Modern Vector Architectures for High-Performance Computing

Andrei Poenaru

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

June 2021

Word count: 42,675

Abstract

Recent generations of general-purpose central processing units (CPUs) for the high-performance segment have had to adopt new approaches in order to deliver increasing performance. Clock frequency has increased little, but the number of cores per chip has increased by several times in a single decade. Inside each core, single instruction, multiple data (SIMD) capabilities have also increased in capacity, resulting in modern vector processors that can achieve peak performance close to that of graphics processing units (GPUs), while maintaining the versatility of a general-purpose processor. These increases in compute power, however, have not been met with similar advances in memory performance.

These architectural changes have coincided with another change in the High-Performance Computing (HPC) landscape: Arm-based processor designs have made their way into supercomputer systems alongside commodity x86 processors. These designs have come in the form of custom implementations from several vendors, and they aim to address deficiencies in both compute and memory performance for the HPC environment. Arm's implementation of wide SIMD is called the Scalable Vector Extension (SVE), and it represents a modern implementation of ideas first seen in the vector architectures of the original Cray supercomputers of the 1970s. For memory bandwidth, the novelty of these Arm-based designs lies in a significant increase in the number of memory channels available, and even in bringing high-bandwidth memory from GPUs to CPUs.

This thesis is a study of modern CPU architectures for HPC. The focus of this research is on the efficacy of the vector capabilities in these new processors, which it investigates from the twin perspectives of performance and programmability. The initial experiments are performed in the context of the first Arm-based hardware adopted in HPC, building up to experiments in simulated and emulated environments on the challenges faced by a wide vector instruction set like SVE, and finally analysing the real-world performance of the first implementation of SVE in hardware. The thesis concludes with an outlook towards the next generations of high-performance processors, highlighting the need for co-design in the quest for performance, and suggesting future research avenues for a new generation of performance tools that can enable informed design decisions for upcoming hardware.

Dedication

Thank you to Prof Simon McIntosh-Smith for six years of guidance, inspiration, and support. You have been a great leader and mentor for my research projects, and I could not have asked for any more patience, flexibility, open-mindedness, or pragmatism from my supervisor. From you I have learned a composed, constructive, and realistic approach to unexpected circumstances that reaches beyond the scope of this thesis, and for that I am grateful.

Thank you to the Bristol HPC Group for treating me as one of their own from my very first day. Your comments, advice, and feedback have been most helpful as I've learned how to be a researcher, and our conversations have made pleasant journeys that were otherwise only long and tiring.

To my family and close friends, thank you for being by my side the whole way. Your continued reassurance and confidence in me have been invaluable throughout these four years.

Thank you to the many people at Arm who have worked with me throughout this PhD: Assad, Chris, David, John, Olly, Phil, Roxana, and Will. You have been nothing but welcoming, and you have made my journey more engaging, more valuable, more unique.

Finally, my thanks go to members of the Bristol Computer Science Department that have made me feel part of a big family throughout my time here: Ben, Bogdan, Dan, David B, David M, and Tilo. It has been a pleasant and rewarding journey being part of this department, learning from you, and later working alongside you. It was your energy and enthusiasm that helped me decide to pursue this PhD.

Acknowledgements

This thesis used the Isambard UK National Tier-2 HPC Service operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1). Access to the Cray XC50 supercomputers Swan and Horizon was kindly provided by Cray through their Marketing Partner Network. Work in this thesis was carried out using the HPC Zoo, a research cluster run by the University of Bristol HPC Group. Some experiments were performed on the University of Bristol supercomputer systems BluePebble and Catalyst. Early work on the A64FX platform was possible thanks to a remote Early Access programme from Fujitsu.

This PhD was sponsored through a Doctoral Training Partnership studentship award by the Engineering and Physical Sciences Research Council (EPSRC) and the EPSRC National Productivity Investment Fund (NPIF). Some of this work was made possible though an Industrial CASE (ICASE) award in collaboration with Arm. In addition to project funding, Arm kindly provided early access to design documents, technical specifications, and developer tools that enabled research around their emerging Scalable Vector Extension (SVE) instruction set.

Author's declaration

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

SIGNED: ..... DATE:.....

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Computational science is an integral part of modern scientific research. It gives scientists tools for early and rapid experimentation at a cost far below what is possible without computer systems. Each scientific domain brings its own set of problem types and challenges to overcome, but the underlying systems have similar characteristics. The field that brings together these challenges and works to improve computational systems for the benefit of all areas of science is *High-Performance Computing (HPC)*.

HPC is concerned with architecting and exploiting computing systems to their fullest in scientific applications. The core research in HPC that eventually benefits the wider fields of science revolves around modelling computer architectures, understanding their weak and strong points, designing software to exploits those, and proposing improvements for future hardware generations. Since the times of the early Cray supercomputers in the 1970s [117], HPC hardware has taken many forms, but one core goal has stayed the same: designing architectures that bring maximum computational power to domain scientists, as effortlessly as possible.

The Cray-1 was a computer that utilised purpose-built hardware to achieve high performance. This approach was common in HPC even in the 2000s, but in the 1990s a different paradigm appeared: instead of large mainframes, powerful computational systems could be built from collections of smaller, general-purpose hardware. The new paradigm gained traction quickly, and by 2010 the majority of HPC systems were built from commodity hardware and

ran the same operating systems as servers and workstations everywhere [132, 64].

One part of the motivation behind this change was production costs, but another part was the increasing overlap with consumer hardware: processor features that were previously used in HPC found their use in home computers. Early examples of such features migrating from enterprise to consumer hardware are floating-point capabilities, error-correcting memory, and 64-bit word sizes; a more recent example is vector processing, the paradigm in which single instructions operate on several operands — or sets of operands — at the same time. The Cray-1 and the earlier systems TI-ASC from Texas Instruments and STAR-loo from Control Data Corporation were the first architecture to utilise vector processing [33], but by the year 2000 desktop processors included functional units for single instruction, multiple data (SIMD) [36] operations, which brought significant speed-up to multimedia applications. For video games and advanced video encoding/decoding, dedicated hardware was added to general-purpose processors: video cards, or graphics processing units (GPUs). The late 2000s then saw GPUs reach back into HPC through a model called general-purpose graphics processing units (GPGPUs), based on the observation that GPUs are at their core wide vector processors and, as had been correctly identified many decades before, scientific workloads can benefit from such processors.

GPUs saw a big rise in adoption over the following years [62], and since then the HPC community has focused on quantifying the benefit of wide vector processing and optimising application code to take advantage of it [65]. On the one hand, many Machine Learning (ML) applications were identified as prime use cases for GPUs, and today most research in the field of artificial intelligence (AI) uses GPUs [96, 54]. On the other hand, porting many traditional applications to GPUs has proved difficult, and so a middle-ground could be more favourable: a general-purpose central processing unit (CPU) with wide vector capabilities may bring a large portion of a GPUs benefit with few of the drawbacks. This design is prevalent in 2021, when all major vendors of high-performance processors integrate SIMD units in their CPUs,

and this hardware often has higher vector processing capabilities than GPUs themselves had a decade ago.

In 2016, Arm introduced the Scalable Vector Extension (SVE), a new vector instruction set that combines modern instruction set architecture (ISA) design with the classic idea of vector processing [128]. This announcement aligned with another step in unifying HPC and consumer-grade hardware: Arm-based CPUs making their way into the x86-dominated supercomputer world [112, 77, 115]. SVE is enabling Arm to exploit an industry-wide move towards accelerated vector computation inside CPUs [122, 31], and in 2020 the fastest supercomputer in the world was powered by this new architecture [119]. This incredible achievement was the result of many years of *co-design*, the process of iteratively making design decisions by considering all the hardware, software, and tooling components involved in a system *all together*, rather than each of them individually [51, 9]. It is a modern approach to HPC research, which benefits chip designers, software developers, and end-users alike, but holistically integrating the many aspects involved is an enormous challenge that uncovers the very limits of our tools and methodology. In this thesis, I explore co-design and its role in creating the next generations of HPC systems, from the hardware to the software used to program it.

1.1 Contributions

This thesis makes several contributions:

- In Chapter 3 I discuss the implications of a diversified landscape of architectures in mainstream HPC and I evaluate the Arm-based ThunderX2 processor, the first general-purpose CPU to compete with the x86 architecture since its establishment in HPC more than ten years ago;
- Chapter 4 expands on top of strengths and weaknesses identified in the previous chapter to estimate the impact of next-generation instructions sets on the performance of scientific applications;

- In Chapter 5 I present an in-depth exploration of the implications for the processor's memory subsystem of the wide vector operations available in contemporary vector instructions sets;
- Chapter 6 analyses the real-world performance of the first modern microarchitectural implementation of a scalable vector instruction set, which at the time of writing powers the machine ranked first in the TOP500;
- Chapter 7 surveys the software frameworks used in HPC and their efficacy at generating performant code on a wide range of modern architectures, including Arm- and x86-based CPUs, as well as GPUs from all major vendors;
- Finally, Chapter 8 gives an outlook towards further research tools and processes that are needed to enable accurate and relevant performance experiments in HPC, keeping to the overarching goals of co-design and performance portability.

CHAPTER 2

Background

Supercomputers, as we know them today, are the result of decades of ongoing research, technological advances, and engineering expertise. Between the 1970s and the mid 1990s, HPC systems were large monolithic machines, but nowadays they are collections of commodity processors, carefully coupled together to create performant machines that can split their computational capacity into arbitrary partitions on demand. In these systems, the smallest building block is a *node*, generally in the form of a blade or rack-mounted server. Nodes are joined together using a high-performance *interconnect*, thus creating clusters that can range in size from only a few nodes in a single rack to datacenter-sized systems comprising tens or hundreds of thousands of nodes, for which purpose-built facilities are needed. Such a design is scalable, cost-efficient, and failure-tolerant.

On the inside, each node is powered by hardware similar to what can be found in web-services datacenters: high-performance general-purpose central processing units (CPUs), often with high core counts, large pools of fast memory, and tiered storage that aims to strike a balance between capacity and speed. In the early 2000s, these processors were based on a number of different architectures, often each with its own operating system. More

recently, the x86-based chips from Intel and AMD[1] have dominated the supercomputer market, with some systems based on IBM's POWER architecture, and over the past few years also Arm-based designs from Fujitsu, Marvell, or Ampere. On the vast majority of systems, the operating system used is Linux [132]. This design has stayed relatively unchanged over the past decade; each component has become individually faster, but the way they are integrated together is similar.

Inside a node, mostly for cost and power efficiency reasons, it has been common for nodes to utilise a *dual-socket* configuration, in which two processors share a single motherboard. Single-socket configurations are used when the chosen CPU cannot be used in multi-socket systems, and some systems have four sockets per node, but more commonly vendors pack several motherboards inside a single blade rather than adding more sockets to a single motherboard [22].

In multi-socket systems, each CPU is connected directly to only a part of the system's main memory. To access memory attached to other (*remote*) sockets, the CPU must send a request over the inter-socket connection and wait to receive back the data. This gives rise to a *non-uniform memory access* (NUMA) architecture, in which the latency of accessing main memory depends on where the data is located. Because all NUMA accesses to other sockets share the same interconnect, the more remote requests are made concurrently, the slower they will perform.

For some workloads, specific types of co-processors can be employed alongside the CPU to improve performance. These are known as *accelerators*, of which common examples nowadays are general-purpose graphics processing units (GPGPUs), field-programmable gate arrayss (FPGAs), and high-performance network interface cards (NICs) with support for in-network computation. Accelerators are attached to a node's CPU, the latter being referred to as the *host* in such a configuration.

[1]The term `x86` is sometimes used to refer to the original 32-bit instruction set, which was expanded to 64 bits with `x86_64` or `amd64`. Because x86_64 is a superset of the 32-bit x86, and 32-bit x86 systems are not in use any more in HPC today, it is common to use "x86" as shorthand for x86_64. In this thesis, I do not use any 32-bit hardware, and the terms x86, x86_64, and amd64 are used interchangeably.

At the CPU level, the last decade has brought many changes. In 2010, the top-end parts — sometimes referred to as *stock keeping units (SKUs)* — of high-performance processors had between 10 and 16 cores; in 2020, a single processor can house up to 64 cores. To support this 4× increase in core count, other components on and off the chip have also become bigger and faster: caches are larger and comprise several levels, interconnects — both within nodes (between sockets) and between nodes — have higher bandwidth and lower latencies due to increased optimisation and lower overhead, and main memory has increased in speed and capacity.

This increase in computation capacity has outlined, around the middle of the previous decade, the next milestone in the HPC world: *exascale*, the capacity to compute 10^{18} floating-point operations within a second (1 EFLOP/s) in an individual system [92]. As systems have got closer to this milestone, which has not yet been reached as of mid-2021, the HPC literature speaks of "the road to exascale", referring to challenges encountered as bigger-and-bigger systems are commissioned, and suggesting potential solutions. Reaching exascale-level performance is such an enormous challenge that we need to be able to extract as much performance as possible from all of a system's components, while still maintaining a clean, productive application design. Thus, a balance needs to be reached between writing high-performance code and not over-specialising for current hardware, because the next iterations of the systems may use different architectures, potentially from different vendors, which might support different programming models.

Regardless of the architectures and configurations used in a supercomputer, one feature has been key for achieving high performance on virtually any recent CPU or GPU: vector processing, or *single instruction, multiple data* (SIMD) computation.

2.1 Vectorisation

Almost all high-performance processors today include vector computing capabilities. Inside each core, dedicated functional units run a single instruction on a whole batch of operands simultaneously. This design allows

a single source of control flow to be fed to several batches of data for each SIMD instruction, thus saving overhead by reducing the number of instructions that need to be decoded. On the other hand, the downside is lower flexibility: compared to a full (scalar) instruction set, generally only a subset of the instructions will be available in SIMD format. Figure 2.1 illustrates the operation of a generic SIMD instruction.

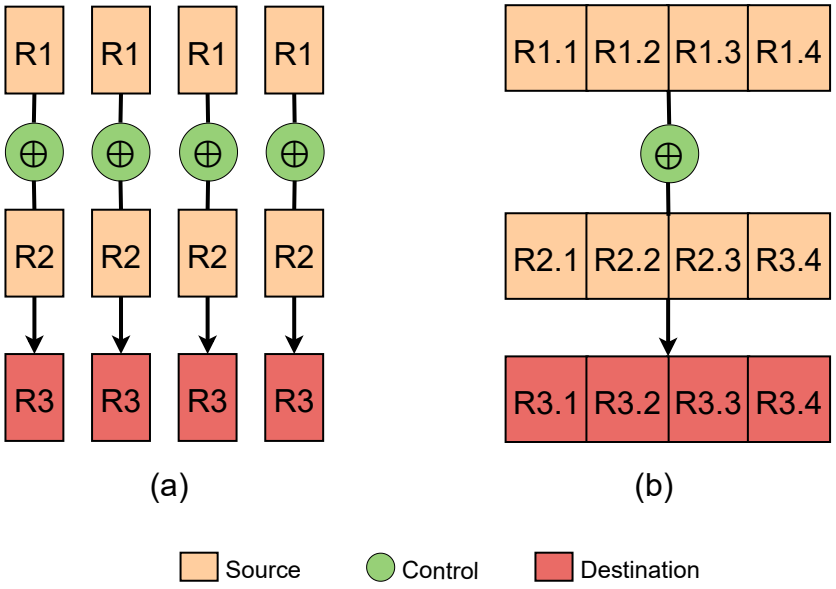


Figure 2.1: Scalar (a) and vector (b) instructions.

Some recent many-core architectures are designed from the ground up around SIMD computation: exploiting these capabilities is critical on GPUs [25] and the Intel Xeon Phi [101, 139], for example, even outside HPC workloads. Compared to general-purpose CPUs, individual cores in these devices are simpler and offer lower performance, but by employing a large number of them in efficient operations, the aggregate computational power of the device becomes substantial. However, with SIMD units having become commonly available in CPUs too, vector code is now important in *all* high-performance processors. Utilising the wide vector units in the latest generations of x86 processors, for example, is the only way to approach peak performance [46, 130].

In consumer-grade processors, SIMD instructions are often used for efficiency [98], for example in applications with structured computation pat-

terns, such as media processing and cryptographic ciphers. Here, SIMD provides a performance boost with a cost to the chip's surface area and power usage that is far smaller than adding whole additional general-purpose cores. These applications have supported the growth of SIMD extensions in common ISAs: x86-based designs started with the Streaming SIMD Extensions (SSE) in 1999 and later moved to the Advanced Vector Extensions (AVX), AVX, and AVX-512, and Arm designs offered the NEON multimedia extension in the ARMv7 architecture, which have become the Advanced SIMD (ASIMD) group of the latest ARMv8 ISA, recently joined by the Scalable Vector Extension (SVE). More recently, general-purpose SIMD capabilities have been used to implement high-precision floating point operations even when the underlying hardware only offers limited precision natively [73].

Scientific applications often exhibit well structured computation patterns [6, 81], and this makes them good candidates for vector processing. In many cases, they consist of a core piece of computation, sometimes referred to as the *kernel*, that is repeatedly applied to a large set of elements in a well-defined sequence. If the kernel can be applied multiple nearby elements simultaneously, then it is a good candidate to benefit from vectorised operations. This is easiest to achieve if the computation for each element does not depend on elements close to it and if elements are iterated through in a contiguous pattern. When successive elements located next to each-other in memory, accesses can be *coalesced*, such that a single memory request can return several contiguous element, but modern vector instruction sets offer alternative solutions when this is not possible. For example, gather operations enable data to be collected into a single vector register from arbitrary locations in memory, scatter operations provide the inverse functionality, and per-lane predication allows instructions to conditionally enable or disable individual operands, which can help work around data dependencies or uncommon access patterns.

2.1.1 Generating and Running Vector Code

There are two common ways of producing vector machine code: manually, through in-line assembly code or compiler intrinsics that map almost-one-to-one onto machine instructions, or automatically, using a vectorising compiler. The former is more tedious and error-prone, because programming languages generally do not provide types that map onto hardware vectors, so it becomes the programmer's responsibility to pack and unpack vector registers. With all but simple applications, this is a very time-consuming task, and it has the major disadvantage that it makes the application code less portable, since intrinsics and assembly instructions are hardware-specific. The advantage of this approach, however, is that one can *ensure* vectors are being used optimally in the most performance-critical parts of the application, and if these parts are relatively small but very commonly used, it may be worth maintaining a version for each platform targetted [93]. This approach is commonly used in optimised maths libraries.

The option of relying on the compiler to auto-vectorise source code requires much less intervention from the programmer, with most modern compilers applying vector optimisation by default in many cases. The most common vector transformation is *loop vectorisation*, where consecutive loop iterations are packed together into vector operations, but some compilers also perform *superword-level parallelism (SLP) vectorisation*, in which groups of similar scalar operations are combined into vector operations. With a vectorising compiler, the same application source code can theoretically take advantage of vector features on any platforms for which a compiler exists. However, compilers are not perfect tools, and so they may not always succeed to automatically vectorise code wherever a skilled programmer could [8].

This latter point is very important for contemporary HPC research. Most applications rely on a compiler at some point in the build process, and the compiler's ability to generate code that is optimised for the target platform is directly linked with the application's run-time performance [150]. Vector optimisations are among the most difficult for compilers to apply, so one factor that often distinguishes compilers in HPC is their ability to under-

stand patterns in high-level source code and transform them into equivalent SIMD machine code. In addition to open-source compilers, of which the GNU Compiler Collection (GCC) and the LLVM Project are the best-known options, processor or system vendors sometime provide their own compiler, for free or under a licence, which in some cases can offer better performance. For example, Intel has held a long track record of providing a robust compiler with good vectorisation ability for its x86 processors, Arm offers the HPC Compiler with a particular focus on its SVE instruction set, and Cray ships the highly regarded Cray Compilation Environment (CCE) with its supercomputer systems. This wide range of choices has led to the standard practice in the field of HPC of *benchmarking* the different compiler options when running performance experiments, because even a simple code change may lead to vastly different performance effects with some compilers but not others.

Unfortunately, most optimising compilers are black boxes from the user's perspective. They are complex system that try to predict the performance of several possible transformations under a model of the target platform, in an attempt to choose the optimal one, but any oversight in either the transformation logic or the model itself can have far-reaching consequences for the generated code. Additionally, compilers will try to obtain the maximum level of performance, but they cannot sacrifice code correctness in this process, so before even attempting many optimisations, they will try to prove that other parts of the code will not be impacted by the changes. Such proofs are hard for human and machine alike, and when the result is uncertain, compilers must choose the cautious approach of disabling any optimisation that have the potential to lead to incorrect code. In practice, this often leads to programmers fighting *against* the optimising compiler, trying hard to demonstrate that optimisations are safe to apply, sometimes through techniques that reduce code clarity or inadvertently reference a specific machine's hardware parameters.

An exception to the two methods above, and in some ways a middle-ground between them, is higher-level programming with explicit vectorisation, as utilised in CUDA and OpenCL. In these languages, kernels are pro-

grammed from the point of view of the smallest *work item*, which is a vector lane in a SIMD processor. Loops over data structures are replaced with kernel invocations over a large thread space, and the programming model restricts the interactions possible within and between kernel instances. This frees the compiler from checking dependencies and promoting scalar code to vector code, with that task instead becoming the responsibility of the programmer. For GPUs, this model has been successful in obtaining high performance, at the cost of porting time when applications need to be converted between CUDA or OpenCL and traditional high-level languages. On CPUs, it is more common to use to libraries or domain-specific programming frameworks that help expose parallelism in the code, such as those built into or on top of modern C++, while leaving explicit vector code generation to the compiler.

2.1.2 An Overview of Modern Vector Instruction Sets

x86. The x86 architecture first gained SIMD support with MMX, an early, 64-bit, integer-only vector instruction set. It was originally introduced for multimedia applications, but because of its overlap with GPU functionality and lack of floating-point support, it was not heavily utilised. In the following years, Intel introduced SSE, and its revisions SSE2, SSE3, and SSE4, which significantly expanded the range of operations supported, including covering floating-point arithmetic, and increased the vector width to 128 bits. More than ten years after SSE appeared, the AVX instruction set superseded it, adding further instructions and introducing 256-bit operations.

AVX came at a time critical for the importance of vector instructions in HPC. General-purpose compute on GPUs had started to grain traction, and compilers and programming models had matured enough to allow for good SIMD support, both through automatic vectorisation and explicit usage. But whereas GPUs ran *all* instructions as vectors, vector extensions on CPUs only covered a small subset of the instructions available. As a result, many computational patterns could only be partially vectorised or required additional code to set up the data in a form suitable for vector instructions.

