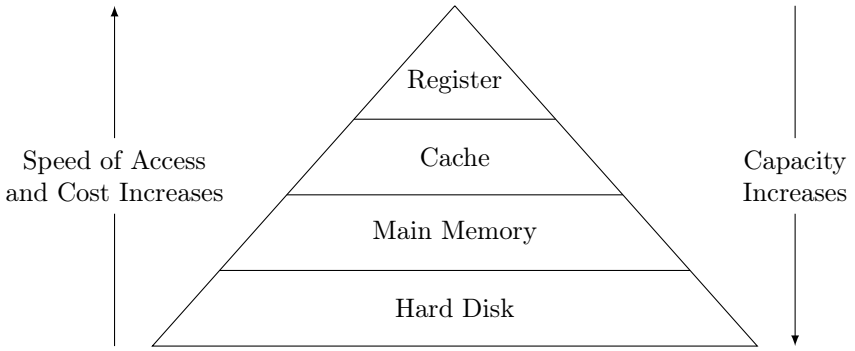


Figure 3.2: Graphical representation of the Memory Hierarchy

Following cache, there is *main memory*. This is often the first level of memory that does not reside on the CPU itself, and, as such, requires a large amount of clock cycles to access. There are many different types of main memory, depending on the purpose. The most common type is known as Double Data Rate (DDR). DDR memory allows for double the data transfer rate of it's predecessor, and is therefore faster. As this type of memory operates on its own clock, the speed of the main memory is also important to the performance of the program. As well as DDR memory, there are other types of main memory which can perform error corrections and can manage larger amounts of bandwidth, which allows for faster transfers through the use of more channels, rather than relying on faster clock speeds.

Finally, *hard disks* allow for bulk storage of large amounts of data. Hard disks also have the benefit of not losing the data stored in the memory when the machine is powered off. However, this comes at a cost as this is the slowest type of memory for the CPU to access. There are many different types of hard drives, depending on the size, speed and economic cost required. These range from magnetic tapes (one of the slowest hard drives, but also one of the longest lasting and most dense), to Hard Disk Drive (HDD) (consisting of a series of spinning, metallic plates that are read with a small armature). HDD are

frequently utilised within computers for larger storage, which is often a specific requirement. In addition, Solid State Drive (SSD) are available, which are faster than HDD due to the lack of moving parts, but, conversely, are more expensive. In a large HPC cluster, there will often be a small hard disk included on each server in order to store the OS and essential programs, and also a networked collection of hard disks, on which user programs can be stored.

In an ideal world, the most relevant piece of data should reside in the fastest memory near the top of the hierarchy. However, most programs could not completely reside in the fastest memory due to there size. As such, data needs to be moved into and out of faster memory as and when required. Ensuring that the CPU does this efficiently is key to an optimally-performing application. When the processor requests a piece of data, the processor will look at each level of the memory hierarchy in turn, requesting the required data at each level. As soon as the relevant data is found, it is moved up the hierarchy as necessary, displacing an older piece of data back down the order. An optimally designed program will do this as little as possible, allowing the CPU to spend more time on the computation, and less time on memory management. A well designed memory structure can help the performance of the program, by allowing the CPU to exploit techniques such as prefetching. In Sections 3.4.1, 3.4.2 and 3.4.3, the three basic data structures are examined, with the advantages and disadvantages of each being considered in turn. Then, in Section 3.4.4, the more abstract data structures are discussed.

3.4.1 Structure of Arrays

Structure of Arrays (SoA) is one of the most common ways to structure the data for any given application, and is often the first that is taught in any computing course. This often consists of a series of arrays, each of which consist of a collection of elements. Each array does not necessarily have the same number of elements, but key groups often will. Figure 3.3 and Listing 3.10 show both a graphical representation and a code example of an SoA in a C-styled pseudocode.

It should be noted that the arrays do not need to be explicitly in a `struct`, but has been included for comparison to other data structures described in Sections 3.4.2 and 3.4.3.

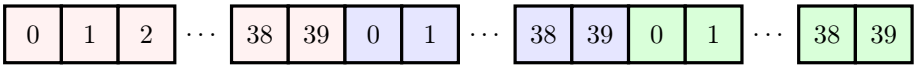


Figure 3.3: Graphical representation of the memory when using a SoA data structure

```
struct SoAExample {
    int foo [40];
    double bar [40];
    char baz[40];
};

SoAExample soaDataStructure;
```

Listing 3.10: Pseudocode example of SoA data structure

SoA allows for all the elements in the same variable to be concurrent in memory. This allows for better cache performance when dealing with a few variables within a given algorithm, as cache lines do not need to be swapped out as often and optimisations such as prefetching can be achieved.. However, once the number of variables used becomes too large, the cache reuse will drop as cache lines containing elements that are still in use, or might be required later, have to be evacuated to make room for the new data.

3.4.2 Array of Structures

The Array of Structures (AoS) arrangement structures the data in the opposite way to SoA. In this data structure, elements from different variables are placed concurrently in memory, looping round until all elements in a variable have been placed. As such, all the variables are required to have the same number of elements. If a variable does not have the same number, then the extra elements would have to be iterated through in a different data structure. Figure 3.4 and

Listing 3.11 show how the data can be laid out in an AoS data structure, both graphically and in C-style pseudocode.

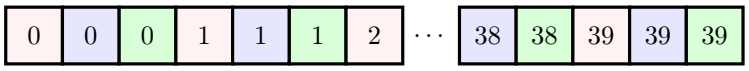


Figure 3.4: Graphical representation of the memory when using a AoS data structure

```
struct AoSExample {
    int foo;
    double bar;
    char baz;
};

AoSExample aosDataStructure[40];
```

Listing 3.11: Pseudocode example of AoS data structure

AoS performs best when there are a larger number of variables used within an algorithm which requires large amounts of data to perform calculations. This allows for data to be close together, thus allowing for better cache reuse. In some languages, padding is added by the compiler. Whilst not required as part of the C standard [45], it is often added as it allows for better memory alignment of the processor, and, as such, enables quicker memory access. However, by doing this, some memory space is lost, which may not be desirable if the system has a small amount of memory or if there are a lot of elements.

3.4.3 Array of Structures of Arrays

Array of Structures of Arrays (AoSoA) is the hybrid approach to both SoA and AoS, and allows for the interleaving of multiple elements and multiple variables. Each array inside the structure can consist of a different number of elements. This means that, unlike AoS, each inner variable does not necessarily need to have the same size. However, each variable must have a common multiple of elements. The common multiple becomes the number of overarching structure

elements. Figure 3.5 and Listing 3.12 shows an example of AoSoA data structure, where each variable inside the structure has an array of two, and a common multiple of 20.

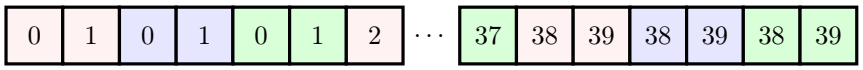


Figure 3.5: Graphical representation of the memory when using a AoSoA data structure

```
struct AoSoAExample {
    int foo[2];
    double bar[2];
    char baz[2];
};

AoSoAExample aosoaVariable[20];
```

Listing 3.12: Pseudocode example of AoSoA data structure

AoSoA is able to provide the benefits of both SoA and AoS, whereby some elements within the same variable are concurrent, and allows for a larger number of variables. However, to achieve these benefits, a large amount of care needs to be taken to ensure the structure fits into cache as optimally as possible. If not, additional padding could be inadvertently introduced meaning that faster cache memory might not be fully utilised.

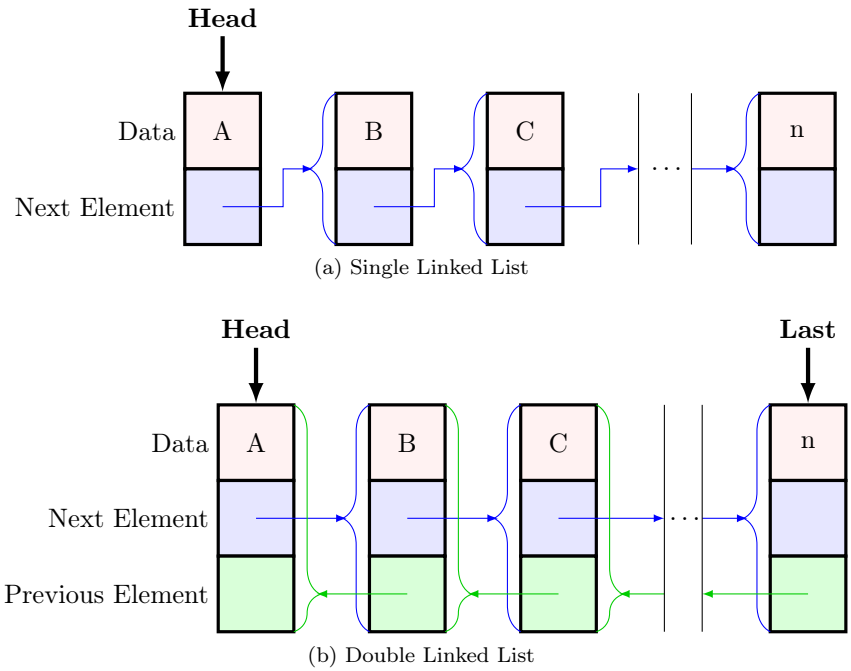
3.4.4 Abstract Data Structures

Whilst SoA, AoS and AoSoA represent different ways to efficiently lay out the data, these structures can often be difficult to apply to a problem. An application could lay out the data in multiple different ways, depending on the needs and which operations are being more commonly carried out. For example, some algorithms require the addition and removal of new pieces of data, whilst others require knowledge of the position of a given element.

Thus, abstract data structures are used to more easily describe how data could be laid out for different problems, independent of their implementation. Each of the data structures presented here can be implemented in multiple ways, including a SoA implementation and an AoS implementation. Because of this, the efficiency of each of these data structures can be given as neither an amount of time nor a definitive number of steps, but, instead, as a notation which gives the order of magnitude of the largest operation. This is called Big-O notation. In Big-O notation, the scale of the efficiency is represented. For example, for $O(n)$, if the number of elements double, the operation would take double the time. However, for $O(n^2)$ or $O(n^3)$, if the number of elements double, the operation would take four (2^2) times or eight (2^3) times longer respectively. As such, the lower the complexity is desired, with the lowest being that the algorithm will take the same time no matter how much data is passed to algorithm, O(1). [4, 73]

Linked Lists

The first abstract data structure examined is a linked list, two different versions of which can be seen in Figure 3.6. In this data structure, the initial data element is stored and is referred to as the head element. Each data element consists of at least two sections. The first is the data itself. The second is the position of the next element in the list. Thus, to get to a given element, the program retrieves the head element and if this is not the element required, the program then looks at the next element. This can repeat until the element is found, or the element does not point to another. Figure 3.6a shows the most basic example of this. Each data element can also be extended to allow for the position of the previous element to be stored as well. Because of this, the last element is also often stored, and is referred to as the tail element. An example of this can be seen in Figure 3.6b. The addition of this facility allows for easier navigation through the list, but comes at the cost of requiring more memory.



(a) Single Linked List

(b) Double Linked List

Figure 3.6: Graphical Examples of Linked Lists data structures

Unlike SoA, AoS and AoSoA, the linked list data structure does not require knowledge about the number of elements it needs to contain. This is because the next or previous elements positions can be any valid memory. Therefore, to append a new element to the list, a new block of memory can be allocated, and the next element in the appropriate node is updated to point to the newly created element. Thus, appending an element to the list can be achieved in O(1). The removal of an element can be carried out in a similar way and with a similar cost, as the elements on either side can be updated to point to each other. However, finding a particular element means going through the entire list. Thus, the efficiency of this is O(n) (where n is the number of elements).

Tree Data Structures

Another type of abstract data structures are trees. Like linked lists, tree data structures consist of data elements (called nodes) with two sections. Similar to linked lists, the first section is the data itself. The second section consists of two or more locations of other nodes. Depending on the type of tree data structure, the number of node locations specified may be fixed. For example, in a Binary Tree data structure (as seen in Figure 3.7), each node can only have a maximum of two nodes that it can point two, referred to as the left and right hand branches. One of the key conditions of tree data structures is that links between nodes cannot become circular, and must be directional. This, followed by fact that traversal must always start from the top node (referred to as the root node), means that the tree can be fully traversed without worry that it could loop.

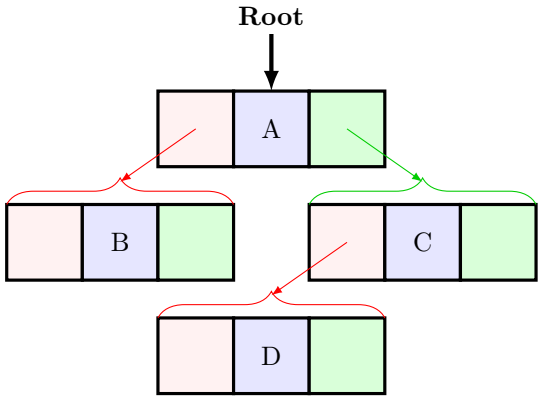


Figure 3.7: Graphical example of a Binary Tree based data structure

Due to the nature of this particular structure, it is often used to store data that requires a strict hierarchy and a particular ordering. The tree data structure allows for a process called re-balancing. Re-balancing enables an ordered tree to be reorganised in such a way that no particular branch has a large proportion of nodes. By so doing, the data structure can ensure faster traversal of the data, taking O($log(n)$), where n is the number of nodes in the entire tree.

Graph Data Structures

Graph data structures are similar to tree data structures, but are more generalised. Like trees, graphs contain nodes linked together (the links are referred to as edges in graph theory). Each node can be linked to multiple different nodes, and in some particular cases, to itself. However, unlike trees, edges on a graph do not have to be directional, they can form loops across the same or multiple nodes, and edges can store data as well as nodes. An example of a graph can be seen in Figure 3.8. In this example, it can be seen that each node has multiple connections, and each edge has a weight attached to it. These data structures have been used widely within different Machine Learning (ML) and Artificial Intelligence (AI) algorithms and programs, as they easily lend themselves to a network of differently connected pieces of data, each with a cost to travel from one to another.

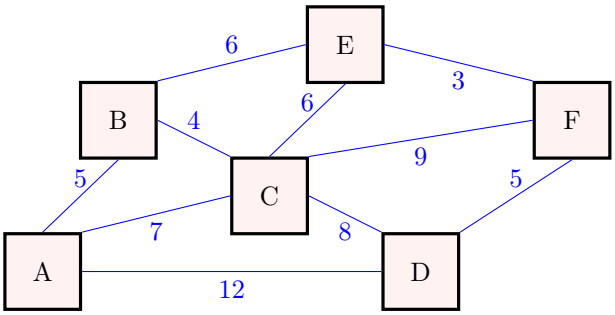
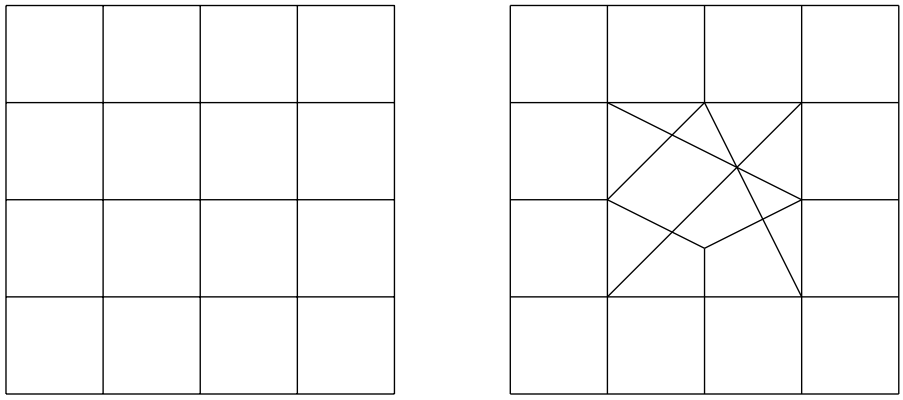


Figure 3.8: Graphical example of a Graph data structure

Meshes

Mesh data structures are often used within physics applications to represent a plane of particles, materials or fluids. In this data structure, it is assumed that the real world physics happens either along intersections or in the centre of a grid cells, depending on the physics being modelled. Figure 3.9 shows the two different types of meshes often seen within physics applications, structured (Figure 3.9a) and unstructured (Figure 3.9b). Structured meshes store all their data in a rigid format, thus allowing for more optimisations. So if one particle is al-

tered the adjacent particles can be quickly determined and adjusted accordingly. However, in order to achieve a higher resolution on a particular area, the entire grid has to change resolution, dramatically increasing the amount of memory required. Unstructured meshes do not have to follow this strict pattern, as they are allowed to make more or fewer connections depending on what is required. This allows for meshes to morph over the course of a program, thus producing more detail in specific areas as required. However, this means that the same assumptions used to optimise structured meshes cannot be applied, as a given cell might have more or less neighbours than before. As a consequence, these structures can be more complex to implement, and can be slower to process.



(a) Structured Mesh (b) Unstructured Mesh

Figure 3.9: Graphical examples of different types of Meshes

3.5 Summary

In this chapter, the use of different techniques to increase the performance of an application through HPC systems were discussed. The concepts of Moore's Law and Flynn's Taxonomy were explored, and how they initially pushed computation and parallelisation respectively. Parallelisation was then explored more deeply, specifically looking at how vectorisation, multithreading, multiprocessing and distributed computed has been used to exploit inherent parallelism in many algorithms.

As well as discussing how a programs performance could be improved through the use of parallelism, the use of memory and data structures was explored. Specifically, how different elements of memory was discussed, along with the rational behind the size, speed and therefore importance of each. This fed into a exploration of a number of data structures, starting with SoA, AoS and AoSoA. Finally, the advantages and disadvantages of abstract data structures were considered.

CHAPTER 4

Analysing the Performance Portability of a Heat-Conduction Mini-Application

Modernising production-grade, often legacy applications, to take advantage of modern multi-core and many-core architectures can be a difficult and costly undertaking. Often, these applications have been developed over decades and consist of code bases with hundreds-of-thousands or even millions of lines of code. Adapting to newer systems may require major re-engineering, depending on the support for different languages, parallel programming models and optimisations across platforms. At the same time, there is a growing range of different systems, each with their own parallelisation methodologies, and all aiming to provide the best performance in the long term. It is therefore clear that manually porting large code-bases to use various different programming models and languages, and then maintaining each of these different versions, is infeasible.

One common strategy is to use small, representative applications to test and evaluate new technologies, programming models, frameworks, and optimisations. The use of such programs, called proxy or mini-applications, is not new. The idea can be traced to the development of small benchmark codes such as LINPACK [17] and the NAS Parallel Benchmarks [5]. More recent efforts include the Mantevo [60] and UK Mini-App Consortium (UKMAC) [118] suites. Due to their small size, mini-apps are much more manageable than production applications and can feasibly be re-written in different programming languages, and with specific optimisations. They are also unrestricted and/or devoid of any commercially sensitive code, allowing them to be readily distributed to many parties and sites. Sections 2.1 and 2.2 contain for more information on benchmarks and mini-apps respectively.

In this chapter, the performance of one such mini-app called TeaLeaf is explored. TeaLeaf implements a set of linear equations which form a sparse structured mesh and then uses a five point stencil and cell-centred temperatures to calculate the conduction coefficient [64]. It has been parallelised using a variety of different programming models and language extensions, including: OpenMP, Message Passing Interface (MPI), Compute Unified Device Architecture (CUDA), and OpenACC. It also has been implemented using the Oxford Parallel Library for Structured mesh solvers (OPS) embedded Domain Specific Language (DSL) [93], and the C++ template libraries Kokkos [18] and RAJA [32]. Whilst many other proxy applications have been written to use a wide range of parallelisation models (such as CloverLeaf [59] and other mini-apps found in the Mantevo suite [60]), TeaLeaf was chosen as it had not been explored as thoroughly. Many of these programming frameworks allow for compilation and execution on multiple different systems and architectures. The chapter therefore compares the performance of different implementations of TeaLeaf, including how manually parallelised and optimised versions compare to those using the frameworks from the OPS, Kokkos and RAJA. By examining these different libraries, the performance impact across different multi-core hardware and their performance portability can be assessed. This is especially important for OPS, which utilised techniques which have been less explored on multi-core machines, and allows for comparisons of this technique compared to both the reference application, and implementations of Tealeaf which use different techniques. The chapter goes on to analyse the performance of TeaLeaf against other multi-core systems such as Intel's Xeon Phi Knights Landing (KNL) processor and NVIDIA's Tesla P100 GPU.

An application is said to be highly performance portable if it achieves the best execution possible (or close to best) on each platform it is tested on. The chapter therefore explores the idea of performance portability through the use of multiple versions of TeaLeaf using a recently developed metric for performance portability, and analyses the achieved performance of TeaLeaf when developed

with the above programming models and frameworks [87].

The rest of the chapter is organised as follows:

- Section 4.1 briefly explores mini-applications, and discusses the development of TeaLeaf.
- Section 4.2 looks into the various implementations of TeaLeaf to achieve parallelism through different techniques.
- Sections 4.3 discusses the performance of the many versions of TeaLeaf.
- Section 4.4 discusses the performance portability relating to the systems of interest when utilising the different implementations of TeaLeaf.

4.1 Motivation

Improving the performance of large-scale, production applications is a significant undertaking. Often, these applications have been developed over decades by multiple teams, using several third party libraries. The resultant developments consist of code bases with thousands or even millions of lines of code. In many cases however, the performance is dominated by a only few units within the application. In order to overcome the issue of the scale and size of the production application, a representative program, often smaller in size, can be created to act as a proxy of the original code. A key benefit of representative applications such as this, is that they can be modified and deployed on a range of systems quickly, and subsequently implemented with multiple parallelisation models and optimised using a wide range of techniques [31].

Notable efforts in developing and using mini-apps include the NAS Parallel Benchmarks in the late 1980s [5], the ASCI applications in the 1990s [75], and more recently the Mantevo [60] and UKMAC [118] benchmark suites. Mini-apps have been developed to represent production applications from a wide range of scientific and engineering areas, including Computational Fluid Dynamics

(CFD) [5, 24, 27], particle transport [15], hydrodynamics [59, 116] and Machine Learning (ML) [106], to name just a few.

The proxy application used in this chapter is the heat conduction solver mini application TeaLeaf [64], part of the Mantevo and UKMAC suites. Martineau et al. [64, 62, 63] discuss several variants of TeaLeaf that have been parallelised using a number of programming models. Furthermore, they compare different solvers within TeaLeaf: Conjugate Gradient (CG), Chebyshev and Polynomially Preconditioned CG (PPCG), on three different Intel Xeon processors, an IBM Power8 processor, an NVIDIA Tesla K20x GPU and an Intel Knights Corner accelerator card [62, 63, 64]. TeaLeaf has also been re-engineered to use the OPS [93] embedded DSL, and the Kokkos [18] and RAJA [32] C++ template libraries.

4.2 Parallelisation of a Heat-Conduction Mini-Application

TeaLeaf is one of 15 mini-applications within the Mantevo suite [60]. The reference version has been written in FORTRAN, and includes both OpenMP and MPI parallisation methodologies that can be run independently or together. In order to make use of other parallel programming models, the application has also been converted to C/C++. In this section we detail the different versions of TeaLeaf used in our study. Section 4.2.1 describes the original reference application and a number of versions ported manually to make use of various parallel programming models. Secondly, Section 4.2.2 details the version parallelised using OPS. Finally, Section 4.2.3 describes versions parallelised by the C++ template libraries, Kokkos and RAJA.

4.2.1 Reference Implementation and Manual Parallelisations

The initial reference version of TeaLeaf employs both OpenMP and MPI to allow parallelisation on both shared and distributed memory systems. Subsequently, it has been manually ported to use other parallel programming models. TeaLeaf's CUDA ports are aimed primarily at accelerator cards. The CUDA implementation specifically targets the NVIDIA Graphics Processing Unit (GPU). Included also is an implementation that uses OpenACC directives, to offload the computation to accelerator devices such as NVIDIA GPUs. Each of these manual ports are standalone programs, replicating the full mini-app that has over 7000 lines of code, and which require maintenance by the authors of the code. The latest versions can be found on the UKMAC website and GitHub repository [119]. Listing 4.1 shows the reference version of TeaLeaf's `cg_calc_w` kernel in FORTRAN, and the manual parallelisation methodologies applied there. In this case, OpenMP has been utilised.

```
pw = 0.0_08

!$OMP PARALLEL
!$OMP DO REDUCTION(+:pw)
DO k = y_min,y_max
  DO j = x_min,x_max
    w(j, k) = (1.0_8                                &
      + ry*(Ky(j,k+1)+Ky(j,k))                      &
      + rx*(Kx(j+1,k)+Kx(j,k)))*p(j,k)              &
      - ry*(Ky(j,k+1)*p(j,k+1) +Ky(j,k)*p(j,k-1))   &
      - rx*(Kx(j+1 k)*p(j+1,k) +Kx(j,k)*p(j-1,k))

        pw = pw + w(j,k)*p(j,k)
    ENDDO
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

Listing 4.1: Reference (FORTRAN) version of TeaLeaf's `cg_calc_w` kernel with OpenMP.

4.2.2 Oxford Parallel Library for Structured-mesh solvers

OPS is a DSL embedded in C/C++ and FORTRAN consisting of a domain specific API that facilitates the development of applications operating over a multi-block structured mesh. [93] Such a mesh can be viewed as an unstructured collection of structured mesh blocks, together with associated connectivity information between blocks. Using OPS, an application developer can write a multi-block structured-mesh application using the API, in the form of calls to a traditional library. The OPS source-to-source translator is then used to parse the API calls and produce different parallelisations. A number of mini-apps have been re-engineered to use the OPS API, including CloverLeaf [71] and TeaLeaf.

OPS is able to automatically produce code that makes use of a range of parallel programming models and extensions such as OpenMP, CUDA, OpenCL, OpenACC and their combinations with MPI. The generated code attempts to use the best optimisations for the given programming model. Examples include the use of cache-blocking tiling to reduce data movement in the OpenMP and MPI versions of the generated code. [94] The key advantage of using OPS is that all these parallelisations and optimisations are produced automatically, from a single high-level source, without the need for maintaining each parallel version. Listing 4.2 demonstrates how the `cg_calc_w` kernel in TeaLeaf (shown in Listing 4.1 is implemented in the OPS version. As can be seen, the kernel is abstracted out in order for OPS to apply different parallelisation methodologies through the use of its source-to-source translator.

4.2.3 Kokkos and RAJA

Kokkos [18] and RAJA [32] are both C++ template libraries, designed with a similar goal to OPS. Through template metaprogramming, they aim to add portability to applications, whilst supporting a wider range of domains.

Both Kokkos and RAJA have unique features. Kokkos is able to select the most appropriate data layout (Array of Structures (AoS) or Structure of

```
*pw = 0.0;

ops_par_loop_tea_leaf_cg_calc_w_reduce_kernel(
  "tea_leaf_cg_calc_w_reduce_kernel",
  tea_grid,
  2,
  rangexy,
  ops_arg_dat(w, 1, S2D_00, "double", OPS_WRITE),
  ops_arg_dat(Kx, 1, S2D_00_P10, "double", OPS_READ),
  ops_arg_dat(Ky, 1, S2D_00_0P1, "double", OPS_READ),
  ops_arg_dat(p, 1, S2D_00_0M1_M10_P10_0P1, "double", OPS_READ),
  ops_arg_gbl(&rx, 1, "double", OPS_READ),
  ops_arg_gbl(&ry, 1, "double", OPS_READ),
  ops_arg_reduce(red_temp, 1, "double", OPS_INC)
);

ops_reduction_result(red_temp, pw);
```

Listing 4.2: OPS (C++) version of TeaLeaf's `cg_calc_w` kernel.

Arrays (SoA)) based on the underlying architecture. However, RAJA can use lambda functions in order to allow for more flexibility when building kernels. Both Kokkos and RAJA are able to produce optimisations with both OpenMP and CUDA, but Kokkos is able to produce a PThread version of the application, and RAJA is able to include MPI within its implementation.

Listings 4.3 and 4.4 show TeaLeaf's `cg_calc_w` kernel implemented into the Kokkos and RAJA versions respectively. Both versions use lambda and template metaprogramming to abstract the parallelisation methodologies away from the kernel, allowing them to applied by the library at compile time.

4.3 Performance of TeaLeaf

In this section, each of the different implementations of TeaLeaf outlined in Section 4.2 are executed across multiple different architectures in order to compare each of the implementations efficiencies, and to explore which frameworks and systems are able to offer the best performance. Section 4.3.1 discusses the experimental setup, whilst Section 4.3.2 explores the results for each of the dif-

```
void cg_calc_w(
    const int x, const int y, const int halo_depth, KView w,
    KView p, KView kx, KView ky, double* pw)
{
    parallel_reduce(x*y, KOKKOS_LAMBDA
        (const int& index, double& pw_temp)
    {
        const size_t kk = index % x;
        const size_t jj = index / x;

        if(kk >= halo_depth && kk < x - halo_depth &&
            jj >= halo_depth && jj < y - halo_depth)
        {
            const double smvp = SMVP(p);
            w(index) = smvp;
            pw_temp += w(index)*p(index);
        }
    }, *pw);
}
```

Listing 4.3: Kokkos version of TeaLeaf's `cg_calc_w` kernel.