Today, x86-based processors use AVX2 and AVX-512, expansions on AVX which are much more flexible than their early counterparts. When code is not vectorised, the cause is usually not the lack of suitable vector instructions, but rather the compiler's inability to understand the code's structure and transform it into vector form [109]. AVX-512 itself is divided into several sets of instructions, and each implementation can choose which part to support. Most operations are 512-bit-wide predicated instructions, but one of the sets, *AVX-512VL*, offers the same core instructions in 256-bit form too. It is easy to combine instructions from all versions of SSE and AVX up to the most recent supported one in an implementation, because vector registers are aliased: a CPU implementing AVX-512 will offer 512-bit registers, called `zmm`, and the `ymm` and `xmm` registers from earlier versions of the ISA simply reference the lowest 256 and 128 bits in the same registers, respectively. Figure 2.2 shows how the register aliases overlap.

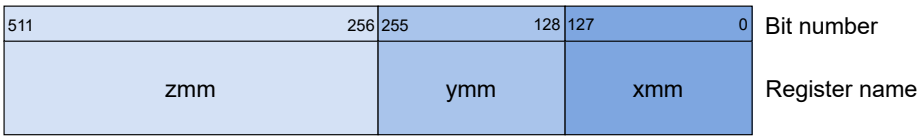


Figure 2.2: x86 vector register aliases: `xmm` and `ymm` refer to the lower 128 and 256 bits, respectively, of the 512-bit `zmm` registers.

The current generation of Intel processors, Cascade Lake, supports AVX-512, including AVX-512VL. The latest x86-based design from AMD, EPYC Rome, supports AVX2.

Arm. Arm processors have traditionally been applied more often in power-limited environments, such as embedded or mobile devices, rather than in high-performance computers. As such, vector extensions were introduced to improve efficiency and focused on media and signal processing, starting with the NEON extension in ARMv7. In ARMv8, NEON was renamed to ASIMD and integrated into the core AArch64 architecture, so it is available on all 64-bit Arm processors. Its origins, however, make it largely unsuitable for modern HPC applications, because it offers very few operations and a vector length of only 128 bits [104].

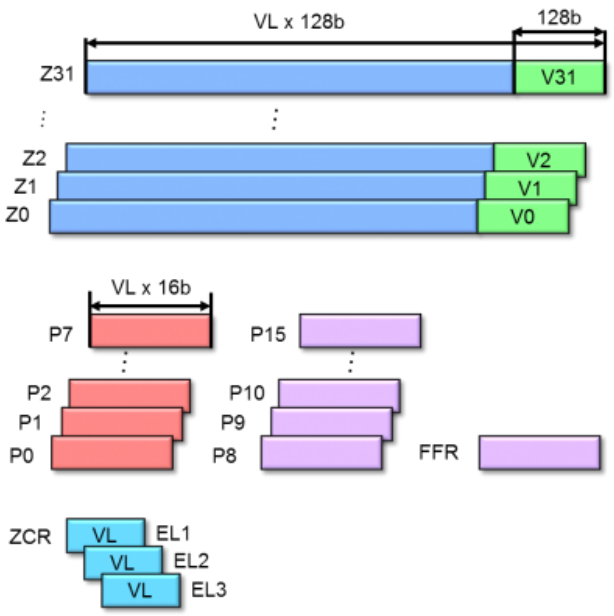


Figure 2.3: SVE vector registers. NEON (`V`) registers are aliased to the lower 128 bits of the full SVE (`Z`) registers. Additional predicate (`P`) and control (`ZCR`) registers also depend on the vector length (`VL`). Source: Arm [127].

The introduction of SVE changes this. SVE is a modern and flexible instruction set, with features relevant for HPC that are on par with AVX-512, such as predication, gather and scatter operations, varied data-type support, and instructions that implement operations across lanes [5]. Unlike the AVX variants, which have a pre-determined vector length, SVE allows each microarchitecture to choose its desired length, in multiples of 128 bits, between 128 and 2048 bits total. SVE code is then *vector-length-agnostic* (VLA) and special instructions are provided to obtain the implementation's chosen (hardware) width at run-time. This concept is illustrated in Figure 2.3, where register sizes depend on the `VL` parameter.

As is the case on x86, compiler support for automatic vectorisation is now the limiting factor in vectorising HPC applications on platforms that implement Arm SVE, rather than instruction availability. The Fujitsu A64FX, released in 2020, is the first processor to implement SVE.

2.2 Modern High-Performance CPU Architectures

Contemporary high-performance processors, while similar from many perspectives, have defining features that make each stand out in certain environments. In this section I introduce the main processor architectures used throughout the experiments in this thesis.

Broadwell. Intel's 5th generation Core architecture was widely used in HPC. Among its top features are 256-bit AVX2 vector instructions, 4 channels of DDR4 memory and up to 24 cores per socket. It features three levels of cache, of which only the last one is shared, with about 2.5 MB of cache for each core in the processor, in an inclusive configuration.

Skylake. Broadwell's successor brought 512-bit AVX-512 vectors to general-purpose CPUs. Memory bandwidth was increased by supporting up to 6 DDR4 channels, and the top SKU offered 28 cores in a single socket. Since it offers more aggregate bandwidth than Broadwell, more cores can run at their full bandwidth; however, since there are also more cores overall, the available bandwidth per core is less than in Broadwell when all cores are used. The last-level cache configuration was changed to an exclusive victim cache, so it is only filled with data that is evicted from the second level, but the amount of last-level cache per core was decreased to 1.375 MB. When running AVX-512 instructions, Skylake *decreases* its clock frequency, which in practice means that applications need a high ratio of vectorisation to benefit from the 512-bit instructions. These different performance characteristics compared to the previous generation, and its increased price, resulted in slow adoption for Skylake, many centres still using Broadwell several years after the launch of the new architecture. Skylake was succeeded by Cascade Lake, an incremental improvement that did not bring significant new features.

Rome. AMD's first-generation EPYC processors, codenamed Naples, were used in datacenters, but less so in HPC; it was the second generation, Rome,

that gained traction. Rome uses the x86 instruction set and supports up to 64 cores per socket connected to 8 channels of DDR4. It brings a novel approach to microarchitectural design in which the chip is built from several *chiplets*, individual compute modules of 8 cores. This design is highly scalable, but because each chiplet is a separate NUMA node, it is also potentially more susceptible to latency issues. Rome supports x86 vector instructions up to 256-bit AVX2.

ThunderX2. Cavium — later acquired by Marvell — released a first generation of Arm-based processors for the datacenter with the ThunderX series. In HPC, these were used for some experiments, but it was only its successor, the ThunderX2, that gained popularity. Built on a design that originated at Broadcomm, the ThunderX2 includes up to 32 ARMv8 cores connected to 8 memory channels. Its memory bandwidth was superior to x86-based alternatives at the time, but its cache was relatively low at 1 MB per core, and it only offered 128-bit vectors.

2.3 Programming Models and Performance Portability

In HPC, programming languages and frameworks are slow-moving. Many applications that are still commonly used today were first released in the 1990s — some even earlier — and so they use some of the same programming languages, concepts, and techniques that were available then. Furthermore, because performance has always been critical in HPC by definition, the common programming languages used are those that incur the least amount of overhead. This has led to the C and Fortran languages being the most prevalent in the industry.

In the wider context, however, programming languages have evolved very rapidly over the past few decades. Advances in hardware and compilers mean that many things are possible now that were not before, but re-writing old applications in a new language is a daunting task. Not only do they involve

codebases that have evolved over many years, but the existing versions have proved over the years that they are reliable and produce correct results; a modern port would need a long time to go through the same verification and validation processes, so that it provides the same level of confidence. Thus, HPC users have not benefited significantly from recent advances in programming [94, 126].

One slight exception to this observation is the C++ language. Some applications adopted C++ early, starting with its initial versions, hoping to utilise the higher level of abstraction to ease the burden on the programmer. But C++, unlike C and Fortran, has evolved very rapidly; C++11 is almost a completely different language from the C++ of the 1990s, and more recent versions such as C++17 and C++20 further bridge the gap in usability between C++ and much higher-level languages which rely on heavyweight runtimes and features like garbage collection in an attempt to provide an efficient programming experience. For applications already using C++, an improvement over time came naturally, at a much lower cost than switching languages altogether.

For the applications that were unable to replace the programming language used, there is still a desire to modernise code. In general, the longer a codebase is used, the larger and harder to maintain it becomes, so programmer productivity is reduced. As compilers have evolved, productivity can further be improved by offloading common tasks from the programmer to the machine. One way to achieve this is through new programming *frameworks* built on the same traditional languages.

HPC applications span many scientific domains, but one characteristic they share is the need to exploit parallelism to achieve high performance. Because this is a very common requirement, it is widespread practice to use libraries to parallelise code instead of implementing everything from scratch every time. Two of the most widely used parallelisation frameworks are the Message Passing Interface (MPI) and OpenMP [67]. Both of these natively support C and Fortran, and while there is some support for C++, often it only covers the language itself and not its standard library of data structures and algorithms. If a framework were to improve upon MPI or OpenMP,

delivering similar performance while improving programmer productivity, its impact would be significant.

To some extent, there have existed frameworks that delivered some of these goals. NVIDIA CUDA, followed later by OpenACC, provides a low-overhead toolchain that is efficient at extracting high performance from NVIDIA GPU hardware, for which it is heavily optimised. However, this comes at the cost of *vendor lock-in*: the future of these tools is entirely in the hands of the company behind them, unlike open standards for programming languages and open-source frameworks. This issue has deterred many developers from investing heavily into a vendor-controlled ecosystem, because their efforts may be wasted if future generations of supercomputers use different hardware, on which these frameworks cannot be used.

In addition to simplifying development of parallel code, there is another area with opportunities for novel programming frameworks to enhance productivity in HPC code: portability. In the second half of the previous decade, the HPC community has placed increasing importance on the *performance portability* of HPC code, with the goal to understand how to write applications in such a way that they achieve a large fraction of peak performance on many different platforms [29]. This endeavour originally appeared as code targeting GPUs — often written using frameworks such as CUDA, which cannot easily be run on CPUs — diverged more-and-more from CPU code, and has grown increasingly important in the context of the upcoming exascale systems, Frontier, El Capitan, Aurora, and Perlmutter, which together combine CPUs from two vendors and GPUs from three vendors, with no established programming framework able to target all these combinations. If code is not performance-portable, it needs to be refactored in large proportions, and sometimes completely rewritten using new libraries, when moving to a system that uses different hardware.

Recently, two frameworks have emerged that aim to improve the portability of HPC code, while maintaining performance and helping programmer productivity: Kokkos [32] and SYCL [45]. These are both frameworks that expand on top of modern C++ and can target both CPUs and GPUs without any change to the source code. They are of particular interest to the HPC

community because they may offer a solution to the fragmentation between hardware targets and programming languages they support. Their adoption is still early, but if it proves successful, another side effect will be increased migration to the C++ language, which some argue is a better choice for the longer term compared to older languages such as C and Fortran.

SYCL is an abstraction layer built on top of C++ that combines the functionality of OpenCL with a single-source approach. An OpenCL program contains two parts: a *host* program, usually written in C or C++, that initialises the OpenCL stack and compiles one or more *kernels*, then schedules them to be run on the target *device*, which may be an accelerator connected to the host, like a GPU or FPGA, or the same CPU as the host itself. In contrast, SYCL code is embedded in the C++ host code, eliminating the need for a kernel compiler that is invoked when the host program runs. This does mean that support for the target device must be present when compiling the SYCL application — as opposed to OpenCL, which only calls the kernel compiler and libraries when the host program is run, targeting the hardware present at that moment — but this is almost always the case in HPC systems. There are currently three major SYCL compilers: Intel DPC++, based on the new-generation Intel ICX compiler (itself based on LLVM), which supports several platforms through the OpenCL, CUDA, and Intel Level Zero backends; ComputeCPP, a compiler developed by Codeplay; and hipSYCL, an open-source implementation that supports CUDA, ROCm, and OpenMP for CPUs and GPUs as backends [3].

Kokkos is packaged as a C++ library. It is distributed as source code, integrated into the application's build process, and built at the same time and using the same compiler. This route has the advantage of not forcing the user to choose a particular compiler — any modern C++ compiler can compile Kokkos. The downside is that Kokkos itself must be updated to support new architectures, so it possible to reach situations in which the underlying compiler is aware of a target architecture, but Kokkos is not, and so the generated parallel code may not be fully optimised.

2.4 Common Classes of HPC Applications

Scientific computing applications are diverse and span a large number of domain sciences, but their computational patterns often fall into one of a few categories. It is common to speak of the "dwarfs" of HPC, each representing a class of common application types [6]. Each class has distinctive computation patterns, and similar algorithmic and optimisation strategies work well across applications within the same class.

From a performance point of view, applications are most often classified by the one factor that prevents the code from running faster. This is sometimes referred to as the *bottleneck* of the application and it represents the one resource which, if increased, would immediately improve the performance of the application. Most HPC applications are bound by one of the following factors:

- Raw arithmetic performance. These applications are called *compute-bound*. To improve their performance, the system needs to be able to perform more calculations — usually floating-point operations (FLOPs) — which implies either faster cores, e. g. through higher clock speeds or wider vectors, or a higher core count.

- Memory bandwidth. This is generally the largest class, because over the years compute performance has improved faster than memory performance in hardware. Depending on the size of the working set and the layout of the system's memory hierarchy, the bottleneck can be either one of the levels of cache (*cache-bandwidth-bound*) or, more commonly, the main memory (*DRAM-bandwidth-bound* or simply *memory-bandwidth-bound*).

- Latency. This bottleneck is most common for applications that perform small operations on many different objects, as opposed to batched computation on large, contiguous data. Examples of this class are graph applications, which are often bound by the latency of local memory as processing moves between vertices [71, 35, 40], and distributed FFT applications, which can be bound by the latency of the network in

all-to-all operations [17]. Latency bottlenecks can also appear in deep NUMA hierarchies, e. g. between NUMA nodes or between different levels of cache. This class is particularly hard to characterise and optimise.

In order to analyse the performance of systems and applications, and quantify the effect of optimisations, it is common to utilise *benchmarking*. The most commonly used HPC benchmarks have been developed to represent individual dwarfs, and because performance characteristics are often shared within classes of problems, they offer good indication of real-world application performance.

2.5 Benchmarking

Benchmarking is the process of a measuring a system's performance using well-defined metrics and test cases. In HPC, benchmarking is commonly used in three different situations:

- In evaluating a system's performance. Systems are often compared against one-another, for example to identify weaknesses or strengths of a different architecture, or to quantify performance improvements versus an older generation of the same system. When comparing between different systems, the benchmarks themselves are usually fixed. A range of benchmarks is chosen from different problem classes to cover as many aspects of the systems as possible — this ensures the benchmarks represent realistic use of the system.
- When selecting optimal configurations, parameters, and software components to solve a problem. For example, different algorithms or implementations could be chosen when using optimised math libraries, and different choices may be fastest on different systems.
- To quantify the performance improvements within an application following an optimisation attempt. In this case, the new version of the application is measured against a snapshot before the changes, recording the performance changes. Because there is a wide variety of HPC

systems available, it is highly valuable to benchmark optimisations on many different platforms, since not all systems will react in a similar way to the same code change.

When applied to several independent parts of a system or application, for the purpose of determining the relative impact of each part on the full run time, benchmarking is referred to as *profiling* [146, 143]. The result of such an experiment is a document detailing the amount of resources — usually time — spent in each component, called a *profile*, and is the *de facto* way of characterising performance empirically [66]. Most HPC experiments, and in particular optimisation attempts, commonly begin with a profile of the initial state, which serves to identify the critical points to be improved in a system [47]. For example, if the goal is to parallelise a serial application, it is most beneficial to first address those sections of the code in which the largest portion of run time is spent — any improvement gained here will be much more valuable than optimisations for rarely visited code.

For benchmarks to be meaningful, a rigorous and objective methodology must be followed thoroughly [53]. Large systems have many moving parts, so it is critical that the testing environment is well documented and kept stable throughout the experiments. To increase confidence in results, tests should be run several times and an average measure produced; this helps isolate noise in the system. If the application is particularly sensitive to the utilisation of the system, for example because of network congestion or I/O wait, then tests should be run on a quiet system to avoid introducing confounders.

The benchmarks used in this thesis have negligible run-to-run variability, in most cases below 2%. They are run within a single compute node, so network performance does not impact the results, and they are configured such that disk I/O is kept to a minimum or completely absent. Unless stated otherwise, each benchmark was run 5 times and the average result presented. Where there was a significant spread ot the results, it is addressed separately. In order to ensure the consistency of the software environment, each benchmark was run in a clean shell session, using environment modules to load exact versions of the libraries used.

It is common to *tune* a benchmark for each system it is run on. Some applications have built-it tuning knobs, which can control, for example, loop unrolling or sizes of the data structures and types used. These tunables can sometimes be chosen by looking at the system's architecture, but often the parameters are harder to understand and map well to the target, so an experimental approach may be more suitable. If the parameter space is large, auto-tuning can be employed to evaluate many combinations procedurally [111, 110].

In HPC, there are also a number of system-wide options that need to be considered. The vast majority of applications are written in compiled languages, so the compiler choice is itself a tunable. As discussed in Section 2.1, a good compiler can make the difference between fully utilising an architecture's features and merely running generic code. Different compilers may make different assumptions or choices by default, so finding the right set of optimisation flags that maximises performance introduces another tuning dimension. For multi-node applications, if the communication framework used is MPI, there are several implementations to choose from, ranging from the open-source Open MPI and MPICH to vendor-specific ones such as Cray MPI, and some may perform better in certain scenarios or when running on certain hardware. If the application uses additional third-party libraries, then the choice of library may impact the performance of the whole application: for optimised maths routines, for example, it is common to utilise either the open-source OpenBLAS or a vendor-provided library like Intel MKL, Cray LibSci, or the Arm Performance Libraries.

When analysing the results of performance experiments, it is often useful to compare the performance obtained against the peak capabilities of the hardware used: if there is a large gap between achieved and peak performance, there may be further opportunities to optimise the code for the target platform. One tool commonly used in HPC research for such comparisons is the *roofline model*, in which the "roofs" represent the hardware's peak memory and arithmetic performance, and the application's achieved performance is shown relative to those [145]. The roofline model can suggest if the application studied is likely to be bound by memory bandwidth or

by arithmetic performance, as well as how far the application is from peak performance.

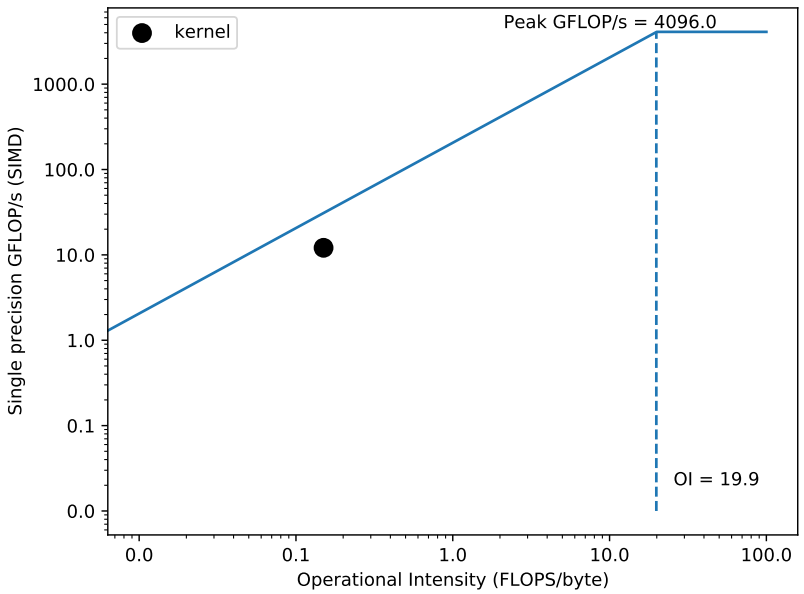


Figure 2.4: Example of a roofline chart, a visualisation produced using the roofline model. The kernel's achieved performance is below the memory bandwidth roof.

2.5.1 Mini-Apps

The large parameter space makes comprehensive benchmarking very resource-intensive. In addition to the time required to run many configurations of the same application, even simply setting up the benchmark configurations can be a lengthy process that needs a lot of developer and processor time, as well as disk space. One of the reasons for this is the compilation process: scientific applications are complex codebases that often rely on third-party libraries, and in order to achieve optimal performance on a system, all of these components need to be compiled from scratch, targeting the system under test. Application code can sometimes be decades-old, sometimes utilising tools that were not designed with the configurations and constraints of modern platforms in mind, and so, in practice, compiling a scientific application on

a new system often first involves finding a compatible compiler version and fixing legacy build system issues.

The other common reason why benchmarking is resource-heavy is input data: many applications rely on large input data sets to process, for example terrain models for weather and climate applications or geometry meshes for fluid dynamics applications. In order to ensure that the benchmarks run are representative of real workloads, it is not always possible to use small inputs, which are sometimes referred to as *toy problems*, but rather full-sized inputs are needed. Again due to the focus on performance, it is not uncommon for input data to need a pre-processing stage before it can be fed to a simulation application, for example by splitting a large geometry in exactly as many components as there are processors in the system. For a regular scientific workflow, in which many runs of the same application will happen in the same configuration and on the same model, this pre-processing step is beneficial, but when benchmarking several configurations across multiple systems, pre-processing often needs to be run as many times — and often takes as long to execute — as the solver itself.

A good way to avoid a large amount of this set-up cost, while still maintaining a relevant testing environment, is avoiding full-scale applications at some stages of benchmarking [142]. Instead, smaller applications known as *mini-apps*, or sometimes *proxy apps*, can be leveraged to provide code that maintains the performance characteristics of a full-size application, but generally does not rely on third-party libraries, large input sets, or complex build processes. Mini-apps are developed from full applications by isolating the performance-critical kernels and adding the minimum amount of wrapper code required to run these kernels on quasi-real data [20, 91]. For example, a mini-app for a fluid dynamics code might include just a subset of commonly used physics solvers, input data that is easy to generate and store, and none of the pre- and post-processing operations that may sometimes be found in full applications for niche situations [12]. In addition, because they are purposely developed for performance analysis, mini-apps often output useful performance metrics out-of-the-box, whereas full applications may need to be profiled to extract the right metrics [15].

Mini-apps have become popular for HPC experiments, and as a side effect of their wide adoption, their code is often more modern and easier to understand than full-scale application code developed over a long time. This makes mini-apps ideal for early system benchmarking: in a new system, where many configurations need to be tried and problems are likely to be encountered along the way, it pays to use as simple a test case as possible. They are also excellent candidates for simulation environments, in which it may not be feasible to run full-scale applications due to the time it would take to simulate such complex code [43, 19, 116, 118]. Because mini-apps are smaller codebases, they are significantly easier to reason about, which in turn makes it easier to predict expected performance and compare it to results obtained from benchmarking. Once most system-related problems have been addressed, and a good selection of tuning parameters has been found for the problem at hand, benchmarking can then move to full applications for more relevant real-world results.

Throughout this thesis, mini-apps are heavily utilised for performance experiments and analysis. I have selected a set of mini-apps that covers a wide range of real scientific workloads, each representing a class of problems. The mini-apps used are:

- **TeaLeaf**, a heat diffusion mini-app that applies a five-point stencil to a regular grid. It uses a conjugate gradient (CG) solver and it is memory-bandwidth-bound [90].

- **CloverLeaf**, which uses a structured grid to solve Euler's equations of fluid dynamics [76]. It is also memory-bandwidth-bound, but contains a higher ratio of compute-intensive operations than TeaLeaf.

- **miniBUDE**, a compute-bound mini-app for virtual drug screening runs using the Bristol University Docking Engine, a well-known molecular docking application. It uses an empirical model to predict the energy of binding drug candidates to target proteins, a process which can occur in a large number of configurations that need to be evaluated individually [106].

- **MiniFMM**, an implementation of the Fast Multipole Method using a tasking dependency model. It is modern code written in C++ and utilising the latest features of OpenMP for task-based parallelism, and while vectorising it is hard, there are good performance benefits to be gained from doing it [7].
- **SNAP**, a mini-app for a deterministic discrete ordinates transport application. The kernel is memory-intensive, and dependencies between elements impose constraints on the sizes of problems that can be run [148]. SNAP has a large memory footprint, so the **MegaSweep** mini-app has been developed to keep the very kernel of SNAP but reduce some of the constraints on run-time configuration [23]. MegaSweep is a good substitute for SNAP in emulated and simulated environments.

Some additional mini-apps are used for a subset of the experiments and are introduced in the the appropriate sections.

CHAPTER 3

Emerging CPU Architectures for HPC

Content from this chapter appears in the following publications:

- Simon McIntosh-Smith, James Price, Tom Deakin and Andrei Poenaru. 'A Performance Analysis of the First Generation of HPC-Optimized Arm Processors'. In: *Concurrency and Computation: Practice and Experience* 31.16 (2019), e5110. DOI: 10.1002/cpe.5110
- Simon McIntosh-Smith, James Price, Andrei Poenaru and Tom Deakin. 'Benchmarking the First Generation of Production-Quality Arm-Based Supercomputers'. In: *Concurrency and Computation: Practice and Experience* (2019), e5569. DOI: 10.1002/cpe.5569

In the early 2010s, x86-based processors from Intel and AMD dominated the data segment. Intel brought the AVX2 vector instruction set alongside higher core counts and improvements in performance-per-watt versus previous generations with Haswell (up to 18 cores/socket), and later Broadwell (up to 24 cores/socket). AMD did not have clear candidates for the server market after the Opteron processors, and Arm-based designs were almost exclusively found in the mobile space and in very-low-power computers such as the Raspberry Pi.

AMD attempted an Arm-based processor for the data center with the A1100 series, but the relatively high TDP of 32 W for only 8 cores with mobile-class performance made it unattractive [59]. Another design was introduced by Cavium in 2014, the ThunderX, which offered up to 48 cores on a single chip, but offered less performance than the high-end x86-based processors of that generation [69]. Neither of these processors was adopted in the HPC world.

In 2018, however, Cavium introduced the ThunderX2 (TX2) processor. Although the name may suggest it was only an iterative improvement on the first-generation ThunderX, it was in fact an entirely different design, based on what was previously known as the Broadcom Vulcan [144]. The TX2 offered up to 32 AArch64 cores per socket running at up tp 2.5 GHz, supporting dual-socket configurations, but the innovation that set it apart from its Xeon competitors was the amount of memory bandwidth available: whereas it was common for server processors, e. g. Broadwell, to have 4 channels of DDR4 memory, the TX2 offered 8 — twice as many. When Cavium was acquired by Marvell, the TX2 became known as the Marvell ThunderX2.

A large number of HPC applications are memory-bandwidth-bound, so the TX2 immediately became an anticipated platform in HPC. In addition, being an Arm-based system, it had the potential to disrupt an x86-dominated market through an increase in competition. Increased competition is desirable not only from a financial standpoint, where it can improve the affordability of HPC systems, but also because it stimulates the development of programming languages, toolchains, and applications in portable, vendor- and platform-agnostic way. In the long term, this prevents research centres becoming locked-in to particular technologies or vendors, improving the flexibility and adaptability of the HPC community to new tools and systems.

This chapter discusses the architecture of the ThunderX2, its relevance for modern HPC, and the lessons learned from its performance characteristics on contemporary HPC workloads that may help shape the future of supercomputing.

3.1 High Performance Arm-based Systems

The UK HPC community adopted the ThunderX2 quickly, albeit to a limited degree. As a novel architecture with the potential to bring sizeable improvements in application performance at a very low cost, research organisations quickly began exploring ways to utilise it to its full potential. The Isambard system[1] was the first supercomputer in the world to offer 64-bit Arm processors in production. It is set up as a *Tier 2* service, i. e. a cluster intended to provide computational facilities to researchers at a regional level, bigger than individual centres but smaller than national-level services. In addition, the Catalyst project[2] recognised the potential of future Arm-based systems in HPC and provided ThunderX2 hardware to researchers in a push to solidify the ecosystem on this platform.

The HPC Group at the University of Bristol contributed heavily to both of these projects. Our work was among the first research published worldwide that looked into how TX2 processors can be exploited for HPC workloads, what types of applications are best suited for their architecture, and how they compare to existing platforms. Outside the UK, one of the most important TX2 system is Astra, at Sandia National Laboratories. The researchers there focused on large-scale applications and used Astra as a test-bed for emerging technologies, such as operating-system-level containers, which can be used to package applications together with their dependencies in portable, low-overhead formats [48, 100].

3.1.1 The ThunderX2 Microarchitecture

The top-end ThunderX2 part has 32 cores running at up to 2.5 GHz, connected to 8 channels of DDR4-2400 memory. Both the Isambard and Catalyst systems use the same variant, installed in a dual-socket configuration. The cores are pipelined, 4-way out-of-order designs based on the 64-bit ARMv8.1-A (AArch64) instruction set. Each core includes two 128-bit

[1] `https://gw4.ac.uk/isambard/`

[2] `https://www.hpe.com/us/en/newsroom/press-release/2018/04/academia-and-industry-collaborate-to-drive-uk-supercomputer-adoption.html`

NEON SIMD units, support for fused multiply–add (FMA) instructions, and can be configured in 1-, 2-, or 4-way simultaneous multithreading (SMT).

Early work on the ThunderX2 focused on quantifying the performance benefit from the different SMT modes. Experience with existing x86-based systems suggested that running more than one SMT thread per core can be beneficial for compute-bound applications. An additional consideration is that if all threads perform network communication, then the total number of messages exchanged grows rapidly when scaling up the node count. Therefore, even if more compute performance may be available, it may not always lead to faster runtime if the network becomes more congested.

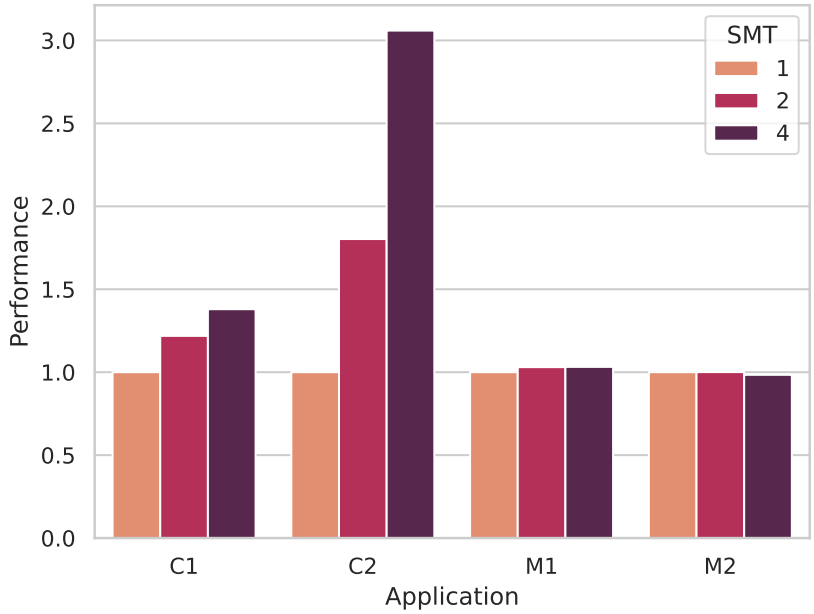


Figure 3.1: Relative performance of four application on the different SMT settings of a TX2 node. Higher numbers represent faster run times.

I observed a similar effect with the TX2's SMT configurations. Going up to 4-way — which presents an impressive 256 logical CPUs at the operating-system level — provided improvements for applications that perform a lot of arithmetic, but could reduce performance when running across more than one node. Figure 3.1 shows the effects of using 1-, 2-, and 4-way SMT on four different HPC applications. The applications labelled M1 and M2 are memory-bandwidth-bound, and C1 and C2 are compute-bound. Perform-

ance is shown relative to using a single thread per core, and it can be seen that using more than one SMT thread per core improves performance for the compute-bound applications, but not for the others. The extent to which performance is improved depends on the applications' computation patterns and how much they are able to hide memory latency by overlapping it with computations. These effects are discussed in-depth in Section 3.4.

The number of hardware threads available can be limited at boot-time to 1 or 2. Regardless of the setting, users can choose to schedule fewer threads on each core, effectively disregarding the SMT capabilities of the processor if they are not useful.

The ThunderX2 includes 32 KB of instruction cache per core, and there are three levels of data cache: 32 KB of private L1, 256 KB of private L2, and 32 MB of shared L3 in a ring arrangement. A block diagram of the TX2's cache configuration is shown in Figure 3.2.

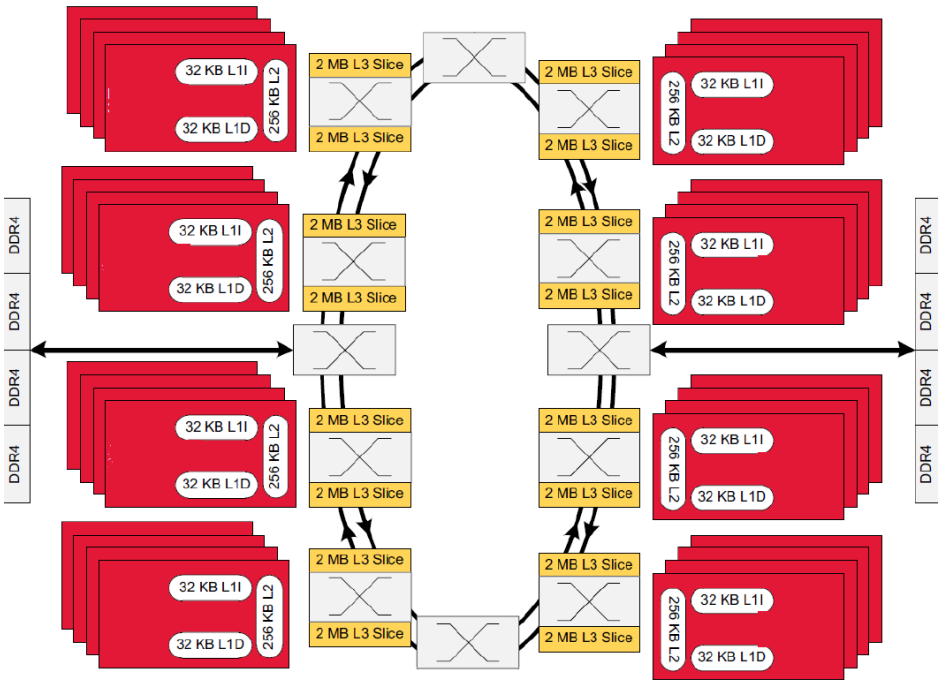


Figure 3.2: The cache configuration of the 32-core ThunderX2 processor. Source: Cavium [16].

Even though the total amount of cache is comparable to a contemporary Intel x86-based processor — the top-end Xeon Platinum 8176 (Skylake) has

38.5 MB of L3 cache — I have found the cache bandwidth available on the TX2 to be lower than on the Skylake. Figure 3.3 shows the aggregate cache bandwidth of these two processors, measured using the University of Bristol's HPC Group's cache-bandwidth tool [78].

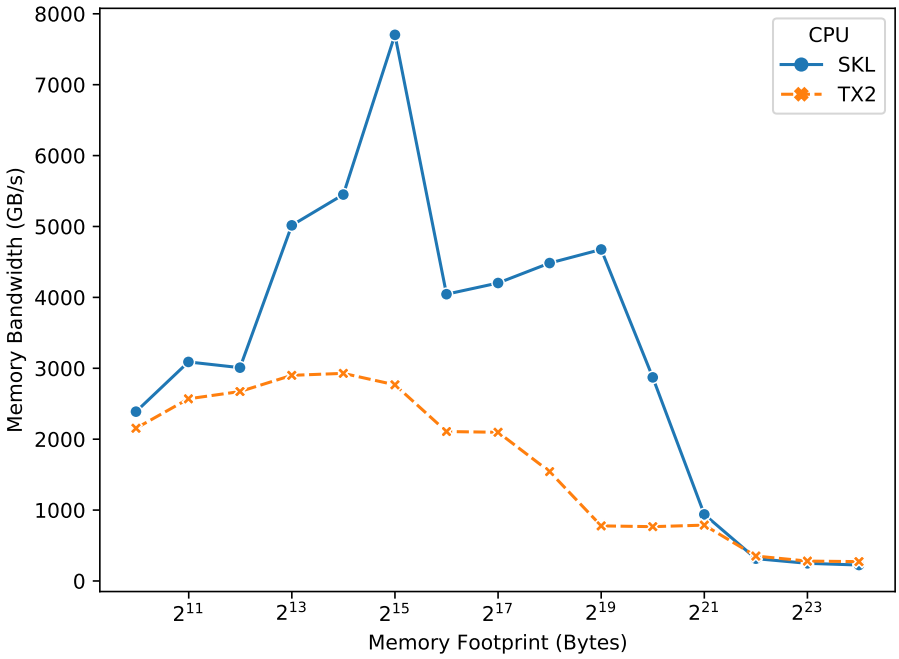


Figure 3.3: Total (aggregate) cache bandwidth achieved on the ThunderX2 (TX2) and Intel Xeon Platinum 8176 (SKL).

3.2 Benchmarks

For the evaluation of the ThunderX2 architecture, I used a diverse set of benchmarks, ranging from simple synthetic kernels to mini-apps and full scientific applications. Because one of this processor's strong points and advantages against its competitors is the high memory bandwidth available, a relevant starting point for performance analysis on TX2 is the **STREAM** benchmark [83]. STREAM is a simple tool that runs four kernels and reports the achieved performance in terms of the amount of memory processed per

unit time. Given arrays A, B, and C and a scalar s, the four STREAM kernels perform the following operations:

- `copy` simply copies data from one array to another: $C = A$;
- `scale` applies a scalar multiplication to data before copying: $C = s \times A$;
- `add` computes vector addition: $C = A + B$;
- `triad` combines the previous two operations in one: $C = s \times A + B$.

STREAM is used to discover an upper bound for the the achievable memory bandwidth on a system — as optimised an application may be, it is unlikely to be able to extract more memory performance than a large sequence of simple and fully independent operations. On modern systems, all four kernels should achieve similar performance, because the difference in computation is negligible for such a bandwidth-bound application. On older hardware, for example, the triad kernel sometimes showed lower performance because fused multiply–add (FMA) instructions were not available. Some architectures may also offer more bandwidth for read operations than for writes, and such a system might see lower performance in the copy kernel — where the ratio of read-to-write operations is 1:1 — versus the other kernels which do several reads for every write.

The STREAM benchmark uses a compile-time parameter to set the sizes of the arrays used in the four kernels. In order to ensure that the bandwidth measured is that of the main memory, and not of cache, the arrays needs to be larger than the size of the cache, at least by a factor of two; there is no upper constraint for the size other than the time taken to run the benchmark. When multiple platforms are compared, the STREAM binary compiled on each platform should use an array size chosen appropriately based on the size of the cache available. For the analysis in this chapter, I used arrays sizes at least 4× larger than the last-level cache on the processor, rounding up to the next power of 2. For example, a dual-socket TX2 system has 64 MB of L3 cache, so the size chosen for the arrays was 2^{25} double-precision elements, which equates to 256 MB of memory per array.

The natural step up from STREAM is benchmarking using **mini-apps**. The strong points of using mini-apps for performance analysis were detailed in Section 2.5.1. This section also introduced a set of mini-apps covering different scientific applications, all of which I used on the ThunderX2.

The final step to comprehensive performance analysis of a new system is benchmarking real-world application performance. For this, I selected a subset of applications from the list of the most heavily utilised scientific applications on ARCHER, the UK's national supercomputer facility [137]. The selected applications cover a representative sample of contemporary HPC use cases, and a description of each benchmark used follows below.

CP2K. CP2K simulates the ab-initio electronic structure and molecular dynamics of different systems such as solids or liquids [138]. Fast Fourier transforms (FFTs) form part of the solution step, but it is not straightforward to attribute these as the performance-limiting factor of this code; the memory bandwidth of the processor and the core count both have an impact. The benchmark used, `H2O-64`, simulates 64 water molecules, consisting of 192 atoms and 512 electrons, in a 12.4 $Å^3$ cell for 10 time steps. This is an often-studied benchmark for CP2K, making it a useful tool for performance exploration.

GROMACS. A molecular dynamics package used to solve Newton's equations of motion, GROMACS is routinely used on systems such as proteins that contain up to millions of particles [1]. It is thought that GROMACS is bound by the floating-point performance of the processor architecture. This has motivated the developers to handwrite vectorised code in order to ensure an optimal sequence of such arithmetic [99]. The hand-optimised code is written using vector intrinsics, which results in GROMACS not supporting some compilers, such as the Cray Compiler, because they do not implement all of the required intrinsics. For each supported platform, computation is packed so that it saturates the native vector length of the platform, e. g. 256 bits for AVX2 and 512 bits for AVX-512.

The molecular system simulated in this benchmark is `ion_channel_vsites`[3]. It consists of the membrane protein GluCl, containing around 150,000 atoms — a small size compared to typical GROMACS runs — simulated in 5 femtosecond time steps. For the ThunderX2 processor, the `ARM_NEON_ASIMD` vector implementation can be used, as it targets the ARMv8.1 architecture. However, this implementation is not as mature as those targeting AVX on x86.

NEMO. The Nucleus for European Modelling of the Ocean code is one ocean modelling framework used by the UK's Met Office, and is often used in conjunction with the Unified Model atmosphere simulation code. The code consists of simulations of the ocean, sea-ice and marine biogeochemistry under an automatic mesh refinement scheme [74]. As a structured-grid code, the performance-limiting factor is typically memory bandwidth at the node level, however communication overheads start to significantly impact performance at scale. The benchmark used was derived from the `GYRE_PISCES` reference configuration, with a $(1/12)^\circ$ resolution and 31 model levels, resulting in 2.72M points, running for 720 time-steps. Version 4.0 of NEMO was used, running with one MPI rank per core for all platforms, without using SMT.

OpenFOAM. In Computational Fluid Dynamics (CFD), OpenFOAM is a well-known modular toolkit written in C++. It is organised as a collection of libraries that can be called individually through thin wrappers or combined in more complex simulation pipelines [58]. Benchmarks with this application used the `simpleFoam` solver for incompressible flow from OpenFOAM v1712+ to model the aerodynamics of the open-source RANS DrivAer model of a generic passenger car. This solver is the *de facto* benchmarking configuration of OpenFOAM, and the model is developed specifically to provide a representative case of real-world aerodynamics simulation [50]. OpenFOAM is largely memory-bandwidth-bound.

OpenSBLI. OpenSBLI is a grid-based finite-difference solver that uses compressible Navier-Stokes equations for shock-boundary layer interactions.

[3] `https://bitbucket.org/pszilard/isambard-bench-pack.git`

The application uses Python to automatically generate code to solve the equations expressed in mathematical Einstein notation, and uses the Oxford Parallel library for Structured mesh solvers (OPS) software for parallelism [56]. As a structured-grid code, it should be memory-bandwidth-bound under the Roofline model, with low computational intensity from the finite difference approximation. The benchmark used is the one developed by the ARCHER community[4], which solves a Taylor-Green vortex on a grid of $1024 \times 1024 \times 1024$ (one billion) cells. One MPI rank was used per core, without using SMT.

UM. The Unified Model is the UK's Met Office code for atmosphere simulation, used for weather and climate applications. Often coupled with the NEMO code, the UM is used for weather prediction, seasonal forecasting, and for climate modelling, with time scales ranging from days to hundreds of years. At its core, the code solves the compressible non-hydrostatic motion equations on the domain of the Earth discretised into a latitude-longitude grid [141]. As a structured-grid code, the performance limiting factor is highly likely to be memory bandwidth. Version 10.8 of the UM was used, with an `AMIP` benchmark [44] provided by the UK Met Office.

VASP. The Vienna Ab initio Simulation Package is used to model materials at the atomic scale, in particular performing electronic structure calculations and quantum-mechanical molecular dynamics. It solves the N-body Schrödinger equation using a variety of solution techniques. VASP includes a significant number of settings which affect performance, from domain decomposition options to maths library parameters. Previous investigations have found that VASP is bound by floating-point compute performance at scales of up to a few hundred cores [14].

The benchmark utilised is known as `PdO`, because it simulates a slab of palladium oxide. It consists of 1392 atoms, and is based on a benchmark that was originally designed by one of VASP's developers, who found that, on a single node, the benchmark is mostly compute-bound; however, there

[4] `http://www.archer.ac.uk/community/benchmarks/archer/`

exist a few methods that benefit from increased memory bandwidth [151]. VASP was run with one MPI rank per core, without using SMT, and an empirically tuned value for `NCORE`, a parameter which describes the parallel decomposition.

3.3 Experimental Set-Up

In order to understand the strong and weak points of the TX2 platform, I compared it to other established HPC systems. The Broadwell and Skylake generations of processors from Intel were the most widely used CPUs in the HPC space during the 2016–2020 period, so they are the best options for comparison. I selected the top-end parts of each of these processors as comparison points, because they represent the most challenging competition for the TX2. However, in practice the top-end (28-core) Skylake Platinum SKU was significantly more expensive than slightly lower-end variants, and so some centres preferred to buy more nodes with a lower core count each, thus achieving a better price-to-performance ratio. In order to keep the comparison representative of the hardware used in real centres, I have also included a lower-end (20-core) Skylake Gold part in the benchmarks. The hardware details for all these processors and their peak performance are given in Table 3.1. In this table, the base value is shown for the clock speed, but dynamic frequency scaling was enabled on all the processors, which may temporarily increase the speed when thermal constraints allow it.

As explained in Section 2.5, the selection of toolchain can result in a significant performance difference. My main goal in this work was to compare the *best* achieved performance achievable on each platform, but it was also useful to know which tools generate the fastest code; in particular, a common question raised in HPC is whether proprietary toolchains are essential for good performance. To address these questions, I experimented with all the available compilers on each platform.