```
void cg_calc_w(
    RAJALists* raja_lists, const int x, const int y,
    const int halo_depth, double* pw, double* p, double* w,
    double* kx, double* ky)
{
    ReduceSum<reduce_policy, double> pw_reduce(0.0);
        forall<policy>(
        raja_lists->inner_domain_list, [=] RAJA_DEVICE (int index)
    {
        w[index] = SMVP(p);
        pw_reduce += w[index]*p[index];
        });

    *pw += pw_reduce;
}
```

Listing 4.4: RAJA version of TeaLeaf's `cg_calc_w` kernel.

ferent libraries analysed. Finally, Section 4.3.3 discusses an analysis of each of the systems when running the proxy application.

4.3.1 Experimental Setup

The results in this section have been collected from three different, single node, multi-core/many-core systems. Each of these systems has been configured with the same set of compilers (where possible) in order to obtain comparable results. The Linux kernel and Operating System (OS) used on each system is 3.16.0-4-amd64 and Debian GNU/Linux 8 respectively. These systems can be found listed in Table 4.1. As can be seen, there are two Central Processing Unit (CPU) processors (an Intel Xeon Broadwell and a KNL), and a GPU processor (an NVIDIA P100).

System	**Key information**
Intel Xeon E5-2660 v4 (Broadwell) [43]	2 processors, each with 14 core and 2 hyperthreads per core. 2.00GHz
Intel Xeon Phi 7210 [41] (KNL)	1 processor with 64 cores and 4 hyperthreads per core. 1.30GHz, Flat memory mode, Quadrant clustering mode
NVIDIA Tesla P100 [76]	3840 single precision CUDA cores (1920 double precision CUDA cores).

Table 4.1: Systems utilised to measure the performance of the different version of TeaLeaf

Where possible, the Intel compiler (17.0u2) and Intel MPI (2017u2) were used when executing the applications on Intel hardware. For the Tesla P100 system, CUDA 8.0.61 was utilised. There were, however, two exceptions to this:

1. When using the C++ template libraries Kokkos or RAJA with CUDA, GNU 5.4.0 was employed instead of the equivalent Intel compiler;
2. When using OpenACC, the PGI compiler (17.3) and OpenMPI (1.10.6) were used in place of the Intel and CUDA compilers, to enable support for OpenACC pragma statements.

. All compilers and flags used for each implementation can be seen in Appendix A.

Some of the versions, such as OPS's CUDA, can take parameters at runtime to further optimise the program. On this implementation, the block size for the kernels can be set by the user to allow for better performance on GPUs. For this chapter, the block size has been set to $(64, 8)$ as this approach was shown to provide the greatest improvements. This was achieved through experimentation with different block sizes.

4.3.2 Results

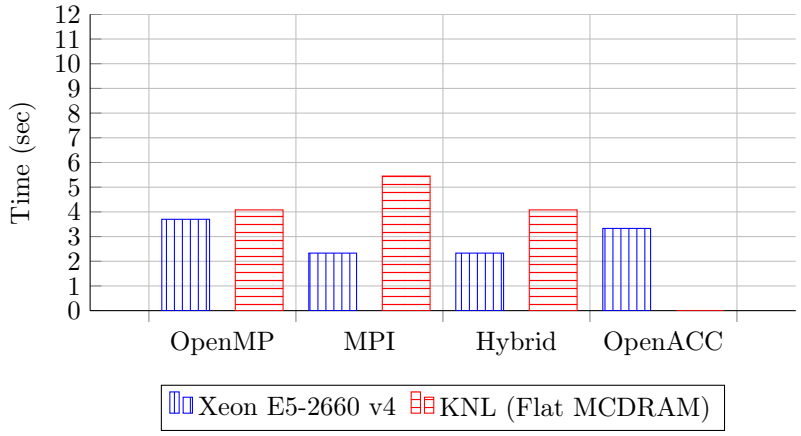
Figures 4.1, 4.2, 4.3 and 4.4 detail the performance on each system across two different problem sizes. Figures 4.1 and 4.2 present the time taken by ten iterations of the main time-marching loop of TeaLeaf solving a 2D problem size of 1000^2, whereas Figures 4.3 and 4.4 show the same but for the larger problem size of 4000^2. In Figures 4.1 and 4.3, the first four sets of columns represent results from manually parallelised versions of TeaLeaf on the Broadwell CPU and the KNL system. The next four groups are from OPS on the same systems, and the final three groups represent the C++ template libraries Kokkos and RAJA. Figures 4.2 and 4.4 show the performance of implementations capable of running on GPU architectures. The first two bars represent the manually parallelised CUDA and OpenACC implementations, the third and fourth bars represent the OPS' CUDA and OpenACC versions, and the final two bars represent the Kokkos and RAJA CUDA implementations.

The times given in Figures 4.1, 4.2, 4.3 and 4.4 are the minimum execution times given all the available options for an implementation. For example, the OpenMP versions were tested over a large range of configurations to find the optimal number of threads. Of particular note, the KNL system consists of multiple Non-Uniform Memory Access (NUMA) regions, one containing the high-bandwidth Multi-Channel Dynamic Random Access Memory (MCDRAM), and another containing the slower Double Data Rate (DDR) memory. The KNL

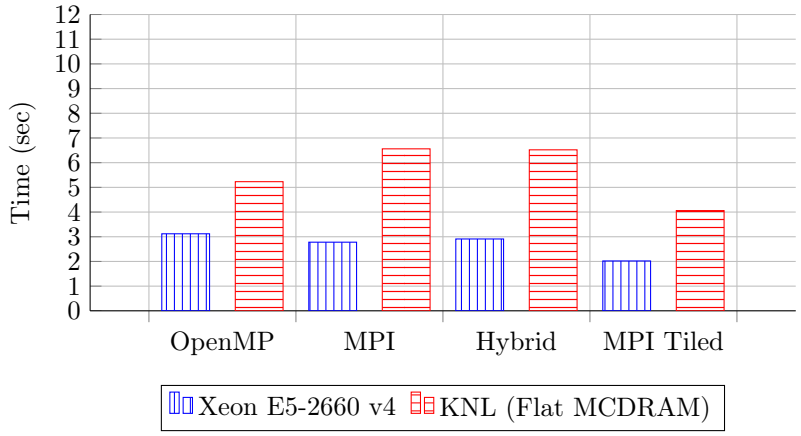
is unique in that the system can be specified at boot to reconfigure the memory into different configurations, including treating the MCDRAM as an additional layer of cache. For the purposes of this chapter, the MCDRAM for the KNL system was set up to be in flat mode, using Quadrant clustering [103]. This enables the memory to be separately addressable and allocates the memory to the closest set of processors. Our experiments showed that this configuration provided the fastest run times compared to the other memory modes. To access this memory, `numactl` was used to allocate all the memory required by the program to the MCDRAM. Should the MCDRAM run out of available memory, `numactl` would start to use the available DDR memory.

OpenMP and MPI

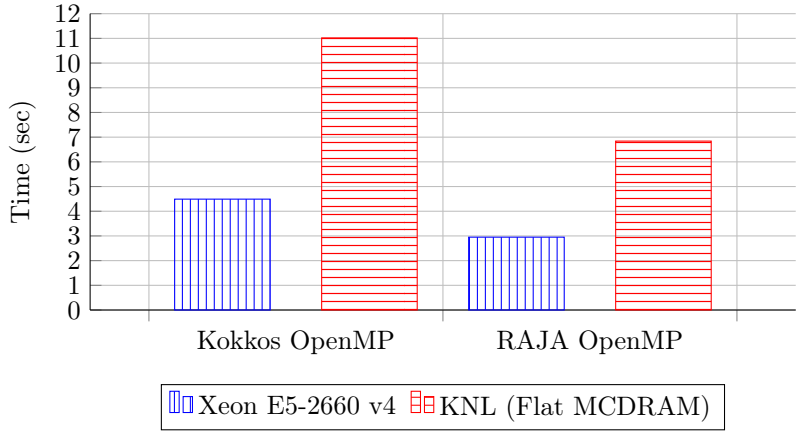
The only parallelisation model used within all the libraries tested is OpenMP. This provides an opportunity to compare each of the libraries with a consistent model. OpenMP was the slowest on all systems utilising CPU architectures when using the small problem set. The slowest two executions were achieved by Kokkos, with a runtime of 4.49 seconds on the Broadwell processor, and 11.02 seconds on the KNL. Out of all the OpenMP versions, the manual implementation of OpenMP on the KNL achieved close to the fastest time for the platform, with OPS's MPI Tiled implementation matching this performance, or performing marginally better. This was not the case when looking at the larger dataset, where the manually parallelised version of OpenMP achieved the worst time out of any implementation when run on the Xeon. However, this appears to be an outlier, being almost 3× slower than any other implementation. In particular, the manually parallelised version using MPI is almost always faster than its OpenMP counterpart. NUMA issues may be contributing to part of this performance degradation, but it is apparent that further optimisations may be required for the manual OpenMP version to improve performance. The best OpenMP performance on the KNL system for the larger dataset is given by the



(a) Manual

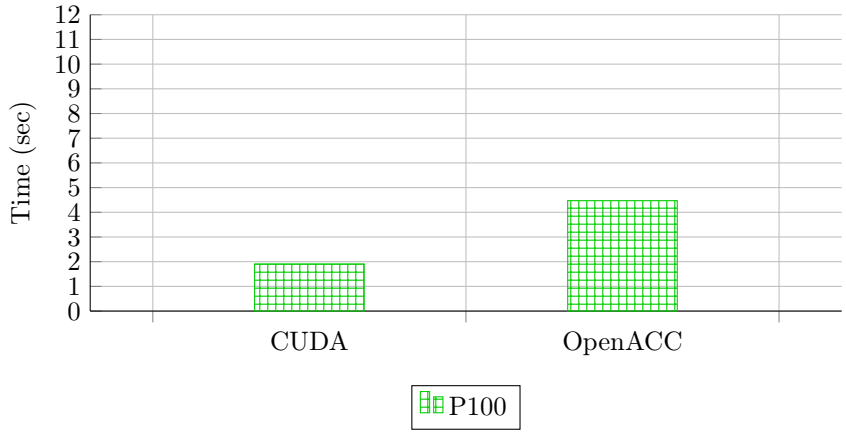


(b) OPS

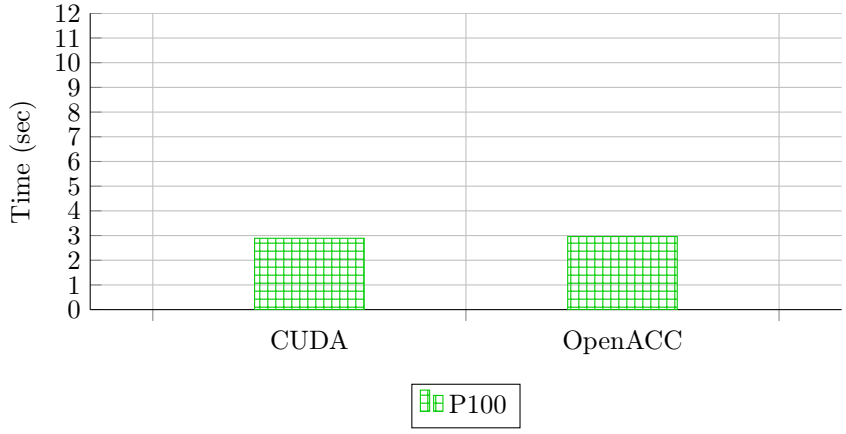


(c) Kokkos and RAJA

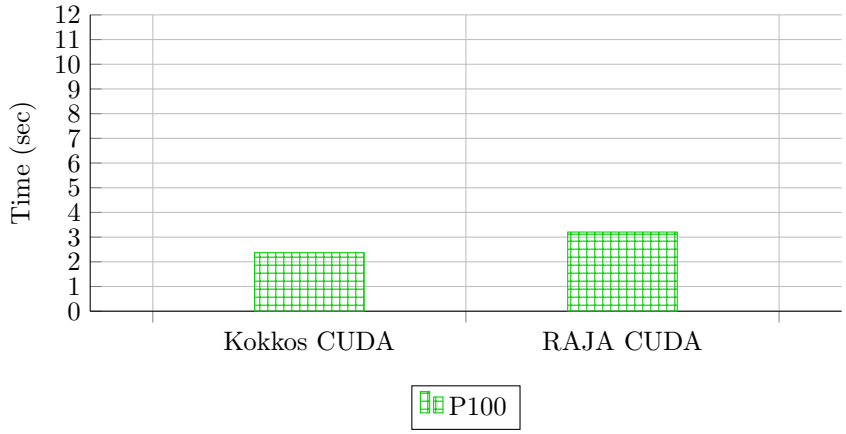
Figure 4.1: Times for TeaLeaf using 1000^2 dataset on CPU systems



(a) Manual

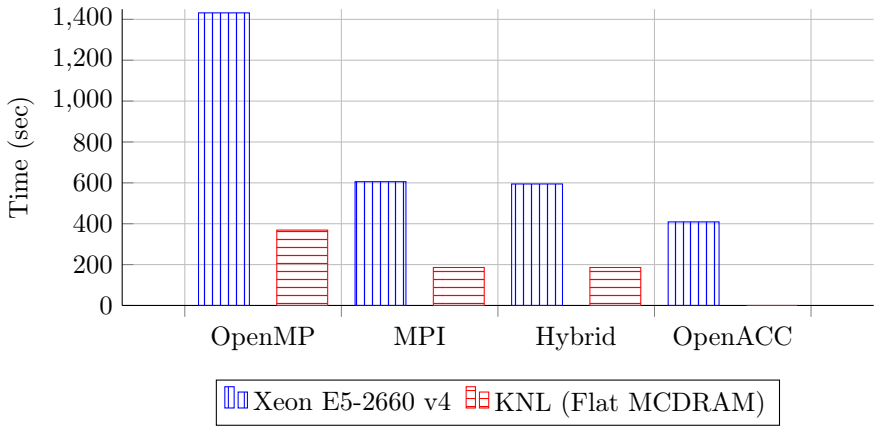


(b) OPS

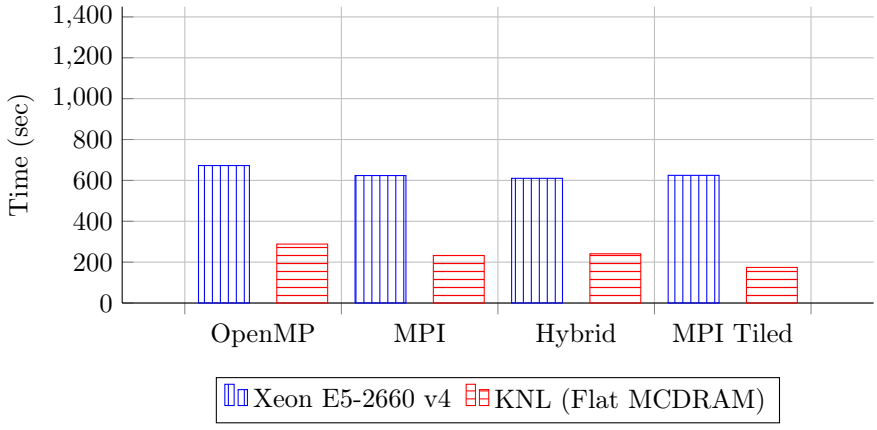


(c) Kokkos and RAJA

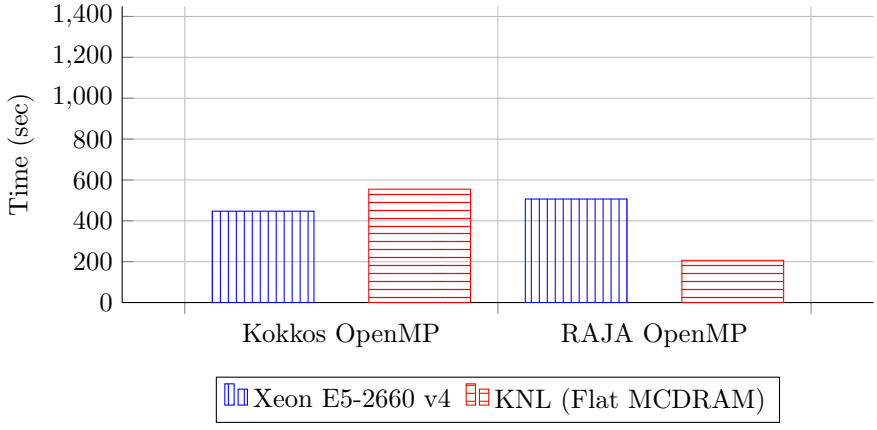
Figure 4.2: Times for TeaLeaf using 1000^2 dataset on GPU systems



(a) Manual

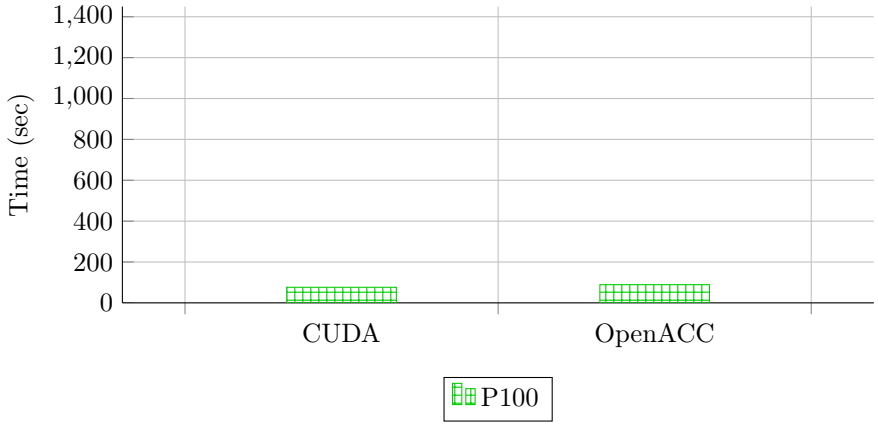


(b) OPS

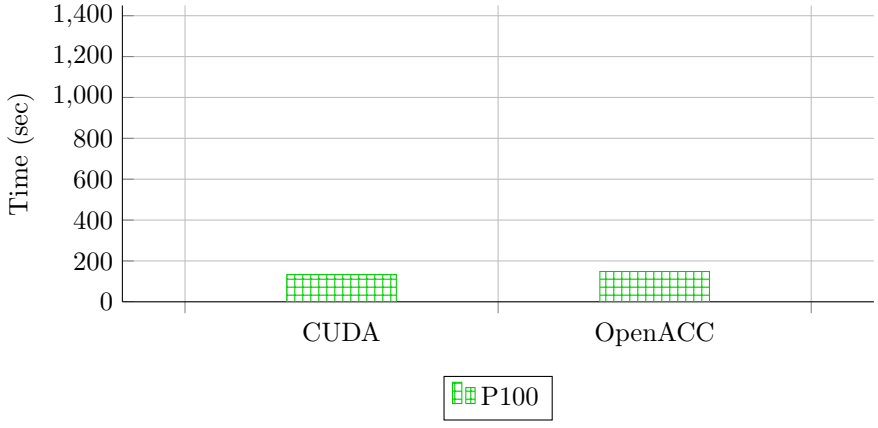


(c) Kokkos and RAJA

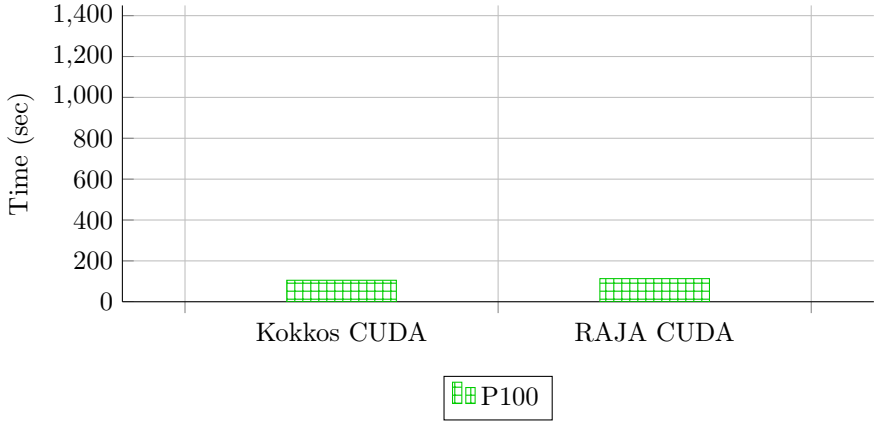
Figure 4.3: Times for TeaLeaf using 4000^2 dataset on CPU systems



(a) Manual



(b) OPS



(c) Kokkos and RAJA

Figure 4.4: Times for TeaLeaf using 4000^2 dataset on GPU systems

version using the RAJA library.

Most of the frameworks used to parallelise TeaLeaf included an MPI implementation. All of the MPI implementations tested also contain an option to use OpenMP alongside MPI. With MPI+OpenMP, TeaLeaf often performed better than the equivalent, OpenMP only implementation. OPS allows the user to generate code with specific optimisations on top of the MPI+OpenMP parallelisation. One such optimisation allows for cache-blocking tiling to reduce data movement [94]. The tiling optimisation made the code faster than the equivalent OpenMP and MPI+OpenMP implementations without tiling. This is especially true for the KNL system, where it gained the fastest time for the small dataset and the second fastest for the larger dataset.

RAJA and Kokkos

Out of all of the OpenMP implementations tested on the CPU architectures, RAJA gave the best performance on the small dataset using the Broadwell system, and the large problem size on the KNL. In contrast, the Kokkos implementation was often the slowest out of all OpenMP implementations, the exception being the large dataset being run on the KNL system.

While Kokkos' OpenMP implementation of TeaLeaf may not perform well on either the Broadwell processor or the KNL system, the CUDA version does perform better on NVIDIA's Tesla P100 GPU. For both problem sets, the Kokkos implementation was faster than the OPS and RAJA versions designed for GPUs. However, the fastest variant of TeaLeaf on a GPU was the manually parallelised implementation using CUDA.

For both the small and large problem sizes, RAJA's CUDA implementation was slower than both the manually implemented CUDA version and the Kokkos implementation. Using the larger dataset, RAJA CUDA was quicker than all of the OPS implementations. However, the same cannot be said for the smaller dataset, where it was slower than all implementations of OPS running on the P100 system.

OpenACC

Another parallelisation model designed primarily for GPU compilation is OpenACC. Two OpenACC implementations were tested on the P100 GPU. One generated using OPS and one which was manually implemented. For the larger problem set, the manually parallelised OpenACC implementation performed very well, achieving the second fastest time, running on the graphics card. However, both OpenACC implementations were slower than the Kokkos CUDA implementation using the smaller dataset. When using both datasets, the CUDA implementations of TeaLeaf was faster than the OpenACC counterparts.

As well as offloading to the GPU, OpenACC can offload to the host processor. This means that the CPU can do all of the processing that would be executed on the GPU. Currently, OPS's OpenACC implementation does not support offload to the host device, so this was tested using the manually parallelised version of TeaLeaf OpenACC. For the smaller dataset, the OpenACC implementation on CPUs performed marginally better than the manually parallelised OpenMP and Kokkos versions. It was, however, slower than both OPS's and RAJA's OpenMP implementations. Regarding the larger problem size, the manually parallelised OpenACC version worked extremely well, with the best performance of any implementation on the Broadwell. OpenACC cannot offload to a KNL as a host device using the PGI 17.3 compilers, so could not be tested with the OpenACC implementation.

4.3.3 System Analysis

Of the two Intel architectures considered, performance on the Broadwell was generally greater than the KNL when the smaller problem size was used. With the 1000^2 dataset, the application requires in the region of 200 MB of memory; for the 4000^2 dataset, this increases to 2.5 GB. Analysing the caching behaviour for the two cases shows that the Broadwell system has a third of the cache misses of the KNL for the small dataset. For the larger dataset, the KNL has less cache

misses, and less cache accesses overall. The application is memory-bound (as will be seen in Section 4.4.1) and the MCDRAM therefore increases performance.

The P100 specific implementations are generally more performant than those that can be run on either the Broadwell or KNL systems, when using the large problem set. However, the percentage difference between the fastest time on a GPU compared to the fastest on a CPU is not as large when the smaller dataset is used (3.04% for the small dataset, 50.57% for the larger dataset). This is an expected performance trait of GPUs where smaller problem sizes benefit less from the increased parallelism available. Overheads (as a proportion of total run time), such as kernel calls and memory copies, further reduce performance when working on smaller problem sizes.

4.4 Performance Portability

Performance portability has been a topic of interest within High Performance Computing (HPC) community for some time; the United States Department of Energy (DoE)'s Centers of Excellence Performance Portability Meeting was set up specifically to discuss how to mitigate the problems with platform diversification and how different laboratories are working on the issue. During and following the April 2016 meeting, an attempt was made to establish a more concrete definition of performance portability. *Performance* and *Portability* are subjective terms, heavily dependent on the user's point of view and the problem being solved [55]. One similarity in all definitions was the inclusion that a performance portable code should be able to run on a variety of machines. There have been many different approaches to arrive at a solution to this, including compiler directives such as OpenACC [97] and OpenMP, languages designed for performance portability such as Chapel [101] and PetaBricks [89], execution models such as EARTH [129], and the utilisation of embedded DSLs such as OPS [93] and Oxford Parallel Library for Unstructured mesh solvers (OP2) [92]. Template libraries have also been used to add performance portability to an ap-

plication, examples of which include Kokkos [18] and RAJA [32].

Assessing the portability of a particular program is usually carried out by measuring performance on multiple machines and then comparing the results. Quantifying the "performance portability" of an application from these results is difficult. To remedy this, Pennycook et al. [87] proposed the metric:

$$P(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

where, H is the set of systems used to test the application, and e is the efficiency of the application a given the input parameters p [87]. The metric uses the harmonic mean to assess either:

1. the application efficiency, i.e., how fast the application runs compared to the best time on each system;

2. the architecture efficiency, i.e., the achieved number of floating point operations per second compared to the maximum possible on each system.

The resultant score ranges between 0% and 100%; should the program not be portable to one or more systems, a score of 0% is achieved.

In this chapter, the metric is used to evaluate the different versions of TeaLeaf. Because the systems tested fall under two distinct architectures: CPUs and GPUs, two sets of performance measures have been taken. The first considers the CPU architectures only and the second looks at all available systems. This means that some of the implementations of TeaLeaf can be compared to the other implementations even though an implementation cannot be run on a particular system.

In order to compare all versions effectively, the manually parallelised implementations have been combined together into one version, which will be referred to as "Manual". When calculating the performance portability, the best performing implementation was then used for the architecture and the application

efficiency. Note that the implementation that achieves the best architecture efficiency may not also achieve the best application efficiency and vice versa. This is because, whilst an implementation might use all the available hardware, it may not use it effectively. Thus, the implementation would have a high architectural efficiency, but low application efficiency. The opposite may also occur, where an implementation may run the fastest out of all implementations on a given architecture, but may not fully utilise the hardware.

When calculating either the application or the architecture efficiency, the equation stated in Chapter 2.4 was utilised. To effectively represent the architecture efficiency, we calculated two metrics. The first is the achieved number of Floating Point Operations per Second (FLOP/s) (i.e. compute intensity) for each parallelisation and the second is the memory bandwidth used. Both measures were obtained using, Intel's *VTune 2017* profiler for the CPU systems, and NVIDIA's CUDA profiler `Nvprof` for the GPU systems. It should be noted that all of the results recorded used the larger 4000^2 dataset.

Sections 4.4.1 and 4.4.2 explore the performance portability of both the architecture efficiency and application efficiency respectively. In each of these sections, the relevent efficiencies are presented, along with the performance portability result for all CPU architectures (Broadwell and KNL) and the performance portability across all three architectures.

4.4.1 Architecture Efficiency

Tables 4.2 and 4.3 show the computational and memory bandwidth efficiencies respectively, as well as the performance portability results, across all tested hardware and frameworks. From these results, it can see that the compute efficiency accounts for a significantly smaller proportion of the system peak, on all systems. Barely 5% of the peak is achieved. However the bandwidth efficiency is mostly over 50%. As such, it is clear that TeaLeaf is a memory-bound application. Therefore, the rest of this section will concentrate only on the memory bandwidth results presented in Table 4.3.

Version	**Broadwell (%)**	**KNL (%)**	$P(\mathbf{CPU})$	**P100 (%)**	$P(\mathbf{CPU} \cup \mathbf{GPU})$
Manual	0.96	1.52	1.18	2.36	1.42
OPS	1.35	3.39	1.93	2.83	2.16
Kokkos	2.73	1.57	2.00	5.30	2.52
RAJA	0.91	1.60	1.16	1.87	1.33

Table 4.2: Computational architectural efficiency (%) and Performance Portability (P) on Xeon Broadwell, KNL (MCDRAM) and a P100 card for the larger dataset (4000^2)

Version	**Broadwell (%)**	**KNL (%)**	$P(\mathbf{CPU})$	**P100 (%)**	$P(\mathbf{CPU} \cup \mathbf{GPU})$
Manual	60.49	91.61	73.19	75.70	74.01
OPS	89.61	95.93	92.66	61.21	79.11
Kokkos	64.11	23.59	34.49	65.86	41.00
RAJA	53.13	60.87	56.74	70.63	60.72

Table 4.3: Memory bandwidth architectural efficiency (%) and Performance Portability (P) on Xeon Broadwell, KNL (MCDRAM) and a P100 card for the larger dataset (4000^2)

With the exception of Kokkos on the KNL, the amount of memory bandwidth used by the different parallelisation models exceeds 60%. The highest bandwidth usage was achieved by OPS on the KNL, utilising 95.93% of the available bandwidth. When looking specifically at the KNL results, the amount of memory bandwidth used correlates with the application efficiency, with models using more bandwidth gaining the higher application efficiency. This is acceptable, as it would be expected that a faster program would better utilise the hardware available. Across all of the CPU architectures, OPS achieved the highest bandwidth, and thus gained the largest performance portability for CPU systems.

Considering the results for the Tesla P100 system, the bandwidth efficiency is relatively high, and spread over a small range (14.49% in range for the GPU, compared to 36.48% for Broadwell and 72.34% for KNL). As with the KNL system, the fastest implementation achieved the highest bandwidth. However, unlike the KNL system, the highest bandwidth utilisation was achieved by the

manually parallelised implementation. This leads to both the manually parallelised and OPS versions having very close performance portability based on the architecture efficiency (74.01% and 79.11% respectively).

4.4.2 Application Efficiency

Version	**Broadwell (%)**	**KNL (%)**	$P(\mathbf{CPU})$	**P100 (%)**	$P(\mathbf{CPU} \cup \mathbf{GPU})$
Manual	100.00	93.73	96.76	100.00	97.82
OPS	67.02	100.00	80.26	57.32	70.81
Kokkos	91.45	31.40	46.74	72.65	53.05
RAJA	80.73	84.25	82.45	67.46	76.77

Table 4.4: Application efficiency (%) and Performance Portability (P) on Xeon Broadwell, KNL (MCDRAM) and a P100 card for the larger dataset (4000^2)

Table 4.4 shows the application efficiency and performance portability of this metric across all tested implementations and architectures. When analysing these results, it can be seen that nearly all the results on the CPU architectures are greater than 80%. The exceptions are OPS on the Broadwell (67.02%), and Kokkos on the KNL (31.40%). These low results are reflected in the performance portability metric for the CPU, where Kokkos is approximately 34% away from the next highest performance portability score across all CPU architectures.

As stated previously, almost all the other implementations of TeaLeaf performed very well, achieving above 80% efficiency. This is reflected in the performance portability metric, with the highest being 96.76% utilising the manual implementation. Both OPS and RAJA achieved very similar performance portability scores across both CPU architectures, with only a 2.19% difference.

However, very few implementations gained a high application efficiency when executed on the P100 system. The manually parallelised versions were the fastest, with Kokkos coming in second with a 72.65% application efficiency.

Due to the low performance portability on the CPU architectures, Kokkos' overall performance portability for application efficiency was the lowest of all

the frameworks assessed, scoring 53.05%. On the other hand, the manually parallelised implementations scored the highest of all models, being the only one to score above 90%. This very much aligns with the intuition that manually optimising and parallelising the code will achieve the best results, even if longer development time is required. Both OPS and RAJA achieved lower performance portability once the GPU architecture was included.

4.5 Summary

In this chapter, the performance of different implementations of TeaLeaf, a mini-application that solves the linear heat conduction equation, was investigated. First, the performance of the mini-app across 3 different multi-core systems: Intel's Xeon E5-2660 v4 CPU; Intel's Xeon Phi Knights Landing processor; and, NVIDIA's Tesla P100 GPU was explored. This showed that the GPU implementations of the different frameworks were faster for larger datasets, with the KNL system closely behind. The best times on the CPU were achieved by the manually parallelised OpenACC implementation and the MPI tiled implementation of OPS.

Secondly, the *performance portability* of different version of TeaLeaf was examined. Overall, the architecture efficiency based on compute intensity was significantly low. However, this was expected, as real-world programs such as TeaLeaf, are usually more complex than traditional benchmarking applications such as LINPACK, which are typically designed to stress the hardware fully. On the other hand, architecture efficiency based on the bandwidth was almost always over 50%, leading to the confirmation that TeaLeaf is a memory bound application.

OPS's architectural efficiency, based on bandwidth, was the highest on CPU architectures. However, for the GPU systems, the manually parallelised version utilised a higher percentage of the peak bandwidth. Overall, both OPS and manual implementations achieved comparable architecture efficiencies.

In terms of application efficiency, the manually parallelised implementations achieved the highest scores, showing that hand-coding of the parallelisations and optimisations will typically produce better results. However, the downside to this method is the need to develop and maintain each separate version. Of all the library based methods, both OPS and RAJA produced good performance results, achieving above 70% overall performance portability.

CHAPTER 5

Creation, Development, Implementation and Optimisations of a Data Structure Abstraction Library

Over recent years, there has been a noticeable shift in the development of new, High Performance Computing (HPC) architectures. The disparity between processor speeds and memory speeds has resulted in an increased focus on the performance of the memory subsystem, as demonstrated by the rising use of high-bandwidth memory in newer Central Processing Unit (CPU) processors such as ARM Fujitsu's A64FX [25] and Graphics Processing Unit (GPU) accelerators such as NVIDIA's A100 [78]. This development is necessary, as it will close the gap between the speed of performing data read and writes compared with the speed of floating point operations. Without this type of development, applications which relied upon large amounts of data movement between the processor and its memory would not see any significant increase in performance when executing on newer architectures. However, even with an improvement in memory performance, the memory efficiency is often lower than its compute counterpart [52]. Combine this fact with the increasingly complex data structures used within applications, and the need for code to be performance portable [50], it can be seen that the structure of the data becomes incredibly important for ensuring high memory and application performance.

This chapter will discuss the design and implementation of a library, which will allow data structures to be abstracted away from applications and algorithms, Warwick Data Store (WDS). There are two benefits to be gained from designing and implementing in such a way. Firstly, large-scale changes to the data structure can be performed without the need for a significant proportion of the program to be re-written. These can range from restructuring the data for

the code, to placing the data onto a different level in the memory hierarchy, or altering the existing data structure. The changes specified here are set by the user. Secondly, the data structures can be tweaked for different applications and hardware, thus enabling better utilisation of the available hardware. Unlike the first benefit, these changes are made within the library itself. To demonstrate the need for a library such as the one described, the focus has been placed on the challenges faced by different physics applications.

The WDS is a template C++ library, designed to replace hard-coded data structures in applications [49]. The resultant library provides a means whereby data structures can be altered and optimised, without the risks associated with large-scale changes to the code. Alongside this, further functionality can be provided that would otherwise be difficult to introduce. Examples of this include the ability to easily switch between different data structures (for example, changing the data structure from Structure of Arrays (SoA) to Array of Structures (AoS) or vice versa, without the need for large, complicated code), and being able to change the data adjacency's of given variables (for example, swapping from row-major to column-major indexing, and vice versa). The changes described above would need to be achieved with the smallest possible cost to an application's performance.

The aim of this chapter is to demonstrate the creation, development and implementation of the data-structure abstraction library, WDS; and thus demonstrating the key criteria of WDS as extensibility, minimal size, ease of implementation and minimal performance impact on an application, as well as additional functionality provided in the library. The rest of the chapter is laid out as follows:

- Section 5.1 discusses similar systems, and how WDS differs from these.
- Section 5.2 explores the initial implementation of the library, and discusses the issues and lessons learnt whilst developing this version of the library.
- Section 5.3 explores the final implementation of the library, going into

detail around the three different areas of the library, the high level functionality (Section 5.3.1), the data storage classes (Section 5.3.2) and the data access classes (Section 5.3.3).

- Section 5.4 discusses the different features achieved by WDS.
- Section 5.5 explores optimisations made to the data structures implemented within the library, in order to ensure minimal performance impact. In addition to the above areas, the implementation of new, specialised data structures is also explored.

5.1 Motivation

WDS is a template C++ library designed to allow for the abstraction of data structures within applications [49]. By adopting this approach, the data structure can be manipulated and optimised without the need to change the program's code. Additional functionality can be provided through the library, that would otherwise be complex, time-consuming, and bespoke to a given program. An example of this is the conversion of variables between different data structures. When developing WDS, the focus was on the core functionality alongside four key criteria: extensibility, minimal size of library, ease of implementation into applications, and the performance impact of the library. By ensuring these four criteria are met alongside the core functionality, it can be ensured that the library maximises its effectiveness and usability.

Two contemporary projects which provide the capability to work with abstract data layouts are Kokkos [18] and RAJA [32]. The goal of both of these tools is to facilitate the development of performance-portable applications that can execute on a wide range of hardware and achieve good performance. Throughout the development of WDS, data storage and manipulation was the primary focus. The aim for WDS is to inject domain-specific knowledge into the library, and make the conversion between different layouts a key feature.

The functionality provided by WDS is therefore orthogonal to that offered by Kokkos and RAJA.

Other specialised data storage libraries have also been designed, but these have been for specific use cases. One such example of this is Atlas, designed by the European Centre for Medium-Range Weather Forecasts (ECMWF) [16]. This library is designed to store unstructured mesh data within climate and weather simulations, and provides a variety of layout options depending on the type of discretisation used. WDS aims to support a wider range of applications.

Another example is Axom, developed by the Lawrence Livermore National Laboratory (LLNL) [56]. This project aims to provide tools for multi-physics applications, with one such tool being a data management tool called Sidre. Sidre's aim is to allow for transparent data accesses for physics applications across a large range of hardware options [57]. Sidre provides similar functionalities and capabilities as WDS and was developed at the same time, but independently to WDS. Sidre's development shows that there is a demand for a library that performs these data structure abstractions.

Libraries have also been created which abstract the data layout, allowing for auto-vectorisation, more utilisation of bandwidth and, as a consequence, higher performance. The most commonly used library for this is Intel's Single Instruction - Multiple Data (SIMD) Data Layout Templates (SDLT). [46] While the main aim of this library is to manipulate the data in order to increase performance, WDS aims to extend the available features, such as the ability to convert between data structures, and to allow more flexibility in how the data is defined. One example of WDS' flexibility is that domain-specific data structures can be created within the library, such as those required for multi-material physics applications. [23]

5.2 Initial Implementation

The initial idea for the WDS library was to have a class which could store all the data for a given variable. This would then be accessed through an object which would store all the variables relating to a given program. It would then be possible to set up a collection of these objects stored in a single data-repository object. By taking this approach, operations which only affect a given level of abstraction could be implemented without the need for large, sprawling changes to the library. C++ was used as the key programming language, as it allows for large amounts of flexibility, whilst being incredibly performant and accessible to a large range of HPC platforms.

The structure of the initial implementation of the WDS can be seen in Figure 5.1. The following key has been used to denote key sections of the library:

- **Bold** lines are interactions between the users applications and the library.
- Green sections are classes which store data. These classes also have functionality to allow access to the data.
- Red sections are classes which manage interactions between the users applications and the library. As such, these are often referred to as controller classes.
- The blue class is the view class, which allows for quicker access to the data, and can be passed to the kernel rather than relying on going through the controller classes.
- Orange sections are interfaces between the library, C and FORTRAN.

One of the first sections to be developed was the storage classes. These classes were used to store all the data relating to a given data structure. This data structure could hold multiple variables (such as AoS), or contain a single variable such as an array (for example, from an SoA data structure). The key class within this collection was the `OBJ` class. The primary aim of this class was

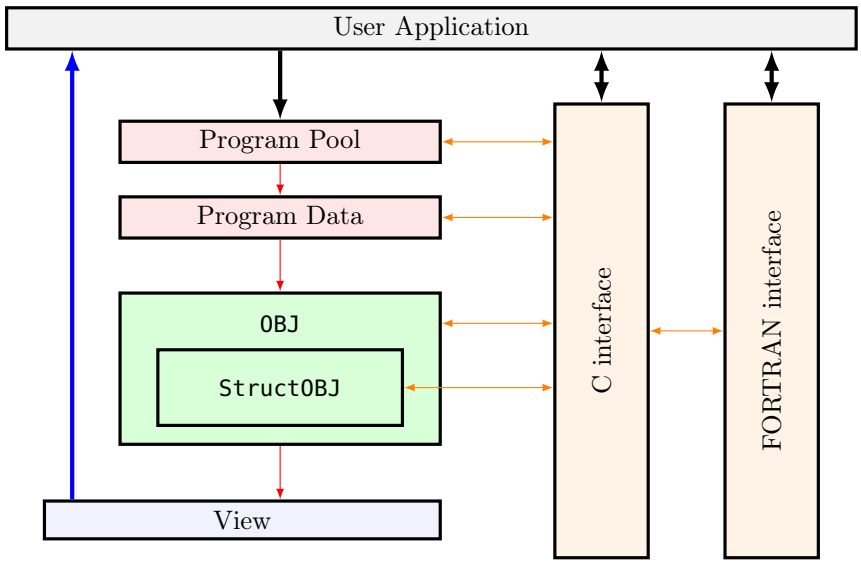


Figure 5.1: Graphical representation of the original structure and control flow of WDS

to store important information about the data structure, such as the variable name, number of items, and variable class information such as the types name and size. It also managed a void pointer to the block of memory where the data is stored. A void pointer stores the beginning address of a block of memory, but does not store the type of data, and therefore cannot be iterated through in the same way an array would be. To overcome this issue, the library used the size of the data class (and the data structure if the structure contained more than one variable), to perform pointer arithmetic to calculate the location of a given element within the data block.

Listing 5.1 shows the key variables used within the `OBJ`. As can be seen, the data is stored in the `dat` pointer, which is allocated memory depending on the arguments passed to the object at construction. `datName` and `datClass` store the name of the variable, and the name of the type of variable being stored. In the initial specification, the guidelines on how these variables should be used was very relaxed, and could be set to anything that was required by a particular data structure, so long as it could be interpreted by a different part of the library. `datClassSize` stores the number of bytes for a given element in the data block,

```
class OBJ {
private:
/* The data stored. */
        void * dat;

/* The variable name for the block of data. */
        string datName;

/* The name of the class for the block of data. */
        string datClass;

/* The size of the class for the block of memory */
        int datClassSize;

/* The number of elements stored in the block of memory */
        int datNumItems;

/* Stores whether or not the block of memory is a structure. */
        bool isStructure;

/* Rest of the OBJ object */
};
```

Listing 5.1: Key variables within `OBJ`

whilst `datNumItems` stores the number of elements in the data block. The `OBJ` can also contain variables required for all other data structures implemented into the library. For example, in Listing 5.1, a variable is used to say whether the data stored is in a structure (`isStructure`).

The `OBJ` class also managed all operations on the data, as well as the functionality needed for the interfaces between the data and other languages. This functionality included extending or shrinking the data structures, printing out a representation of the contents of the data structure, as well as testing if the object is empty and generating `View` objects. The function signatures for these can be seen in Listing 5.2. It should be noted that template functions have been utilised to allow for compilation optimisations (as the compiler can then know what types are required, and how to interpret these requests), but only in cases where the data is being directly accessed. This allows for type checking to be achieved, without the class having to be converted to a template class.

```
/* Sets an element in the block of data to a new value. */
template <class T> bool setData(T newDataItem, int place=-1);
template <class T> bool setData
                (T * newDataItems, int numItems, int place=-1);

/* Provides the data stored in the block. */
template <class T> T * getData();
template <class T> T getData(int place);

/* Removes an element of data from the block. */
bool removeData(int element);
bool removeData(int * elements, int count);

/* Adds a piece of data to the block. */
template <class T> bool addData (T newDataItem, int place = -1);
template <class T> bool addData
                (T * newDataItems, int numItems, int place = -1);

/* Adds a piece of data to the block. */
bool add (int element);
bool add (int * elements, int count);

/* Returns the name of the block of memory. */
string getName();

/* Returns the name of the class of the block of memory. */
string getClassName();

/* Returns the size of the class of the block of memory. */
int getClassSize();

/* Returns the number of elements stored in the block of memory. */
int getNumItems();

/* Frees the data in object. */
bool freeDat();

/* Returns whether the block of memory is classed as "empty" */
bool isEmptyOBJ();

/* Prints out the OBJ */
void print();

/* Generates a View object */
template <class T> View<T> getView(string varName = "");
```

Listing 5.2: Key functions within `OBJ`

When initially implemented, the `OBJ` class was only meant to handle SoA data structures. However, in order to allow AoS data structures to be incorporated, the metadata stored within `OBJ` was extended to keep track of whether it was a structure or an array, and could store specific data relating to the structure. This structure-specific metadata was stored in a separate class, called `StructOBJ`. In this class, a name, the number of elements, and the variable class name and size could be stored, as well as a vector (a specialised array which can have its size changed) of `StructOBJ` objects. As such, the initial level stored a specific name for the structure, the overall size of the structure, and the number of elements at this given level. If we use Figure 3.4 and Listing 3.11 in Chapter 3.4.2 as an example, the `OBJ` object would contain the name of `foo_bar_baz`, a size of 13 (8 bytes for a double, 4 bytes for an integer and 1 byte for a char), the number of elements would be set to 40, and a void pointer, pointing to a block of memory 520 (13×40) bytes long. In the vector of `StructOBJ`, three objects would be found, containing data relating to one of the three variables `foo`, `bar` and `baz`. Due to its recursive design, Array of Structures of Arrays (AoSoA) data structures could also be easily represented in this format, along with any combination of AoSoA, SoA and AoS.

Listing 5.3 shows the key variables and function signatures within `StructOBJ`. Applying the above example to this `StructOBJ` interface, `structName` would be `foo_bar_baz`, the `className` variable would be left blank, `numItems` would equal 40, `structSize` would equal 13, and `innerStruct` would contain an array of three `StructOBJ` objects. The functions within this class allow for key information and data to be retrieved and set, depending on what is required. Functionality such as generating `View` objects and storing the data is left to the `OBJ` class. This allows for the `StructOBJ` class to only contain the functionality that allows for the interpretation of the data block.

In order to manage the potentially large number of variables that could be stored within an application, and to control potential high-level functionality, two controller classes were developed called `ProgOBJ` (referred to as Program

```
class StructOBJ {
private:
/* The name of the structure */
  string structName;
/* The class of the element */
  string className;
/* The number of items at this element */
  int numItems;
/* The size of the structure/element from this point */
  int structSize;
/* A vector that stores the structure tree. */
  vector<StructOBJ> innerStruct;

public:
/* Constructors */

  string getName() { return structName; }
  string getClassName() { return className; }
  int getNumItems() { return numItems; }
  int getSize() { return structSize; }
  int getInnerStructSize() { return innerStruct.size(); }
  bool isInnerEmpty() { return innerStruct.empty(); }

/* Gets the StructOBJ object at a given position. */
  StructOBJ getInnerStructOBJ(int pos);

/* Sets the pointer to a given position. */
  static void * setPointer
    (void * data, int offset, int totalIncrements, int totalSize);

/* Inserts some data into a set pointer. */
  template <class T> static bool insertDataPtr
    (void * setDataBlock, T data);

/* Gets some data in a set data block. */
  template <class T> static T getDataPtr (void * setDataBlock);

/* Finds a given variable within the structure. */
  bool findInStruct (string varName);

/* Sets the number of items at a given variable name. */
  bool setNumItems (string varName, int newNumItems);

/* Gets the amount of bytes to the beginning of a variable. */
  int getOffset(string varName, int * places, int numLevels);
  vector<int> getOffsets(string varName, int offset);
};
```

Listing 5.3: Key variables and functions within `StructOBJ`

Data in Figure 5.1) and `ProgPool` (referred to as Program Pool in Figure 5.1). `ProgOBJ` was designed to store all variables for a given collection of kernels. As such, this class stored the `OBJ` objects in a vector, along with another vector storing the data structure type for the given `OBJ` object. In order to identify the `ProgOBJ` object later, a name field was used to act as a unique key, in the same way the name field in the `OBJ` object was used. `ProgPool` was designed to control all the different `ProgOBJ` objects, and to manage all user interactions between the library. As such, the only data this class stored was a vector of the `ProgOBJ` objects. From this class, the user application could request `OBJ` to be created or removed from given `ProgOBJ`, or make a request for a print out of details relating to a particular variable, data structure or a collection of data structures. Data can also be examined and set through this interface, though due to the fact that at least two searches would be required for each request (one to find the specific `ProgData` object, and one to find the specific `OBJ` within this, and potentially more if the data structure is either AoS or AoSoA), this was not the preferred method.

Instead of accessing data through the library interface, a specialised object could be requested. Thus, the view class was created, which became the preferred method for accessing data quickly. The class could be requested through the library interface for a particular variable, and would be generated by the corresponding `OBJ` class. `View` objects were only allowed to be created through an `OBJ` object, as this contained all the required information such as the pointer to the data, the structure of the data, and the amount of data. A reference to the corresponding `OBJ` object was also stored in the `View` object. The `View` class was built as a template class (a class which can take a type of data as a parameter when first created), to ensure that the data was being consistently accessed and to reduce the amount of pointer arithmetic the library was required to undertake.

Initially, the `View` class would access all the data through the `OBJ` functions. However, this proved to be a slow process as the data could not be immediately

accessed. This, therefore, caused a level of indirection, meaning that the compiler could not perform optimisations on the data. As such, the `View` class was designed so that these calculations could be performed within the required functions, with as few steps as possible. In order to access the data, the functionality of both `[]` and `()` operators were overridden, and set to access the data in the correct pattern. This proved to be more efficient, and because this interface is closer to how native C/C++ data structures are implemented, re-implementing a program to use the library was simpler.

In order to allow for the library to be used by a wide range of HPC applications, an interface section was built. This allowed the library to be used by both native C and FORTRAN applications, alongside the C++ interface. These were designed such that the overall functionality of WDS can be provided across as many different applications as possible. However, the functionality for the C and FORTRAN interfaces had to be limited due to the restrictions on the languages. Because the C language does not have classes or objects, access to the data could not be achieved by passing the `View` object to the application, as the application would not know how to interpret the object. As such, multiple functions outside the `ProgPool` were created to access the functionality that the class provided. As well as classes and objects, C does not allow for templates. To resolve this issue, functions were duplicated to allow for specific types of data to be added, removed and accessed by the library. In particular, these types were doubles, integers and long integers.

The FORTRAN interface had similar constraints due to the way it had to be implemented. In order to interface the FORTRAN and C++ languages, the C language has to be used as an interim step. This meant that the same restrictions on the type of data had to be applied. The types of data also had to be limited in FORTRAN, as an integer in FORTRAN could be a different size than the integer in C/C++. As such, the FORTRAN interface had to use specific types in order to ensure the data was being read correctly. However, newer versions of FORTRAN allow for classes and objects to be created and

passed between functions. Theoretically this could have allowed view objects containing the required information to be generated. This was not carried out due to the fact that all the required information would have to be passed through the C interface and then built (meaning that any changes to the `View` class would have to be reproduced in the FORTRAN view class), and also because classes within FORTRAN are not well supported by FORTRAN HPC compilers or applications at the time.

Due to these limitations, the FORTRAN interface for WDS was very verbose. The majority of this interface can be seen in Listing 5.4. This interface contains both functions and subroutines that allow for all the functionality of the WDS to be utilised, as long as the data was either a `INTEGER` or a `REAL` type. This includes the creation and deletion of program pools and variables, as well as expanding and shrinking variables and accessing data within requested variables. In order for the FORTRAN interface to be connected to WDS, `ISO_C_BINDING` had to be used. This allows for functions and subroutines in FORTRAN to be connected to C functions with the same function signature, by using the `BIND` keyword. To ensure the function signatures are correct, `ISO_C_BINDING` provides types for each C primitive, which can be used to set the corresponding FORTRAN type to the same size. This is done through the use of the keyword `KIND`. The interface for accessing `INTEGER` types is shown within Listing 5.4, but not the version for `REAL` types. This is because both use the similar function names, but with different function signatures.

Whilst this version of the library showed that some key points could be achieved, there were a number of issues. One of the most important issues was the way in which the variables were stored and accessed. Due to all the data structure information having to be contained within a single class, if the data structure was to become more complex than a simple SoA/AoS, then the class would become far more complex and difficult to ensure the performance impact would be minimal. The functionality would also become slower and slower, as there would need to be more checks to determine how to perform certain

```
! The Fortran interface with WDS.
MODULE WDS
  USE ISO_C_BINDING
  INTERFACE
! Create a program in the pool
    FUNCTION createProgram(newProgName) BIND (C, name="createProg")
      CHARACTER(KIND=C_CHAR)  :: newProgName(*)
      LOGICAL(KIND=C_BOOL)    :: createProgram
    END FUNCTION createProgram
! Finds the position of a program in the pool
    FUNCTION findProgram(progName) BIND (C, name="findProg")
      CHARACTER(KIND=C_CHAR)  :: progName(*)
      INTEGER(KIND=C_INT)     :: findProgram
    END FUNCTION findProgram
! Checks whether a program exists
    FUNCTION programExists(progName) BIND (C, name="progExists")
      CHARACTER(KIND=C_CHAR)  :: progName(*)
      LOGICAL(KIND=C_BOOL)    :: programExists
    END FUNCTION programExists
! Deletes the pool
    FUNCTION destroyProgramPool() BIND (C, name="destroyProgPool")
      LOGICAL(KIND=C_BOOL)    :: destroyProgramPool
    END FUNCTION destroyProgramPool
! Prints the entire pool
    SUBROUTINE printPool () BIND (C, name="printPool")
    END SUBROUTINE printPool
! Prints all details about a program.
    SUBROUTINE printProg (progName) BIND (C, name="printProg")
      CHARACTER(KIND=C_CHAR)     :: progName(*)
    END SUBROUTINE printProg
! Prints all details about a variable.
    SUBROUTINE printOBJ (progName, varName) BIND (C, name="printOBJ")
      CHARACTER(KIND=C_CHAR)     :: progName(*)
      CHARACTER(KIND=C_CHAR)     :: varName(*)
    END SUBROUTINE printOBJ

! -- DELETE --
! Removes an element of data from a variable.
    FUNCTION removeInOBJ_Val(progName, varName, posInOBJ) BIND (C, name="removeInOBJ_Val")
      CHARACTER(KIND=C_CHAR)     :: progName(*)
      CHARACTER(KIND=C_CHAR)     :: varName(*)
      INTEGER(KIND=C_INT), VALUE :: posInOBJ
      LOGICAL(KIND=C_BOOL)       :: removeInOBJ_Val
    END FUNCTION removeInOBJ_Val
! Removes a collection of elements of data from a varaiable.
    FUNCTION removeInOBJ_Ptr(progName, varName, posInOBJ, countPos) BIND (C, name="removeInOBJ_Ptr")
      CHARACTER(KIND=C_CHAR)     :: progName(*)
      CHARACTER(KIND=C_CHAR)     :: varName(*)
      INTEGER(KIND=C_INT)        :: posInOBJ(*)
      INTEGER(KIND=C_INT), VALUE :: countPos
      LOGICAL(KIND=C_BOOL)       :: removeInOBJ_Ptr
    END FUNCTION removeInOBJ_Ptr