On the Intel platforms, the compilers used were Cray, Intel, and GNU. Cray and GNU are also available for Arm, but Intel is not. Instead, the Arm HPC Compiler — later renamed to the Arm Compiler for Linux (ACfL) —

Table 3.1: Processor model details and their peak performance.

Processor	Cores	SMT	Clock Speed (GHz)	TDP (W)	Arithmetic (FP64 TFLOP/s)	Mem. Bandwidth (GB/s)
Intel Xeon E5-2699 v4 (Broadwell)	2×22	2	2.2	145	1.55	154
Intel Xeon Gold 6148 (Skylake)	2×20	2	2.4	150	3.07	256
Intel Xeon Platinum 8176 (Skylake)	2×28	2	2.1	165	3.76	256
Marvell ThunderX2	2×32	4	2.2	175	1.13	320

Table 3.2: Compilers available for each platform.

Compiler	Version	Target Platforms
Arm HPC Compiler (ACfL)	18.2	aarch64 only
Cray Compilation Environment (CCE)	8.7	x86 and aarch64
Intel (ICC)	18	x86 only
GNU (GCC)	7.2	x86 and aarch64

Table 3.3: Third-party libraries, the benchmarks that use them, and the available variants.

Library	Used in Benchmarks	Options
BLAS	CP2K, VASP	ArmPL, Intel MKL, OpenBLAS
FFT	NAMD, VASP	ArmPL, Intel MKL, FFTW

can be used. Table 3.2 shows the versions of the compilers used in this study, alongside the platforms they can target.

Where libraries were utilised and multiple alternatives were available, these were also compared. Table 3.3 shows a summary of the libraries used and which applications need them.

The systems used for this work were all built by Cray. One of the core innovations of Cray systems is their high-performance network, Aries [4]. Along with the network hardware, Cray provide optimised MPI libraries that can take full advantage of the network to perform operations effectively. For scaling experiments, the Cray MPI library, which is based on MPICH2 [42], was used for inter-process communication.

Within each node, all the available cores were always utilised. Where SMT was available, I tried all possible numbers of threads per core and settled on the best result. For applications that use MPI together with OpenMP — in particular the mini-apps — I tried the following common configurations:

- Flat OpenMP: a single MPI rank per node, which spawns enough OpenMP threads to fill all the cores;
- Flat MPI: one MPI rank for each core, with a single thread inside each rank;
- Hybrid per-socket: one MPI rank per socket and enough OpenMP threads inside each rank to fill the socket, all bound so that threads cannot migrate between sockets;
- Hybrid per-core: one MPI rank per physical core and enough OpenMP threads inside each rank to fill all the hardware threads of the core, all bound so that threads cannot migrate between cores.

MPI ranks were mapped and pinned to CPU cores using Cray's `aprun` launcher. This ensured that each rank is assigned its own, dedicated CPU cores and that ranks are not migrated between cores. To pin threads to cores within each MPI rank, I set the `OMP_PROC_BIND` environment variable to `true`.

3.4 Results

In order to fairly asses the performance of the TX2 versus the other platforms, I first looked at the best performance obtained on each benchmark. Together, these form a good expectation of the performance, on average, of each platform on real-world workloads.

The results presented in Section 3.4.1 are the best case for each platform: where several compilers or parameters could be used, I tried all combinations and selected the best result. The following sections go into more details about the impact of the compilers and libraries used.

3.4.1 Best Application Performance

Figures 3.4 and 3.5 show the performance of the four platforms studied when running mini-apps and full applications, respectively. The results are normalised to Broadwell, which was the *de facto* architecture utilised in HPC when the ThunderX2 became available [133].

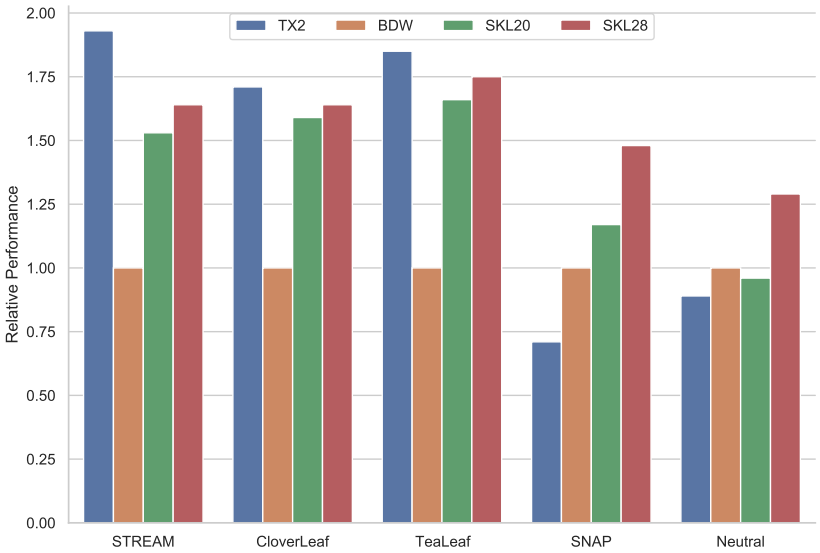


Figure 3.4: Relative performance of mini-apps compared to Intel Broadwell. Higher numbers represent better performance.

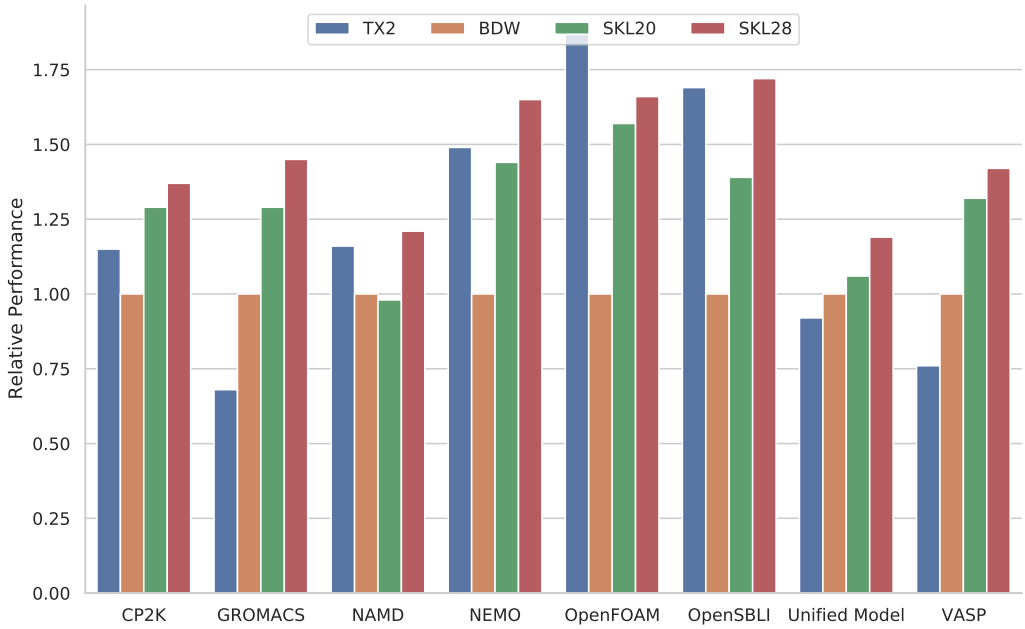


Figure 3.5: Relative performance of applications compared to Intel Broadwell. Higher numbers represent better performance.

STREAM. For the processors tested, the available memory bandwidth is determined by the number of memory channels. Skylake achieved a 1.64× improvement over Broadwell, which was expected, given Skylake uses 6 channels of 2666 MHz DDR4 vs. Broadwell's 4 channels at 2400 MHz. The 8 memory channels per socket on the Marvell ThunderX2 resulted in a 1.93× speed-up over Broadwell.

Broadwell and Skylake achieved 84.4% and 83.9% of theoretical peak memory bandwidth, respectively. At 253 GB/s, ThunderX2 achieved 79.2% of theoretical peak memory bandwidth. On ThunderX2, the best STREAM Triad performance was achieved with running 16 OpenMP threads per socket (32 in total), rather than one thread per core (64 total). With 32 total threads, the CrayPAT tool reported cache hit rates of 69.4% for L1 and 66.9% for L2, whereas for 64 threads, the cache hit rates were 66.6% for L1 and 38.5% for L2; notice that the L2 hit rate is lower when using more threads. On the 18-core Broadwell CPUs in the Isambard Phase 1 system, the cache hit rates were 66.7% for L1 and 11.3% for L2, both with 36 threads,

and the numbers were similar with only half the number of threads. This suggests that on high-core-count processors like the ThunderX2, if memory bandwidth is the sole concern, sometimes better results may be achieved by not filling all the available cores.

The use of non-temporal store instructions is an important optimisation for the STREAM benchmark, as Raman et al showed: for the Triad kernel, for example, it provided a 37% performance improvement on Intel Broadwell and Xeon Phi (Knights Landing) processors [113]. On the Intel architectures, using these instructions for the write operations in the STREAM kernels ensures that the cache is not polluted with output values, which are never re-used. As such, if this data occupied space in the cache, it would reduce the capacity available for the other arrays which are being prefetched into cache. The construction of the STREAM benchmark — arrays allocated on the stack, with the problem size known at compile-time — allowed the Intel compiler to generate non-temporal store instructions for all the Intel architectures in this study.

Although the GCC compiler does not generate non-temporal stores for the ThunderX2 architecture — in fact, it cannot generate non-temporal store instructions for *any* architecture — the implementation of these instructions within the ThunderX2 architecture does not result in a bypass of cache anyway. Instead, these stores still write to L1 cache, but in a way that exploits the write-back policy and *least recently used* eviction policy to limit the disruption on cache. This lack of true streaming stores may be a limiting factor in achieving higher architectural efficiency with STREAM on ThunderX2, as memory bandwidth is being wasted evicting the output arrays of the kernels. However, the additional memory controllers on ThunderX2 processors still provided a clear memory bandwidth advantage over Broadwell and Skylake processors.

CloverLeaf. The normalised results for the CloverLeaf mini-app are consistent with those for STREAM on all the processors studies. CloverLeaf is a structured-grid code and the majority of its kernels are bound by the available memory bandwidth, and it has been shown previously that using

GPUs with high memory bandwidth results in proportional improvements for CloverLeaf performance [85]. The same was true on the processors in this study: the improvements on ThunderX2 came from its greater memory bandwidth, reaching the highest performance of the processors tested.

The time taken to execute each iteration increased as the simulation progressed on the ThunderX2 processor. This was due to the data-dependent need for floating-point intrinsic functions, such as `abs`, `min` and `sqrt`, which can be seen in the `viscosity` kernel, for instance. As the iterations progress, the need for such functions became higher and, therefore, the kernel increased in runtime. Although these kernels are memory-bandwidth-bound, the increased number of floating-point operations increase the computational intensity, and CloverLeaf as a whole is therefore slightly less bandwidth-bound under the Roofline model. On the x86-based processors, which have less memory bandwidth available but more arithmetic resources, this change in the iteration time was not observed, suggesting that the extra arithmetic was entirely hidden by the slower memory operations.

On all the platforms studied, I found that the best run configuration for CloverLeaf was per-core hybrid MPI, with OpenMP threads used to fill all the available hardware threads. On the ThunderX2, this meant running 64 MPI ranks, each bound to a physical core and splitting its computation over 4 OpenMP threads. This was the best way to utilise the processor's SMT capabilities; without, it was better to run a single MPI rank per physical core without any OpenMP threading, for a total of 64 ranks of one thread each, rather than one MPI rank for each hardware thread available.

TeaLeaf. The TeaLeaf mini-app again tracked the memory bandwidth performance of the processors, as previously shown on x86 and GPU architectures [90]. The additional memory bandwidth of the ThunderX2 processor clearly improved the performance over processors with fewer memory channels. As was the case with CloverLeaf, the most efficient run configuration was one MPI rank per physical core, each using 4 threads, for a total of 256 threads on a dual-socket ThunderX2 node.

SNAP. Understanding the performance characteristics of the SNAP proxy application is difficult [23]. If truly memory-bandwidth-bound, the extra bandwidth available on the ThunderX2 processor would increase the performance as it did for the other memory-bandwidth-bound codes discussed. However, the ThunderX2 processor was almost 30% *slower* than Broadwell for the input problem in this benchmark. The Skylake processor, on the other hand, did show an improvement, and memory bandwidth was not the main factor here: Skylake has 512-bit vectors, which create a wide data path through the cache hierarchy. In comparison, Broadwell has 256-bit vectors and the ThunderX2 has 128-bit vectors, moving half and a quarter of a 64 byte cache line per load operation, respectively.

The main sweep kernel in the SNAP code requires that a cache hierarchy support both accessing a very large data set and simultaneously keeping a small working set in low levels of cache. To evaluate the performance of the caches, the CrayPAT profiler was used to obtain the cache hit rates for L1 and L2 caches. On the ThunderX2, the hit rates for the caches were both at around 84%, which is much higher than the hit rate for the STREAM benchmark — the latter is truly main-memory-bandwidth-bound. This shows that the SNAP proxy application heavily reuses data in cache, and so performance of the memory traffic to and from cache is a key performance factor. On the Broadwell processors in Isambard Phase 1, CrayPAT reported cache hit rates of 89.4% for L1 and 24.8% for L2. Again, the L1 cache hit rate was much higher than in the STREAM benchmark, indicating high reuse of data in the L1 cache, but that the L2 was not used as efficiently. It is thus clear that main memory bandwidth alone is not the performance limiting factor, but rather it is aggregate bandwidth to the caches where the x86 processors have an advantage [28].

Neutral. In previous work, it has been shown that the Neutral mini-app has algorithmically little data reuse, due to the random access to memory required for accessing the mesh data structures [79]. Additionally, the access pattern is data-driven, and thus not predictable, so any hardware prefetching of data into cache according to common access patterns is likely to be

ineffective, resulting in a relatively high cache miss rate. Indeed, CrayPAT showed a low percentage (27.3%) of L2 cache hits on the ThunderX2 processor and a similar percentage on Broadwell. The L1 cache hit rate was high on both architectures, with over 95% hits. As a result, the extra memory bandwidth available on the ThunderX2 processor did not provide an advantage over the Intel Xeon processors. Note that the ThunderX2 processor still achieved performance close to that of Broadwell, with 28-core Skylake only offering a performance improvement of about 29%.

CP2K. CP2K comprises many different kernels that have varying performance characteristics, including some floating-point-intensive routines and some that are affected by memory bandwidth. While the compute-intensive routines ran up to 3× faster on Broadwell compared to ThunderX2, the improved memory bandwidth and higher core counts provided by ThunderX2 allowed it to reach a 1.15× final speed-up over Broadwell for this benchmark. The 28-core Skylake processor provided even higher floating-point throughput and closed the gap in terms of memory bandwidth, yielding a further 19% improvement over ThunderX2. The `H2O-64` benchmark has been shown to scale sublinearly when running on tens of cores [11], which impacts on the improvements that ThunderX2 can offer on its 64 cores. When running the benchmark on a single socket, Skylake was just 5% faster than ThunderX2.

GROMACS. The GROMACS performance results were influenced by two main factors. First, the application is heavily compute-bound, and the x86 platforms were able to exploit their wider vector units and wider datapaths to cache. Performance did not scale perfectly with vector width due to the influence of other parts of the simulation, in particular the distributed FFTs. Secondly, because GROMACS uses hand-optimised vector code for each platform, x86 benefits from having the more mature implementation, one that has evolved over many years. Since Arm platforms were new in HPC at the time, the NEON implementation did not achieve peak efficiency on ThunderX2.

NAMD. As discussed in Section 3.2, NAMD is not clearly bound by a single factor, and thus it is hard to underline a specific reason why one platform was slower — or faster — than another. It is likely that results were influenced by a combination of memory bandwidth, compute performance, and other latency-bound operations. The results observed do correlate with memory bandwidth, making Broadwell the slowest platform of the three for this application. Running more than one thread per core in SMT increased NAMD performance, and this is the most pronounced on the ThunderX2, which can run 4 hardware threads on each physical core. Furthermore, due to NAMD's internal load balancing mechanism, the application was able to efficiently exploit a large number of threads, which conferred yet another advantage to ThunderX2 for being able to run more threads (256) than the x86 platforms (112 on the 28-core Skylake). As a result, while the 32-core ThunderX2 did not match the top-bin 28-core Skylake, it did outperform the mainstream 20-core Skylake by about 18%.

NEMO. For the NEMO benchmark, ThunderX2 was 1.49× faster than the 22-core Broadwell, and slightly faster than the 20-core Skylake, while not matching the 28-core Skylake. While the benchmark should be mostly memory-bandwidth-bound, leading to significant improvements over Broadwell, the greater on-chip cache bandwidth of Skylake gave the top-bin part a performance advantage over ThunderX2. Running with multiple threads per core to ensure that the memory controllers are saturated provided a small improvement for ThunderX2.

OpenFOAM. The OpenFOAM results followed the STREAM behaviour of the three platforms closely, confirming that memory bandwidth is the main factor that influences performance here. With its eight memory channels, ThunderX2 yielded the fastest result, at 1.87× the Broadwell performance. Skylake ran 1.57×–1.66× faster than Broadwell, i. e. a bigger difference than in plain STREAM, because it is likely able to get additional benefit from its improved caching, which is not a factor in STREAM. This benchmark

strongly highlights ThunderX2's strength in how performance for HPC workloads can be improved significantly through higher memory bandwidth.

OpenSBLI. The OpenSBLI benchmark exhibited a similar performance profile to OpenFOAM, providing another workload that directly benefits from increases in memory bandwidth. The ThunderX2 system produced speed-ups of 1.69× and 1.21× over 22-core Broadwell and 20-core Skylake, respectively, and almost matched the 28-core Skylake.

Unified Model. Comprising two million lines of Fortran, the Unified Model is arguably the most challenging of the benchmarks used in this study, stressing the maturity of the compilers as well as the processors themselves. The 28-core Skylake only reached 1.19× the performance of the 22-core Broadwell, indicating that the performance of this test case is not entirely correlated to memory and cache bandwidth or floating-point computational performance, and that the relatively low-resolution benchmark may struggle to scale efficiently to higher core counts. The ThunderX2 result was around 8% slower than the top-bin Broadwell, but demonstrated the robustness of the Cray software stack on Arm systems by successfully building and running a complex, long-lived piece of software without requiring any modifications.

Interestingly, when running on just a single socket, ThunderX2 provided a ∼15% improvement over Broadwell, suggesting a potential scaling issue of this application to high-core-count nodes — a dual-socket node of TX2 has 64 cores total, but a dual-socket node of Broadwell only has 44 — or perhaps that inter-socket communication may be less optimal on the TX2 system than it is on the Intel systems. I also observed performance regressions in the more recent versions of CCE on all three platforms: the Broadwell result was fastest using CCE 8.5, which could not be used for either Skylake or ThunderX2, because CCE only gained support for these with version 8.6.

VASP. The calculations performed by the VASP benchmark are dominated by floating-point-intensive routines, which naturally favour the x86 processors with their wider vector units. While the higher core counts provided

by ThunderX2 made up for some of the difference, the VASP benchmark exhibited a similar profile to GROMACS, with ThunderX2 reaching around ¾ the performance of the 22-core Broadwell and around half that of the 28-core Skylake.

Application Performance Summary. Many of these results highlight the superior memory bandwidth offered by the ThunderX2's eight memory channels, which deliver 253 GB/s for the STREAM Triad benchmark — twice that of Broadwell and 18% more than Skylake. This performance increase can be seen in the memory-bandwidth-bound mini-apps such as CloverLeaf and TeaLeaf, with the ThunderX2 processor showing similar improvements to STREAM over the x86 processors. The SNAP and Neutral mini-apps, however, rely more on the on-chip memory architecture — the caches — and so they are unable to leverage the memory bandwidth on all processors. As such, the additional memory controllers on the ThunderX2 processors do not seem to improve the performance of these mini-apps relative to processors with less memory bandwidth.

The remainder of this chapter dives deeper into insight on running HPC workloads on the ThunderX2. In the following chapters of this thesis, I explore the challenges of SVE-enabled processors: Chapters 4 and 5 address issues that implementations of SVE have to consider in making optimal design choices; Chapter 6 evaluates the first real-world implementation of SVE; and Chapter 7 surveys modern programming models and their applicability to a wide range of architectures, including Arm SVE. Finally, in Chapter 8 I present insight I have gained that could suggest exploration avenues and methodology for further studies involving modern vector architectures.

3.4.2 Compiler Performance Comparison

Early toolchain comparison. The results discussed in Section 3.4.1 are the best achieved on each platform. They were obtained by compiling each application with all the supported compilers, using a set of compiler flags to enable the highest level of optimisation for the target platform, e. g.

Table 3.4: Best compiler for each application on the platforms studied.

Benchmark	ThunderX2	Broadwell	Skylake
STREAM	Arm 18.3	Intel 18	CCE 8.7
CloverLeaf	CCE 8.7	Intel 18	Intel 18
TeaLeaf	CCE 8.7	GCC 7	Intel 18
SNAP	CCE 8.6	Intel 18	Intel 18
Neutral	GCC 8	Intel 18	GCC 7
CP2K	GCC 8	GCC 7	GCC 7
GROMACS	GCC 8	GCC 7	GCC 7
NAMD	Arm 18.2	GCC 7	GCC 7
NEMO	CCE 8.7	CCE 8.7	CCE 8.7
OpenFOAM	GCC 7	GCC 7	GCC 7
OpenSBLI	CCE 8.7	Intel 18	CCE 8.7
UM	CCE 8.6	CCE 8.5	CCE 8.7
VASP	GCC 7.2	Intel 18	Intel 18

`-mcpu=native -O3 -ffast-math -ffp-contract-fast`, then selecting the best-performing one. Table 3.4 shows which compiler produced the fastest binary in each case.

For mini-apps, the Intel compiler often produced the fastest results on the x86 platform: it was able to match patterns in the kernels and generate highly optimised code sequences for them, and because the mini-apps contain relatively little kernel code, the performance gained this way was significant. On the other hand, when moving to full-scale applications, the Cray compiler was the fastest, by virtue of higher vectorisation performance, on x86 and Arm alike. The best-performing compiler was weakly correlated with the programming language of the application: the Cray compiler performed the best on Fortran programs, such as NEMO and UM, while GCC was fastest with C++, for example on OpenFOAM and GROMACS.