! -- INTEGER --
! Adds a new variable to a program.
    FUNCTION addOBJ_Int_Val(progName, newData, varName, dataNumItems) BIND (C, name="addOBJ_Int_Val")
      CHARACTER(KIND=C_CHAR)     :: progName(*)
      INTEGER(KIND=C_INT),VALUE  :: newData
      CHARACTER(KIND=C_CHAR)     :: varName(*)
      INTEGER(KIND=C_INT), VALUE :: dataNumItems
      LOGICAL(KIND=C_BOOL)       :: addOBJ_Int_Val
    END FUNCTION addOBJ_Int_Val
! Adds a new variable to a program.
    FUNCTION addOBJ_Int_Ptr(progName, newData, varName, dataNumItems, isMalloc) BIND (C, name="addOBJ_Int_Ptr")
      CHARACTER(KIND=C_CHAR)       :: progName(*)
      INTEGER(KIND=C_INT)          :: newData(:)
      CHARACTER(KIND=C_CHAR)       :: varName(*)
      INTEGER(KIND=C_INT), VALUE   :: dataNumItems
      LOGICAL(KIND=C_BOOL), VALUE  :: isMalloc
      LOGICAL(KIND=C_BOOL)         :: addOBJ_Int_Ptr
    END FUNCTION addOBJ_Int_Ptr
! Adds a new variable to a program.
    FUNCTION addOBJ_Int_Ptr_2D(progName, newData, varName, dataNumItems, isMalloc) BIND (C, name="addOBJ_Int_Ptr_2D")
      CHARACTER(KIND=C_CHAR)        :: progName(*)
      INTEGER(KIND=C_INT)           :: newData(:,:)
      CHARACTER(KIND=C_CHAR)        :: varName(*)
      INTEGER(KIND=C_INT), VALUE    :: dataNumItems
      LOGICAL(KIND=C_BOOL), VALUE   :: isMalloc
      LOGICAL(KIND=C_BOOL)          :: addOBJ_Int_Ptr
    END FUNCTION addOBJ_Int_Ptr_2D
! Adds a data element to a variable
    FUNCTION addToOBJ_Int_Val(progName, newData, varName, posInOBJ) BIND (C, name="addToOBJ_Int_Val")
      CHARACTER(KIND=C_CHAR)     :: progName(*)
      INTEGER(KIND=C_INT),VALUE  :: newData
      CHARACTER(KIND=C_CHAR)     :: varName(*)
      INTEGER(KIND=C_INT), VALUE :: posInOBJ
      LOGICAL(KIND=C_BOOL)       :: addToOBJ_Int_Val
    END FUNCTION addToOBJ_Int_Val
! Adds a collection of data elements to a variable
    FUNCTION addToOBJ_Int_Ptr(progName, newData, sizeOfData, varName, posInOBJ) BIND (C, name="addToOBJ_Int_Ptr")
      CHARACTER(KIND=C_CHAR)     :: progName(*)
      INTEGER(KIND=C_INT)        :: newData(*)
      INTEGER(KIND=C_INT), VALUE :: sizeOfData
      CHARACTER(KIND=C_CHAR)     :: varName(*)
      INTEGER(KIND=C_INT), VALUE :: posInOBJ
      LOGICAL(KIND=C_BOOL)       :: addToOBJ_Int_Ptr
    END FUNCTION addToOBJ_Int_Ptr
! Sets a data element within a variable.
    FUNCTION setInOBJ_Int_Val(progName, newData, varName, posInOBJ) BIND (C, name="setInOBJ_Int_Val")
      CHARACTER(KIND=C_CHAR)     :: progName(*)
      INTEGER(KIND=C_INT),VALUE  :: newData
      CHARACTER(KIND=C_CHAR)     :: varName(*)
      INTEGER(KIND=C_INT), VALUE :: posInOBJ
      LOGICAL(KIND=C_BOOL)       :: setInOBJ_Int_Val
    END FUNCTION setInOBJ_Int_Val
! Sets a collection of data elements to a variable
    FUNCTION setInOBJ_Int_Ptr(progName, newData, sizeOfData, varName, posInOBJ) BIND (C, name="setInOBJ_Int_Ptr")
      CHARACTER(KIND=C_CHAR)     :: progName(*)
      INTEGER(KIND=C_INT)        :: newData(*)
      INTEGER(KIND=C_INT), VALUE :: sizeOfData
      CHARACTER(KIND=C_CHAR)     :: varName(*)
      INTEGER(KIND=C_INT), VALUE :: posInOBJ
      LOGICAL(KIND=C_BOOL)       :: setInOBJ_Int_Ptr
    END FUNCTION setInOBJ_Int_Ptr
! Gets a data element from a variable.
    FUNCTION getInOBJ_Int_Val(progName, varName, posInOBJ) BIND (C, name="getInOBJ_Int_Val")
      CHARACTER(KIND=C_CHAR)     :: progName(*)
      CHARACTER(KIND=C_CHAR)     :: varName(*)
      INTEGER(KIND=C_INT), VALUE :: posInOBJ
      INTEGER(KIND=C_INT)        :: getInOBJ_Int_Val
    END FUNCTION getInOBJ_Int_Val
! Gets a collection of elements from a variable.
    SUBROUTINE getInOBJ_Int_Ptr(progName, varName, posInOBJ, countPos, dataReturn) BIND (C, name="getInOBJ_Int_Ptr_F")
      CHARACTER(KIND=C_CHAR)           :: progName(*)
      CHARACTER(KIND=C_CHAR)           :: varName(*)
      INTEGER(KIND=C_INT)              :: posInOBJ(*)
      INTEGER(KIND=C_INT), VALUE       :: countPos
      INTEGER(KIND=C_INT), INTENT(OUT) :: dataReturn(*)
    END SUBROUTINE getInOBJ_Int_Ptr
! Gets all the elements within a variable.
    SUBROUTINE getAllInOBJ_Int(progName, varName, dataReturn) BIND (C, name="getAllInOBJ_Int_F")
      CHARACTER(KIND=C_CHAR)           :: progName(*)
      CHARACTER(KIND=C_CHAR)           :: varName(*)
      INTEGER(KIND=C_INT), INTENT(OUT) :: dataReturn(*)
    END SUBROUTINE getAllInOBJ_Int

! -- DOUBLE --
! Repetition of INTEGER functions and subroutines

    END INTERFACE
END MODULE
```

Listing 5.4: FORTRAN interface for the original implementation of WDS

operations. There would also be a knock on effect to the `View` class, which would have to become more complex in order to handle the different cases. Due to the amount of data accesses within a kernel, any additional computation would slow the program down dramatically. The structure of the library also became difficult to control, as different pieces of required data were stored in different places. As such, the library was redesigned whilst retaining key lessons learnt from this implementation, and improving on the design and flexibility.

5.3 Library Structure

Whilst the initial implementation showed that the creation of a data structure abstraction library was achievable, some of the key principles were not realised in this implementation. One of the key principles that was not achieved in the initial implementation was to allow for flexibility in creating new data structures, whilst ensuring that the performance was not severely impacted. This needed to be achieved, whilst ensuring the interfaces between different data structures remained consistent. When developing the current version of the library, it was key to ensure these objectives for the library was met, alongside the minimal size and the ease of implementing the library into an existing code.

Figure 5.2 shows an overview of the current library structure, split across multiple sections. These are:

- **Bold arrows** are interactions between the user and the library.
- **Bold text** are substitutes for different data structure names. **A** and **B** are not the only data structures within the library, but show how key interactions between classes of the same and different data structures occur.
- Red elements are high level functionality classes, or interactions managed by high level classes.
- Green elements are classes that manage the storage to variables, as well as any large changes to the data structures such as expansion or shrinking

of the variables data structure.

- Blue elements are classes that manage fast data accesses between the user application and the library.

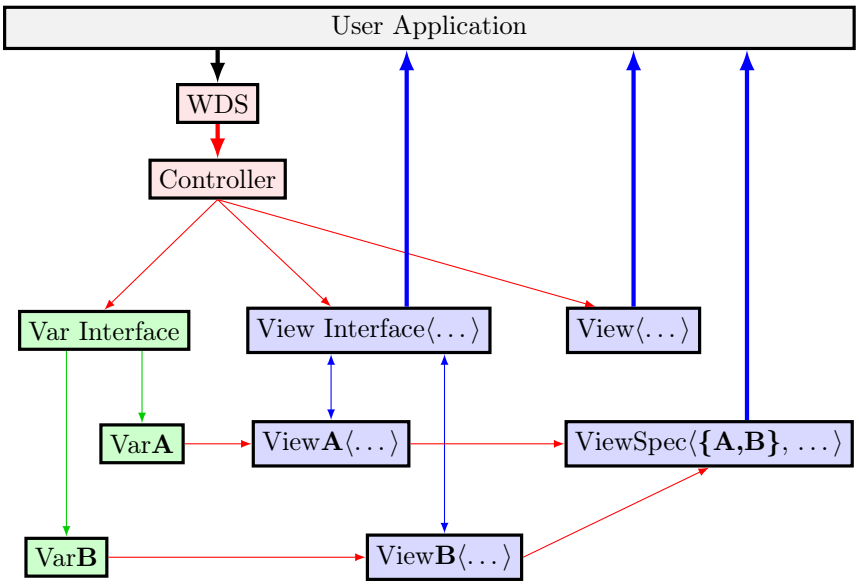


Figure 5.2: Graphical representation of the structure and control flow of the final version of WDS

Comparing this list to the original library implementation shows that some of the key principles have remained the same. In particular, that there is collection of high-level classes, a collection of low level memory management classes and a collection of data access classes. By dividing the library in this way, it is possible to extend one type of class without impacting the functionality of the others. This allows for new data structures to be implemented with relative ease. In the following sections, each one of these collections will be explored in turn (high-level functionality in Section 5.3.1, data storage classes in Section 5.3.2 and data access classes in Section 5.3.3).

5.3.1 High-Level Functionality Classes

In order to control and manage the variables stored within the library, and to efficiently build tools that can alter or change a variables data structure, col-

lection of high-level functionality classes were created. This resulted in other classes being more focused on specific tasks, and meant that the addition of more high-level functionalities did not rely on the modification of a large amount of classes. Instead, only one or two classes had to be altered. Similarly to the initial implementation, there are two classes in this collection, each performing similar functionality. These are the `Controller` class (as a replacement to the Program Data class) and the `WDS` class (replacing the Program Pool class). Each of these classes controlled different functionality within the library. This split ensured that the appropriate functionally could be developed without changing the interface for implemented functionality within the library. In addition, by not allowing the user direct access to the variable management classes, an extension of the data structures available to the user can be achieved, without affecting applications which do not require this facility. Because this abstraction can view all variables and possible data structures, adding and extending high-level functionality becomes simpler, compared to when this was carried out within, or between, specific programs.

The `Controller` class deals with user interactions and transformations within variables, including the management thereof. Focusing on the management, the class contains a vector of a structure called `Variable`. This is a different class to those used to store the variable data. In this `Variable` structure, a pointer to a given data structure variable object, and a value representing the data structure are both stored. This arrangement is required owing to the way in which the data storage classes are implemented; a list of variables with data structure in their native classes is not possible, as this would mean that differently sized classes would be stored in the same list. To resolve this problem, the variable object stored is denoted as the parent class, the interface which they inherit from. However, this means that some of the data is lost. So, to ensure correct interactivity with the variables, the data structure type is stored with the variable object, to ensure that it can be cast into the correct class before it is utilised. Once this is done, the other functionalities of the class, the

transformations, can then be correctly applied to the variable.

The `WDS` class controlled all functionality that was undertaken externally to variables, such as the movement of variables and changing variables data structures. However, unlike its predecessor, the `WDS` class does not store multiple `Controller` classes for any longer than required. Instead, it retains a single `Controller` object, and generates new ones only when new variables are required that might clash with the functionality of the permanent `Controller` object. An example of this is where two variables with the same key are required, even if one is later going to be deleted. This method is used when a variable or collection of variables is changed to a new data structure. The `WDS` class also acts as the main interface between the user application and the library.

Due to the simplification of the library by removing the concept of multiple `Controller`/`ProgData` objects, the user interface can be stream-lined. Listings 5.5 and 5.6 show the differences between the C++ interface for the initial implementation of WDS, and the final version. The key difference is that the data is stored within the object (called `data` in Listing 5.6), rather than globally with a key to access specific sections (`"example"` in Listing 5.5). In addition, the final implementation utilises a `wds` namespace (which ensures all the WDS classes are contained and do not conflict with other classes) and only a single file has to be included for the entire library, rather than one for each class.

```
#include "Prog_Pool.hpp"
#include "View.hpp"

int main() {
  createProg(string("example"));

/* Rest of code here... */

  return 0;
}
```

Listing 5.5: Initialisation step for a WDS based application using the initial implementation

```
#include <wds.hpp>

int main() {
  wds::WDS data = wds::WDS();

/* Rest of code here... */

  return 0;
}
```

Listing 5.6: Initialisation step for a WDS based application using the final implementation

5.3.2 Data Storage Classes

The variable collection of classes controls the allocation and management of the data itself. These classes are referred to as `Var` within the library. Each of the data structures implemented within WDS is created within a separate class in this collection of classes, and inherits from the parent interface. This ensures the data structures are sandboxed from each other, and that the high level classes can access all of the necessary functionality. These variable classes include everything that would be needed for the data structure, except for how the user accesses the data. This action is managed by the final collection of classes, the views.

The `Var` class acts as an interface on which data structure classes can be based. This ensures that, no matter which data structure is required for a given application, the high-level classes are able to create, store and alter variables using these data structures. It was a requirement that each data structure class had to be capable of implementing the following functionalities:

- Sanitise meta data - This would ensure that the meta data is in the correct format for the data structure, thus it can be easily interpreted by the variable class and its corresponding view class. This may involve combining multiple meta data objects into one (or vice versa), altering the dimension data or ensuring that the data adjacency list is correct.

- Check that a given variable reference is located within the current data structure. This may be complex, as it may be located within the inner data structure, or multiple variable references may be merged.

- Add/Remove elements - This determines how the data structure can be expanded or contracted given how many elements are required to be added/removed; the dimension in which this is to occur; and the position in which this occurs within the given dimension. The functions may also alter the dimension list to better reflect the current state

- Calculate the number of elements within a given data structure.
- Generate `View` and `ViewInterface` objects - This builds a `View` and `ViewInterface` object, depending on what is requested. Through the `View` functionality, only information present within the meta data object can be passed. The functionality for building `ViewInterface` objects is more flexible, as it is defined by the corresponding view for the given data structure.
- Delete variable name - This variable manages how a variable will be deleted within a data structure. For some data structures such as SoA, this is fairly trivial (the array with the variable can be deleted). However, for others such as AoS or AoSoA, this is more complicated (the variables data is interleaved with other variables, so cannot be easily removed).

The `Var` class also stores the meta data object, ensuring that all data structures and the variables using them have the meta data object.

As well as the ability to determine how variables are created and stored, this collection of classes can also determine the storage of data relating to a variable. Two classes in particular are central to managing the meta-data of a variable or collection of variables. The first is the `DataStoreType`. This class manages the storage of a value corresponding to a set list of data structures. As such, this is often used within all the sections to determine which data structure is being used by a given variable. To make this as small and optimal as possible, a C++ enum class is used for `DataStoreType`. The second class is the `VarMeta` class. This manages the storage of all the data that may be required by a `Var` class, including:

- Variable Name - Used to identify a given `Var` or `View` object. The default for this is an empty string, but should always be set
- Type of data structure stored - Used to identify the data structure stored if required. Whilst this would be the same as the `Variable` structure

stored in the `Controller` object, it is stored here to ensure that the user application can query this, if required. It is also used to ensure the data has been checked and is ready for a given data structure (discussed later). As such, the default state for this is undefined

- Class name - Used to validate that the `View` is passed the correct class. The default for this is an empty string, but should always be set
- Class size - Used both to validate the `View` is passed the correct class, and to calculate where in a memory block a given piece of data resides. By default, this is set to zero, but should always be set
- List of dimensions - Used to identify how the data can be accessed, and how to calculate the position of a given element of data. By default, this is set to a single element list with the number of elements in the variable
- Data adjacency list - Used to determine where an element of data is stored, by determining the order in which the dimensions should be used. By default, this is set to be in the order the dimensions are initialised as
- Inner structure (stored as a vector of `VarMeta` objects) - Used to store recursive meta data. This is usually left empty

The `VarMeta` class also has a few functions, such as calculating the total size of a given variable based on the dimensions. Due to its importance, the class is used by all three categories for different purposes. The high-level section uses this class to search for the correct variable. The data access section uses this class to determine how the data should be accessed, and the data storage section uses the class to determine the way in which a given variable is modified or built.

As discussed in previous sections, the data storage classes should not be able to be accessed by the user. As such, the only interactions should be through the user interface. In the initial interface, all the elements of the data structure had to be given in the manner in which they needed to be created. This was fine for SoA data structures, but made constructing more complex data structures

such as AoS and AoSoA more difficult. The example in Listing 5.7 shows how two SoA variables `a` and `b` can be created. If the user required a more complex data structure utilising the initial data structure, then the arguments within the `addOBJ` function would have to be completely modified. The other issue is that the initial implementation of WDS was not able to handle multi-dimensional arrays. Instead, it had to be implemented as a 1D array with a mapping. This can be seen in variable `b`, which is a 2D array of double values with dimensions 1000×4.

```
#define NEL 1000
#define NSHAPE 4
/* Creation of variables "a" and "b" in Program Pool "example" */
addOBJ("example", int(), "a", NEL);
addOBJ("example", double(), "b", NEL*NSHAPE);
```

Listing 5.7: Addition of variables `a` and `b` to WDS using the initial implementation

Listing 5.8 shows how the final interface accomplishes the same as the original interface as shown in Listing 5.7. The key differences is that, rather than multiple interfaces for different data structures, only one interface is required. A list of different variables is created by passing `VarMeta` into the `addMeta` function. Each variable is contained within a `VarMeta` object. These objects are not processed until WDS is told which data structure is to be built from the given `VarMeta` objects.

```
#define NEL 1000
#define NSHAPE 4
/* Creation of WDS variables "a" and "b" */
data.addMeta(wds::VarMeta("a", {NEL}, int()));
data.addMeta(wds::VarMeta("b", {NEL, NSHAPE}, double()));
data.buildVar{WDS_DT::SOA};
```

Listing 5.8: Addition of variables `a` and `b` using the `addMeta` and `buildVar` method within WDS' final implementation

To build the data structure, and to generate the required specialised `Var` objects, the `buildVar` function should be used. This takes all the `VarMeta` objects previously added to the data store, calculates how these should be combined for the required data structure, and generates the required specialised `Var` object(s). The data structure is decided by the user, by passing in a `DataStoreType` (`wds::DataStoreType` has been aliased to `WDS_DT` through type definitions) value to the `buildVar` function. Unlike the initial interface, if the data structure needs to be changed, then this argument needs to be changed and the code recompiled.

5.3.3 Data Access Classes

The view collection of classes controls how the user accesses the data stored within the library. These classes intentionally use a very similar design to the variable collection of classes. When a variable class is created for a new data structure, at least one new view class should also be created. This enables a data structure with multiple parts to be represented easily, as each part would be accessed through its own view class. Each of these view classes inherit from a common interface, to allow for the same flexibility as the variable collection, and to allow this interface to be passed to the user. The interface is defined in the appropriately-named `ViewInterface` class. This arrangement allows for the appropriate view class to be called, whilst being data structure agnostic.

Whilst the view interface is a good way to access the variable without worrying about its data structure, it comes at the cost of the code using VTable lookups. This means that, for every data access, the application has to lookup how to access the data and thus adds unnecessary computation whenever a data access is required. To overcome this issue, two separate view classes have been created. The first, simply named `View`, is designed to provide data access for any data structure with a uniform striding pattern, and as such, is data structure agnostic. This is very common within data structures, as it allows for memory optimisations such as cache prefetching. The second class is `ViewSpec`. This class allows for direct access to the original view, without having to use

VTables. To do this, each of the functions are inlined within specific versions of `ViewSpec`, defined through template specialisation. As such, the data structure is required to be passed to the library by the user in order for this to be used.

Both `View` and `ViewSpec` are designed to have common functions to access the data. Specifically, both classes override the functions `operator[](int i)`, `operator()(int i)` and `operator()(int i0, int i1)`. These versions allow for array accesses within applications to be readily replaced with WDS views. The only change would be to replace `[]` with `()` where appropriate. In more complex data structures within WDS, a distinction is made between `operator[](int i)` and `operator()(int i)`. The first function accesses the raw pointer without calculation, whereas the second calculates where the element at that position should be. This allows for data structures which contain irregular data access patterns to be accessed differently, depending on the scenario presented by the algorithm. All of this ensures that WDS can be easily implemented into existing applications.

The simplicity of adding WDS to an application can be seen through the development of the interface. Listings 5.9 and 5.10 show the function-based interface used predominately for C and FORTRAN programs (Listing 5.9) and templated view based interface (Listing 5.10). The function-based interface required large amounts of adjustments to the kernels, which meant that each access would require two searches (one for the correct Program Pool, and one for the variable within the Program Pool). The template view interface allows for fewer changes to the kernel, as the `operator[]` function can be used on the object, making it resemble an array. However, accesses were limited to 1D, so if a variable was multi-dimensional (such as `b` in these examples), then additional calculations would have to be carried out within the kernel. In addition, due to the limitations of the initial implementation of WDS; the `operator[]` function called a corresponding function within the `OBJ`, meaning that multiple levels of indirection and function calling had to be taken for each data access.

```
for (int i = 0; i < NEL; i++) {
    double total = 0.0;
    for (int j = 0; j < NSHAPE; j++) {
        total += getInOBJ<double>("example", "b", i*NSHAPE+j);
    }
    setInOBJ("example", (int) (total*total), "a", i);
}
```

Listing 5.9: Calculations using `a` and `b` through WDS' initial function-based interface

```
/* Creation of View objects */
aView = getView<int>("example", "a");
bView = getView<double>("example", "b");

/* Calculation loop using only access function */
for (int i = 0; i < NEL; i++) {
    double total = 0.0;
    for (int j = 0; j < NSHAPE; j++) {
        total += bView[i*NSHAPE+j];
    }
    aView[i] = (int) (total*total);
}
```

Listing 5.10: Calculations using `a` and `b` through WDS' initial view-based interface

The final implementation and interface for WDS, as shown in Listings 5.11 and 5.12, demonstrate the improvements to the library against the initial implementation. Both code examples utilise multi-dimensional access directly though inlined functions, minimising the amount of function and memory indirection. Therefore, both `View` and `ViewSpec` can be compiled more effectively. The only difference between the two examples is that the first (Listing 5.11) uses the generic view class, whereas the second (Listing 5.12) uses the specialised view class. Where the variables are AoS, AoSoA or (as they are in this case) SoA, either technique will provide equivalent results. However, in more specific cases, such as the ones described in Section 5.5.3, Listing 5.11 may not be viable and only Listing 5.12 can be used.

```
/* Creation of generic view (wds::View) objects */
auto aView = data.getView<int>("a");
auto bView = data.getView<double>("b");

/* Calculation loop using natural access function */
for (int i = 0; i < NEL; i++) {
    double total = 0.0;
    for (int j = 0; j < NSHAPE; j++) {
        total += bView(i, j);
    }
    aView(i) = (int) (total*total);
}
```

Listing 5.11: Calculations of `a` and `b`, using the `View` objects from WDS' final implementation

```
/* Creation of specialised view (wds::ViewSpec) objects */
auto aView = data.getViewSpec<int, WDS_DT::SOA>("a");
auto bView = data.getViewSpec<double, WDS_DT::SOA>("b");

/* Calculation loop using natural access function */
for (int i = 0; i < NEL; i++) {
    double total = 0.0;
    for (int j = 0; j < NSHAPE; j++) {
        total += bView(i, j);
    }
    aView(i) = (int) (total*total);
}
```

Listing 5.12: Calculations of `a` and `b`, using the `ViewSpec` objects from WDS' final implementation

5.4 Library Features

Due to the way in which the library has been developed and structured, the data structure can be abstracted in such a way that a kernel does not need to know the underlying data structure. This can be seen in Listing 5.13, where the variables `a` and `b` are set up in WDS and passed to the kernel, without the kernel knowing the underlying data structure. In addition to its intended function, the library is able to provide further functionality that would otherwise be difficult and time-consuming to implement, and which would be bespoke to the application. These additions include extensions to low level functionalities such as more control over the way in which data structures are expanded and shrunk (the dimension, position and size can be specified).

```
#include <wds.hpp>

int main(int, char **) {
//Create data store
    wds::WDS datastore;
//Specify the variables "a" and "b"
    const int len = 250, depth = 4;
    datastore.addMeta("a", {len}, double()));
    datastore.addMeta("b", {len, depth}, int()));
//Build the variables given the metadata provided, specifying the
//layout desired. In this case, SOA has been specified.
    datastore.buildVar(WDS_DT::SOA);
    {
//Views used to access and modify data
        auto a = datastore.getView<double>("a");
        auto b = datastore.getView<int>("b");
//Kernels operate on views without knowledge of the underlying
//layout
        kernel(a, b, len, depth);
    }
    return 0;
}
```

Listing 5.13: C++ example of how WDS can be used to pass two variables `a` and `b` into kernel, without kernel knowing the data structure.

Enhancements to high level functionality are also presented by WDS, such as the ability to extend the functionality of current data structures and to implement new data structures. This will be discussed later in Section 5.5.3. In this section, two difficult functionalities are described. These are: the ability to convert a variable or a collection of variables from one data structure to another (Section 5.4.1), and the ability to change the adjacency of data within a given variable (Section 5.4.2).

5.4.1 Conversion of Variables

The way in which the library has been developed and designed, functionality that would otherwise be difficult to implement can be provided with quickly to the application developer. This additional functionality ranges from the ease of implementing new data structures, to additional functionality that can change and manipulate existing variables. One of the extra features is the ability to convert a variable or collection of variables from one data structure to another. To do this, the `WDS` object builds two lists. The first containing all the meta data of the variables that are to be converted, and the second containing `View<void>` objects relating to the variables to be converted. The `View<void>` class is a specialisation of the generic `View` class, where the data type is set to void. Without this specialisation, the `View` object would treat the data as a void type when requesting data, potentially resulting in issues arising when the data is read. Due to this, and because the void datatype is very rarely used with respect to variables, the functionality of `View<void>` has been overridden to provide a specific data type (an unsigned char), and to provide a 1D access pattern across each element and between each element.

Once these two lists are generated, a new, temporary `Controller` object is built to store and maintain the new variables. In this temporary object, the new variables based on the required data structures and the list of meta data previously generated. From this, a collection of new `View<void>` objects relating the the new variables is generated. Both lists have the variable keys

in the same order, to minimise searching through for corresponding keys. The data is then copied from the old view to the new, iterating through every single byte and placing it in the new variable. The old variables are then removed from the permanent `Controller` object using the `removeVar` functionality (as discussed in Section 5.3.2), and the new variables are migrated from the temporary `Controller` object to the permanent one (as discussed in Section 5.3.1), before the temporary object is destroyed.

This whole process is carried out within the library, with the user required to provide very little in the way of information. In fact, the user is only required to provide information relating to which data structure they would like the variable(s) to be converted to, and the list of variable keys to be converted. A full example of this process being used can be seen in Listing 5.14. In this example, variables `a` and `b` are created as SoA variables, and the kernel is executed with these variables. The variables are then converted to an AoS data structure (which would interleave one value of `a` with four values of `b`), and the kernel is executed again, unaware of the change.

As discussed in Section 5.3.3, the `View` object has some limitations. Even with the `View<void>` specialisations, the `View` object is not designed to handle data structures that do not, or may not have a uniform access pattern. In these cases, the developer of a data structure that would like to include this functionality has two options. The first is to build a specialisation of the corresponding `ViewSpec` class to act in the same manor as `View`. The other is to implement the functionality within the necessary `Var` class. Whilst the first option is the easiest to implement, the second allows for a faster conversion. The latter option has been used for the specialised data structure conversion discussed later in Chapter 7.

```
#include <wds.hpp>

int main(int, char **) {
//Create data store
    wds::WDS datastore;
//Specify the variables "a" and "b"
    const int len = 250, depth = 4;
    datastore.addMeta("a", {len}, double()));
    datastore.addMeta("b", {len, depth}, int()));
//Build the variables given the metadata provided, specifying the
//layout desired
    datastore.buildVar(WDS_DT::SOA);
    {
//Views used to access and modify data
        auto a = datastore.getView<double>("a");
        auto b = datastore.getView<int>("b");
//Kernels operate on views without knowledge of the underlying
//layout
        kernel(a, b, len, depth);
    }
//Convert variables "a" and "b" from one data layout to another. In
//this case, the variables "a" and "b" are being converted from SoA
//to AoSoA
    datastore.convertData(WDS_DT::AOS, {"a", "b"});
    {
        auto a = datastore.getView<double>("a");
        auto b = datastore.getView<int>("b");
//Unmodified kernel
        kernel(a, b, len, depth);
    }

    return 0;
}
```

Listing 5.14: C++ example showing how two variables `a` and `b` can be converted from one data structure to another utilising WDS.

5.4.2 Data Adjacency

Different algorithms may prefer to iterate through a given variable in different ways, depending on the need of the users application. Changing the adjacency of data elements can lead to improvements in the speed of an algorithm, by ensuring the next element is closer to the current element and thus allowing for more efficient data prefetching. Much like the conversion between variables,

this can be complex to implement for large variables. As such, it is possible to implement this functionality into the library, and to allow for a wide range of data accesses. However, owing to the nature of implementing such a functionality, it can only be applied in certain cases where the data can be re-ordered without affecting the underlying data structures. For example, take an SoA representation of a 2D structured mesh. As can be seen from Figure 5.3, the data can be structured in a column-major or row-major way whilst still being represented in an SoA data structure.

The variable can be set with a given data adjacency when first created. If none is provided, then the data adjacency is set to follow the order of dimensions. However, the library also allows for the data adjacency to be changed once the variable has been created. This is achieved in the library in the same way as the conversion of variables. As such, the user can utilise the functionality in the same way, as it takes the variable key, and a list of the new data adjacency order. For example, in Figure 5.3, the column-major access pattern would be `{1, 2}`, and the row-major access pattern would be `{2, 1}`. This data adjacency information can then be used by the variable and view classes to adjust the way in which the data is stored, and by extension, the data access pattern.

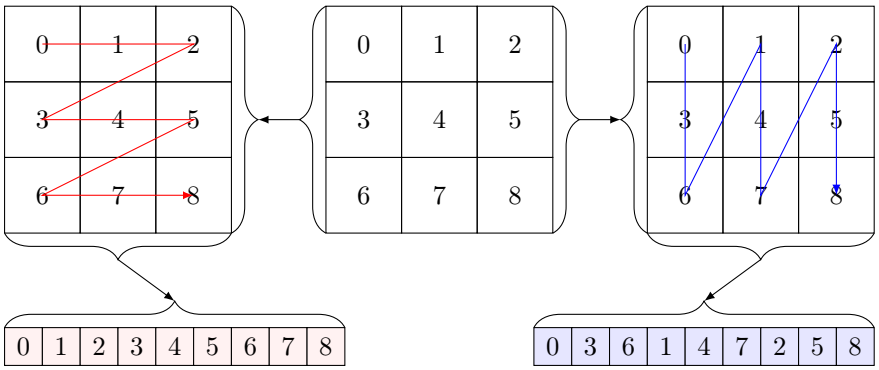


Figure 5.3: Graphical example of how the order of the data can differ, without changing the underlying data structure

Whilst the transformation depicted in Figure 5.3 is trivial, WDS allows this to apply to variables with two or more dimensions. As such, this can be very useful in restructuring variables with a large number of dimensions, in order to achieve a better data adjacency. This can also be useful if the application uses data from other program languages which use different access pattern (such as FORTRAN and C++). Listing 5.15 shows how this could be applied to a 3D variable `c`, which can be altered with different data adjacency, transparently to the kernel.