Figures 3.6 and 3.7 compare the three major compilers on the ThunderX2 platform, normalised to the best performance observed for each benchmark. The benefit of having multiple compilers for Arm processors is clear, as none of the compilers dominated performance, and no single compiler was able to build all of the benchmarks. Performance for the mini-apps was broadly

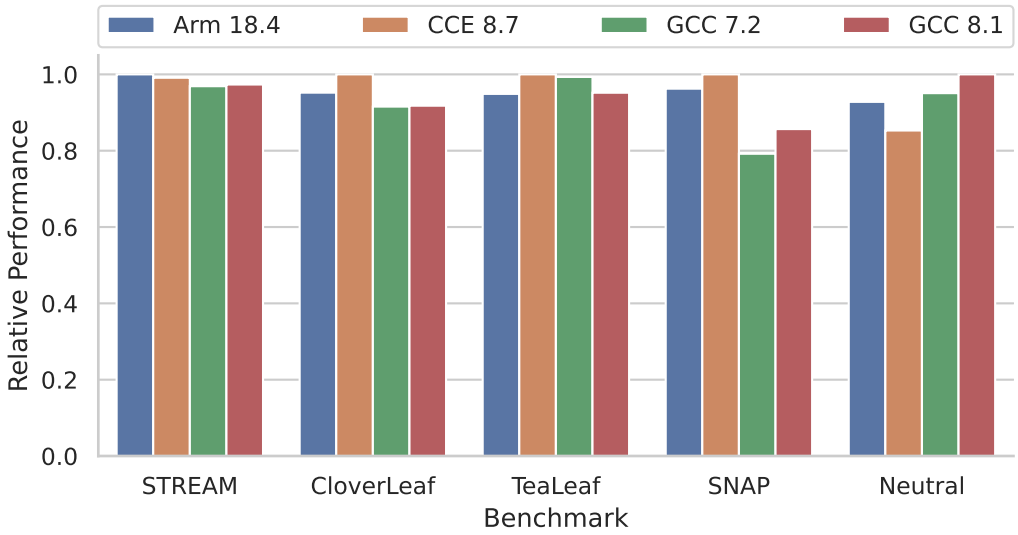


Figure 3.6: Relative performance of mini-apps running on ThunderX2 when compiled with different toolchains.

similar across all of the compilers, with 15–20% variations for SNAP and Neutral, where the more complex kernels drew out differences in the optimisations applied by the compilers. The Arm HPC compiler uses the LLVM-based Flang, a relatively new Fortran frontend, which at the time of writing produced an internal compiler error while building CP2K. Both CP2K and GROMACS crashed at runtime when built with CCE; this issue also occurred on the Broadwell system and so was not specific to Arm processors. While the NAMD benchmark built and ran correctly with GCC 7, it froze after initialisation with GCC 8. It is unclear whether these issues are a result of bugs in the applications themselves or errors in the compilers.

NAMD failed to build with CCE because Charm++ uses inline assembly syntax which is not supported by the Cray compiler. OpenFOAM exhibited multiple syntax errors in its source code, which are only flagged as issues by GCC 8 and CCE; this syntax was only valid in legacy versions of C++ and more modern compilers do not accept it any more. The largest performance difference I observed between the compilers was with the OpenSBLI benchmark, where the code generated by CCE was 2.5× faster than any of the other compilers. On x86, performance was much closer between all of

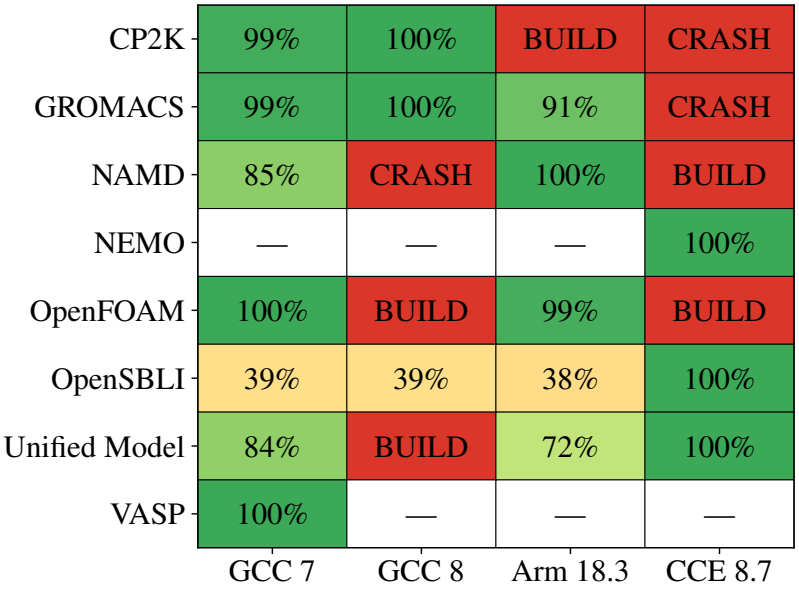


Figure 3.7: Relative performance of full applications running on ThunderX2 when compiled with different toolchains. For each application, the fastest result is labelled "100%". Build- and run-time errors are marked in red, and dashes indicate build configurations not supported at the time of writing.

the compilers, and the main factor causing a performance discrepancy is the MPI library used. On systems where Cray MPI can be used with non-CCE compilers, the final performance difference should be minimal.

The results above were produced using the latest versions of the toolchains available when the Isambard system was installed in 2017–18. Those were, in most cases, the very first release versions of these tools targeting an HPC environment and the ThunderX2 processor. However, toolchains have continuously improved since then, and the four years of running the Isambard service have directly contributed to their maturity.

Updated toolchain comparison. In 2021, I repeated the same compiler experiments with the latest versions of the tools to quantify the improvement over time. The initial and updated versions of the toolchains are given in Table 3.5.

Table 3.5: Initial TX2 compiler versions from 2018 compared to the latest available releases in 2021.

Compiler	Early version (2018)	Modern version (2021)
Arm HPC Compiler (ACfL)	18.2	21.0
Cray Compiler (CCE)	8.7	11.0
GNU (GCC)	7.2	11.1

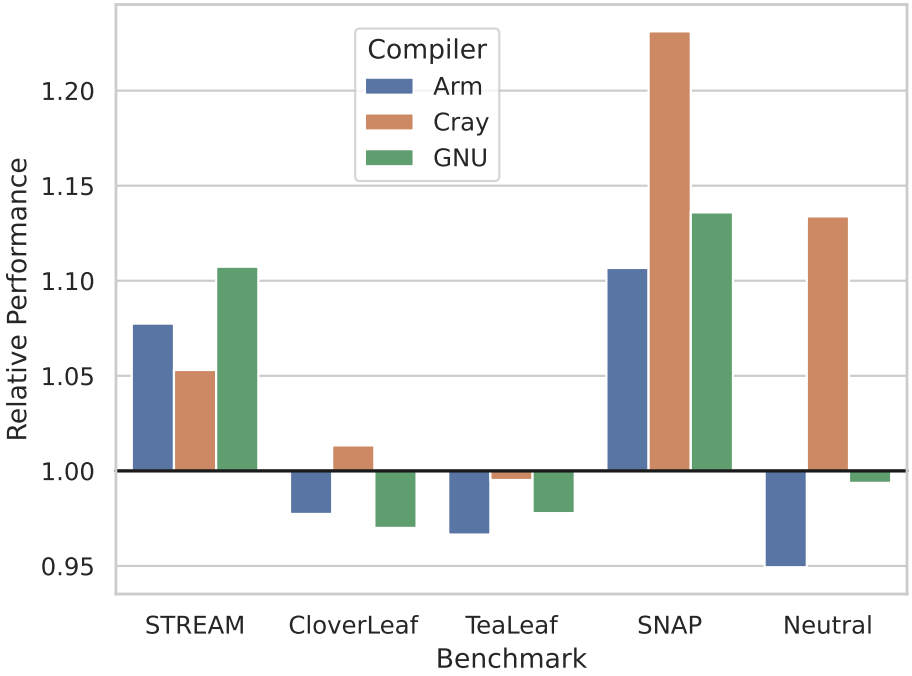


Figure 3.8: Relative performance of the latest version of TX2 compilers in 2021 compared to the initial releases in 2018. Numbers above 1 represent an increase in performance.

I observed 5–10% increases in the memory bandwidth reported by STREAM, but no significant performance change for any of the bandwidth-bound mini-apps. There were increases in the performance of SNAP — particularly with the Cray compiler, which improved by more than 20% — which suggest that the compilers have more mature cost models of the TX2 microarchitecture. CCE also generated better code for Neutral, closing the gap to the other compilers. The performance of the code generated by the Arm Fortran compiler was 5% lower than 3 years before. Figure 3.8 shows the performance achieved for the mini-apps with the newer toolchains.

Full applications have since been updated to support the latest compilers. Build errors have been reduced, and I did not find any Arm-target-specific build failures — where an application failed to build with a compiler, it did so on both Arm and x86. Performance did improve for some applications, but that was the result of manual optimisation work rather than compiler improvement: in GROMACS, for example, the `ARM_NEON_ASIMD` implementation has been improved and an `ARM_SVE` variant has been added. As for libraries, ArmPL and FFTW produced virtually similar results on the ThunderX2 after undergoing optimisations for these platforms. These observations are consistent with a stable, developed ecosystem and are what I expected to find after years of improvement.

3.4.3 Library Performance Comparison

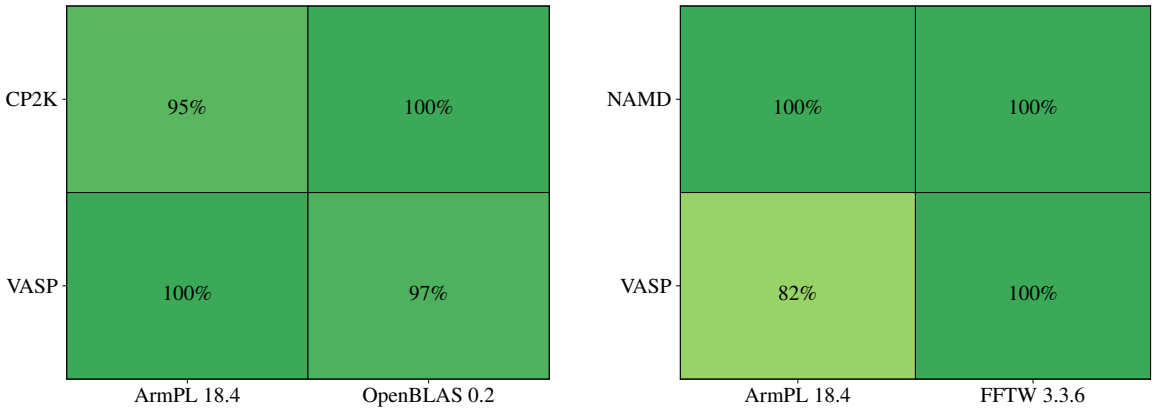
In addition to the compiler's ability to generate optimised code for the target processor, some of the benchmarks presented in Section 3.4.1 utilise external libraries for optimised maths routines. This is standard practice in scientific applications, because it lets application developers focus on their specific problem, while ensuring that optimisation work put into common routines can benefit everyone. The x86 architecture is well established in HPC, and so it is a common expectation that maths libraries provide good performance, both in open-source packages and in vendor-specific toolchains. On Arm-based platforms, however, the introduction of the ThunderX2 in the

HPC space provided the first real opportunity to evaluate the performance of the libraries.

The benchmarks in this work use external libraries for two types of operations: FFTs and basic linear algebra subprograms (BLAS). In each case, there are two alternatives that can often be used interchangeably:

- for FFTs, Cray provide an optimised build of FFTW, and Arm implement the same FFT interface as part of the Arm Performance Libraries (ArmPL);
- for BLAS, the proprietary ArmPL implementation can be used, or the open-source OpenBLAS package can be built from source.

Figures 3.9a and 3.9b show heatmaps of the relative performance of these options in the benchmarks where there are used, for BLAS and FFT, respectively. For each benchmark, the best-performing choice is marked 100% and the other option is given as a relative percentage of the former. The compiler here is the Arm HPC compiler in all cases, because it is compatible with all these library options.



(a) BLAS (b) FFT

Figure 3.9: Relative performance of optimised maths libraries on ThunderX2.

The results show that while performance differences did exist between the different libraries, in most cases these affected the overall benchmark result by less than 10%. The one exception to this rule was VASP when run with the FFT implementation in ArmPL, which was identified as a weakness in

the Arm implementation and was fixed in subsequent release of the library. The small differences observed increased confidence that the Arm toolchain ecosystem was of high, production-ready quality even from the first releases of software targeting the very first Arm-based processor aimed at HPC workloads.

3.5 ThunderX2 Performance Summary

Overall, the results presented in this chapter demonstrate that the Arm-based Marvell ThunderX2 processors are able to execute a wide range of important scientific computing workloads with performance that was competitive with state-of-the-art x86 offerings at the time of its release. The ThunderX2 processors can provide significant performance improvements when an application's performance is limited by memory bandwidth, but are slower in cases where codes are compute-bound. When processor cost is taken into account, ThunderX2's proposition was even more compelling. With multiple production-quality compilers now available for 64-bit Arm processors, the software ecosystem has reached a point where developers can have confidence that real applications will build and run correctly, in the vast majority of cases with no modifications.

Some of the applications tested highlighted the lower floating-point throughput and L1/L2 cache bandwidth of ThunderX2. Both of these characteristics stem from the narrower vector units relative to AVX-capable x86 processors, but Arm SVE enables hardware vendors to design processors with much wider vectors of up to 2,048 bits, compared to the 128 bits of today's NEON. Therefore, SVE-enabled Arm-based processors are anticipated to likely address most of the issues observed in this study, enabling Arm processors to deliver even greater performance for a wider range of workloads. I explore the extent to which this potential is fulfilled in the first generation of SVE hardware in Chapter 6.

Another difference between the processors used in this study is the total cache available. The longer the vector width, the faster the cache can be filled, but even at the same vector widths difference cache configurations

can affect application performance. I further investigate the impact of a processor's cache structure on application performance in Chapter 5.

3.6 Reproducibility

The benchmarks presented in this chapter cover a large set of applications, each with specific build-time and run-time options and potentially different flag choices on each platform. This parameter space has been captured in a set of scripts that can be used to reproduce the results in this chapter. They are set up for the systems used in this work, but can be easily modified for other environments. The scripts are available online[5].

[5] `https://github.com/UoB-HPC/benchmarks`

CHAPTER 4

Next-Generation Vector Instruction Sets

Content from this chapter appears in the following publication:

- Andrei Poenaru and Simon McIntosh-Smith. 'Evaluating the Effectiveness of a Vector-Length-Agnostic Instruction Set'. In: *Euro-Par 2020: Parallel Processing*. Euro-Par 2020 (Warsaw, Poland, 24–28 August 2020). Ed. by Maciej Malawski and Krzysztof Rzadca. Cham: Springer International Publishing, 2020, pp. 98–114

Modern processors rely on SIMD hardware to provide high performance for scientific applications. Vector hardware is not a new concept, with its origins reaching back to the CRAY-1 in 1975, but taking advantage of such capabilities has become increasingly important over the past few years.

Current x86-based processors offer SIMD capabilities through the 256-bit AVX2 and 512-bit AVX-512 instruction sets. Arm-based alternatives, however, have so far only offered 128-bit vectors through the instruction set previously known as NEON, which is now part of the ARMv8 ASIMD instruction group. The relatively short width of ASIMD vectors, combined with the reduced flexibility of this instruction set originally designed for media and signal processing, has limited the performance of Arm-based processors on a number of scientific applications [87]. In Chapter 3, this effect was observed on some of the more compute-bound benchmarks.

The next generations of high-performance Arm processors will use the Scalable Vector Extension (SVE) to provide more powerful vector operations [128]. Unlike other current mainstream SIMD implementations, SVE is a vector-length-agnostic (VLA) instruction set, allowing each implementation to choose a vector width between 128 and 2048 bits, in increments of 128 bits, with SVE binaries being portable between implementations. The first SVE-capable hardware did not become available until 2020 [134], but a number of tools that enable SVE experiments through either emulation or simulation were available earlier. In this chapter, I use these SVE performance tools to assess the efficacy of the new vector instruction set across a range of common HPC problem classes.

First, I compare the vectorisation efficiency of several HPC mini-apps on contemporary vector platforms from Arm and Intel; this sets a baseline for expectations. Then, I analyse how SVE mini-apps representing different classes of scientific applications are able to exploit SVE, by inspecting executed vector code and memory access patterns. Because the vector width is not fixed in SVE, this can be done across a range of chosen widths, examining the changes at every step. Finally, I evaluate the state of early SVE compilers and performance analysis tools, which are critical for the adoption of this new platform in HPC.

The goal of these experiments is to study the applicability in HPC of SVE as a target instructions set, without relying on a specific hardware implementation. Lessons learned here can be used to guide future implementations towards making design decisions that bring the biggest benefit possible to HPC applications. This work is a good example of how the *co-design* of hardware and software can build towards more advanced, more efficient, and more performant systems.

4.1 Modern Vector Instructions Sets

Section 2.1.1 explained that it is common for vector code to be produced by optimising compilers. However, compiler-backed auto-vectorisation cannot be assumed to be optimal [75, 109]. Therefore, it is important to eval-

uate its effectiveness on new hardware platforms. Furthermore, differences in instruction sets and their implementation in hardware can cause different behaviour on two distinct processors, even when the same benchmark and toolchain are used.

On x86 processors there are many variants of AVX available, and the optimal code for each variant may be significantly different [149], but with the Arm SVE instruction set, the generated machine code does not depend on a fixed vector width. Instead, executables automatically exploit the widest vector size available at run-time, using an approach similar to that of the very first vector computers [129]. This is particularly attractive for benchmarks based on real-world scientific applications, as they tend to steer clear of platform- or vendor-specific optimisations and instead opt for portable code[1].

SVE is implemented in new and upcoming generations of Arm-based HPC processors, including the recently announced Fujitsu A64FX [147] and the Marvell ThunderX4 [121]. Because SVE supports vector widths between 128 and 2048 bits, chip designers need to select the vector width to be used in their implementation. It is, thus, important to estimate how this choice will affect the performance of applications run on such future processors, and experiments are already being run to determine the impact of SVE width on scientific kernels [63].

4.2 SVE Evaluation Methodology

To study the efficacy of SVE in the field of HPC, I used mini-apps. These use only OpenMP or MPI, require no external libraries, and rely on automatic vectorisation by the compiler, i. e. no platform-specific intrinsics are used. Each mini-app is representative of a different class of common scientific problems, and they were individually introduced in Section 2.5.1. The same set was used to characterise the performance of Isambard, the first production-ready system based on ARMv8 processors. This work was

[1] One notable exception to this is GROMACS, introduced in Section 3.2. Implications of their choice to implement vectorisation using platform-specific intrinsics will be presented in Chapter 6.

presented in Chapter 3 and gives a good overview of their performance on modern HPC systems.

I performed the experiments described in this chapter using a combination of static and dynamic analysis tools. The compilers used were the latest versions of the three main SVE toolchains available at the time of writing: Arm HPC Compiler 19.2, GCC 8.2, and Cray Compiler (CCE) 9.0; for SVE, a pre-release version of the Cray Compiler, 9.0a, was used. I enabled most compiler optimisation with the flags `-O3 -ffast-math -mcpu=thunderx2t99+sve`; full reproducibility details can be found in Section 4.7. In all experiments, I used a single OpenMP thread and MPI process (where applicable), and the inputs were chosen such that the non-instrumented run time is below 5 seconds on a single core of a ThunderX2 processor. I used compiler optimisation listings and annotated source code to count vectorised loops in each mini-app, and I confirmed that vector instructions are run using hardware counters.

These experiments were performed prior to any SVE-equipped hardware becoming available, and so do not focus on any particular microarchitectural implementation. Instead, I ran the SVE versions of the mini-apps using the Arm Instruction Emulator (ArmIE)[2]. ArmIE runs base AArch64 instructions natively on the host, and switches to emulation when encountering SVE instructions. It also allows user-defined instrumentation code, known as instrumentation *clients*, to be run over both the native and emulated parts of the application.

For these experiments, I wrote custom instrumentation clients to record data about the instructions executed and the memory accesses performed by the programs. I limited instrumentation to the core computation kernels in the mini-apps, such that data is not collected for the initialisation and shut-down stages of the applications, because these are generally not important when measuring real-world performance. Recording data outside the kernels can skew the results by showing a misleadingly high number of scalar instructions if these sections are not optimised for vectorisation. To define the regions where data was collected, I inserted special instructions to

[2] `https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator`

start and stop instrumentation, which are invalid AArch64 instructions but are recognised and honoured by our ArmIE client.

I classified dynamically recorded instructions into several categories: scalar AArch64 (A64), vector AArch64 (i.e. Advanced SIMD/NEON), SVE arithmetic, SVE memory loads, SVE memory stores, SVE moves, and SVE control flow. I used the memory access trace data to describe each operation as ⟨load/store, contiguous/non-contiguous, some/all vector lanes active⟩. The SVE vector width was set by stepping through the powers of two between 128 and 2048.

4.3 Results

4.3.1 Compiler Vectorisation Efficiency

I analysed the static vectorisation efficiency of SVE compared to AVX by looking at the loops in the core computation kernels of each mini-app. I selected loops to cover the majority of the mini-apps' run times, as reported by a profiled run on a real ThunderX2 processor. For targeting Arm, both with SVE and NEON, I used the three main HPC compilers: Arm's HPC compiler, GCC, and the Cray Compiler; for x86, I used the same versions of GCC and Cray, but I used the Intel Compiler 19.0 instead of the Arm HPC Compiler.

Table 4.1 shows, for each application, the number of loops considered, the percentage of run time that they represent, and the number of loops vectorised by each compiler on each platform. I show TeaLeaf twice — once using a CG solver, once using a PPCG solver — because the two runs cover very different code paths, and both are representative of real workloads. There are no MiniFMM results with the Cray Compiler because the application's build system does not currently support the Cray Compiler.

Aggregating the results across mini-apps, I observed that the compilers which can generate code for all the instructions sets vectorised the highest number of loops on SVE.

Table 4.1: Number of loops vectorised by each compiler on the top loop-nests, selected by percentage of total run time on a ThunderX2 processor, in the mini-apps studied. The results for AVX2 and AVX-512 were identical; here they share the *AVX* label.

Application	% Time (Total Loops)	SVE			NEON			AVX		
		Arm	Cray	GCC	Arm	Cray	GCC	Intel	Cray	GCC
STREAM	92.4 (4)	4	4	4	4	4	4	4	4	4
miniBUDE	98.6 (4)	4	3	3	3	4	3	4	4	3
TeaLeaf (cg)	87.2 (8)	5	6	8	5	6	8	8	6	6
TeaLeaf (ppcg)	91.2 (6)	6	6	6	6	6	6	6	6	6
CloverLeaf	62.5 (10)	9	10	6	8	9	6	10	9	8
MegaSweep	70.3 (4)	1	4	0	1	1	0	4	1	0
Neutral	85.8 (2)	0	0	0	0	0	0	0	0	0
MiniFMM	98.1 (8)	7	—	5	3	—	5	7	—	5
Total	(46)	36	32	32	30	30	32	43	28	32

I then studied the factors influencing vectorisation on each mini-app individually. TeaLeaf with the PPCG solver was fully vectorised on all the platforms, by all compilers. TeaLeaf with CG and miniBUDE achieved 80% or more vectorisation with all compilers; it should be possible to achieve full vectorisation, as shown by the Intel compiler on AVX and GCC on Arm. CloverLeaf and MiniFMM showed all loops except one vectorised with Arm, Cray, and Intel, but only about half with GCC; GCC reports that further vectorisation is not beneficial according to its cost model, on all platforms, due to indirect access. MegaSweep was not vectorised by GCC on any platform, but fully vectorised by Cray on SVE and Intel on x86, which suggests vectorisation is possible, but not all compilers understand the loops' structure. Neutral was not vectorised at all, on any platform, due to the deeply nested branching in its algorithm.

The differences in vectorisation between SVE and NEON are due to the higher flexibility of SVE: there are more general-purpose vector instructions available, compared to the multimedia-focused NEON instruction set, so more operations can be implemented efficiently using vectors. When targeting x86, all compilers vectorised the same number of loops on both AVX2, e. g. for Broadwell, and AVX-512, e. g. for Skylake. An ideal compiler should be able to vectorise all 46 loops studied, although it is likely that the performance gained by vectorising the remaining three loops — two in Neutral and one in MiniFMM — depends on the data supplied at run-time.

4.3.2 Dynamic Instruction Analysis

After I obtained vectorised code for the mini-apps, I recorded dynamic instruction execution traces at each power-of-two SVE vector length between 128 and 2048 bits. I added a NEON-only and a non-vectorised (scalar) run for each application, to serve as baselines against which to compare the SVE results. The traces allowed us to identify the types of SVE instructions executed and how their dynamic count varies with the chosen vector length.

Figure 4.1 shows the dynamic instruction count analysis for the **STREAM** benchmark, where instructions are grouped by type: scalar AArch64, NEON

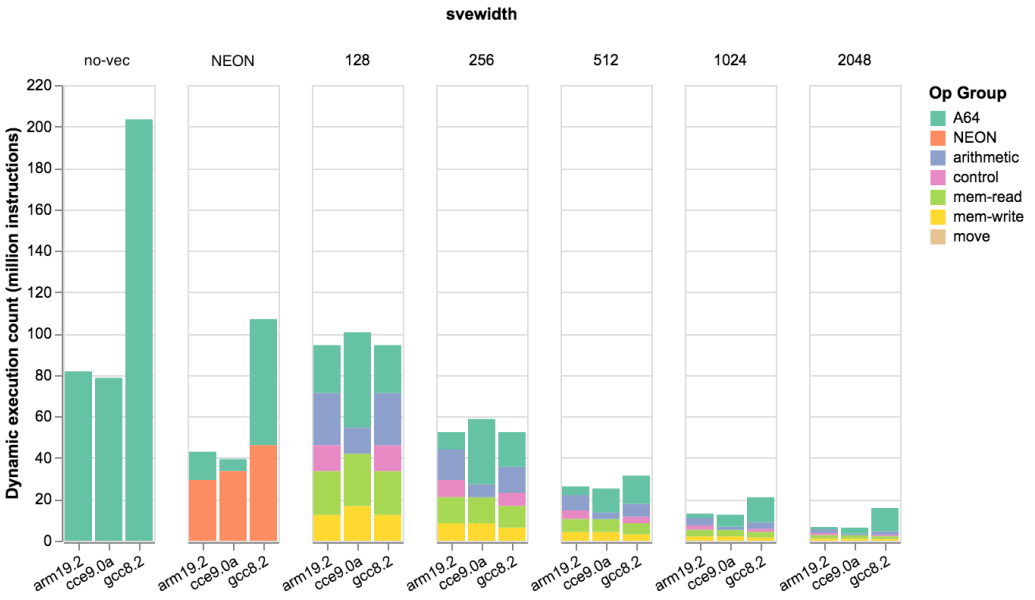


Figure 4.1: Dynamic instruction count and grouping for STREAM. Lower is generally better. *A64* refers to scalar instructions; *NEON* refers to base-AArch64 ASIMD vector instructions; the remaining groups are all SVE instructions.

(AArch64 ASIMD), and several groups of SVE operations; a lower number of instructions executed is generally better. In the scalar and NEON-only cases, the Arm and Cray Compiler showed similar behaviour, but the GCC version ran more than twice as many instructions because it did not make use of load/store pair instructions, an operation in which two 64-bit values can be read from/written to memory in a single instruction. When targeting SVE, all three compilers performed similarly, and I saw a decrease in the total instruction count as I increased vector length, since each instruction had increasingly more active lanes. No compiler generated load/store pairs for SVE, so the instruction count at 128 bits — the same vector length that NEON uses — is close to that observed for GCC when targeting NEON. The Arm and Cray compilers, but not GCC, chose to use scalar A64 instructions for loop control flow, which resulted in the scalar instruction count also varying with SVE width.

miniBUDE, a heavily compute-bound application, ran vector code almost exclusively, which results in a clear inverse relation between the dy-

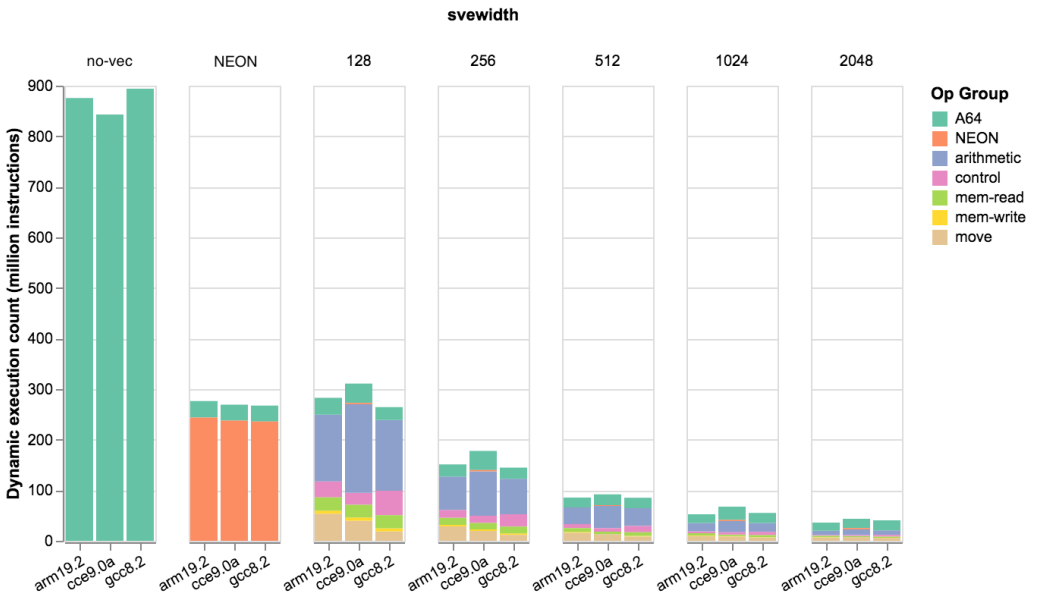


Figure 4.2: Dynamic instruction count and grouping for miniBUDE. Lower is generally better. *A64* refers to scalar instructions; *NEON* refers to base-AArch64 ASIMD vector instructions; the remaining groups are all SVE instructions.

namic instruction count and the vector length. All compilers performed very similarly for this application. The results are shown in Figure 4.2.

TeaLeaf and **CloverLeaf** exhibited similar behaviour: the code was only partially vectorised, leading to a mixture of SVE and scalar instructions. As the SVE length was increased, the number of executed SVE instructions decreased, but the number of scalar instructions executed stayed constant. The non-SVE part comes largely from outer-loop code, since in these cases only the innermost loop is vectorised by the compilers. Figures 4.3 and 4.4 show the results for TeaLeaf and CloverLeaf, respectively. The TeaLeaf results shown are for the CG solver, which is the default and most commonly utilised option for the mini-app.

MegaSweep was only vectorised by the Cray Compiler. As with STREAM, CCE performed control flow using scalar instructions, so the instruction counts followed a similar profile here. Because the GCC- and Arm-compiled versions were not vectorised, all instructions run were scalar A64 and their

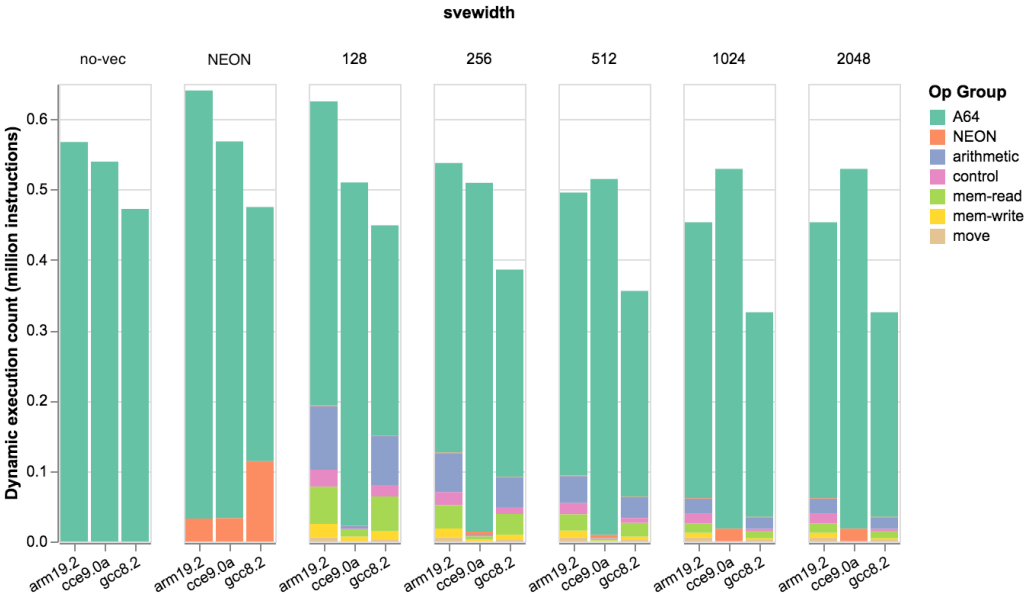


Figure 4.3: Dynamic instruction count and grouping for TeaLeaf.

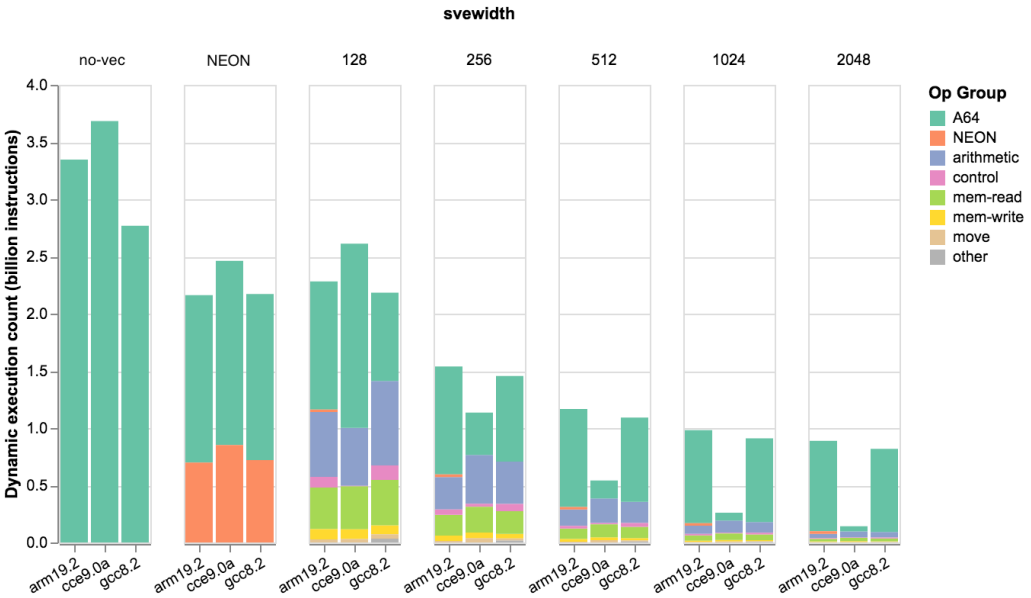


Figure 4.4: Dynamic instruction count and grouping for CloverLeaf.

execution count did not change with SVE width. Figure 4.5 shows the results for all three compilers.

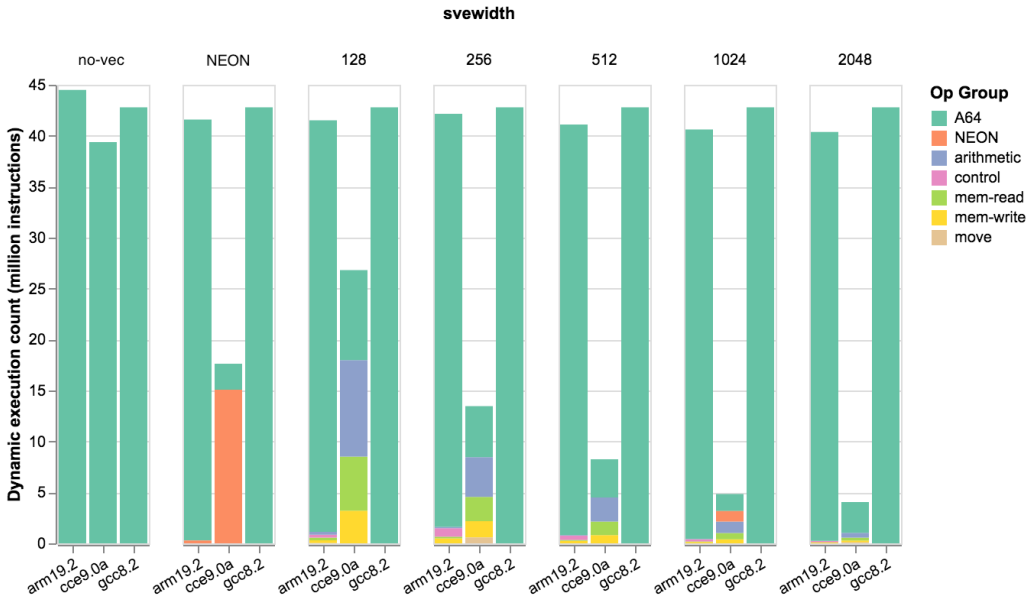


Figure 4.5: Dynamic instruction count and grouping for MegaSweep.

Figure 4.6 shows the dynamic instruction analysis for **MiniFMM**. This application's build system does not currently support the Cray Compiler, so results are only shown for GCC and Arm. Even though the application was (partially) vectorised, the instruction count did not decrease significantly when increasing the SVE vector width over 512 bits, in contrast to the applications presented previously. Due to an interaction between the way MiniFMM vectorises over particles and the small scale of the problem run, not all the lanes in SVE registers were being utilised at high vector lengths; since the vectors were partially empty, the total instruction count did not decrease linearly.

Neutral is excluded from this analysis because it was not vectorised at all.

4.3.3 SVE Vector Lane Utilisation

Because SVE instructions employ per-lane predication, observing that SVE instructions are being *executed* is not enough to conclude that the ap-

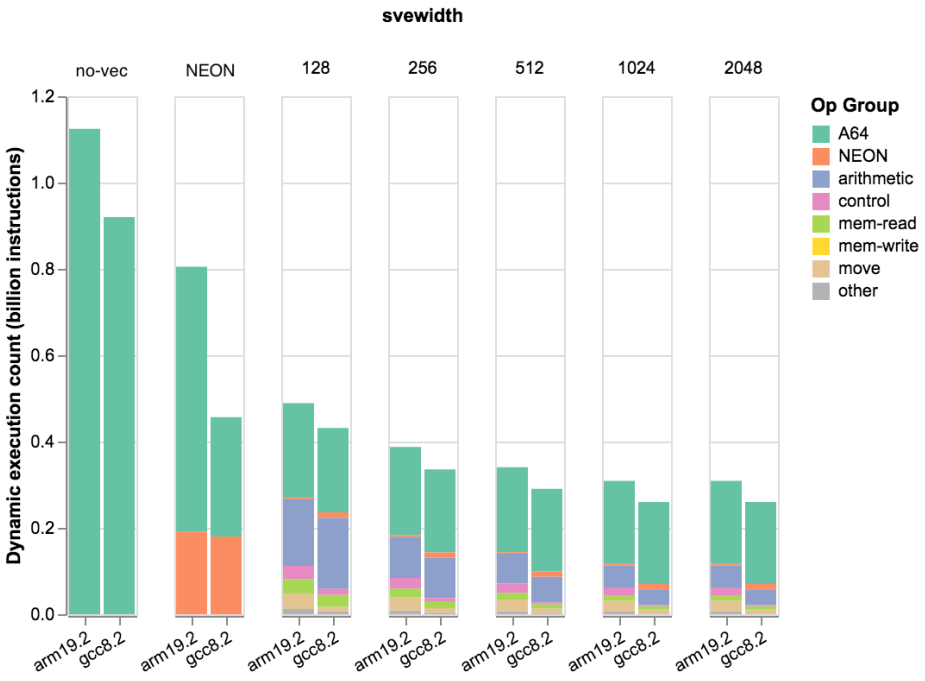


Figure 4.6: Dynamic instruction count and grouping for MiniFMM.

plication is using vector operations efficiently — it is possible that a large portion of the elements, potentially all but one, are masked out. This means that vector register can be underpopulated, almost empty. To investigate this, I looked at per-lane utilisation of SVE registers when running the mini-apps.

For applications with a high degree of vectorisation, e. g. miniBUDE, TeaLeaf, or CloverLeaf, vector operations were performed using all the lanes, i. e. at maximum utilisation. For MiniFMM, however, the number of active lanes varied: at 512-bit-wide SVE and below, most instructions used 80% or more of the lanes available, but when increasing the SVE length further, vector register utilisation peaked between 512 and 768 bits. Vector utilisation was virtually identical across both compilers tested, Arm and GCC.

Figure 4.7 shows a histogram of the number of active bits in SVE operations, grouped in 128-bit-wide bins. Increasing the SVE width past 512 bits brings little benefit for MiniFMM, as only a minority of the operations performed use more than 512 bits. When the vector width is set to 1024 bits,

less than 5% of the instructions use the full available width, and further increasing the width to 2048 bits produces no change in vector utilisation.

In contrast, Figure 4.8 shows how miniBUDE, a mini-app that vectorises efficiently, was able to fully utilise vectors in *all* operations, even at the highest widths allowed by SVE. The other mini-apps investigated in this chapter showed the same perfect vector utilisation efficiency as miniBUDE. These results cover both 32- and 64-bit floating-point data types: miniBUDE uses 32-bit data (`float`), and the other mini-apps use 64-bit types (`double`).

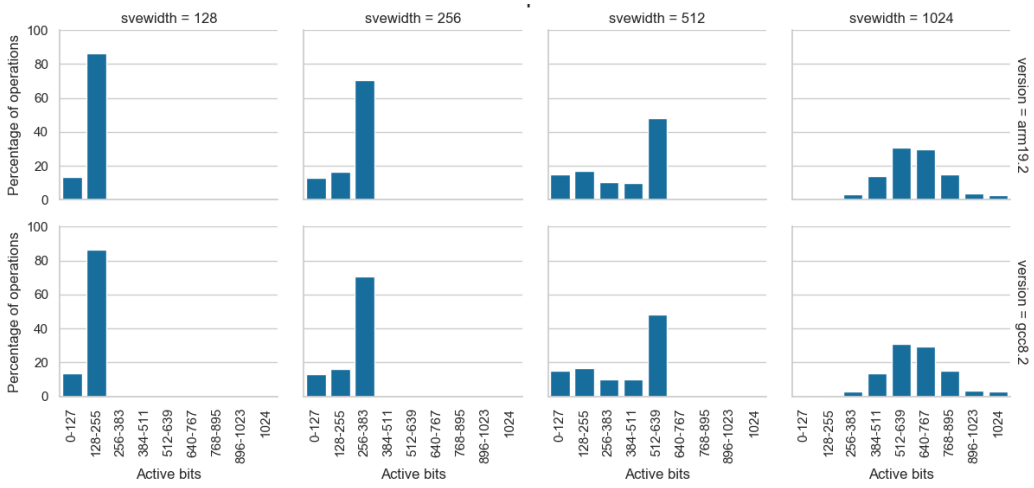


Figure 4.7: Histogram showing the number of active bits in the SVE operations performed by MiniFMM. The application cannot saturate the full widths of the vectors when the SVE length is 512 bits or higher.

4.3.4 SVE Memory Operations

Finally, I looked at how the mini-apps are able to take advantage of SVE for memory operations. Since all SVE instructions are predicated per-lane, including contiguous and strided memory operations, every SVE memory instruction can differ in the number of bytes transferred.

I found that SVE usage for memory operations varied greatly between applications. Mini-apps with lower degrees of vectorisation, such as MegaSweep, used little SVE for memory accesses, but even applications with a higher degree of vectorisation showed a mixture of SVE and non-SVE

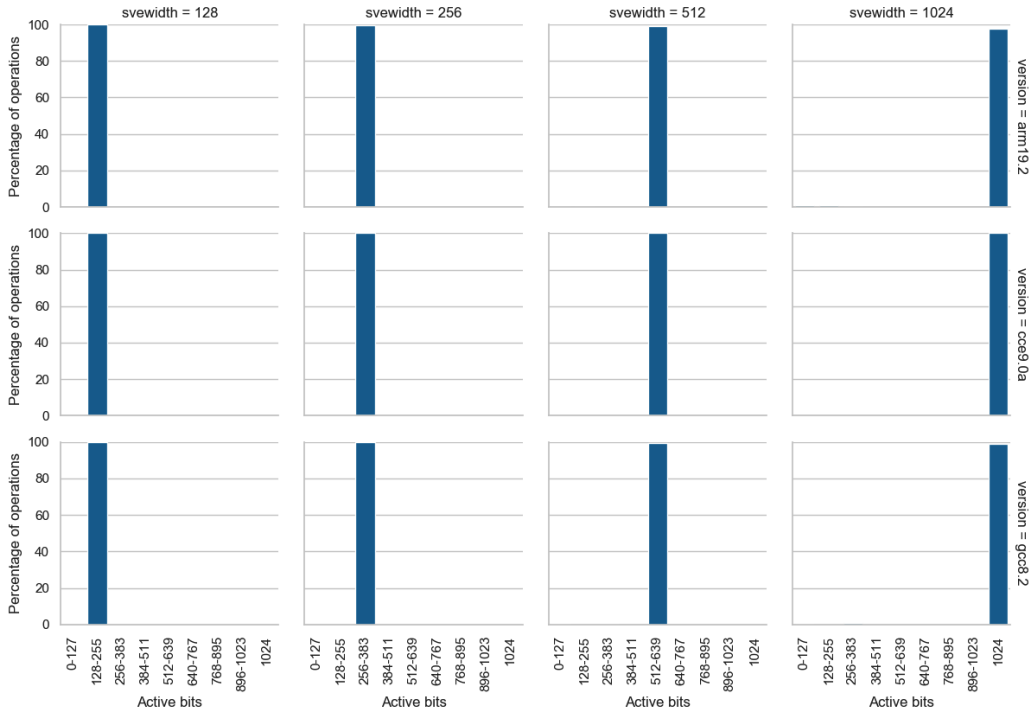


Figure 4.8: Histogram showing the number of active bits in the SVE operations performed by miniBUDE. Vectorisation is perfectly efficient at all SVE widths.

memory operations. In miniBUDE, about three quarters of the memory instructions were SVE instructions; in CloverLeaf, TeaLeaf, and MiniFMM, between a quarter and a third of the memory operations were SVE. In MiniFMM, of the SVE operations, about a third were gathers, while there were no scatters; the other applications utilised contiguous accesses almost exclusively. All applications utilised all the SVE lanes in their memory operations, except for MiniFMM, where about half the SVE memory operations, including all the gathers, were only partially filled.