```
#include <wds.hpp>

int main(int, char **) {
//Create data store
    wds::WDS datastore;
//Specify the variable "c"
    const int len = 250, depth = 4, width = 20;
    datastore.addMeta("c", {len, depth, width}, double()));
//Build the variables given the metadata provided, specifying the
//layout desired. Default data adjacency is {1,2,3}
    datastore.buildVar(WDS_DT::SOA);
    {
//Views used to access and modify data
        auto c = datastore.getView<double>("c");
//Kernels operate on views without knowledge of the underlying
//layout
        kernel(c, len, depth, width);
    }
//Change the adjacency of the variable "c". In this case, the data
//adjacency of the first two dimensions have been reversed
    datastore.changeAdjacency("c", {2,1,3});
    {
        auto c = datastore.getView<double>("c");
//Unmodified kernel
        kernel(c, len, depth, width);
    }

    return 0;
}
```

Listing 5.15: C++ example showing how a variable `c` can have it's data adjacency's altered, transparent to the kernel through the utilisation of WDS.

5.5 Data Structures and Optimisations

Two of the most used data structures, SoA and AoSoA have been implemented within the library. This allowed for easier testing and debugging of the system, as they are some of the least complicated structures to examine. These data structures also enabled the library to be easily validated, as the outputs could be easily matched to a reference version. A set of specialised data structures were also implemented, in order to validate that this implementation could be undertaken with minimal impact to the application. Details of the specialised data structures and the impact the abstraction had can be seen in Chapter 7.

From the initial implementation to the final implementation, optimisations and code practises were changed to ensure that the library ran as fast as possible, thus minimising the impact of the library on the application. Whilst some of these actions only applied to particular data structures, some of the largest impacts came from generalised optimisations. One of the key optimisations carried out involved not including branching statements in the libraries view classes. These branching statements were introduced into the initial implementation to facilitate parameter checking and validation, whilst allowing for exceptions to be thrown should these checks fail. In cases where multiple dimensions were required, loops were used within data access functions to determine the position of the data element. This caused data access to be incredibly slow compared to reference implementations. To rectify this, several changes were made. The use of exception objects was replaced by the use of assertion function found in the C++ standard library [13]. These are only compiled and used when the application is built with a debugging flag, thus ensuring that they are only included when necessary. The use of branching and statements was also minimised. In so doing, less computational steps are included with a data access, and it is easier for the compiler to achieve vectorisation and parallelisation. In nearly all cases, looping statements were replaced by unrolling the loop.

Listings 5.16 and 5.17 show the multi-dimensional access operators for the `View` class in both the initial and final implementations respectively. As can be seen, these take two very different forms. The initial implementation shown in Listing 5.16 relies on the use of branching statements to check to see if the function should be used. This can be incorporated into an `assert` function, as shown in Listing 5.17. The other key difference is that the initial implementation used a variadic function for the access operator. In this way, the function can be generalised to allow for any number of dimensions. However, this approach came with the disadvantage of the requirement for a loop through the list of parameters, which made the access operator slow. Therefore, separate functions were built for different numbers of dimensions. The 2D version is shown in Listing 5.17.

```
DataType &operator() (int n, ...) {
        if (dimensions.empty()) {
                return this->operator[](n);
        } else {
                assert(n <= dimensions.size()+1);
                va_list dim;
                va_start(dim, n);
                int pos = va_arg(dim, int);
                for (int i = 0; i < n-1; i++) {
                        int var = va_arg(dim, int);
                        pos = (pos * dimensions.at(i))+var;
                }
                va_end(dim);

                return this->operator[](pos);
        }
}
```

Listing 5.16: Multi-dimensional access operator for the `View` class within WDS used the initial implementation

Another optimisation that helped to improve performance involved ensuring that any data access functions were inlined. By doing this, the compiler places any functionality of the function directly into the code, rather than pushing the current state of the program onto the stack and running the function indepen-

```
inline T& operator()(int i0, int i1)__attribute__((always_inline)){
    assert(meta.dim.size() == 2);
    assert(i0 >= 0);
    assert(i0 < meta.dim[0]);
    assert(i1 >= 0);
    assert(i1 < meta.dim[1]);
    unsigned long pos = (dimMultiplier[0] * (unsigned long)i0) +
                        (dimMultiplier[1] * (unsigned long)i1) +
                        aosOffset;
    return *((T*)(((unsigned char*)data) + pos));
}
```

Listing 5.17: Multi-dimensional access operator for the `View` class within WDS used the final implementation

dently. However, if the function is large or complex, inlining this function can impact the performance. Hence why this was only applied to data access functions, which are designed to be as light-weight as possible. This often involves including only essential calculations and with no, or only very small branching statements. It is not always possible for compilers to ascertain whether the function should be inlined. As a consequence they may take an overly cautious approach when making this decision. This can mean that, for some compilers, no functions in WDS are inlined. In order to overcome this problem, WDS will inform the compiler which functions must be inlined. In order to do this, both the C++ keyword `inline` [14] and the compiler attribute `always_inline` [28] were applied to the functions. This can be seen in Listings 5.16 and 5.17. The function shown in Listing 5.16 would be too large to effectivly inline due to the large branching statement and the loop contained within the function. It is also reliant on other functions, which is a good indication that inlining is not appropriate in this situation. On the other hand, Listing 5.17 contains no branching statements or loops, and only contains basic arithmetic calculations (as the `assert` statements will be removed by the compiler if not being compiled in debug mode). As such, inlining should be used, and will be forced on through the use of `inline` and `__attribute__((always_inline))`.

Some optimisations were carried out on specific data structures, due to the way in which they were initially implemented. For example, this includes using functions from other classes (in the case of the SoA data structure), or over complication in order for generalisation (in the case of the AoSoA data structure). In order to continue support for these data structures, and to ensure performance gains were being achieved and could be validated, each data structure class was copied into a new class. This new class was duly optimised, then once this was achieved, the old class was replaced by the new class. Sections 5.5.1 and 5.5.2 discuss the optimisations applied to the SoA and AoSoA data structures respectively. In Section 5.5.3, the process of implementing a new data structure in the library is discussed.

5.5.1 Structure of Arrays

In the initial implementation of the SoA data structure, all accesses to the data had to be achieved through the variable class. Originally, in order to access a piece of data from a variable through a view class (either specialised or generic), the view object would calculate the amount of bytes into a data block in which the requested piece of data resided. The relevant information was then passed to the relevant variable object through the 1D access function, which would then return the memory address back to the view object. The view object would then cast the data to the required data type, and then return the result to the application. If the variable data access was more than a single dimension, then further calculations were carried out by the view object, before passing the result onto the 1D access function within the variable object.

Even with the inlining of all data access functions including the single-dimension data access, the performance of data access was much lower than expected. To correct this, the pointer to the data block was passed into the view, and was cast to the correct data type at the creation of the view object. This action reduced the amount of repetitive computation carried out at data access time. Furthermore, each data access function, regardless of the number

of dimensions requested, performed all the calculations and returned the result on the same line. The computation of each line was minimised, using only additions and multiplications. Listings 5.18 and 5.19 demonstrate these changes to the 2D data access function for the SoA specialised view classes in the initial and final implementations respectively.

```
inline T& operator[](int i) override final {
    return variable->getRef<T>(i);
}
/* Other data access functions */
inline T& operator()(int i0, int i1) override final {
    return this->operator[](this->meta.dim[0] * (i1) + i0);
}
```

Listing 5.18: SoA 2D data access function used in WDS' initial implementation

```
inline T& operator()(int i0, int i1) override final {
    assert(this->meta.dim.size() == 2);
    return rawData[(this->meta.dim[1] * i0) + i1];
}
```

Listing 5.19: SoA 2D data access function used in WDS' final implementation

5.5.2 Array of Structures of Arrays

Optimisations of the AoSoA data structure had some similar issues to the SoA data structures, but were further exacerbated by the way in which the flow of data was managed. Much like the SoA data structure as discussed in Section 5.5.1 and demonstrated in Listings 5.18 and 5.19, the AoSoA data structure accessed all the data by requesting the memory position with a given offset, via the single-dimension data access function. This meant that this function had to retroactively calculate the structure element requested, alongside the required element within the given structure. This required the use of modulus calculations and divisions; two operations that are drastically more expensive when compared to the cost of a multiplication or an addition. This situation was only

made worse by the fact that multi-dimensional data access functions had to perform calculations to put it into this 1D format, only for some of the calculations to then be undone.

Listing 5.20 demonstrates the issue with the initial implementation of the AoSoA data structure. In the initial implementation of the data structure in WDS, all data access were limited to the 1D access function `operator[](int i)`. Therefore, to access a particular data point within a structure, the 2D access function had to encode the required information. This could then be decoded by the 1D access function. However, to decode the data point required, expensive calculations such as modulus calculations and divisions were required.

```
inline const T& operator[](int i) const override final {
  int offset =
    offsetCycle[i % offsetCycle.size()] +
    (this->meta.size * floor((double)i/(double)offsetCycle.size()));
  return variable->getFromOffset<T>(offset);
}
/* Other data access functions */
inline T& operator()(int i0, int i1) override final {
  return this->operator[](this->meta.dim[0] * (i1) + i0);
}
```

Listing 5.20: AoSoA 2D data access function used in WDS' initial implementation

Multiple steps were taken in order to correct this problem. Firstly, the pointer to the block of memory was copied into the view class, resulting in a reduction in the number of calls to other functions and objects. Secondly, to reduce the amount of calculations required when accessing a given data element, the view class performed more calculations in the creation of the object. Predominantly, this involved calculating the amount of offset for each dimension necessary. Thirdly, each data access function was replaced by a 2D access function. This eliminated all the modulus and division calculations from the data access functions, and just used the pre-calculated values with additions and multiplications of the required data. This last optimisation however, required a

change in definition of what the single-dimension functions achieved. Instead of iterating through every single element in the variable, these functions returned the first element in a requested structure. Listing 5.21 shows the final, optimised version of the AoSoA 2D data access function within WDS. Due to the removal of the encoding and decoding stages, the final version solely relies on additions and multiplications, improving the performance of the access function.

```
inline T& access(int i0, int i1) {
    assert(i0 > -1 && i0 < this->meta.dim[0]);
    assert(i1 > -1 && i1 < this->innerNumItems);
    unsigned long pos = (sizeOfStruct * (unsigned long)i0) +
                        (sizeOfClass * (unsigned long)i1) +
                        offset;
    assert(variable->totalOffset > pos);
    return *((T*)(((char*)data) + pos));
}
/* Other data access functions */
inline T& operator()(int i0, int i1) override final {
    assert(i0 >= 0);
    assert(i1 >= 0);
    assert(this->meta.dim.size() == 2 && this->meta.dim[1] != 1);
    return access(i0, i1);
}
```

Listing 5.21: AoSoA 2D data access function used in WDS' final implementation

5.5.3 Specialised Data Structures

Within certain algorithms, the required data structure cannot be expressed with the SoA or AoSoA data structures. As a consequence there will be applications and algorithms that will require their own data structure in order to perform computations efficiently such as the data structures described in Chapter 7. The way in which the library has been developed allows for this to be carried out relatively easily. To create a specialised data structure, first create a new variable class, inheriting from the variable class interface `Var`. This ensures that all the necessary functionality is included, and that it can be stored in the `Controller` class. As discussed in Section 5.3.2, this class manages the storage

of the data, as well as any large scale changes. It should be noted that at this stage, the variable class cannot be linked to a view or set of views. Once this process has been completed, the necessary view classes can be created. Like the variable class, the view class must inherit from the `ViewInterface` class. Whilst there should only be one variable class for a given data structure, there can be multiple views for a given data structure. This may be because there are multiple ways of iterating through the data or because the data structure can be split into multiple smaller sections. Following this stage, the variable class and the view class(es) can be linked, so that the variable class can pass all the required information to each of the view classes. If a variable is able to create multiple views, then either the view-building function can generate the correct view depending on a given key, or multiple view-building functions can be created, one for each view class.

After these classes have been created, there is then a need to extend the relevant functions to include the new data structure. Firstly, the `DataStoreType` should be extended with identifying names for each new view created. This is created first, as all other extensions rely on the ability to identify the data structure and its views. The functions within `WDS` and `Controller` classes can then be expanded to include the new data structure, using the new identifiers in `DataStoreType`. Without this step, the library would not be able to build, let alone find or perform actions on, any variable with this new data structure. Finally, new specialisations of `ViewSpec` need to be created, to ensure that data is accessed faster. This is usually quick and easy to implement, as the function names will be the same. Each view is created and has an identifier in `DataStoreType` as the data structure will require a specialisation within this class.

With this completed, all the required functionality of the library is in place, and the data structure can be used and validated. However, additional functionality can also be implemented if the data structure allows it and is required. One of the additional functionalities involves the extension of `View` class to al-

low the class to utilise the new data structure. This allows for the user to be more data-structure independent in their code, thus allowing for easier switching of data structures. However, this can only be done if the data structure can access the data in a uniform and consistent pattern. Another additional functionality with the potential to be extended is the extra high level functionality built within the `WDS` and `Controller` classes. This includes the conversion of variables and the changing of data adjacency's. Whist it gives extra functionality to the developer, there may be no logical way to convert to or from the new data structure. Finally, additional functions may need to be built into `WDS` and `Controller` to allow for better utilisation of the data structure. This extra functionality should not interfere with the operations of other data structures or variables.

All of the aforementioned processes have been put into practice when building specialised data structures for multi-material problems. The development and functionalities of this data structure has been outlined in Chapter 7, along with its performance.

5.6 Summary

In this chapter, the creation, development, implementation and optimisations of the data structure abstraction library WDS have been presented. The initial implementation showed how some of the key features could be implemented, and proved as a functional prototype showing that it was possible to abstract the data structure away from an application. However, this initial implementation came with some caveats. Firstly, the library was not flexible. Trying to extend this version of the library to include specialised data structures would have been very difficult, and would have created fragmented and ill-performing code. Secondly, and more importantly, the version of the library heavily impacted the performance of the program.

These problems were fixed within the current version of WDS. This version was designed in such a way that data structures could be added trivially, with minimal impact to the performance of the library or the application it is being utilised in. Through the use of interfaces, consistency in functionality was maintained between data storage classes. The same principle, alongside template programming, was also applied to data access classes whilst maintaining performance. The design of the current data structure abstraction library also allowed for high-level functionality to be implemented more easily, such as the conversion of variables from one data structure to another, and the alterations of a variables data adjacency.

CHAPTER 6

Performance Analysis of the Data Structure Abstraction Library

In Chapter 5, the aim, implementation and development of the data structure abstraction library Warwick Data Store (WDS) was discussed, and how it can be used to abstract data layout from an application. However, this did not cover how improve programmer productivity, minimising the impact on application performance, as these are also significant targets for the library. As such, the aim of this chapter is to show how the library can be used and the overhead incurred by WDS. In particular, the implementation of the data structure abstraction library into different kernels and proxy applications is discussed. Alongside this, the overhead of the implementation of WDS into these different codes is examined. It is also important to show how the library scales as an application runs over multiple nodes on a supercomputer. This chapter has been broken up into the following sections:

- Section 6.1 discusses the implementation of a benchmark suite, and the overhead of these kernels across different compilers and architectures.
- Section 6.2 explores the implementation of WDS into three different physics proxy applications, and the overhead associated with these. These mini-apps are examined across different parallelism methodologies, problem sizes, compilers and architectures.
- Section 6.3 discusses how the performance of a physics mini application across multiple nodes with strong scaling, to show how the overhead changes.

6.1 Benchmark Testing and Overhead

In order to start testing and validating that the library worked, the library was developed alongside multiple example codes. These examples built trivial data structures, ensured the data could be accessed properly accessed and validated key features. However, these example codes were too small and trivial to be used to generate meaningful results with respect to the amount of overhead the library would create. As such, a collection of benchmarks were utilised.

The benchmarking suite was made to test all the benchmarks within a single application. In this suite, different benchmark kernels were implemented alongside there WDS counterpart. When a benchmark kernel was tested, the WDS counterpart used the same data structure. This allowed for just the overhead of the library to be measures, and not count any performance differences gained or lost from the use of a different data structure. The suite then measured the amount of time each kernel took, and would repeat the process multiple times, storing the maximum time for each kernel. These times were then used to calculate the overhead.

In order to ensure that the results were as comparable as possible, thus increasing the confidence in the results; the benchmarking suite generated the same starting data for both the reference kernel and the WDS kernel, and validated this through inspection. The benchmarking suite would also validate that the kernel ran correctly by comparing the end state of all the data of both the reference and WDS version for any discrepancies. Throughout this process, the benchmarking suite was used to replace the small example programs, and became the primary way to test the library. In order to ensure the library worked well across a range of environments, the benchmarking suite was tested on multiple architectures and compilers. Table 6.1 lists both of the systems tested, and both of the compilers used on each of these systems, alongside key statistics. To calculate the bandwidths, the STREAM benchmark [67] was used with optimisation flags. In particular, both the `Ofast` and appropriate OpenMP

flags across all systems. Where a compiler with streaming stores was available, this was utilised along with the appropriate flag. Thus, for Orac, this meant using the Intel 19.1.0 compiler with `qopt-streaming-stores` flag. It should be noted that Orac is the very similar to the system utilised in Chapter 4, the only difference being the speed of the processor.

		Isambard	**Orac**
Processor		ARM Marvell Thunder X2 [65]	Intel Xeon E5-2680 v4 [44]
Sockets per Node		Dual	Dual
Per Socket	Cores	32	14
	Bandwidth (BW)	116.5 GB/s	59.1 GB/s
	Cache	32 MB L3	35 MB L3
Compilers		CCE 9.1.3 GNU 9.2.0	Intel 19.1.0.166 GNU 8.3.0
Message Passing Interface (MPI)		Cray MPICH 7.7.12	Intel - IMPI Build 20191121 GNU - OpenMPI 3.1.4

Table 6.1: Systems used to measure the performance impact of WDS when testing benchmarks

The benchmarking suite consisted of key kernels from a variety of benchmarks and mini-applications, specifically STREAM [67], HLP [88], and SNAP [128]. The STREAM benchmark is a memory focused benchmark comprising of four kernels, and was utilised in order to calculate the memory bandwidths of the processors being tested. The HPL benchmark is used to calculate the peak Floating Point Operations per Second (FLOP/s) of large supercomputers, and consists of a large number of kernels. For the purposes of testing, a single kernel (DGemm) was extracted, converted and utilised. Finally, SNAP is a discrete ordinates proxy application developed by Los Alamos National Laboratory (LANL). The key kernel from the SNAP (Sweep) was utilised to test the effectiveness of WDS. As well as testing the overhead, the HPL and SNAP kernels were also used to validate different data structures. Table 6.2 shows the overhead percentages (described in Chapter 2.6) from each of the kernels, across

both architectures and compiler.

From Table 6.2, it can be seen that across all STREAM kernels, the overhead

Benchmark	**Kernel**	**Overhead(%)**			
		Isambard		**Orac**	
		Cray	**GNU**	**Intel**	**GNU**
STREAM	Copy	1.52	-1.28	-0.92	1.28
	Scale	-0.52	-1.00	-1.62	0.47
	Add	0.68	-0.56	-0.22	0.32
	Triad	0.95	0.46	-0.10	2.13
SNAP	Sweep	5.91	-8.51	-7.11	-12.9
HPL	DGemm	4.23	-0.67	-10.5	-0.41

Table 6.2: Results for different benchmark kernels across architectures, compilers and data structures

peaks at 1.52%, and for the majority of cases, the overhead is close to 0%. For the other kernels, the highest overhead is 5.91%, with the majority being close to 0%, showing that the library has a minimal impact on these benchmark kernels.

6.2 Mini-Application Performance and Overhead

Whilst benchmark kernels can show particular issues, and are useful for validation due to there size, it can be difficult to predict the actual overhead cost of the library when used in a production application. Thus, in order to demonstrate how WDS performs in a more realistic settings, the library has been implemented into three different mini-applications [31]. These mini-applications are small, self-contained codes that are designed to be representative of larger applications, and are therefore perfect to test and develop new ideas in an agile way. The objective of this is to show that the library incurs a low overhead cost, when compared to the reference application under the same conditions.

To fully explore this, multiple different parameters were tested to see the full range of effects of WDS. Firstly, a variety of different architectures and compilers were used, to see if there were differences in how the hardware and compilers handled the library (Section 6.2.1). Multiple different mini-applications

were also used to test the library across each different permutation of architecture and compiler. These mini-applications were BookLeaf [116] (Section 6.2.2), TeaLeaf [62] (Section 6.2.3) and MiniMD [31] (Section 6.2.4). Finally, two input decks were created for each mini-application, a small and large variant. By doing this, problems could be tested when more memory or computationally bound, thus exploring how this affected the overhead of WDS. Each problem size was also ran on all architectures and compilers. The different problem configuration for each mini-application can be seen in Table 6.3.

Mini-app	**Input Deck**	**Small**	**Large**
BookLeaf	Noh Problem Size	200×60	2530×126
	Sedov Problem Size	179×179	566×566
TeaLeaf	Problem Size	1000×1000	8000×8000
	Timestep	20	10
MiniMD	Problem Size	$64 \times 64 \times 64$	$128 \times 128 \times 128$
	Timesteps	1000	500

Table 6.3: Input sizes for small and large problems across all mini-aplications

In order to ensure fair results, each configuration of architecture, compiler, mini-application and problem size was executed five times, with the average of these results being used to calculate the overhead. Similarly to how the benchmarks were tested, the WDS version of the mini-applications were ran using the same data structure configuration as the reference version, so that the difference in time related only to the additional time required by the library, thus ensuring a fair comparison. As few changes to the applications logic were made as possible, in order to ensure a direct comparison between both of the versions.

For each configuration, two sets of results are presented. The first set of results is the time taken for the reference version to be executed, given in seconds. The second is the percentage overhead of the library version. For all results tables within this section, a colour scheme has been used for overheads to show the difference in results. All green cells are values below 10%, orange cells are values between 10% and 30% inclusive, and red cells are values above 30%. The

aim for the library is to get the overhead as low as possible though it is expected to see trends across compilers and architectures.

6.2.1 Hardware and Compilers

To ensure that the performance impact of WDS is small across a variety of HPC systems, all problems were ran across three different systems, each of which use a different processor, and across two different compilers on each system. Specifically, an ARM ThunderX2 system, an Intel Xeon Cascade Lake AP system and an AMD Rome Epyc system were utilised. Details for each of these systems can be seen in Table 6.4. Each of these systems have varying amounts of cache, bandwidth and processing power, allowing for a wide range of architectural differences to be inspected. To calculate the bandwidths, we used the STREAM benchmark [67] with optimisation flags. We used both the `Ofast` and appropriate OpenMP flags across all systems. Where a compiler with streaming stores was available, this was utilised along with the appropriate flag. For Kingfisher, this meant using the Intel compiler with `qopt-streaming-stores` flag. For Rome, the AOCC compiler was used, with the `fnt-store` compiler flag and transparent huge pages tuned off.

For all systems, all MPI problems were executed across all physical cores in a node, all OpenMP problems across a single Non-Uniform Memory Access (NUMA) region within a node, and all hybrid (MPI + OpenMP) problems such that the MPI ranks are allocated to seperate NUMA regions, with OpenMP threads filling each NUMA region. For Isambard, each socket consists of a NUMA region. So, when running hybrid problems, two MPI ranks are used, with each rank consisting of one thread per core. As such, each Isambard NUMA region has 32 threads. Each socket in a Kingfisher node consists of two NUMA regions. This means that when running hybrid problems on Kingfisher, four MPI ranks are used with each NUMA region containing 24 threads. For Rome, the processor was split into four NUMA regions. The hybrid runs were achieved by splitting the processor further, and using a single MPI rank per

		Isambard	**Kingfisher**	**Rome**
Processor		ARM Marvell Thunder X2 [65]	Intel Xeon 9242 [42]	AMD EPYC 7742 [1]
Sockets per Node		Dual	Dual	Dual
Per Socket	Cores	32	48	64
	BW	116.5 GB/s	187.3 GB/s	176.4 GB/s
	Cache	32 MB L3	71.5 MB Smart Cache	256 MB L3
Compilers		CCE 9.1.3 GNU 9.2.0	Intel 19.1.1 GNU 8.2.0	AOCC 2.2.0 GNU 9.2.0
MPI		Cray MPICH 7.7.12	IMPI 7.217	AOCC - Open-MPI 4.0.3 GNU - Open-MPI 4.0.2

Table 6.4: Systems used to measure the performance impact of WDS when tesing mini applications

Level 3 Cache Memory (L3) region, consisting of four cores. Each NUMA region consists of four L3 regions, so a configuration of 32 MPI ranks, each with four OpenMP threads, was used. It should be noted that the AMD processor can be configured to consist of one NUMA region if required. To ensure fairness when comparing both MPI and MPI + OpenMP implementations of the mini-applications, the implementation of all WDS versions of the proxy applications mirrored the reference implementations as closly as possible. In addition, and for all experiments, thread balancing was used through the job management system for the machine. This ensured that all the cores were being used efficiently.

6.2.2 Unstructured Physics Mini-Application

BookLeaf [54, 116] solves the compressible Euler equations on an unstructured grid using an Arbitrary Lagrangian-Eulerian (ALE) formulation. These equations describe the dynamics of inviscid fluids, and are used widely to solve many problems in science and engineering. Two classic test problems are used to test

WDS; Sod's shock tube [102] and Noh's cylindrical artificial viscosity problem [74]. In this subsection, the C++ OpenMP version of the code is utilised as the base for the WDS version, and also the reference version used to compare its performance.

Creation of the WDS version of BookLeaf was relatively simple. The C++ version of BookLeaf contains a data storege system, complete with its own view class. As such, the majority of code was simple to change, as the reference data storage and view objects were swapped for the WDS equivalents. Listings 6.1 and 6.2 show how this was achieved through the use of the `getEnergy` kernel within BookLeaf. Listing 6.1 shows the reference version of this particular kernel, whilst Listing 6.2 shows the WDS version. As can be seen, the only change is the function signature; the computation carried out by the kernel has not been altered. The initialisation of the variables in the data structure could not be treated in the same way. For this, a simple function was used to generate all the variables that could be used, at the required sizes.

Most routines within BookLeaf have a low arithmetic intensity, meaning that the code is typically memory-bound. As such, it is expected that the library will have a larger overhead with smaller problem sizes, and a relatively small overhead for larger problems. This is due to the fact that on larger problems, the processor will be more memory constrained, allowing for computation to be done in the time the processor is waiting on data.

Table 6.5 shows the time taken to complete the reference version, and the overheads for all variations of problem decks, processors and compilers. The average time taken for the WDS implementations can be calculated from the reference execution time and the percentage overhead. This additional data is not presented however, as the key elements are already presented in the form of the relative execution time, and the scale of overhead incurred by implementing WDS into the application. The colour scheme discussed at the end of Section 6.2 has been applied to Table 6.5, where overhead values below 10% are highlighted in green, overhead values between 10% and 30% are highlighted in orange, and