Figures 4.9 and 4.11 show the distributions of memory accesses in miniBUDE and MiniFMM, respectively. These two mini-apps form the most contrasting pair in the set of mini-apps evaluated. The observations here are consistent with Sections 4.3.2 and 4.3.3: miniBUDE vectorises very efficiently, and MiniFMM utilises some SVE-specific features but does not always utilise all vector lanes available. CloverLeaf, shown in Figure 4.10, performed both SVE and non-SVE memory operations, with the large majority of accesses being contiguous. More analysis of the non-contiguous scatter and gather accesses in CloverLeaf is presented in Chapter 5.

These results are collected from the version of the applications compiled with the Arm Compiler 19.2 and run on 512-bit SVE, which is the vector length utilised in the Fujitsu A64FX processor. The absolute numbers of vector operations varies between the versions built with different compilers and when adjusting the SVE width, but the same important characteristics can be seen in all cases, and the conclusions drawn are similar.

4.4 SVE Usage Discussion

The **STREAM** benchmark runs simple, predictable memory operations. All the compilers tested were able to successfully use SVE — at all vector lengths — to vectorise this code, and at run-time the vectors were fully utilised. This is the expected behaviour for the benchmark.

miniBUDE is a heavily compute-bound benchmark, and thus complementary to STREAM. This application shows very efficient utilisation of SVE: the main kernels all execute vectorised operations, which scale with

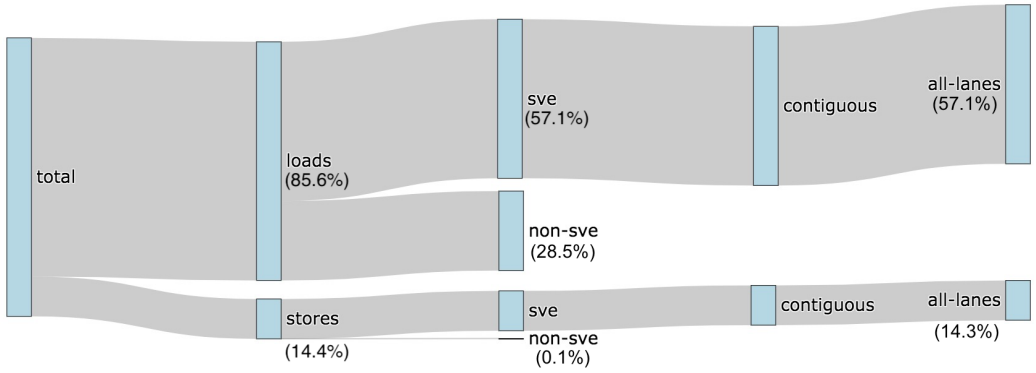


Figure 4.9: Relative counts, by number of instructions, of memory operations in miniBUDE. All memory accesses are contiguous and most are performed through SVE instructions.

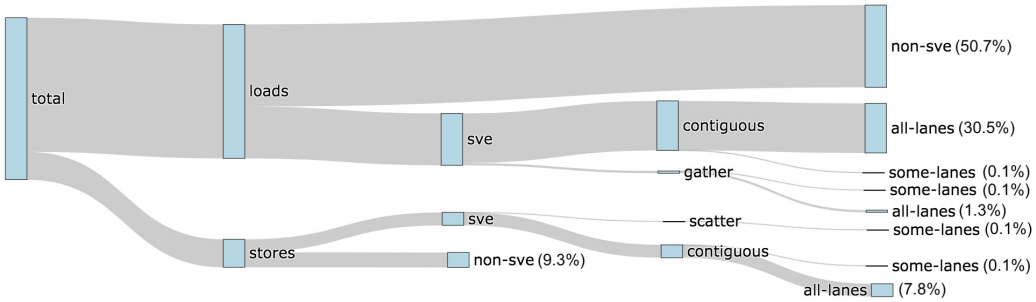


Figure 4.10: Relative counts, by number of instructions, of memory operations in CloverLeaf. Memory accesses are split between SVE and non-SVE instructions. In the vast majority of cases where SVE is used, accesses are contiguous and all the lanes are being utilised.

the chosen SVE length. At 128 bits, the amount of code run — both vector and scalar — is almost identical to the established NEON version, which indicates that good code is generated by all the compilers. Increasing the vector length by 2× reduces by half the number of instructions run up to 1024 bits; at 2048 bits, the total number of executed vector instructions becomes smaller than the number of scalar instructions.

Even though more than half of the main loops in **TeaLeaf** are vectorised by all the compilers, only relatively few vector instructions are executed at run-time: for 128-bit SVE, these represent less than a third of the total instructions run for the Arm and GCC versions. Increasing the vector length

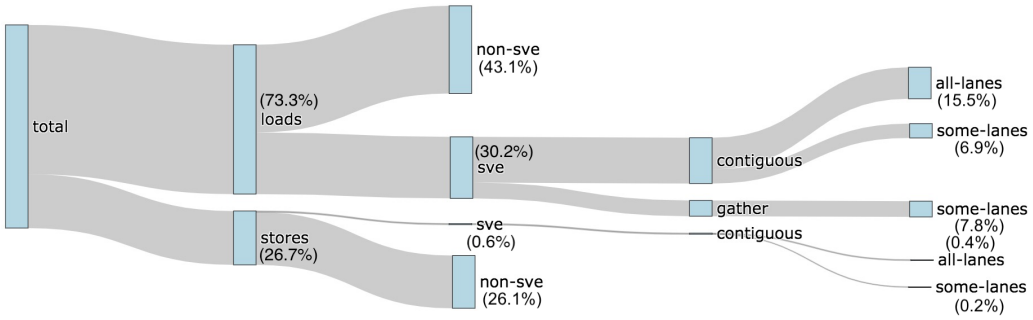


Figure 4.11: Relative counts, by number of instructions, of memory operations in MiniFMM. This applications shows a mixture of SVE and non-SVE operations, and the SVE ones show a further split between contiguous and non-contiguous accesses. Not all lanes are always used in SVE operations for MiniFMM.

decreases the count, but only with around 50% efficiency and up to 1024 bits; there is virtually no change going to 2048 bits. The Arm-compiled executable runs comparatively more instructions than the GCC version, by 35–40%, depending on the chosen vector length. With the Cray executable, less than 10% of the instructions run are vector operations, even though the compiler vectorised the same loops as Arm and GCC; at 1024 and 2048 bits, the vector code run is NEON, and not SVE, which I suspect is due to a compiler bug.

The **CloverLeaf** benchmark shows characteristics similar to TeaLeaf, but with more vector instruction utilisation. In all three versions, vector instructions account for between a third and half of the total instruction count at 128 bits; all three compilers produce a similar total dynamic instruction count. The SVE instruction count scales as expected up to the largest vector width possible, 2048 bits. The Cray-compiled version initially runs the highest number of total instructions, but it decreases sharply at 256 and 512 bits; at 512 bits more than two thirds of the code executed is SVE, and at 2048 bits the total count constitutes 22% of those of the Arm and GCC versions, suggesting that the Cray compiler optimises better for higher vector lengths.

This also hints at the importance of the loop chosen for vectorisation: if a compiler is able to vectorise the outer loop, as CCE is, and perhaps also to collapse the inner loop when doing so, the reduction in instruction count

at high vector lengths can be considerable. On the other hand, the same strategy may not be desirable at smaller vector lengths, where vectorising the inner-most loop may be optimal. This would imply that, for optimal code generation, the compiler either needs to know the hardware vector width at compile-time, or it needs to generate several code paths and dynamically choose the optimal one when the vector length information becomes available at run-time.

A related issue is that the compilers tested in the study use a generic cost model for SVE, which stays the same even when targeting different vector lengths. This is a limitation in the current study, since it may not accurately reflect any real implementation. With access to the cost model of a different SVE implementations, the compilers may generate different code to take advantage of each implementation's strengths.

In CloverLeaf, SVE memory accesses represent about half the total memory operations performed, both when reading and writing, and the vast majority of those are contiguous operations.

Of the mini-apps included in this study, **MegaSweep** shows the most notable difference between the three compilers: Cray is the only one that successfully vectorises the code, both on NEON and SVE. The binary it produces runs 2.5× fewer total instructions than Arm and GCC at 128 bits, and the amount of SVE instructions executed scales almost perfectly up to 2048 bits, although the 1024-bit binary highlights a compiler issue where some of the code run is NEON, not SVE, which reduces the scaling efficiency in this particular case. At 2048 bits, the Cray version runs 10.5× fewer instructions than the GCC alternative. The Cray version also successfully utilises SVE for memory access, all of which are contiguous and are able to exploit the full lengths of the vectors.

Neutral does not vectorise with any of the compilers, so no SVE is being run. Due to the nature of the Monte-Carlo algorithm, there is little structure in the access patterns in the kernels. As Martineau and McIntosh-Smith explained, it is possible to force vector code generation, but it will be comprised almost entirely of indirect, variable-stride accesses that do not

improve performance [79]; the compilers make the right choice to generate scalar instructions in this case.

In general it is desirable to utilise as much of the available vectors as possible, but partial utilisation does not always signal a problem. The **MiniFMM** result exhibits the flexibility of SVE: even though the parallelisation strategy in the application cannot fill the vectors above 512 bits, the hardware can still efficiently utilise its resources by executing partially masked operations. These operations should not be any more expensive than regular operations with full vectors, and so are more efficient than falling back to scalar code. It is possible to construct an input for MiniFMM that can utilise longer vectors, but this would require either a larger problem scale, with many more total particles, which would be intractable under an emulated environment, or a configuration of the FMM parameters that would not be representative of real FMM runs.

4.5 Relevance of SVE for HPC

The results presented in Sections 4.3 and 4.4 show that SVE is a viable, competitive vector instruction set for HPC applications. For HPC workloads, it represents a noticeable improvement over NEON, bringing high-performance Arm processors in line with current-generation x86 processors, both in terms of the available vector length and the flexibility of the operations.

Even before SVE hardware was available, I have found the SVE toolchains to be mature already. Generating SVE code only required enabling the SVE extension in the target architecture flag, and the compilers were successful in utilising SVE where expected. Compared to NEON, more loops were vectorised with SVE by *all* compilers. In addition, the Arm and Cray compilers achieved a similar or higher degree of vectorisation with SVE compared to AVX-512, a significant improvement compared to what was previously possible with NEON.

One of the main advantages of SVE arose from its per-lane predication, which allowed loops with heavy control flow to be vectorised without addi-

tional cost. This additional flexibility meant it was sometimes beneficial to vectorise loops on SVE even when it was not on other instructions sets.

In the wider context, these results suggest that many HPC applications should be able to utilise SVE and benefit from doing so. The flexibility of SVE allows a wide range of loops to be turned into vector code, including cases where vectorisation is not possible with NEON or AVX, e. g. with irregular and unpredictable access patterns. Compute-bound applications can exploit high vector widths, bringing the number of instructions required significantly lower than on (128-bit) NEON. Partially filled operations allow vector instructions to be generated and executed even when the application cannot fill whole vector registers, a more efficient alternative than falling back to scalar code.

While in this study I have shown that SVE HPC applications behave well in an *emulated* environment, I cannot yet make any claims regarding their performance on *real* hardware. Implementations of SVE are likely to come with caveats and performance characteristics which cannot be determined *a priori*, and so it is impossible to predict which types of operations will be fast and which will bring little improvement over scalar code. Until several SVE implementations, with different native vector widths, become available and supported by the compilers, such a study is infeasible.

4.6 Towards Accurate Performance Modelling

The analysis presented in this study covers the three main SVE compilers available at the time of writing. However, Fujitsu A64FX systems ship with a proprietary compiler supplied by Fujitsu to accompany their processor. Optimisations applied by this compiler may be key in extracting high performance from the A64FX, so analysing the binaries it produces should prove a valuable research direction. Chapter 6 studies the performance of the A64FX, the first hardware implementation of SVE, using real-world HPC applications.

One of the shortcomings of the compilers available at this stage was their lack of cost models for real SVE platforms, and further work will be enabled when the compilers are able to generate *tuned* binaries. The early versions used in this study only use a generic model of an SVE processor, because neither the compilers nor ArmIE currently allow the user to specify microarchitectural details, except the SVE width. Once a tuned binary can be generated, running it on its target platform will enable quantifying of the tuning benefit, and an even wider range of experiments is possible if these tuning parameters can be adjusted dynamically.

4.7 Reproducibility

The tools developed to perform the analysis in this chapter have been released as open-source software. Detailed build and run instructions for each application, the custom ArmIE instrumentation clients used, and scripts to aggregate and plot the collected data can be found online[3].

4.8 Conclusion

In this work, I have presented an analysis of SVE usage across a number of mini-apps that span several common HPC problem classes. I have looked at how currently available compilers are able to utilise SVE to automatically vectorise the mini-apps' code, how much of the executed code is SVE, the efficiency of the executed SVE vector instructions, and whether new ways of accessing memory introduced with SVE are utilised in these mini-apps.

I found that SVE was generally well targetted by the compilers: in most cases, compilers were able to utilise SVE at least as well as AVX and NEON, and often better. The available compilers for SVE were only surpassed by the Intel compiler targeting AVX on select few occasions. Most SVE binaries used wide vectors efficiently, with all lanes being active for the vast majority of the run time; MiniFMM was the only exception, where SVE efficiency

[3] `https://github.com/UoB-HPC/sve-analysis-tools/tree/euro-par-2020`

varied depending on the SVE width utilised. In terms of memory accesses, vectorised mini-apps were able to use SVE instructions to efficiently load and store data, and MiniFMM also made use of gather operations, either fully or partially filled. I saw little use of SVE scatter instructions, but this is expected given the optimised memory access patterns on the mini-apps studied.

I conclude that SVE is a promising instruction set, and HPC applications and toolchains appear ready to take advantage of it to deliver performant code running on upcoming generations of Arm-based high-performance processors.

With wider vector lengths comes an additional consideration when designing processors: the impact of these wide vector operations on the caches. In Chapter 5, I will examine how cache utilisation relates to the vector width and what parameters of the cache are most critical for high performance. I also explore the impact of non-contiguous memory operations, which are a new addition that came to the Arm instruction set with SVE. Finally, in Chapters 6 and 7 I explore the performance of the first implementation of SVE in hardware, the Fujitsu A64FX, and compare it with other mainstream processors, both from a microarchitectural point of view, but also from the perspective of programming, performance portability, and productivity.

CHAPTER 5

The Effects on Cache of Wide Vector Operations

Content from this chapter appears in the following publication:

- Andrei Poenaru and Simon McIntosh-Smith. 'The Effects of Wide Vector Operations on Processor Caches'. In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 2020, pp. 531–539. DOI: 10.1109/CLUSTER49012.2020.00076

In high-performance processors, the speed at which data can be processed is generally limited by how fast the processor can perform operations on it, how quickly it can be read from and written to memory, or a combination of both. To increase the overall throughput of operations, several architectural techniques can be employed to enable having more than a single instruction in-flight at a time, or the instructions themselves can be extended to several operands. These wide instructions, known as vector or single instruction, multiple data (SIMD) operations, require suitably high memory throughput in order to sustain their ability to consume operands. Main system memory — usually dyamic random access memory (DRAM) — does not on its own meet the latency and raw bandwidth requirements of high-performance CPUs, so caches are employed to bridge this gap.

Caches are additional memories that sit between the processor and its main memory. They are optimised for speed rather than size, so they are generally smaller but faster than main memory. This trade-off can be heavily variable, so it is common for processors to have several *levels* of cache arranged in a *cache hierarchy*, where the fastest — but smallest — cache is connected directly to the CPU and the largest — but slowest — comes right before main memory. When more than two levels are used, any other levels between the first and the last maintain this trend of increasing size and decreasing speed.

In varying the speed–size compromise, there are several parameters of a cache that can be tuned. Any changes in the cache's architecture will affect how the cache performs under load, not only in terms of raw bandwidth and latency, but also in its capacity to provide data to the CPU without needing to fetch it from higher up the memory hierarchy [82]. Two of the most common metrics used to describe cache efficiency look at how often requested data is not found in the cache (*cache misses*) and how much data is replaced when bringing in new data (*cache evictions*).

The effects on performance of changing cache parameters are particularly important when running vector instructions, because they have the potential to exhaust small caches very quickly. For example, the latest generations of Arm processors can use the Scalable Vector Extension (SVE) instruction set to processes up to 2048 bits of data in a single instruction. Because the size of a typical first-level cache is in the order of kilobytes, only a few tens of SVE instructions can be enough to fill up the cache. Hence, it is critical that caches are designed with not only micro-architectural constraints in mind, but also with the performance implications of the design choices made.

In this chapter, I investigate the lifetime of data in cache — its evict distance — at different SVE widths, for three different scientific mini-apps. I apply visualisations to present the effects of varying cache parameters in modern high-performance processors under typical scientific workloads, and I give detailed insight into the interaction of non-contiguous memory operations with the cache hierarchy in two contemporary AArch64-based processors.

5.1 Processor Cache Design Space

In modern systems, all requests from the processor to the main memory must pass through the cache hierarchy. Every possible memory address must therefore have at least one possible location in cache, but caches are generally orders of magnitude smaller than main memory, so several memory addresses could share the same cache location. To determine how main memory addresses are mapped to cache addresses, a *mapping policy* is used. Some caches in processors available today use *direct mapping*, where each address can only have one possible location in cache, but most use *set-associative mapping*, where a memory address can take any of a set of cache addresses. The size of each cache set is sometimes called the *associativity*, or the "number of ways" of the cache; a direct-mapped cache is a 1-way set-associative cache. Direct-mapped caches are faster — and simpler — but set associativity can decrease cache miss rate.

Once there is no more space for new data to be placed into cache, older data needs to be evicted to make room. Caches choose what to evict according to an *eviction policy*, of which the most common is the *least-recently used (LRU)* policy [140].

Regardless of the mapping and replacement policies used, caches are organised in *blocks*, or *lines*, where a cache block is the smallest structure that can be inserted or evicted in an operation. A common cache line size in use today is 64 bytes, which is used in many x86 processor and in the Marvell ThunderX2 processors, but other options exist, such as the 256-byte cache lines in the Fujitsu A64FX [38]. The choice of line size is particularly important in processors that support wide SIMD operations, because these can quickly touch large amounts of data.

All these variables create a large parameter space from which designers must choose a single option to implement in a processor. Each option comes with its own architectural constraints, strengths, and weaknesses, but the needs of the *software* running on the processor should also be considered. It is particularly important for HPC applications that hardware is designed to not only offer good raw performance, but to make a good portion of the peak

performance achievable in practice [29]. In recent years, this has led to the *co-design* of hardware and software, an iterative process in which hardware design choices are made so that it best supports the software running on it and, in turn, software is optimised for its target architecture.

SVE, Arm's next-generation vector instruction set, presents new challenges for software and hardware architectures due to its wide maximum vector length [107], but also offers timely opportunities for co-design [134]. In order to encourage experiments with SVE ahead of hardware release, Arm have created the Arm Instruction Emulator (ArmIE), a software tool that emulates the execution of real SVE binaries on any given vector length between 128 and 2048 bits [21]. ArmIE can instrument the binaries it runs to collect arbitrary user-defined metrics, among which is recording traces of memory the operations performed. The data from these memory traces can be used to investigate the behaviour cache hierarchies under the selected SVE width.

5.2 Cache Analysis Methodology

To investigate how cache design choices affect the performance of SVE applications, I used a cache simulator. I developed the simulator from scratch to process memory traces produced with ArmIE, with full support for both contiguous and non-contiguous operations, i. e. scatters and gathers, at any SVE vector length. The simulator was validated using hardware counter data obtained on the latest Arm-based CPUs used in HPC, the Marvell ThunderX2 (TX2) and the Fujitsu A64FX. The cache configurations of these two processors are shown in Table 5.1 [38, 144].

Data was collected on real hardware using the PAPI library [57] and was compared to the equivalent metrics obtained from the simulator. Comparisons were performed at the hardware's native vector width, so TX2 data was compared to simulated 128-bit SVE data and A64FX data was compared to simulated 512-bit SVE output. Table 5.2 presents the difference between data from hardware counters and the equivalent data produced by our cache simulator on a number of benchmarks. Total access and misses

are recorded as absolute numbers, and miss percentages represent the fraction of total cache accesses that missed. Averages between applications are computed from the raw numbers. These metrics are the same metrics used for the experiments throughout this study.

The differences between simulated and hardware data arise mainly because the simulator does not perform prefetching. I do not use prefetching because it is an implementation detail that can vary widely between processors — and one that is often not transparently described — and I aim to draw conclusions relevant to a wide range of processors, not a particular implementation. The memory traces generated by ArmIE also do not accurately model time, and prefetching mainly affects the *time* at which hits and misses occur: even without prefetching, data requested by the application will still generate the same number of hits and misses. Because time is not modelled in the simulator, prefetching has little effect on its output when *useful* data is prefetched. The remaining cases, however, where the prefetcher loads data that is not needed by the application, will generate additional misses on real hardware, which the simulator will not account for. This makes the simulated data slightly optimistic when counting misses, by up to 10%.

I benchmarked three different mini-apps, each representative of a different class of HPC applications: **CloverLeaf**, a hydrodynamics code that solves Euler's equations of compressible fluid dynamics [76]; **MegaSweep**, a STREAM-style benchmark that uses the main kernel from SNAP, a deterministic discrete ordinates transport proxy application [23]; and **MiniFMM**, a Fast Multipole Method mini-app [7]. Results from **STREAM** [83], the *de facto* benchmark for measuring memory bandwidth, were also used for validating the simulator. These applications were introduced in detail in Section 2.5.1.

The mini-apps were run in a single-core configuration, by setting the both the number of OpenMP threads and MPI processes to 1. The inputs to the mini-apps were derived from typical inputs for full-node runs, but with the problem size reduced by using a smaller grid and fewer iterations. I simulated two levels of set-associative private cache using a LRU replacement policy.