```
void getEnergy(
    double dt,
    double zerocut,
    ConstView<double, VarDim, NCORN> cnfx,
    ConstView<double, VarDim, NCORN> cnfy,
    ConstView<double, VarDim, NCORN> cnu,
    ConstView<double, VarDim, NCORN> cnv,
    ConstView<double, VarDim>        elmass,
    View<double, VarDim>             elenergy,
    int nel)
{

    #pragma omp parallel for
    for (int iel = 0; iel < nel; iel++) {
        double w1 = cnfx(iel,0)*cnu(iel,0)+cnfy(iel,0)*cnv(iel,0)+
                    cnfx(iel,1)*cnu(iel,1)+cnfy(iel,1)*cnv(iel,1)+
                    cnfx(iel,2)*cnu(iel,2)+cnfy(iel,2)*cnv(iel,2)+
                    cnfx(iel,3)*cnu(iel,3)+cnfy(iel,3)*cnv(iel,3);

        w1 = -w1 / std::max(elmass(iel), zerocut);
        elenergy(iel) += w1 * dt;
    }
}
```

Listing 6.1: Reference BookLeaf `getEnergy` kernel

overhead values above 30% are highlighted in red.

As expected, for both Noh and Sedov problems, the small problem sets have a larger overhead, than the large problem sets. This is independent of both the system and the compiler, though there is some fluctuations in how the compilers performed on the small problem sets. For the large problem set, the compilers produced close to the same overhead on the same system and across architectures. Across both BookLeaf problem sizes, the WDS times are very similar across the different compilers within a given architecture. However, with the smaller problem size, the runtimes are shorter than there large problem size counter part, making the overhead more sensitive to differences. This is why there is a larger range of overheads for the smaller problem sizes, and why the slower runtime within an architecture has the smaller overhead. Even with these factors, it is clear that the overheads for the smaller problem is greater than the

```
void getEnergy(
    double dt,
    double zerocut,
    wds::ViewSpec<double, WDS_DT::SOA>&  cnfx,
    wds::ViewSpec<double, WDS_DT::SOA>&  cnfy,
    wds::ViewSpec<double, WDS_DT::SOA>&  cnu,
    wds::ViewSpec<double, WDS_DT::SOA>&  cnv,
    wds::ViewSpec<double, WDS_DT::SOA>&  elmass,
    wds::ViewSpec<double, WDS_DT::SOA>&  elenergy,
    int nel)
{

    #pragma omp parallel for
    for (int iel = 0; iel < nel; iel++) {
        double w1 = cnfx(iel,0)*cnu(iel,0)+cnfy(iel,0)*cnv(iel,0)+
                    cnfx(iel,1)*cnu(iel,1)+cnfy(iel,1)*cnv(iel,1)+
                    cnfx(iel,2)*cnu(iel,2)+cnfy(iel,2)*cnv(iel,2)+
                    cnfx(iel,3)*cnu(iel,3)+cnfy(iel,3)*cnv(iel,3);

        w1 = -w1 / std::max(elmass(iel), zerocut);
        elenergy(iel) += w1 * dt;
    }
}
```

Listing 6.2: WDS BookLeaf `getEnergy` kernel

larger problems across all architectures and compilers.

6.2.3 Heat Conduction Mini-Application

TeaLeaf [119] solves the linear heat conduction equation on a structured grid. Parabolic equations like this are often solved using implicit methods, requiring the use of a linear solver. TeaLeaf's primary purpose is to support experimentation with different types of linear solver in a simple setting [62]. A C/C++ version has been created by the University of Bristol [120], which we use here, specifically the MPI and OpenMP variant. We focus on the Conjugate Gradient (CG), Polynomially Preconditioned CG (PPCG) and Chebyshev solvers.

TeaLeaf contains a structure called `Chunk`, which stores all the data for the chunk of the mesh contained within the MPI rank. The kernels then pull out relevant references from this structure, to be used within the kernels. In order

System	Compiler	Result	Noh		Sedov	
			Small	Large	Small	Large
Isambard	Cray	Ref (sec)	32.0	62.7	8.00	57.1
		Overhead (%)	16.3	6.69	19.7	4.31
	GNU	Ref (sec)	35.4	63.2	12.4	3.12
		Overhead (%)	6.21	1.51	4.91	3.12
Kingfisher	Intel	Ref (sec)	15.9	48.9	4.69	46.1
		Overhead (%)	26.3	4.16	37.7	-1.50
	GNU	Ref (sec)	19.9	49.1	5.59	43.5
		Overhead (%)	18.2	6.08	19.0	5.85
Rome	AOCC	Ref (sec)	16.8	112	5.27	100
		Overhead (%)	31.9	1.64	31.0	2.16
	GNU	Ref (sec)	20.4	112	5.68	99.8
		Overhead (%)	17.4	0.99	20.9	1.49

Table 6.5: Results for BookLeaf input decks across architectures, compilers and input decks.

to implement WDS into TeaLeaf, the `Chunk` data structure was replaced with a WDS object, and rather than passing pointer references to kernels, view objects were passed instead. Initially, the view objects were being copied into functions when using some compilers. This negatively affected the performance of smaller functions that did not scale by the mesh size, but rather scaled by the number of MPI ranks. Thus, when implementing WDS into TeaLeaf, it was ensured that the view object was passed by reference in all cases. Listings 6.3 and 6.4 show the same kernel for both the reference version and the WDS version of TeaLeaf's `cg_calc_w` kernel respectively. As can be seen, the `Chunk` ID had to be passed in, as the WDS required it in order to access the correct values. The other main change involved altering the data access function from `[index]` to `(chunk, index)`, in order to use the WDS views correctly. All solvers were validated using the same methodology as used when validating BookLeaf.

Two optimisations were also implemented in order to ensure consistency between runs and compilers. The first optimisation included an MPI barrier before both AllReduce functions. This was done in such a way that it did not count towards the final runtime, and ensured that any load imbalance did not

```
void cg_calc_w(
    const int x,
    const int y,
    const int halo_depth,
    double* pw,
    double* p,
    double* w,
    double* kx,
    double* ky)
{

    double pw_temp = 0.0;

#pragma omp parallel for reduction(+:pw_temp)
    for(int jj = halo_depth; jj < y-halo_depth; ++jj) {
        #pragma ivdep
        for(int kk = halo_depth; kk < x-halo_depth; ++kk) {
            const int index = kk + jj*x;
            const double smvp = SMVP(p);
            w[index] = smvp;
            pw_temp += w[index]*p[index];
        }
    }

    *pw += pw_temp;
}
```

Listing 6.3: Reference (C++) TeaLeaf `cg_calc_w` kernel

affect the results. The second optimisation was to add `pragma ivdep` to compute functions that could safely be vectorised. This ensured that all functions were being optimised in the same manner, meaning that differences in time were caused solely by the addition of WDS. This particular optimisation can be seen in both Listings 6.3 and 6.4. Both of these optimisations were added to both the reference and WDS versions across all architectures and compilers.

TeaLeaf, like BookLeaf, is typically memory bound. As such, the smaller problem size is expected to have a larger overhead than the larger problem size, as less computation can be shadowed by memory accesses. Tables 6.6, 6.7 and 6.8 show the overhead of the WDS version of TeaLeaf against the reference version across different architectures, compilers and solvers. The tables also

```
void cg_calc_w(
    const int x,
    const int y,
    const int halo_depth,
    double* pw,
    wds::ViewSpec<double, WDS_DT::SOA>& p,
    wds::ViewSpec<double, WDS_DT::SOA>& w,
    wds::ViewSpec<double, WDS_DT::SOA>& kx,
    wds::ViewSpec<double, WDS_DT::SOA>& ky,
    const int chunk)
{

    double pw_temp = 0.0;

#pragma omp parallel for reduction(+:pw_temp)
    for(int jj = halo_depth; jj < y-halo_depth; ++jj) {
        #pragma ivdep
        for(int kk = halo_depth; kk < x-halo_depth; ++kk) {
            const int index = kk + jj*x;
            const double smvp = SMVPWDS(p);
            w(chunk, index) = smvp;
            pw_temp += w(chunk, index)*p(chunk, index);
        }
    }

    *pw += pw_temp;
}
```

Listing 6.4: WDS TeaLeaf `cg_calc_w` kernel

System	Compiler	Result	CG		Chebyshev		PPCG	
			Small	Large	Small	Large	Small	Large
Isambard	Cray	Ref (sec)	5.82	1877	1.82	1583	2.54	2007
		Overhead (%)	12.1	0.45	33.8	0.25	23.1	1.46
	GNU	Ref (sec)	3.97	1441	1.96	1501	2.74	1970
		Overhead (%)	12.2	-1.38	8.64	0.19	8.03	4.41
Kingfisher	Intel	Ref (sec)	0.65	862	0.33	808	0.41	1055
		Overhead (%)	39.5	-1.27	65.5	-2.07	66.5	0.83
	GNU	Ref (sec)	0.99	854	0.53	805		1055
		Overhead (%)	30.5	1.81	43.0	-0.54		2.34
Rome	AOCC	Ref (sec)	1.51	1181		1111	0.59	1474
		Overhead (%)	3.14	-0.52		-0.10	39.8	-0.25
	GNU	Ref (sec)	0.75	1182	0.37	1111	0.55	42.2
		Overhead (%)	53.1	-0.36	39.1	-0.32	42.2	-0.24

Table 6.6: Results for TeaLeaf MPI, across all input decks, solvers, architectures and compilers.

System	Compiler	Result	CG		Chebyshev		PPCG	
			Small	Large	Small	Large	Small	Large
Isambard	Cray	Ref (sec)	12.3	3614	4.45	3238	4.01	3562
		Overhead (%)	3.25	2.05	4.46	0.83	7.09	2.68
	GNU	Ref (sec)	11.3	3106	5.35	3454	6.25	4407
		Overhead (%)	-3.68	0.63	-1.13	0.95	-3.16	0.26
Kingfisher	Intel	Ref (sec)	4.17	3398	2.48	3179	1.98	3669
		Overhead (%)	1.78	-2.01	2.92	-0.96	1.63	-0.38
	GNU	Ref (sec)	7.75	3369	3.02	3204		3675
		Overhead (%)	2.81	-0.48	-7.40	-0.40		-0.42
Rome	AOCC	Ref (sec)	8.22	9362		8855	1.89	10348
		Overhead (%)	0.54	-0.05		0.00	10.4	-0.12
	GNU	Ref (sec)			2.46	8856	2.47	10291
		Overhead (%)			0.33	0.00	2.02	0.04

Table 6.7: Results for TeaLeaf OpenMP, across all input decks, solvers, architectures and compilers.

System	Compiler	Result	CG		Chebyshev		PPCG	
			Small	Large	Small	Large	Small	Large
Isambard	Cray	Ref (sec)	6.96	1885	1.88	1647	2.33	1824
		Overhead (%)	4.24	0.16	10.5	1.03	6.15	8.26
	GNU	Ref (sec)	7.90	1591	3.32	1763	4.33	2240
		Overhead (%)	-8.44	0.61	-8.15	2.02	-8.40	4.16
Kingfisher	Intel	Ref (sec)	1.27	838	0.66	813	0.74	911
		Overhead (%)	12.1	0.99	13.9	-2.64	8.64	0.88
	GNU	Ref (sec)	4.63	839		814		920
		Overhead (%)	1.92	1.25		0.90		2.59
Rome	AOCC	Ref (sec)	1.56	1178		1113	0.58	1379
		Overhead (%)	3.84	-0.17		-0.28	34.9	-0.22
	GNU	Ref (sec)	0.82	1180	0.41	1113	0.60	1373
		Overhead (%)	6.47	-0.33	16.9	-0.15	24.1	-0.08

Table 6.8: Results for TeaLeaf MPI and OpenMP, across all input decks, solvers, architectures and compilers.

contain information on the time taken for the reference application to execute. WDS execution times have not been included for the same reason why they were not included when investigating the performance of implementing WDS into BookLeaf (described in Section 6.2.2). These tables utilise the same highlighting as used in the BookLeaf results, where overhead values below 10% are highlighted in green, overhead values between 10% and 30% are highlighted in orange, and overhead values above 30% are highlighted in red.

Each table focuses on a different parallisiation methodology; Table 6.6 focuses on the MPI results, Table 6.7 focuses on the OpenMP results and Table 6.8 focuses on the MPI+OpenMP results. The configuration for the number of ranks

and threads for each parallelisation methodology is described in Section 6.2.1. Across all of these results, for the majority of cases, the smaller problem sizes incur a higher overhead than the larger problem size counterpart regardless of architecture.

Of the three parallelisation methodologies used, it can be seen that the MPI implementation incurs the highest overheads, followed by the the hybrid implementation then the OpenMP version. This is especially true in the smaller problem sizes. For the smaller problem sizes, this is due to the fact that the communications take up a large fraction of the runtime. However, in the larger problems, the computation kernels take up a much larger proportion of the runtime, compared to the communications. Because this is not as much of an issue for the hybrid version, and not an issue at all for the OpenMP implementation, we see lower overheads.

Much like BookLeaf, the runtimes for TeaLeaf are consistent across compilers on the same architecture for the majority of cases. This is true for both problem sets, but is more prominent in the smaller problem size. When examining the variance for the larger problem size, it can be seen that the results overlap for the majority of cases. This means that these results will have a small, if not 0% overhead.

6.2.4 Molecular Dynamics Mini-Application

MiniMD is a proxy-application for the much larger LAMMPS Molecular Dynamics (MD) code developed and maintained by Sandia National Laboratory (SNL) [90]. MD codes such as LAMMPS are widely used by scientists to study the microscopic properties of matter. MiniMD is designed to use the same algorithms as its parent code, but has been structured to be much simpler to support co-design. The mini-application supports two inter-atomic potentials: the Lennard-Jones potential and the Embedded Atom Model (EAM). For the purposes of this paper, we test WDS with a simulation using the Lennard-Jones potential and MPI+OpenMP reference version.

In order to create the WDS version, the arrays within MiniMD's data structure were swapped for WDS variables. View objects were then created to replace the variables, which were then passed to kernels in a similar way the original references were passed. This made the conversation process simple. Listings 6.5 and 6.6 show the `Thermo::temperature` kernel for both the reference and WDS versions of MiniMD respectively. Because the WDS view classes were added directly into MiniMD's data structure class (called `Atom`) and replaced the key arrays, the function signature did not need to be altered. Instead, rather than copying the pointer of the arrays required from the `Atom` class, the WDS views can be accessed directly. In order to make the WDS version more readable, the access function was altered to represent 2D notation by changing `[i * PAD + 0]` to `(i, 0)`. This alteration would not affect the performance of the program, as the code would be compiled to the same assembly instructions.

System	Compiler	Result	MPI		OpenMP		Hybrid	
			Small	Large	Small	Large	Small	Large
Isambard	Cray	Ref (sec)	19.7	81.4	59.9		33.9	124
		Overhead (%)	6.33	5.54	3.61		3.58	3.00
	GNU	Ref (Sec)	17.7	73.5	71.1		40.1	147
		Overhead (%)	1.37	-0.02	2.22		2.42	1.76
Kingfisher	Intel	Ref (sec)	6.89	28.5	48.6	198	14.1	51.3
		Overhead (%)	0.41	2.93	11.5	12.0	10.1	11.7
	GNU	Ref (sec)	6.20	26.5	39.7	164	11.5	42.9
		Overhead (%)	6.20	5.58	10.1	9.31	8.02	8.05
Rome	AOCC	Ref (sec)	5.74	22.6	45.4	192	6.49	26.0
		Overhead (%)	0.72	0.54	6.50	6.26	7.52	7.42
	GNU	Ref (sec)	5.77	22.3	46.6	197	6.59	26.6
		Overhead (%)	0.81	1.63	5.21	5.74	6.69	6.30

Table 6.9: Results for MiniMD input decks, across all architectures and compilers.

The results for MiniMD across all systems and compilers running on a single node can be seen in Table 6.9, with **Hybrid** representing OpenMP + MPI. The configurations used for ranks and threads can be found in Section 6.2.1. The colour scheme discussed at the end of Section 6.2 has been applied to Table 6.9, where overhead values below 10% are highlighted in green, overhead values between 10% and 30% are highlighted in orange, and overhead values above 30% are highlighted in red. As can be seen, the overhead for the application

```
MMD_float Thermo::temperature(Atom &atom) {
    MMD_int i;
    MMD_float vx, vy, vz;
    MMD_float t = 0.0;
    t_act = 0;
#pragma omp barrier
    MMD_float* v = atom.v;
OMPFORSCHEDULE
    for(i = 0; i < atom.nlocal; i++) {
        vx = v[i * PAD + 0];
        vy = v[i * PAD + 1];
        vz = v[i * PAD + 2];
        t += (vx * vx + vy * vy + vz * vz) * atom.mass;
    }
#pragma omp atomic
    t_act += t;
#pragma omp barrier
    MMD_float t1;
#pragma omp master
    {
        if(sizeof(MMD_float) == 4)
            MPI_Allreduce(&t_act, &t1, 1, MPI_FLOAT, MPI_SUM,
                MPI_COMM_WORLD);
        else
            MPI_Allreduce(&t_act, &t1, 1, MPI_DOUBLE, MPI_SUM,
                MPI_COMM_WORLD);
    }
    return t1 * t_scale;
}
```

Listing 6.5: Reference MiniMD `Thermo::temperature` kernel

does not go above 12% for any configuration of either problem size. Some of the overheads are slightly negative. This is most likely due to machine fluctuations. In addition, the OpenMP 3 implementation has the largest range of overheads, with the OpenMP + MPI approach coming in second for all systems.

6.3 Scaling Performance and Overhead

While the previous results show that the single node performance impact of WDS is minimal for a variety of different scenarios, it is also appropriate to examine the performance at scale of the data structure abstraction library. This

```
MMD_float Thermo::temperature(Atom &atom) {
    MMD_int i;
    MMD_float vx, vy, vz;
    MMD_float t = 0.0;
    t_act = 0;
#pragma omp barrier
OMPFORSCHEDULE
    for(i = 0; i < atom.nlocal; i++) {
        vx = atom.wds_v(i, 0);
        vy = atom.wds_v(i, 1);
        vz = atom.wds_v(i, 2);
        t += (vx * vx + vy * vy + vz * vz) * atom.mass;
    }
#pragma omp atomic
    t_act += t;
#pragma omp barrier
    MMD_float t1;
#pragma omp master
    {
        if(sizeof(MMD_float) == 4)
            MPI_Allreduce(&t_act, &t1, 1, MPI_FLOAT, MPI_SUM,
                MPI_COMM_WORLD);
        else
            MPI_Allreduce(&t_act, &t1, 1, MPI_DOUBLE, MPI_SUM,
                MPI_COMM_WORLD);
    }
    return t1 * t_scale;
}
```

Listing 6.6: WDS MiniMD `Thermo::temperature` kernel

allows us to analyse the impact of the library in a scenario which more accurately represents how applications are typically executed on large parallel systems. To show this, MiniMD and the larger problem size was used in a strong scaling manner, from one node to 16 nodes across each two architectures and systems, as can be seen in Table 6.1. It should be noted that Orac only allows for up to and including 14 nodes to be utilised for a single job.

Scaling has been examined with both the small and large problem sizes, and use the same reference and WDS versions as the single node performance analysis in Section 6.2.4. Each run uses only MPI parallelisation, using one rank per core. The maximum number of ranks that would ensure all cores have a

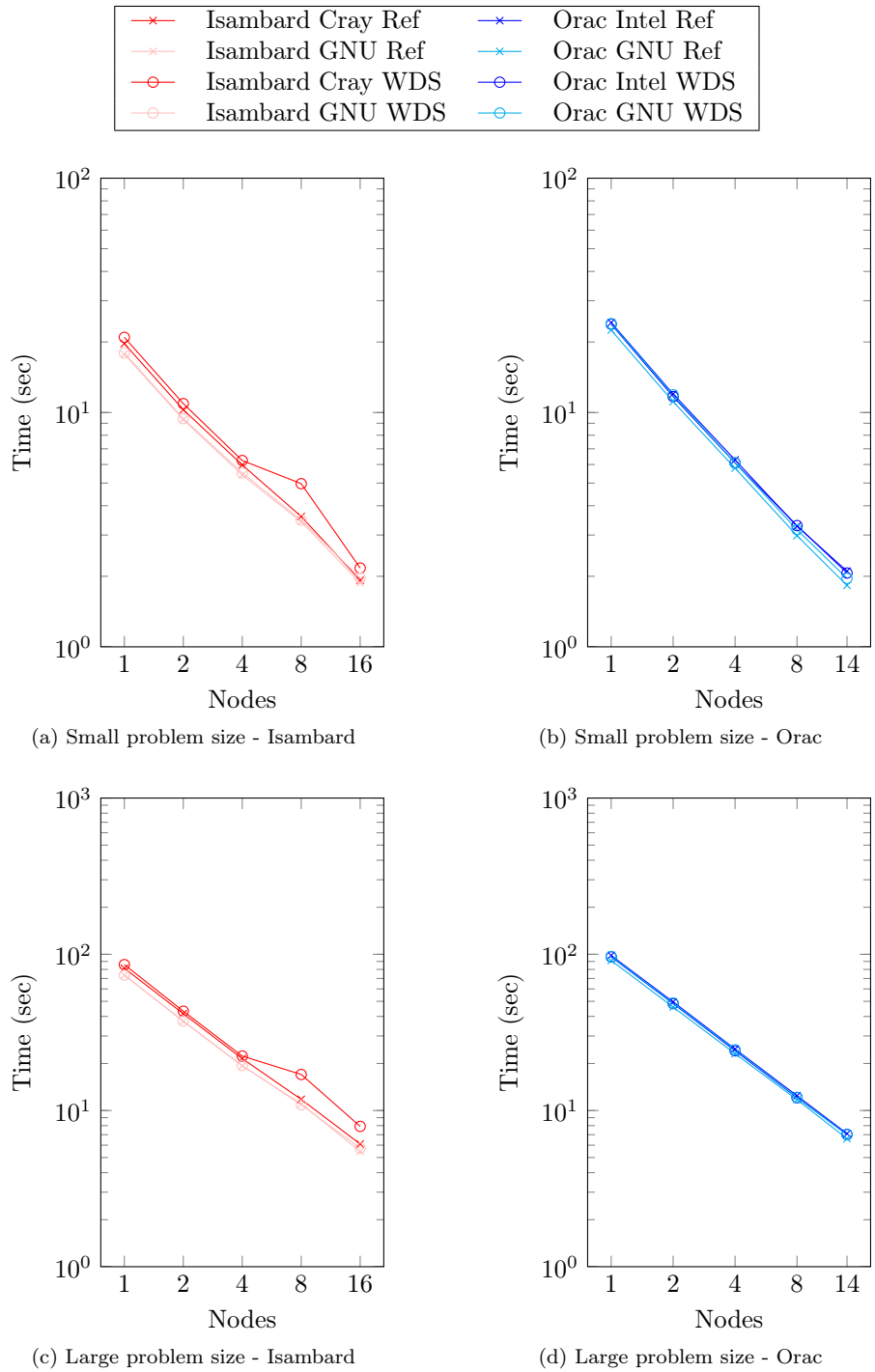
single rank has been used. Similarly to the testing of the mini-applications on a single node, each configuration of system, compiler and problem size was tested five times, and an average was used to calculate the runtime.

Nodes	Problem Size	Overhead(%)			
		Isambard		Orac	
		Cray	GNU	Intel	GNU
1	Small	6.33	1.37	-1.55	6.54
	Large	5.54	-0.02	-2.14	6.16
2	Small	6.18	1.06	-2.58	6.98
	Large	3.85	0.59	-2.52	6.13
4	Small	4.45	2.12	-2.96	5.30
	Large	3.86	0.48	-2.61	5.93
8	Small	38.1	1.65	0.63	6.06
	Large	44.3	-1.30	-2.87	6.25
14	Small			-1.85	7.05
	Large			-1.14	5.94
16	Small	12.7	4.81		
	Large	29.7	5.00		

Table 6.10: Results showing the overhead for all strong scaling results utilising MiniMD

The graphs in Figure 6.1 show the average time for each experimental setup across both problem sizes. The overheads for each configuration have also been presented in Table 6.10. Much like the tables presented in Section 6.2, a colour coding scheme has been used for Table 6.10. Results below 10% have been highlighted in grren, results between 10% and 30% have been highlighted in orange, and results above 30% have been highlighted in red. Across both data sets, we see that the difference in time decreases across the majority configurations, as the number of nodes increases. However, because the execution time also decreases as the number of nodes increases, the overhead value is consistent, or slightly increases with the node count. Even so, the overhead does not go over 8%, with the exception of Isambard Cray on 8 and 16 nodes.

The runtimes for the results presented in Table 6.10 can be seen in Figure 6.1. It is expected that the runtimes for both the reference and WDS ver-



(a) Small problem size - Isambard

(b) Small problem size - Orac

(c) Large problem size - Isambard

(d) Large problem size - Orac

Figure 6.1: Strong scaling results for MiniMD across all architectures and compilers for one to 16 nodes on Isambard and one to 14 nodes for Orac, utilising both problem sizes (64^3, 1000 timesteps for small problem size, 128^3, 500 timesteps for large problem size).

sions follow the same trend as the number of nodes increase. It is also expected that the trend will be inverse square, thus producing a straight line in a log-log graph. Figure 6.1 depicts the execution times on a log-log graph, and shows that most of the results follow the expected trend. This is especially true on the larger datasets, as the parallelisable region is larger. Figures 6.1a and 6.1c allow for the exploration of the higher overheads on the Isambard system when Cray compiler is utilised. It can be seen that this spike in overhead is not caused by additional parallelism in the reference version, but by an increase in time taken by the WDS version. From a brief exploration into the problem, this seems to be an issue with the implementation of the Cray compilers used to run these experiments. However, further exploration is required.

6.4 Summary

In this chapter, the performance impact for the data structure abstraction library WDS as described in Chapter 5, has been tested in multiple ways. These tests were carried out using a range of different Central Processing Unit (CPU) architectures, and each processor was tested using two compilers. Along with the fact that an average of five runs were taken when presenting the final runtime. All of this ensured that WDS was fairly and exhaustively tested.

Across the majority of the tests carried out, the overhead incurred for using the library is small. When exploring the benchmark kernels tested, the overhead does not go over 6% for both architectures and all compilers tested on. When the library was tested using mini-applications, the overhead decreased as the problem size increased. As such, on the larger problem sizes, the majority of the results were below 5%. For smaller problems, this overhead is larger, but still within reason. Finally, a scaling study was used with one of the mini-applications. This showed that the overhead did not fluctuate as the problem was scaled across multiple nodes.

CHAPTER 7

Data Structure Abstraction Library Specialisation

A key aim for a library within the High Performance Computing (HPC) space is for the library to either improve the performance it is implemented into, or at the very least, minimise the performance impact whilst providing new functionality that would not otherwise be possible. When developing the data structure abstraction library Warwick Data Store (WDS), the aim was to provide the functionality of being able to swap data structures, without making large alterations to an application. As such, it was expected that the library will impact the performance of a program, even if it is very slight, as shown in Chapter 6. However, the aim for WDS was that it should be flexible enough for new, specialised data structures to be implemented into the library, whilst maintaining this minimal impact to the library. In this chapter, specialised multi-material physics data structures have been used to show how WDS can be extended whilst maintaining the small performance impact.

The rest of the chapter is broken up into the following:

- Section 7.1 discusses the need to explore specialised data structures within WDS.
- Section 7.2 explores two multi-material data structures that have been implemented into WDS, and the key elements within each.
- Section 7.3 discusses how each of these data structures were implemented into the data structure abstraction library, with particular emphasis on usability and performance.
- Section 7.4 explores the performance impact of specialised data structures built within the library compared to a reference version.

7.1 Motivation

In real-world problems of interest, it is common to find highly specialised data structures that have been carefully designed to address a particular issue. Examples include lock-free hash tables for k-mer counting in Bioinformatics [66], Morton-ordered texture caches in computer graphics [125], and Compressed Sparse Row (CSR) data structures for sparse linear algebra [123].

Another example, which this chapter will focus on, are interface tracking algorithms in solid-fluid mechanics applications. Numerical methods designed for such applications often run into difficulties treating the sharp discontinuities in state variables that occur at boundaries between two distinct physical materials. Interface tracking methods are a broad family of approaches designed to ameliorate these issues by keeping a record of exactly where such boundaries are located, and applying correction terms to the solution variables in these areas. The methods used to store this boundary information are sometimes termed multi-material data structures [23].

A key design goal for WDS is to provide sufficient flexibility that specialised domain-specific data structures such as those required for multi-material problems can be efficiently described and stored using its mechanisms.

7.2 Multi-Material Data Structures

One of the key requirements of the data structure abstraction library is that it should be extensible. To demonstrate this, a specialised data structures for multi-material applications have been implemented into WDS. A multi-material problem is a particular subset of physics applications where the cells are not constrained by each cell occupying only one material. Figure 7.1 shows a simple multi-material problem. In this problem, a small 3×3 mesh can be seen, containing four different materials. Some of the cells in the mesh, such as Cell 0 and Cell 6, contain only one material. However, in the majority of cells, two or more materials occupy the same cell. If this was a single-material problem, the

mesh resolution (the number of cells within a mesh) would have to be increased, requiring large amounts of memory to store. Alternatively, the mesh resolution could remain the same, and the accuracy of the results would be decreased. Through the use of specialised data structures, an accurate representation of the problem can be achieved, with minimal extra use of memory. This example shows the key difficulties in representing multi-material problems. When describing and demonstrating data structures within this section, this example will be used as the basis.

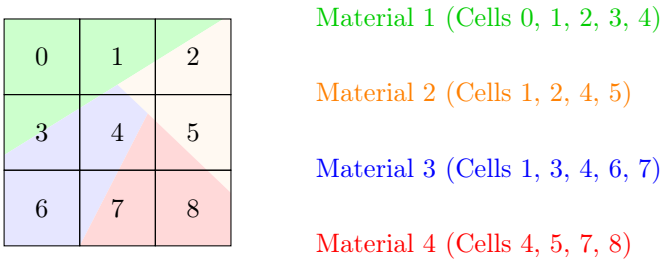


Figure 7.1: Graphic representation of multi-material mesh 3×3 mesh with four materials

To demonstrate that specialised data structures for multi-material applications can been implemented into WDS, two multi-material data structures have been utilised. In particular, these are the Compact Cell data structure outlined by Fogerty et al. [23] and a variant called Compact Cell Flat. In order to define a mesh in either data structure, WDS has been extended to include functions that allow for the addition and removal of materials from cells. Much like other WDS functions described in Sections 5 and 6, these take the variable name as the key identifier for the variable. Listing 7.1 shows how the materials for the cells described in Figure 7.1 can be added to a variable `"var"`. WDS is told that it is `COMPACTCELL` (a multi-material data structure, described and used in Sections 7.2.1 and 7.2.2) variable type, and thus expects two dimensions. The first is the number of cells in the grid, and the second is the maximum number of materials possible within the grid. In this case, this would be `{9, 4}`. Materials can then be added to the given variable using the `addMaterial` function.

This function takes the variable name and checks to see if it is a correct type (the variable is `COMPACTCELL`). WDS then extends the data structure for the given variable to include the material for the required cell, making any necessary adjustments to other cells, such as altering indirect access indexes. The function `removeMaterial` uses the same interface as `addMaterial`, but removes the required material from the given cell.

```
#include <wds.hpp>
/* Initialise dataStore, number of cells (numCells=9) and max
* number of materials (numMats=4) */

//Create var with the number of cells and max number of materials
dataStore.buildVar(WDS_DT::COMPACTCELL,
    wds::VarMeta("var", {numCells, numMats}, double()));

//Assign materials to variables and cells
dataStore.addMaterial("var", 0, 0);
dataStore.addMaterial("var", 1, 0);
dataStore.addMaterial("var", 1, 1);
dataStore.addMaterial("var", 1, 2);
dataStore.addMaterial("var", 2, 0);
    /* Materials added here */
dataStore.addMaterial("var", 7, 3);
dataStore.addMaterial("var", 8, 1);
dataStore.addMaterial("var", 8, 3);
```

Listing 7.1: WDS psudocode for adding materials to cells according to Figure 7.1

7.2.1 Compact Cell Multi-Material Data Structure

Figure 7.2 shows the Compact Cell data structure outlined by Fogerty et al. [23], and represents the mesh outlined in Figure 7.1. This data structure consists of two parts, one for storing all cells containing only a single material and associated metadata, and another for storing multi-material cells in the form of a packed linked list. The single-material portion of the data structure consists of the data for all single-material cells and the number of materials in each cell. It also stores either the material used in the cell, or the position of the

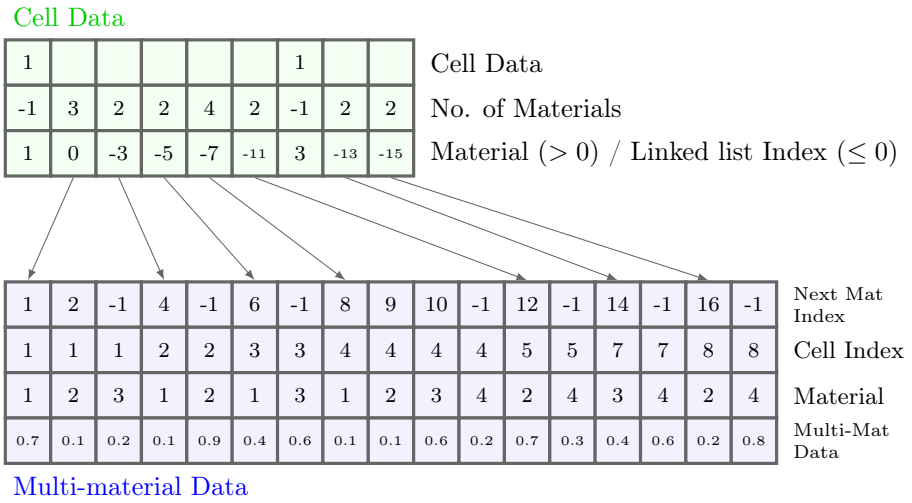


Figure 7.2: Graphical representation of Fogerty et al. Compact Cell [23] data structure using the example mesh shown in Figure 7.1

first multi-material element stored in the multi-material portion of the data structure. These links are represented by the arrows in the figure. The multi-material portion acts as a single linked list, where the head is provided by the single-material portion. Alongside the data for a given portion of the cell, the cell index (the cell corresponding to the segment) and the material is stored in the linked list.

In order to implement this data structure into WDS, two View classes were created that utilised the data from the `COMPACTCELL` variable type. One was created for the single-material cell data, and the other for the multi-material data. To request a particular view, the enum `WDS_DT` was extended with two new variables, `COMPACTCELL_SINGLE` and `COMPACTCELL_MULTI`. This allowed for each key section of the data structure to be accessed. Listing 7.2 shows how views can be created for both the single-material cell data, and the multi-material data. Each of the views contain specialised functions that allow for access to each element within each block. The key functions are discussed in Section 7.3.

```
auto varSingle =
    dataStore.getViewSpec<double, WDS_DT::COMPACTCELL_SINGLE>("var");
auto varMulti  =
    dataStore.getViewSpec<double, WDS_DT::COMPACTCELL_MULTI>("var");
```

Listing 7.2: Construction of WDS Views for Compact Cell

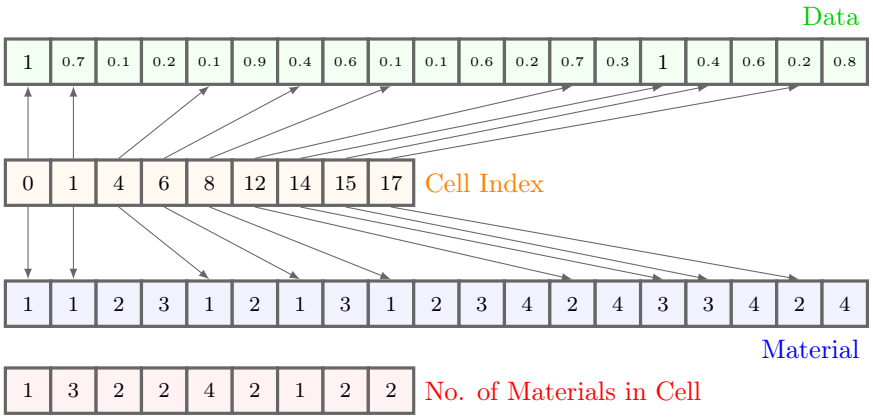


Figure 7.3: Graphical representation of WDS' Compact Cell Flat data structure using the example mesh shown in Figure 7.1

7.2.2 Compact Cell Flat Multi-Material Data Structure

As well as Compact Cell, a variation on this data structure was developed entitled Compact Cell Flat, shown in Figure 7.3. Much like Figure 7.2, Figure 7.3 shows how the Compact Cell Flat data structure can be used to represent the multi-material mesh in Figure 7.1. In this data structure, all the data is placed in a single, concurrent block of memory, placed in cell order, then in material order. A corresponding material array allows each segments material to be identified. The cell index array builds the link between a given cell and the first segment related to this cell. In order to iterate through a cell without having to perform additional calculations, a fourth and final array is provided with the number of elements.

```
auto varFlat =
    dataStore.getViewSpec<double, WDS_DT::COMPACTCELL>("var");
```

Listing 7.3: Construction of WDS Views for Compact Cell Flat

Due to the data structure having a simpler access pattern, only a single WDS view is required. As such, the view took the name of the variable type, `COMPACTCELL`. The view can be created in the same way as any other view, an example of which can be seen in Listing 7.3.

The rationale behind developing Compact Cell Flat was to combat some of the issues with Compact Cell with regard to iterating over data. The original Compact Cell implementation described in Section 7.2.1 relies on the single and multi-material data being split, and the fact that the multi-material data may not be in order with regards to the cells or their materials. Thus, Compact Cell is useful if the algorithm being used is one which looks at the single material data and multi-material data separately. However, if the algorithm requires iterating over single element materials, then Compact Cell will incur a heavy performance cost as indirection would be encountered. In addition, if the algorithm needs to apply a function on all data without needing to know if the cell is single or multi-material, then using Compact Cell would mean applying the same function on both, separate blocks of memory. Compact Cell Flat aims to overcome this problem by maintaining the data in contiguous memory (allowing it to be iterated over without the need to access multiple blocks of memory), in cell, and then in material order (ensuring minimal indirection memory accesses). However, this comes at a cost to performance if the algorithm only requires single or multi-material data.

7.3 Implementation of Abstract Data Structures

Out of both Compact Cell and Compact Cell Flat, the first data structure to be implemented into the library was Compact Cell Flat. As discussed in Chapter 5.5.3, the variable class was created first, storing each of the arrays. The arrays would be expanded and contracted as more materials were added and removed respectively. In order to maintain the ordering, the data was shifted where appropriate to fill in any gaps that could form. A corresponding view

class was then created. To do this efficiently, the data was placed into the data structure out-of-order, setting a flag in the process. When the view class was requested, the flag was checked and the data was re-ordered to the correct order if required.

In order to facilitate interactions between the user and the data structure, WDS treated the data as a two-dimensional array, where the first dimension is the cell index and the second is the material. The library passes this information to the variable and view classes, which then interprets it appropriately, depending on the required functionality. As an example: when adding a new material to the cell, the variable class will use this data to expand the arrays rather than altering the dimensions of the variables. Another example can be seen in the way in which the data can be accessed through the view class. As discussed in Chapter 5.3.3, `ViewSpec` has multiple ways in which the data can be accessed. Specialised ways in which the data can be accessed can therefore be created, depending on what is passed to the Compact Cell Flat view class. For Compact Cell Flat, `[int]` was designed to allow for direct access to the data array, `(int)` allowed for access to the first element of the given cell index thus allowing the user to iterate through the cell without needing to know the materials in the cell, and `(int, int)` allows for access to the given material for a given cell. If an incorrect element is given, the WDS specification specifies that this should be classed as undefined behaviour. Specialised functions were also created to allow for access to the material and number-of-materials arrays. All of this demonstrates the flexibility of WDS and can be seen in the example shown in Listing 7.4.

Compact Cell was also implemented into WDS. For this data structure to be implemented, a Structure of Arrays (SoA) implementation was used for both the single-material and multi-material data. Because a linked list is used in this data structure, the addition or removal of materials is simpler. Due to the arrangement of this particular data structure, it would not be possible to easily implement Compact Cell in a single view class. Instead, two views are required,

```
/* Compact Cell Flat data */
/* Set the third piece of data in the data structure (corresponds
 * to Cell 2, Material 2 in this example) */
varFlat[2] = 0.1;

// Set first material in cell 2 (Material 1 in this example)
varFlat(1) = 0.7;

//Set the third material in Cell 2 (may not be Material 3)
varFlat(1, 2) = 0.2;

//Prints the number of pieces of data, 17
std::cout << "Size: " << varFlat.size() << std::endl;

/* Prints whether a given cell contains multiple materials.
 * For Cell 2, this is true */
std::cout << "Check if Cell 1 contains multiple materials: "
    << varFlat.isMM(1) << std::endl;

/* Prints the number of materials for a given cell.
 * For Cell 2, this is 3 */
std::cout << "Number of materials in Cell 1: "
    << varFlat.sizeMM(1) << std::endl;

/* Get a pointer to the first element in material list for a
 * given cell */
auto matPointer = varFlat.getMaterials(1);
```

Listing 7.4: Uses of WDS Views for Compact Cell Flat

one for accessing data in the single material data, and one for the multi-material data. Because of the flexibility WDS has, this can easily be achieved. This is achieved by having two values in `DataStoreType` outlined in Chapter 5.3.2 and demonstrated in Listing 7.2 found in Section 7.2.1, one corresponding to the single-material section and one corresponding to the multi-material section. Much like the view class created for Compact Cell Flat, both of these view classes are specialised in order to allow the user to access the other arrays stored in the data structure as well as the data. The use of `[int]` and `(int)` mirrors that of the Compact Cell Flat data structure, but only the multi-material view has the need for the `(int, int)` access function, as the single-material view only has to worry about the cells. Example uses for both the single material and

multi-material WDS views can be seen in Listings 7.5 and 7.6 respectively.

```
/* Compact Cell Single material data */
// Set a single cell value. Can also use varSingle(0)
varSingle[0] = 1;

//Prints the number of cells, 9
std::cout << "Size:␣" << varSingle.size() << std::endl;

/* Prints the number of materials for a given cell.
 * For Cell 2, this is 3 */
std::cout << "Number␣of␣materials␣in␣Cell␣1:␣"
    << varSingle.sizeMM(1) << std::endl;

/* Prints whether a given cell contains multiple materials.
 * For Cell 2, this is true */
std::cout << "Check␣if␣Cell␣1␣contains␣multiple␣materials:␣"
    << varSingle.isMM(1) << std::endl;
```

Listing 7.5: Uses of WDS Views for Compact Cell (Single Material)

Because the two data structures are used for similar purposes, one of WDS' key features can be utilised, this being the conversion of variables from one data structure to another as described in Chapter 5.4.1. This functionality was built into the variable class, and was carried out when the user requested a given view class. When the user requested a Compact Cell Flat view object, a check would be done to see whether the variable was already in required the data structure. If the requested data structure differed from the data structure in use, then the variable class would convert the data structure, ordering the data as it was being converted. The converse would be true for the Compact Cell data structure.

```
/* Compact Cell Multi-material data */
// Set a fraction of a cell. Can also use varMulti(0)
varMulti[0] = 0.7;

/* Sets a particular cell and material value.
 * If material is not in cell, default value is given. */
varMulti(1, 2) = 0.1;

/* Gets the cell ID for a given index.
 * For [0], this would be cell 1. */
std::cout << "Cell␣ID:␣" << varMulti.getElement(0) << std::endl;

/* Gets the material ID for a given index.
 * For [0], this would be material 0. */
std::cout << "Cell␣ID:␣" << varMulti.getMaterial(0) << std::endl;

//Prints the number of multi-materials, 17
std::cout << "Size:␣" << varMulti.size() << std::endl;
```

Listing 7.6: Uses of WDS Views for Compact Cell (Multi-Material)

7.4 Performance of Data Structure Abstraction Library

As discussed in Section 5.5.3, one of the key features of WDS is that new data structures can be added with little cost to performance, and are able to perform transformations transparently to the user. To demonstrate this, the two multi-material data structures implemented into WDS, as described in Section 7.3 are going to be tested through two kernels.

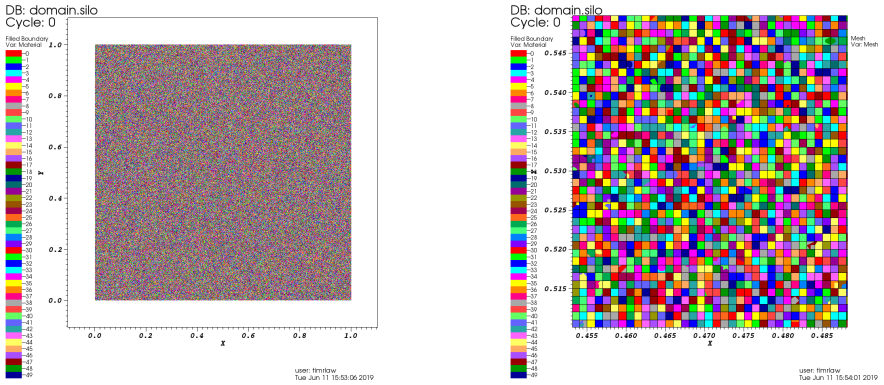
The first takes an average of all multi-material cells in a mesh, and stored this in a single-material array. As such, it is expected that the Compact Cell data structure would perform better than the Compact Cell Flat data structure. This due to the fact that, in the Comapct Cell data structure, the multi-material data has been independent of the single-material data, and so can be iterated easily. However, in the Comapct Cell Flat data structure, all data has to be iterated through, looking for mutli-material cells. Because of this, the average kernel was used to test the overhead of using WDS for specialised data structures.

The second kernel performs an Equation of State (EOS) material lookup. Two versions of the second kernel were made; one which iterated though the cells and found the EOS material it corresponded to, and the other iterated through the EOS materials and then searched for cells with that material. In the first version, the kernel iterates through all cells materials, performs a lookup and then a calculation based on this material. The second kernel performs a similar action, but iterates through the material list then finds cells with that material. Because both of these kernels do not need to know whether the cell is single-material or multi-material (only what materials a cell has), the Compact Cell Flat data structure should perform better than the Compact Cell data structure. This is because the Compact Cell Flat data structure has all the materials in concurrent memory, making it quick and fast to iterate through. As such, these kernels will be used to show how altering the data structure can increase the performance of certain kernels.

7.4.1 Experimental Setup

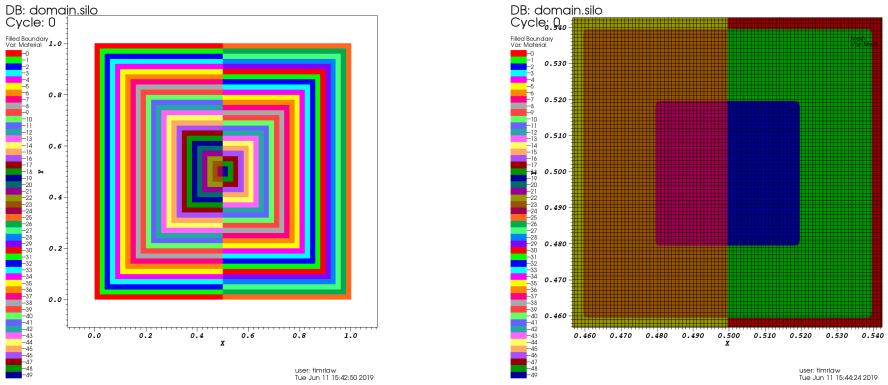
In order to exhaustively test the multi-material data structure and kernels, two meshes outlined by Fogerty et al. [23] were used. These meshes sit at the extremes of possible multi-material meshes. The first is a randomised mesh, with a given proportion of cells containing two, three and four materials within a cell and can be seen in Figure 7.4. In particular, 20% of cells were randomly picked to be multi-material cells. Of these multi-material cells, 62.5% were allocated two materials, 25% were allocated three materials, and the remaining 12.5% were allocated four materials.

The second multi-material mesh is a geometric patterned mesh, as seen in Figure 7.5. This consists of a much lower portion of multi-material cells to single material cells, compared to the randomised mesh (95% single-material cells and 5% multi-material cells). To ensure the kernels validate when using both meshes, and in order to guarantee a fair comparison, the mesh is generated once, and then duplicated for both the reference and WDS versions.



(a) Whole Mesh (b) Zoomed Mesh

Figure 7.4: Graphical example of a randomised multi-material mesh



(a) Whole Mesh (b) Zoomed Mesh

Figure 7.5: Graphical example of a geometric multi-material mesh

In order to measure the performance of the kernels specified, with both meshes, the kernels were built into the benchmarking suite outlined in Chapter 6.1. This allowed for quick testing of the kernels and data structures. Unlike the benchmarks, the multi-material kernels were tested on a larger range of Central Processing Unit (CPU). In fact, these kernels were tested on the same range of architectures and compilers as the mini-applications outlined in Chapter 6.2. A full breakdown of the architectures and compilers used can be seen in Table 6.4 in Chapter 6.2.1.

7.4.2 Results

The overhead results presented in this section (Tables 7.1 and 7.2) have been colour-coded with the same scheme as the overhead results presented in Chapter 6, and were calculated using the formula described in Chapter 2.6. Table 7.1, shows the average kernel across four different architectures, each using two different compilers. As can be seen, the overhead across the board is very low, less than 3% for most systems, compilers and across both types of mesh. Whilst expected, this shows that WDS can be extended easily, with very little impact to the performance.

System	**Compiler**	**Overhead (%)**	
		Random Mesh	**Geometric Mesh**
Isambard	Cray	2.03	-1.79
	GNU	-2.74	-3.32
Kingfisher	Intel	-1.54	-2.44
	GNU	-7.46	-6.35
Rome	AOCC	5.30	7.78
	GNU	11.7	16.2

Table 7.1: Results of multi-material average kernel within Benchmarking suite, across different architectures and compilers

Table 7.2 shows the overhead for the EOS kernels. In this table, it can be seen that a majority of systems and configurations have a negative overhead,

with the lowest being −32.1%. This negative overhead is most likely due to the fact that under the reference version, the EOS kernel has to be called twice (once for the single-material data, then again for the multi-material data), whereas the Compact Cell Flat version only needs to be called once. As well as this, the Compact Cell Flat data structure ensures that all the valid data is contiguous allowing for better utilisation of vectorisation. Whilst the data is accessed in contiguous order for Compact Cell, there are gaps where data does not need to be processed. Finally, when looking at the version of the EOS kernel which iterates through the cells first, the reference version is required to check that there is a valid material at all given positions before performing the calculations. This is not required in the Compact Cell Flat version, as the data structure ensures all pieces of data has a corresponding material.

System	**Compiler**	**Overhead (%)**			
		EOS (Cell)		**EOS (Mat)**	
		Random	**Geometric**	**Random**	**Geometric**
Isambard	Cray	-14.5	-3.57	4.74	-6.61
	GNU	-8.74	-1.95	-10.6	-27.5
Kingfisher	Intel	-30.8	-5.83	17.8	-23.7
	GNU	-25.3	-2.09	-0.55	2.33
Rome	AOCC	-25.2	6.37	8.81	-45.5
	GNU	-21.4	7.48	3.42	-32.1

Table 7.2: Results of multi-material EOS kernels within Benchmarking suite, across different architectures and compilers

There is also a difference between the two EOS kernels for each of the meshes. For the random mesh, the method of iterating through cells and then locating the materials performs better than the reverse. The opposite is true for the geometric mesh. However, on the kernels where the mesh does not match the best EOS kernel, the overhead is not large.

7.5 Summary

In this chapter, WDS has been shown to be flexible enough for new, specialised data structures to be implemented within it. This has been shown through the use of two multi-material data structures. Specifically, it has been shown how the library is able to expand to include the data structures in key functionality such as the conversion of variables from one data structure to another. It has also been shown that it is flexible enough to allow for different access patterns, depending on the requirements of the user application.

The performance of these data structures were compared to the reference multi-material data structure, across multiple architectures, compilers and meshes. It was shown that, in a like-for-like comparison, the overhead of utilising WDS is minimal, approximately 3% in most cases. The variant data structure built into the library was then compared to the same reference data structure for a different use case, and was shown to be a performance improvement.

CHAPTER 8

Conclusion and Future Work

Throughout this thesis, an emphasis has been placed on the development of memory within a High Performance Computing (HPC) system. This emphasis on memory can be seen in the increase in high bandwidth memory hardware developed in recent years, such as Intel's Xeon Phi Knights Landing (KNL) [103], the ARM Fujistu A64FX [25] and the NVIDIA A100 [78]. The emphasis on memory development, coupled with the growing diversity of hardware in the HPC space highlights the importance of the performance portability of these heterogeneous systems. This thesis has set out to show that more efficient and effective performance portability can be achieved through the use of libraries that can abstract concepts from a program, thus applying specialisations without the need for large rewriting of code. Hence the need for the data structure abstraction library Warwick Data Store (WDS).

Chapter 4 presented the way in which the performance of a heat-conduction proxy application differs between parallelisation libraries and architectures. It was shown that higher bandwidth memory can improve the performance of largely memory bound problems. The need for performance portability was also emphasised, and information was provided as to how this can be measured effectively from both an application and architectural viewpoint.

Chapter 5 focused on the development of a data structure abstraction library, WDS. The chapter documented how flexibility was incorporated into the design, without sacrificing performance. The additional features gained though the abstraction of the data structures highlighted another benefit of the library. Finally, the data structures implemented into the library were discussed, alongside steps taken to ensure minimal performance impact. Furthermore, ways in

which specialised data structures could be built, was discussed.

Chapter 6 explored the performance of WDS. The performance assessment of WDS was achieved through the implementation of the library into a collection of benchmark kernels and mini-applications. The collections were then tested across a multitude of different processors and compilers in order to ensure that the cost of utilising the library was low. WDS was also tested at scale to ensure that the library did not impact performance when utilised across multiple nodes. It was shown that the overhead was larger for smaller problems, but was still rather small (less than 30% for most settings). The overhead for larger problems decreased to less than 10% for nearly all cases, showing that WDS does not majorly impact the performance.

Chapter 7 described a specialised case for WDS, this being the use of multi-material data structures. In this chapter, two multi-material data structures were implemented into the library, and through the use of two kernels and a collection of architectures and compilers, it was shown that the performance of these custom data structures through the library is similar to that of the reference version. It was also shown that changing the data structure could further enhance performance.

The remainder of this chapter will focus on three areas. Firstly, in Section 8.1, the limitations of the work are discussed. Section 8.2 then explores how work carried out in this thesis could be taken forward, expanding upon the limitations discussed in Section 8.1. This includes expansions to WDS and the work undertaken on the multi-material kernels. A final reflection on the thesis is then presented in Section 8.3.

8.1 Limitations

The primary limitation of this thesis relates to the restricted scope of some of the areas that each chapter focused on. One such example of this is the range of systems examined in Chapter 4. Within this chapter, the performance

portability of the mini-application was measured across two different Intel systems as well as a NVIDIA Graphics Processing Unit (GPU). Whilst this covers the majority of HPC systems, there is an increasing variety of different processors available. This includes a growing presence of AMD processors, IBM Power 8 [30] and Power 9 [34] systems, and a larger variety of ARM processors such as Marvell's Thunder X2 [65] and Fujitsu's A64FX Central Processing Unit (CPU) [25] processors. The latter of these is now utilised to run the most powerful supercomputer in the world. [114]

Another limitation, driven by the necessity for a smaller scope, is seen in the development of the data structure abstraction library WDS which was discussed in Chapter 5. When developing the library, the main focus was on memory accessible to the CPU, and in particular, main memory. By adopting this approach, the possibility of using the library in Compute Unified Device Architecture (CUDA) applications, or in any application that utilises GPUs to achieve parallelism, was limited. This also comes with the added caveat that the library is only performance portable across CPU architectures, such as Intel Xeon CPUs, as well as KNL, AMD and ARM processors.

Furthermore, WDS was also designed with an additional limitation following it's initial implementation. As discussed briefly in Chapter 5.2, the initial implementation of WDS had a FORTRAN and C interface that could be used to interact with the library from a wider range of applications. However, due to their performance within the library, these interfaces were depreciated when transitioning to the current implementation. This meant that support for these languages were dropped, in favour of more development and optimisation in the core, C++ implementation.

8.2 Future Work

Whilst a significant amount of work in this area has already been achieved, there are still more areas of interest that can be explored. Such work would mainly

expand from the data structure abstraction library itself, and would include work on how improvements could be made and how more functionality could be included. The following sections discuss in detail the different aspects of the future work.

8.2.1 Warwick Data Store

Even with the feature set outlined in Chapter 5, the functionality provided by WDS has the potential to be extended in order to encompass more data operations. One of the key limitations of WDS is that it can only handle main memory of a CPU centric system, meaning that specialisations for different memory hierarchies are not accounted for. An example of this is the KNL system. In this system, a separate block of Multi-Channel Dynamic Random Access Memory (MCDRAM) can be configured into multiple different configurations. Depending on the memory configuration of the KNL system being utilised, it may be useful for the library to manage which data resides on the higher bandwidth MCDRAM, and what data is located on the larger-capacity main memory. Through this approach, processors with multiple Non-Uniform Memory Access (NUMA) regions, such as the KNL could be better supported.

Another area for expansion with the WDS library is the exploration of allowing data to reside across multiple devices on the same node, such as an accelerator such as a GPU. Whilst, in the case of CUDA capable GPUs, unified memory techniques can be utilised to get around this issue with minimal development time, this approach has been shown to be slower than manual allocation. [58] As such, expanding the capabilities of WDS such that it can handle memory on different devices, and transfer data more efficiently would be a beneficial feature in ensuring better performance portability.

Whilst WDS can handle Message Passing Interface (MPI) applications, these have been limited to Structure of Arrays (SoA) data structures. For Array of Structures (AoS) data structures, this would mean either sending individual values one at a time, packing and unpacking the data into an SoA buffer, or

knowing the underlying data structure and building an MPI data type that could iterate through the variable properly. None of these options are ideal when the library is designed to handle data structure interactions. Therefore, a good improvement to WDS would be to include mechanisms into the `View` and `ViewSpec` classes that would allow for better interactions with MPI. For example, this could include WDS generating the relevant MPI data type automatically.

Another feature that would be helpful when developing a mini-application with WDS, is the idea of grouping variables together. This would involve altering the data structure for one variable which would alter the data structure for the other variables, even if the the variables are not part of the same data structure. This is useful if, for example, the sizes of multiple SoA variables change together. Without using this feature, the application developer would have to iterate through each variable and apply the alteration to each individual variable. However, such an approach could easily lead to variables being missed, or incorrectly altered.

Finally, a useful additional feature for those working with WDS, would be to include a mechanism for the collection of statistics that are collated centrally and are reported either when requested, or when the data store object is destroyed. In order to eliminate the effect on performance on every run, this collection process would only be included when the program is compiled in debug mode, in the same way the C++ `assert` statements are handled. This could be very useful when analysing the data access pattern of a given program, and could even lead to recommendations on those data structure changes which may improve the performance of a program.

8.2.2 Multi-Material Data Structures

Multi-material physics within the HPC space is still relatively unexplored, especially when discussing their data structures. It would be a useful exercise to explore this area further and to examine the other data structures for multi-

material physics outlined by Fogerty et al. [23], in order to assess the performance in actual kernels, and to understand the effect on these data structures of different levels of parallelism and memory bandwidth. These data structures, outlined in Chapter 7.3, could be implemented into WDS and may be able to follow the same format as the already-implemented Compact Cell and Compact Cell Flat data structures. From this, the data structures could be swapped between themselves by changing a single value, and recompiling the code.

In addition, it would be interesting to explore whether it is possible to create a variation of Compact Material (a similar data structure to Compact Cell outlined in Chapter 7.3 where the data structure orders the data in material order through a linked list, both designed by Fogerty et al. [23]). This variant would be similar to the differences between the Compact Cell Flat and the Compact Cell data structure. As such, the Compact Material Flat data structure outlined would aim to place all the data in concurrent memory, at the cost of being able to add or remove cells from materials easily.

As shown in Chapter 7.4, each of the six data structures has the potential to perform best for a given kernel. It is therefore apparent that there is a requirement to convert the multi-material data structure from one to another. Whilst WDS has mechanisms in place to handle such conversions, care would have to be taken in order to ensure that the conversion process is applied efficiently and in a way that ensures that data is not lost between conversions. This approach could result in the provision of a data structure optimisation analysis for a given collection of multi-material kernels or potentially for a mini-application.

8.2.3 Data Structure Optimisations

As alluded to in Section 8.2.2, the work covered in the thesis has not included a performance analysis on the conversion of variables from one data structure to another. This is, however, an interesting area of research, and one that would need to be explored over multiple different scenarios. This would involve identifying kernels within a mini-application that could be improved by altering

the data structure. Altering the data structure could involve a complete restructuring of the data structure (for example, SoA to AoS or vice versa) or an alteration within a variable (for example, changing the data adjacency as outlined in Chapter 5.4.2). The performance of the reference version, the original WDS version and modified WDS version would then be measured, examining both the hardware utilisation through the use of profiling tools and the runtimes of the kernels and conversion process. By applying this method across a range of architectures and problem sizes, it can be ascertained at what point it is worth altering the data structure for better performance on a particular kernel, and also when the conversion process is more expensive than the gains provided by the alteration.

In order for this analysis to be useful, it would be necessary to explore the utilisation of the current data structure and the performance of the conversion process. Otherwise, the effective point at which the conversion process would be better in terms of the performance, would be skewed and may be higher than required. As such, the performance of this conversion process needs to be as optimal as possible.

8.2.4 Just-In-Time Compilation

Within WDS, the majority of calculations performed are done at runtime. This is due to the fact that the information required for these operations, such as calculating the offset and access patterns for a given data structure, are only provided once the program has begun execution. However, once the data structure is created, this information is not often changed. As such, it would be beneficial for the performance of the library to calculate these values prior to compilation, and use them as static, constant values. In cases where the data structure is altered, a new data structure would be created and the old one discarded. This process would help the performance of the library, as the compiler knows what will be constant, and can ensure that the program does not make unnecessary checks to see if these values have changed.

The aforementioned scenario highlights where newer techniques such as Just-In-Time (JIT) compilation could be beneficial to WDS. JIT compilation is formed of two stages. The first is the compilation of only the required and necessary code. The remainder of the code, or code which has been labelled, is then packaged together with a small program. While the program is executed, the second stage is executed. During execution, the program will reach a section which has not yet been compiled. When this occurs, the small program bundled with the uncompiled code is executed, and compiles the next section of code. The original program continues executing, now with the newly compiled code which is specialised to the given state of the program. The use of JIT compilation could be explored for WDS, to ascertain whether a performance improvement could be gained, bringing the library closer to a 0% overhead for more situations.

8.3 Reflections

With the rapid pace of development seen within the HPC research community, the architecture of future systems is not, nor has it ever been, certain. Some trends, such as the growth of memory-focused systems and the reliance on performance portable code, will continue into the foreseeable future. This thesis only scratches the surface on memory-based improvements from both a software and hardware perspective, and touches on some of the key issues within the area.

Bibliography

[1] AMD. AMD EPYC 7742. https://www.amd.com/en/products/cpu/amd-epyc-7742 (Accessed: 28th August 2020), 2020.

[2] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Spring Joint Computer Conference (AFIPS '67 (Spring))*, pages 483–485, Atlantic City, NJ, April 1967. Association for Computing Machinery, New York, NY.

[3] Arm. Arm Forge | Profiling with Arm MAP - Arm Developer. https://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge/resources/tutorials/profiling-with-arm-map (Accessed: 10th August 2020), 2020.

[4] P. G. H. Bachmann. *Die analytische Zahlentheorie. Dargestellt von Paul Bachmann.* Leipzig B.G. Teubner, 1894. (in German).

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks—Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*, pages 158–165, Albuquerque, NM, August 1991. Association for Computing Machinery, New York, NY.

[6] B. Barney. Message Passing Interface (MPI). https://computing.llnl.gov/tutorials/mpi/ (Accessed: 26th July 2020), 2020.

[7] B. Barney. POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/ (Accessed: 25th July 2020), 2020.

[8] Berkeley Lab. 2019 International Workshop on Performance, Portability and Productivity in HPC (P3HPC). https://p3hpc2019.lbl.gov/ (Accessed: 14th August 2020), 2019.

[9] Berkeley Lab. DOE Performance, Portability and Productivity Annual Meeting. https://doep3meeting2019.lbl.gov/ (Accessed: 14th August 2020), 2019.

[10] C. Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[11] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz. Caliper: Performance Introspection for HPC Software Stacks. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 550–560, Salt Lake City, UT, November 2016. IEEE Computer Society, Los Alamitos, CA.

[12] M. Ciżnicki, M. Kierzynka, P. Kopta, K. Kurowski, and P. Gepner. Benchmarking Data and Compute Intensive Applications on Modern CPU and GPU Architectures. *Procedia Computer Science*, 9:1900–1909, April 2012.

[13] cppreference.com. assert. `https://en.cppreference.com/w/cpp/error/assert` (Accessed: 24th July 2021), 2020.

[14] cppreference.com. inline specifier. `https://en.cppreference.com/w/cpp/language/inline` (Accessed: 9th August 2020), 2020.

[15] T. Deakin, S. McIntosh-Smith, and W. P. Gaudin. Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale. In *High Performance Computing*, volume 9697, pages 429–448. Springer-Verlag, Berlin, Germany, June 2016.

[16] W. Deconinck, P. Bauer, M. Diamantakis, M. Hamrud, C. Kühnlein, P. Maciel, G. Mengaldo, T. Quintino, B. Raoult, P. K. Smolarkiewicz, and N. P. Wedi. Atlas: A library for numerical weather prediction and climate modelling. *Computer Physics Communications*, 220:188–204, November 2017.

[17] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, July 2003.

[18] H. C. Edwards and C. R. Trott. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 18–24, Boulder, CO, August 2013. IEEE Computer Society, Los Alamitos, CA.

[19] Exascale Computing Project. ECP Proxy Applications. `https://proxyapps.exascaleproject.org/` (Accessed: 13th August 2020), 2020.

[20] F H McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical report, Lawrence Livermore National Lab, United States Department of Energy, Livermore, CA, December 1986.

[21] P. J. Fleming and J. J. Wallace. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *Communications of the ACM*, 29(3):218–221, March 1986.

[22] M. J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[23] 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*, 78(2):565–581, July 2019.

[24] K. Franko. MiniAero and Aero: An Overview. Technical report, Sandia National Laboratories, Albuquerque, NM, 2015.

[25] Fujitsu Limited. Fujitsu Presents Post-K CPU Specifications. `https://www.fujitsu.com/global/about/resources/news/press-releases/2018/0822-02.html` (Accessed: 5th July 2020), 2018.

[26] S. Futral and LLNL. ASC Sequoia Benchmark Codes. `https://asc.llnl.gov/sequoia/benchmarks/` (Accessed: 13th August 2020), 2013.

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

[28] GNU. 6.45 An Inline Function is As Fast As a Macro. `https://gcc.gnu.org/onlinedocs/gcc/Inline.html` (Accessed: 9th August 2020), 2020.

[29] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982.

[30] D. Henderson. POWER8 Processor-Based Systems RAS. Technical report, IBM Server and Technology Group, Armonk, NY, March 2016.

[31] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications. Technical report, Sandia National Laboratories, Albuquerque, NM, September 2009.

[32] R. D. Hornung and J. A. Keasler. The RAJA Portability Layer: Overview and Status. Technical report, Lawrence Livermore National Lab, Livermore, CA, November 2014.

[33] IBM. Fact Sheet & Background: Roadrunner Smashes the Petaflop Barrier. `https://www-03.ibm.com/press/us/en/pressrelease/24405.wss` (Accessed: 23rd September 2020), 2008.

[34] IBM. POWER9 processor chip | IBM. https://www.ibm.com/it-infrastructure/power/power9 (Accesssed: 6th September 2020), 2020.

[35] IEEE Computer Society. 1003.1-2008 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)). Technical report, IEEE Standards Association, September 2008.

[36] Intel. Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor White Paper. Technical report, Intel, Santa Clara, CA, March 2004.

[37] Intel. Intel Advanced Vector Extensions Programming Reference. Technical report, Intel Corporation, January 2009.

[38] Intel. Optimizing Performance with Intel Advanced Vector Extensions. Technical report, Intel Corporation, September 2014.

[39] Intel. Intel MPI Library. https://software.intel.com/content/www/us/en/develop/tools/mpi-library.html (Accessed: 26th July 2020), 2020.

[40] Intel. Intel VTune Profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html (Accessed: 10th August 2020, 2020.

[41] Intel. Intel Xeon Phi Processor 7210. https://ark.intel.com/content/www/us/en/ark/products/94033/intel-xeon-phi-processor-7210-16gb-1-30-ghz-64-core.html (Accessed: 28th August 2020), 2020.

[42] Intel. Intel Xeon Platinum 9242 Processor. https://ark.intel.com/content/www/us/en/ark/products/194145/intel-xeon-platinum-9242-processor-71-5m-cache-2-30-ghz.html (Accessed: 28th August 2020), 2020.

[43] Intel. Intel Xeon Processor E5-2660 v4. https://ark.intel.com/content/www/us/en/ark/products/91772/intel-xeon-processor-e5-2660-v4-35m-cache-2-00-ghz.html (Accessed: 28th August 2020), 2020.

[44] Intel. Intel Xeon Processor E5-2680 v4. https://ark.intel.com/content/www/us/en/ark/products/91754/intel-xeon-processor-e5-2680-v4-35m-cache-2-40-ghz.html (Accessed: 22nd September 2020), 2020.

[45] International Organization for Standardization. ISO/IEC 9899:1999 Programming languages - C. Technical report, International Organization for Standardization, December 1999.

[46] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*, pages 251–267. Elsevier, Amsterdam, Netherlands, 2016.

[47] K. Thompson, D. M. Ritchie. *UNIX PROGRAMMER'S MANUAL*. Bell Telephone Laboratories, Incorporated, 5 edition, June 1974.

[48] X. ke Liao, K. Lu, C. qun Yang, J. wen Li, Y. Yuan, M. che Lai, L. bo Huang, P. jing Lu, J. bin Fang, J. Ren, and J. Shen. Moving from exascale to zettascale computing: challenges and techniques. *Frontiers of Information Technology & Electronic Engineering*, 19(10):1236 – 1244, October 2018.

[49] R. O. Kirk, T. R. Law, S. Maheswaran, and S. A. Jarvis. Warwick Data Store: A HPC Library for Flexible Data Storage in Multi-Physics Applications. In *Super Computing 19 (SC19)*, Denver, CO, November 2019. Association for Computing Machinery, New York, NY.

[50] 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 *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 834–841, Honolulu, HI, September 2017. IEEE Computer Society, Los Alamitos, CA.

[51] R. O. Kirk, M. Nolten, R. Kevis, T. R. Law, S. Maheswaran, S. A. Wright, S. Powell, G. R. Mudalige, and S. A. Jarvis. Warwick Data Store: A Data Structure Abstraction Library. In *11th IEEE International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS20)*, pages 71–85, Atlanta, GA, November 2020. IEEE Computer Society, Los Alamitos, CA.

[52] P. Kogge and J. Shalf. Exascale computing trends: Adjusting to the "new normal" for computer architecture. *Computing in Science & Engineering*, 15(6):16–26, October 2013.

[53] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[54] T. R. Law, R. Kevis, S. Powell, J. Dickson, S. Maheswaran, J. A. Herdman, and S. A. Jarvis. Performance Portability of an Unstructured Hydrodynamics Mini-application. In *Proceedings of 2018 International Workshop on Performance, Portability, and Productivity in HPC (P3HPC)*, Dallas, TX, November 2018. Association for Computing Machinery, New York, NY.

[55] Lawrence Livermore National Laboratory. DOE Centers of Excellence Performance Portability Meeting. Technical report, Lawrence Livermore National Laboratory, Glendale, AZ, 2016.

[56] Lawrence Livermore National Laboratory. Axom. `https://github.com/LLNL/axom` (Accessed: 4th July 2020), 2020.

[57] Lawrence Livermore National Laboratory. Sidre User Documentation. `https://axom.readthedocs.io/en/develop/axom/sidre/docs/sphinx/index.html` (Accessed: 4th July 2020), 2020.

[58] W. Li, G. Jin, X. Cui, and S. See. An Evaluation of Unified Memory Technology on NVIDIA GPUs. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1092–1098, Shenzhen, China, May 2015. IEEE Computer Society, Los Alamitos, CA.

[59] A. C. Mallinson, D. A. Beckingsale, W. P. Gaudin, J. A. Herdman, J. M. Levesque, and S. A. Jarvis. CloverLeaf: Preparing Hydrodynamics Codes for Exascale. In *A New Vintage of Computing (CUG2013)*, pages 1–15, Napa, CA, May 2013. Cray User Group.

[60] Mantevo Organization. Mantevo Project | Mantevo. `https://mantevo.github.io/` (Accessed: 13th August 2020), 2020.

[61] I. L. Markov. Limits on fundamental limits to computation. *Nature*, 512(7513):147–154, 2014.

[62] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. P. 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)*, pages 1–10, Barcelona, Spain, March 2016. Association for Computing Machinery, New York, NY.

[63] M. Martineau, S. McIntosh-Smith, M. Boulton, W. P. Gaudin, and D. Beckingsale. A Performance Evaluation of Kokkos & RAJA using the TeaLeaf Mini-App. In *Supercomputing 2015 (SC15)*, Austin, TX, November 2015.

[64] 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), January 2017.

[65] Marvell. Manufacturing Applications on Marvell ThunderX2. Technical report, Marvell, Hamilton, Bermuda, June 2019.

[66] G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, January 2011.

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

[68] S. McIntosh-Smith and T. Mattson. Chapter 22 - Portable Performance with OpenCL. In J. Reinders and J. Jeffers, editors, *High Performance Parallelism Pearls*, pages 359–375. Morgan Kaufmann, Boston, 2015.

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

[70] G. E. Moore. Progress in digital integrated electronics. *International Electron Devices Meeting*, pages 11 – 13, 1975.

[71] G. R. Mudalige, I. Z. Reguly, M. B. Giles, A. C. Mallinson, W. P. Gaudin, and J. A. Herdman. Performance Analysis of a High-Level Abstractions-Based Hydrocode on Future Computing Systems. In *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS 2014)*, pages 85–104, New Orleans, LA, November 2014. Springer International Publishing, New York City, NY.

[72] NASA. NAS Parallel Benchmarks. `https://www.nas.nasa.gov/publications/npb.html` (Accessed: 12th August 2020), 2020.

[73] A. A. Nasar. The history of Algorithmic complexity. *The Mathematics Enthusiast*, 13(3):226–227, August 2016.

[74] W. F. Noh. Errors for Calculations of Strong Shocks Using an Artificial Viscosity and an Artificial Heat Flux. *Journal of Computational Physics*, 72(1):78–120, September 1987.

[75] D. A. Nowak and R. C. Christensen. ASCI Applications. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, November 1997.

[76] NVIDIA. NVIDIA Tesla P100. Technical report, NVIDIA, Santa Clara, CA. WP-08019-001_v01.1.

[77] NVIDIA. CUDA Zone | NVIDIA Developer. `https://developer.nvidia.com/cuda-zone` (Accessed: 14th August 2020), 2020.

[78] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. Technical report, NVIDIA, Santa Clara, CA, 2020. V1.0.

[79] NVIDIA Developer Zone. Profiler :: CUDA Toolkit Documentation. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html` (Accessed: 10th August 2020, 2020.

[80] OpenACC. OpenACC. `https://www.openacc.org/` (Accessed: 14th August 2020), 2020.

[81] OpenMP. OpenMP. `https://www.openmp.org/` (Accessed: 25th July 2020), 2020.

[82] OpenMP Architecture Review Board. OpenMP Application Program Interface. Technical report, OpenMP, July 2013.

[83] OpenMPI. Open MPI: Open Source High Performance Computing. `https://www.open-mpi.org` (Accessed: 26th July 2020), 2020.

[84] OpenPBS. OpenPBS Open Source Project. `https://www.openpbs.org/` (Accessed: 26th July 2020), 2020.

[85] L. A. Parnell, D. W. Demetriou, V. Kamath, and E. Y. Zhang. Trends in High Performance Computing: Exascale Systems and Facilities Beyond the First Wave. In *2019 18th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*, pages 167 – 176, Las Vegas, NV, May 2019. IEEE Computer Society, Los Alamitos, CA.

[86] A. Peleg and U. Weise. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[87] S. J. Pennycook, J. D. Sewall, and V. W. Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 92:947–958, March 2019.

[88] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. `http://www.netlib.org/benchmark/hpl/` (Accessed: 14th June 2020), 2018. Version 2.3.

[89] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable Performance on Heterogeneous Architectures. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13)*, pages 431–444, Houston, TX, March 2013. Association for Computing Machinery, New York, NY.

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

[91] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro*, 20(4):47–57, 2000.

[92] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. J. Kelly, and D. Radford. Acceleration of a Full-scale Industrial CFD Application with OP2. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1265–1278, July 2015.

[93] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Design and Development of Domain Specific Active Libraries with Proxy Applications. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 738–745, Chicago, IL, September 2015. IEEE Computer Society, Los Alamitos, CA.

[94] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Loop Tiling in Large-Scale Stencil Codes at Run-time with OPS. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):873–886, November 2017.

[95] J. Reinders. Intel AVX-512 Instructions. `https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html` (Accessed 24th July 2020), 2013.

[96] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas. The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications. In *2011 18th International Conference on High Performance Computing (HIPC 2011)*, pages 1–10, Bangalore, India, 2011. IEEE Computer Society, Los Alamitos, CA.

[97] S. Sawadsitang, J. Lin, S. See, F. Bodin, and S. Matsuoka. Understanding Performance Portability of OpenACC for Supercomputers. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pages 699–707, Hyderabad, India, 2015. IEEE Computer Society, Los Alamitos, CA.

[98] SchedMD. Slurm Workload Manager - Overview. `https://slurm.schedmd.com/overview.html` (Accessed: 26th July 2020), 2020.

[99] Science History Institute. Gordon E. Moore. `https://www.sciencehistory.org/historical-profile/gordon-e-moore` (Accessed: 22nd July 2020).

[100] J. M. Shalf and R. Leland. Computing beyond Moore's Law. *Computer*, 48(12):14–23, 2015.

[101] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzar'n, and D. Padua. Performance Portability with the Chapel Language. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 582–594, Shanghai, China, May 2012. IEEE Computer Society, Los Alamitos, CA.

[102] G. A. Sod. A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws. *Journal of Computational Physics*, 27(1):1–31, April 1978.

[103] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, April 2016.

[104] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. *SIGPLAN Notices*, 29(6):196–205, June 1994.

[105] T. Sterling, M. Anderson, and M. Brodowicz. Chapter 16 - The Essential OpenACC. In T. Sterling, M. Anderson, and M. Brodowicz, editors, *High Performance Computing*, pages 483–508. Morgan Kaufmann, Boston, 2018.

[106] S. R. Sukumar, M. A. Matheson, R. Kannan, and S.-H. Lim. Mini-apps for High Performance Data Analysis. In *IEEE International Conference on Big Data (Big Data)*, pages 1483–1492, Washington, DC, December 2016. IEEE Computer Society, Los Alamitos, CA.

[107] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173, Dresden, Germany, 2010. Springer, Berlin Heidelberg.

[108] S. Thakkur and T. Huff. Internet Streaming SIMD Extensions. *Computer*, 32(12):26–34, 1999.

[109] The Khronos Group Inc. OpenCL Overview - The Khronos Group Inc. https://www.khronos.org/opencl/ (Accessed: 14th August 2020), 2020.

[110] The Khronos Group Inc. SYCL Overview - The Khronos Group Inc. https://www.khronos.org/sycl/ (Accessed: 14th August 2020), 2020.

[111] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 878–883, Portand, OR, 1993. Association for Computing Machinery, New York, NY.

[112] T. N. Theis and H.-S. P. Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science Engineering*, 19(2):41–50, 2017.

[113] Top500. Home - Top500. https://www.top500.org/ (Accessed: 26th July 2020), 2020.

[114] Top500. Japan Captures TOP500 Crown with Arm-Powered Supercomputer. https://www.top500.org/news/japan-captures-top500-crown-arm-powered-supercomputer/ (Accessed: 6th September 2020), 2020.

[115] Top500. The Linpack Benchmark. https://top500.org/project/linpack/ (Accessed: 12th August 2020), 2020.

[116] D. R. Truby, S. A. Wright, R. Kevis, S. Maheswaran, J. A. Herdman, and S. A. Jarvis. BookLeaf : an unstructured hydrodynamics mini-application. In *Third International Workshop on Representative Applications*, pages 1–8, Belfast, United Kingdom, September 2018. IEEE Computer Society, Los Alamitos, CA.

[117] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing on-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 392–403, S. Margherita Ligure, Italy, 1995. Association for Computing Machinery, New York, NY.

[118] UK Mini-App Consortium. UK Mini-App Consortium. https://uk-mac.github.io/ (Accessed: 13th August 2020), 2014.

[119] UK Mini-App Consortium. UK-MAC/TeaLeaf. https://github.com/UK-MAC/TeaLeaf (Accessed: 16th August 2020), 2020.

[120] University of Bristol. TeaLeaf. `https://github.com/UoB-HPC/TeaLeaf` (Accessed: 18th June 2019), 2019.

[121] Valgrind Developers. Valgrind. `https://valgrind.org/` (Accessed: 12th August 2020), 2020.

[122] D. W. Walker. Standards for Message-Passing in a Distributed Memory Environment. In *Center for Research on Parallel Computing (CRPC) workshop on standards for message passing in a distributed memory environment*, Williamsburg, VA, April 1992. U.S. Department of Energy Office of Scientific and Technical Information, Washington, DC.

[123] J. B. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *Proceedings Fourth International Conference on High-Performance Computing*, pages 66–71, Bangalore, India, December 1997. IEEE Computer Society, Los Alamitos, CA.

[124] R. S. Williams. What's Next? [The end of Moore's law]. *Computing in Science & Engineering*, 19(2):7–13, 2017.

[125] D. S. Wise. Ahnentafel Indexing into Morton-Ordered Arrays, or Matrix Locality for Free. In *Euro-Par 2000 Parallel Processing*, pages 774–783, Munich, Germany, August 2000. Springer, Berlin, Heidelberg.

[126] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31(2):50–59, 2011.

[127] S. A. Wright, S. J. Pennycook, S. D. Hammond, and S. A. Jarvis. RIOT : a parallel input/output tracer. In *UK Performance Engineering Workshop (UKPEW'11)*, Bradford, United Kingdom, July 2011.

[128] J. Zerr and R. Baker. SNAP: SN (Discrete Ordinates) Application Proxy. `https://github.com/lanl/SNAP` (Accessed: 14^{th} June 2020), 2020.

[129] W. Zhu, Y. Niu, and G. R. Gao. Performance Portability on EARTH: A Case Study Across Several Parallel Architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, April 2005. IEEE Computer Society, Los Alamitos, CA.

Appendices

APPENDIX A

Compilers and compiler flags used for *Analysing the Performance Portability of a Heat-Conduction Mini-Application* (Chapter 4)

Version	**Compiler**	**Flags**
OpenMP, Message Passing Interface (MPI), Hybrid	Intel 17.0u2, IMPI 2017u2	`-O3 -no-prec-div -fpp -align array64byte -qopenmp -ip -fp-model strict -fp-model source -prec-div -prec-sqrt`

Compute Unified Device Architecture (CUDA)	Intel 17.0u2, CUDA 8.0.61	`ifort -O3 -fpp -no-prec-div -qopenmp -fp-model strict -fp-model source -prec-div -prec-sqrt`
		`nvcc -gencode arch=compute_60,code=sm_60 -restrict -DNO_ERR_CHK -O3`
		`icc -O3 -qopenmp -fp-model strict -fp-model source -prec-div -prec-sqrt`

OpenACC	PGI 17.3, OpenMPI 1.10.6	`-O3 -acc (-ta=multicore` **`or`** `-ta=tesla:cc60) -mp`

Table A.1: List of the manual implementation of TeaLeaf with compilers and corresponding flags used on the single node, multi-core systems

Version	Compiler	Flags
OpenMP, MPI, Hybrid, MPI Tiled	Intel 17.0u2, IMPI 2017u2	`-O3 -ipo -fp-model strict -fp-model source -no-prec-div -prec-sqrt -vec-report2 -xHost -parallel -restrict -fno-alias -inline-forceinline -qopenmp`
CUDA	Intel 17.0u2, IMPI 2017u2, CUDA 8.0.61	`icc -O3 -ipo -fp-model strict -fp-model source -no-prec-div -prec-sqrt -vec-report2 -xHost -parallel -restrict -fno-alias -inline-forceinline -qopenmp nvcc -O3 -restrict -use_fast_math -gencode arch=compute_60,code=sm_60`
OpenACC	PGI 17.3, OpenMPI 1.10.6	`-acc -ta=tesla:cc60 -O2 -Kieee -Minline -ldl`

Table A.2: List of the Oxford Parallel Library for Structured mesh solvers (OPS) implementation of TeaLeaf with compilers and corresponding flags used on the single node, multi-core systems (CUDA ran with a block size of 64 by 8)

	Version	Compiler	Flags
Kokkos	OpenMP	Intel 17.0u2	`-O3 -no-prec-div -fpp -fp-model strict -fp-model source -prec-div -prec-sqrt`
	CUDA	GNU-5.4.0, CUDA 8.0.61	`-O3 -march=native -funroll-loops -DKOKKOSP_ENABLE_PROFILING -ffloat-store`
RAJA	OpenMP	Intel 17.0u2, IMPI 2017u2	`-O3 -no-prec-div -restrict -fno-alias -xhost -std=c++11 -qopenmp -DNO_MPI -DENABLE_PROFILING`
	CUDA	GNU-5.4.0, CUDA 8.0.61	`g++ -march=native -funroll-loops -std=c++11 -ffloat-store -fopenmp nvcc -ccbin g++ -O2 -expt-extended-lambda -restrict -arch compute_60 -std=c++11 -Xcompiler -fopenmp -x cu`

Table A.3: List of both the Kokkos and RAJA versions of TeaLeaf with compilers and corresponding flags used on the single node, multi-core systems