Table 5.1: Cache configurations of current-generation server-class processors based on Arm architecture. Level 2 is shared on A64FX, but private on TX2; TX2 has a shared cache at Level 3.

Processor	Level 1			Level 2		
	Total size	Line size	Set size	Total size	Line size	Set size
A64FX	64 KB	256 B	4	8 MB	256 B	16
ThunderX2	32 KB	64 B	8	256 KB	64 B	8

Table 5.2: Percentage differences between data from simulation and equivalent statistics obtained from querying hardware counters on real processors. The simulated results are within 10% of the data collected from hardware.

Application	128 bits						512 bits					
	Level 1			Level 2			Level 1			Level 2		
	Accesses	Misses	Miss %	Accesses	Misses	Miss %	Accesses	Misses	Miss %	Accesses	Misses	Miss %
STREAM	−6.9%	+4.7%	+13.5%	+8.0%	+10.3%	+2.1%	−4.5%	−2.4%	+1.9%	−2.3%	−0.1%	+2.1%
CloverLeaf	−6.6%	−8.9%	−2.2%	+7.0%	−9.7%	−13.9%	+2.6%	−8.0%	−10.8%	−8.0%	−9.7%	−1.6%
MegaSweep	+3.1%	+6.1%	+3.0%	−8.4%	−13.6%	−4.8%	+5.5%	−2.3%	−8.3%	−6.6%	−4.7%	+1.7%
MiniFMM	−7.0%	−3.4%	+3.4%	−5.7%	−7.4%	−1.5%	+4.8%	−2.9%	−8.1%	−6.4%	−3.3%	+2.9%
Average	5.9%	5.7%	5.5%	7.2%	10.2%	5.5%	4.3%	3.9%	7.2%	5.8%	4.4%	2.1%

This configuration is consistent with the design used in the ThunderX2 and similar to the A64FX, with the exception that in the latter, the second-level cache is shared between groups of cores (CMGs).

When scaling down from using all the cores available to just one, both on the TX2 and the A64FX, the performance characteristics of the first two levels of cache remained the same, bar a scaling factor applied to the absolute numbers. At scale, when more than one core is used, the applications distribute the computation between threads and processes. Each thread then executes a smaller fraction of the total amount of computation, but the operations it performs stay the same. Since I am only studying the effects on private caches, interactions *between* threads would also not affect the performance characteristics.

I performed several experiments on each mini-app to understand its cache behaviour. First, looking at 512-bit SVE, the vector width used in the A64FX, the first hardware implementation of SVE, I varied the associativity and the line size of the cache. This experiment aimed to show how these parameters help or hinder performance. Then, using the parameters of the caches used in the TX2 and the A64FX, in turn, I investigated how changing the SVE length from 128 to 2048 bits affects the caches. To describe the effects on the caches I looked at hit/miss ratios and at the length of the intervals between data being loaded in the cache and the same data being evicted. Finally, I investigated how SVE non-contiguous memory operations interact with these two cache configurations.

When counting caches access and misses, I considered the sizes of the memory requests used. Contiguous requests were counted as a single access, unless they spanned more than one cache line, in which case the number of cache lines touched was the number of accesses recorded. Non-contiguous requests were split into sequences of contiguous access, and each such contiguous part was treated as above.

5.3 Results

5.3.1 Cache Parameters

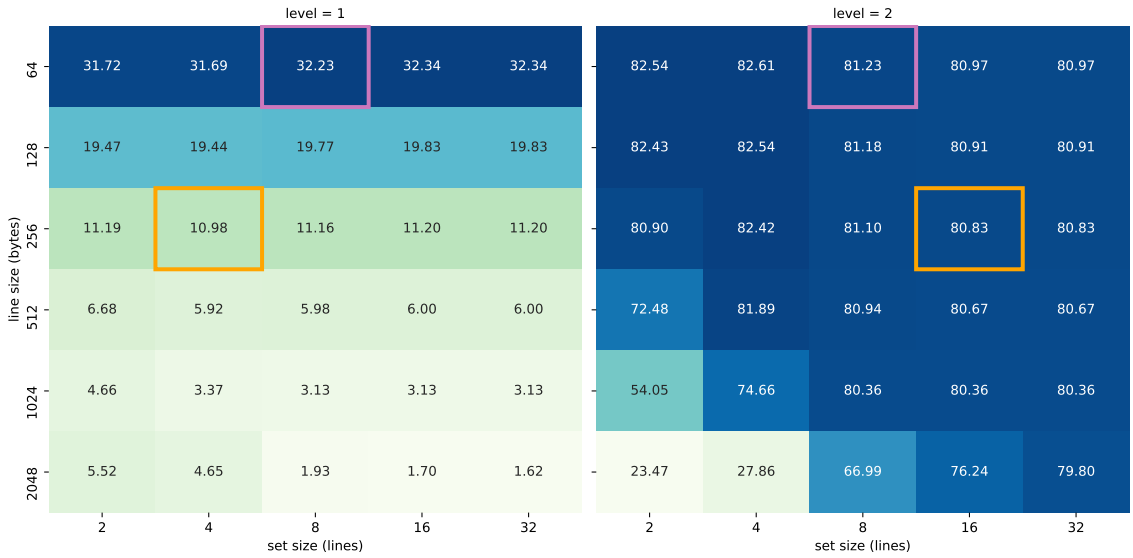


Figure 5.1: Cache misses, as a percentage of total cache accesses, for **CloverLeaf** in different cache configurations, at the two levels of cache. The A64FX and TX2 configurations are highlighted in orange and pink, respectively.

The choice of cache parameters had a significant impact on hit/miss ratios across all the applications. This was particularly evident for the first level of cache (L1), where there was a direct correlation between larger cache lines and fewer misses. The applications' memory accesses are arranged in such a way to do as much contiguous access as possible, so a single load at larger cache line sizes will service more subsequent requests for data.

When varying the set size of the cache, a similar trend was noticed, where bigger sizes corresponded to fewer misses, but the effects were less pronounced compared to changing line size. This parameter had a bigger effect when memory access were *less* structured, so MiniFMM and MegaSweep showed more benefit from higher associativity that CloverLeaf did, and in both cases the difference was higher at higher line sizes. Compared to the smallest set size tested, 2 lines per set, and with all the other parameters kept constant,

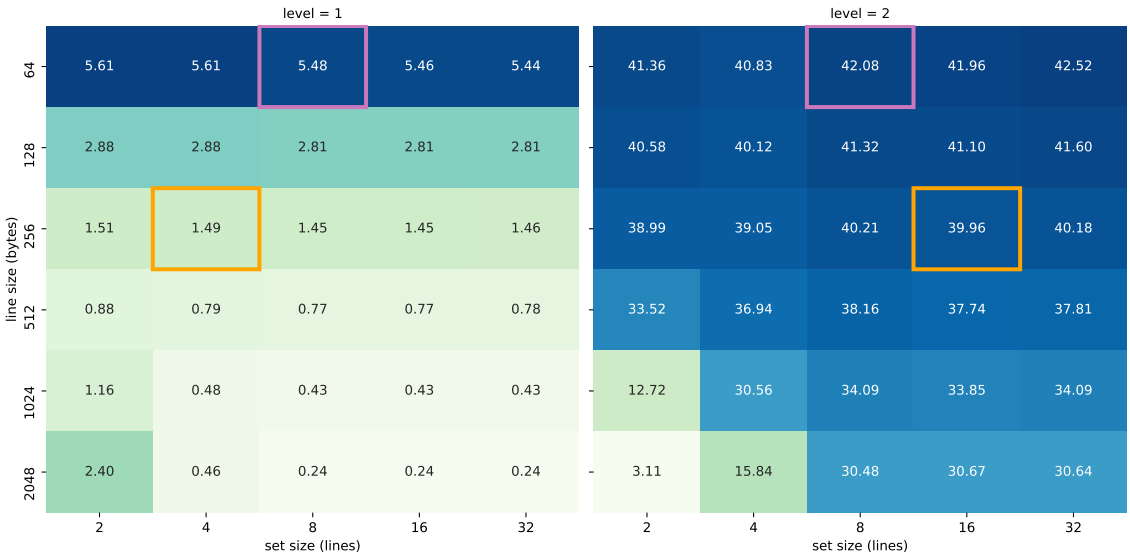


Figure 5.2: Cache misses, as a percentage of total cache accesses, for **MegaSweep** in different cache configurations, at the two levels of cache.

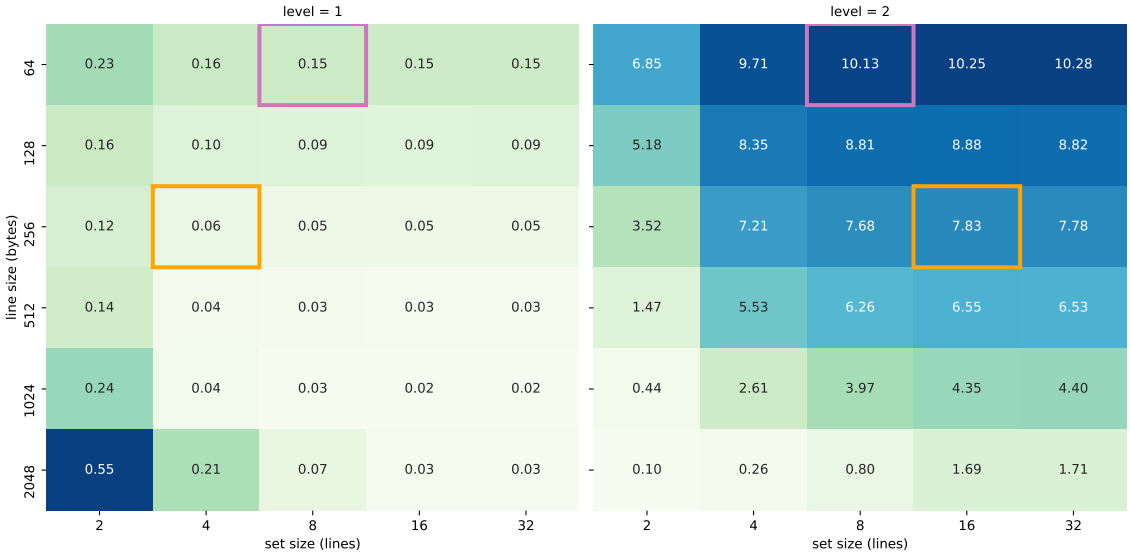


Figure 5.3: Cache misses, as a percentage of total cache accesses, for **MiniFMM** in different cache configurations, at the two levels of cache.

the biggest improvement for CloverLeaf from using bigger sets was 3.4× fewer misses, whereas for MiniFMM it was 18.3×, both at 2048-byte lines.

An interaction was observed between the line size and the set size when using longer lines. At low associativity, e. g. 2- or 4-way, the number of misses was an order of magnitude higher than at high associativity (32-way). For line sizes of 512 bytes or lower, using 2- or 4-way associativity did not result in a steep increase in misses, but this changed above 512 bytes, where the difference was up to 5× between 2-way and 4-way associativity and up to 10× between 2- and 8-way associativity. Increasing the set size above 8 still resulted in fewer misses, but the largest factor observed was 2.5×, for MiniFMM, and it did not change the miss rate for MegaSweep.

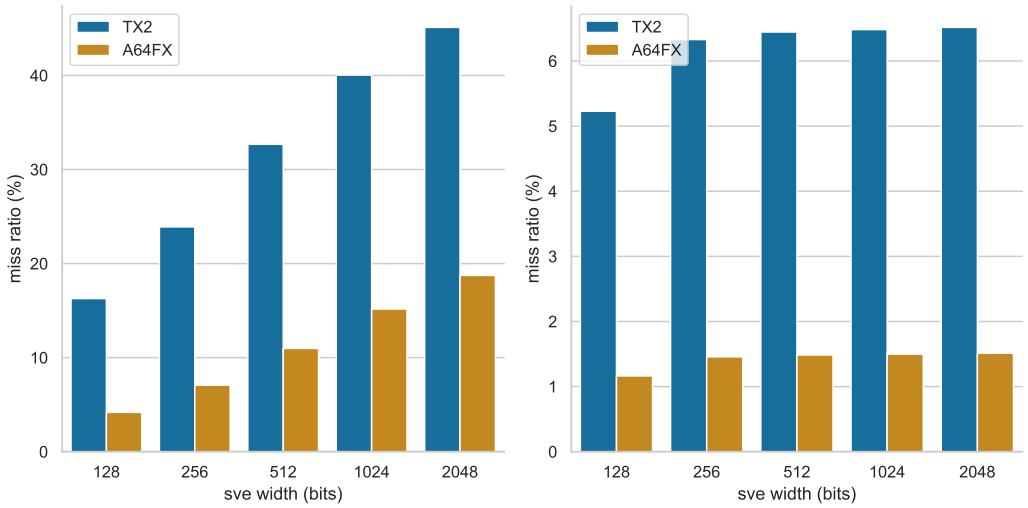
At the second level of cache (L2), the relation between higher line sizes and fewer misses persisted. The set size had little impact when the line size was below 512 bytes, but the difference between the best and the worst setting at 2048 bytes was more than 10× for MiniFMM and MegaSweep and 3.4× for CloverLeaf. However, unlike in the case of L1, higher set sizes were not always better: at high line sizes, *smaller* set sizes produced fewer misses. The best configuration, for all the applications benchmarked, was 2048-byte lines (the maximum tested) with 2-way associativity (the *minimum* tested).

The previous result is particularly significant for wide vector processors. As the cache line size is increased, there will be fewer cache lines — and sets — available if the total cache size is kept constant. For both MiniFMM and MegaSweep, and to some extent for CloverLeaf, having *more* sets available proved more useful than having *larger* sets. All of these applications work with several data structures at once, so more separate sets, i. e. lower associativity, enabled parts of more data structures to be held in cache at once. At higher associativity, when there were fewer sets overall, more aliasing led to one structure evicting more data from others, thus increasing the number of misses.

Figures 5.1, 5.2, and 5.3 show the miss ratios across a range of line and set sizes for CloverLeaf, MegaSweep, and MiniFMM, respectively. Both levels are of cache are shown independently in these figures. In these heatmaps, line

sizes are given in bytes, set sizes represent the number of ways of associativity, and darker colours represent more misses.

5.3.2 SVE Width



(a) **CloverLeaf**. (b) **MegaSweep**.

Figure 5.4: Cache miss rates for at different SVE lengths, for the cache configurations in the Marvell ThunderX2 (**TX2**) and the Fujitsu A64FX (**A64FX**).

Changing the vector length affects cache accesses in different ways based on whether the instructions being run are vector instructions or not. For the vectorised parts of the application, each (vector) memory access can carry more data at higher lengths, so fewer total access are needed to transfer the same amount of data. For scalar parts of the application, the vector length does not change memory accesses directly, but there are cases where vectorisation indirectly impacts the number of scalar accesses performed, e. g. nested loops where a fixed number of scalar access are executed per loop iteration and the number of iterations depends on vectorisation.

Most real applications run a combination of vector and scalar code, and their performance profile will change with vector length, depending on the proportion of the code that is vectorised. Figure 5.4a shows how cache misses scaled with the selected SVE length when using the L1 cache configuration

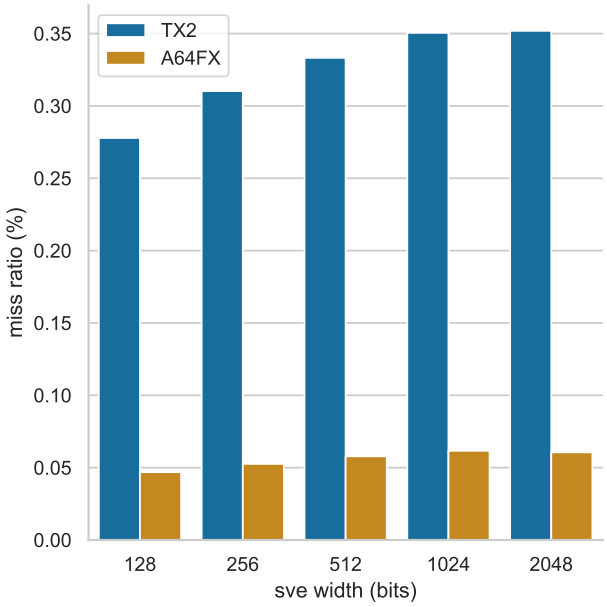


Figure 5.5: Cache miss rates for at different SVE lengths, for the cache configurations in the Marvell ThunderX2 (**TX2**) and the Fujitsu A64FX (**A64FX**) on the **MiniFMM** benchmark.

of the Marvell ThunderX2 and the Fujitsu A64FX when running CloverLeaf. As the vector width was increased, the overall number of *accesses* decreased, because each operation touched more data. The number of *misses*, however, depends on how many times cache lines have to be (re)loaded, which does not directly correlate to vector width, but rather with the size of the lines. Therefore, the miss ratio increases as the vector width is scaled up.

In the case of MegaSweep, shown in Figure 5.4b, vector operations only form a small part of the total number of memory operations, so the effect of scaling the vector width was much smaller. A similar effect was observed for MiniFMM, but here the miss rates were lower overall, due to the smaller working set size. The results for MiniFMM are shown in Figure 5.5.

In both of those cases, the miss rate recorded was lower for the configuration corresponding to the A64FX than the one for TX2. The main contributing factor was the line size, which is 4× as large in the former.

5.3.3 Lifetimes

Cache misses show an aggregated view of cache performance, but not all cache misses have the same impact on real-world performance. Since caches attempt to exploit temporal locality in the data, the less often data is reloaded into cache, the more time it spends being available in faster memory, and so the more "useful" the cache is. To capture the behaviour of the A64FX and TX2 caches from the point of view of how long data spends in cache, I measured the *lifetime* of each cache entry, i. e. the total number of memory accesses to any address between the moment the data was loaded and when it was evicted. This metric is sometimes called the *evict distance* of a cache line and it is an empirical variant of the *reuse distance* or *stack distance* [13].

Figures 5.6 and 5.7 show normalised histograms of cache lifetimes at different SVE widths for CloverLeaf and MegaSweep, respectively. For both applications, the higher line size and overall cache size of A64FX cache allowed data to be held, on average, for twice as long as the TX2 cache did, which in the histograms above is represented by wider curves, with the peaks shifted to the right; the mean of the distributions is also given. This effect was visible at all the tested SVE vector widths, but was more pronounced at higher widths, where the TX2 curves are thinner and packed toward the origin of the time axis.

For CloverLeaf, the distribution of lifetimes showed two distinct peaks on the TX2 configuration. CloverLeaf implements a multi-dimensional stencil algorithm, and so the rightmost peak corresponds to the higher cache-friendliness of the inner dimensions of the data structure. At higher SVE widths, each vector memory access touches more data, so data was evicted from cache sooner, shifting the curve to the left. These effects were not observed in the A64FX configuration, where the higher line size and total size allowed more of the outer dimensions to persist in cache, leading to more homogeneous cache usage overall.

In the case of MegaSweep, the curve was narrower on the TX2 configuration. There is a second peak here as well, but it is less distinct than in the case of CloverLeaf and it is located to the *left* of the main peak. On real

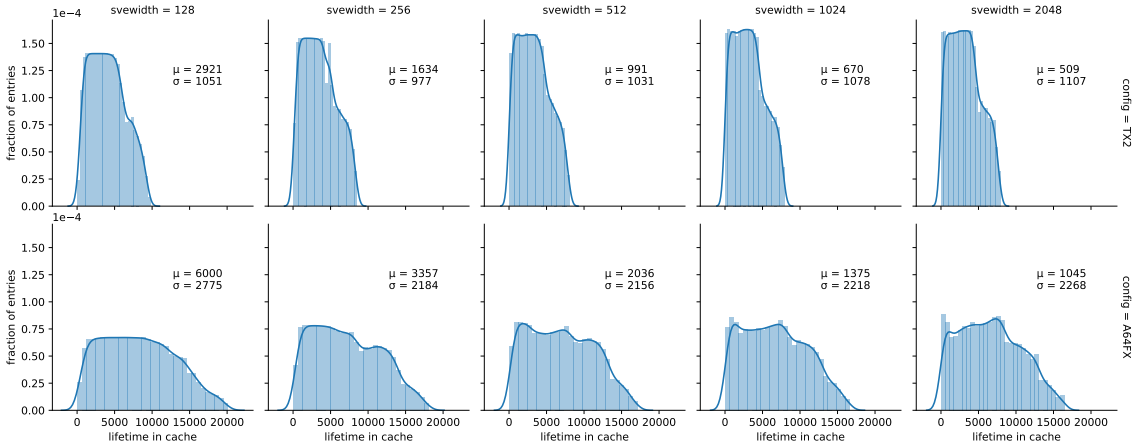


Figure 5.6: Level 1 cache lifetimes for **CloverLeaf** at different SVE lengths under the configurations of the A64FX and the TX2. A higher mean ($\boldsymbol{\mu}$) shows more time spent in cache on average; $\boldsymbol{\sigma}$ is the standard deviation.

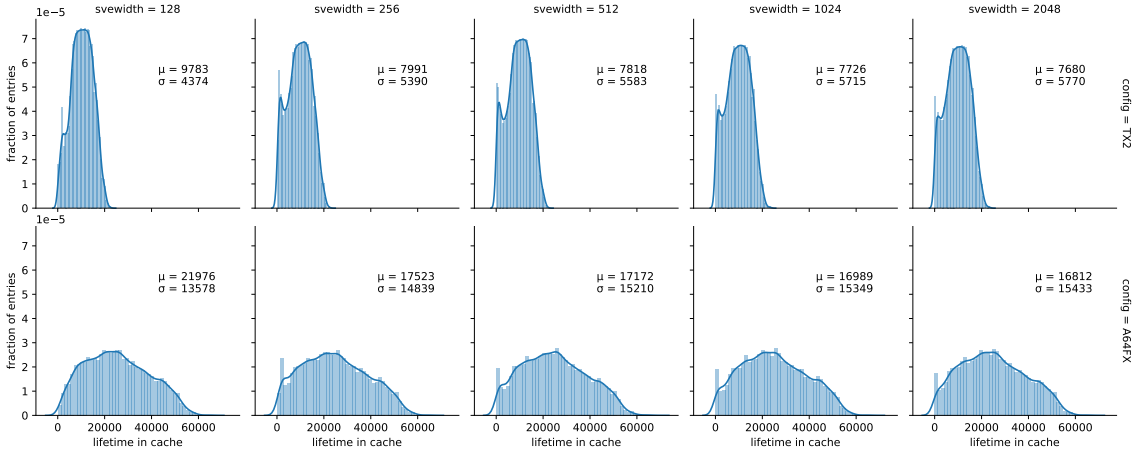


Figure 5.7: Level 1 cache lifetimes for **MegaSweep** at different SVE lengths under the A64FX and TX2 configurations.

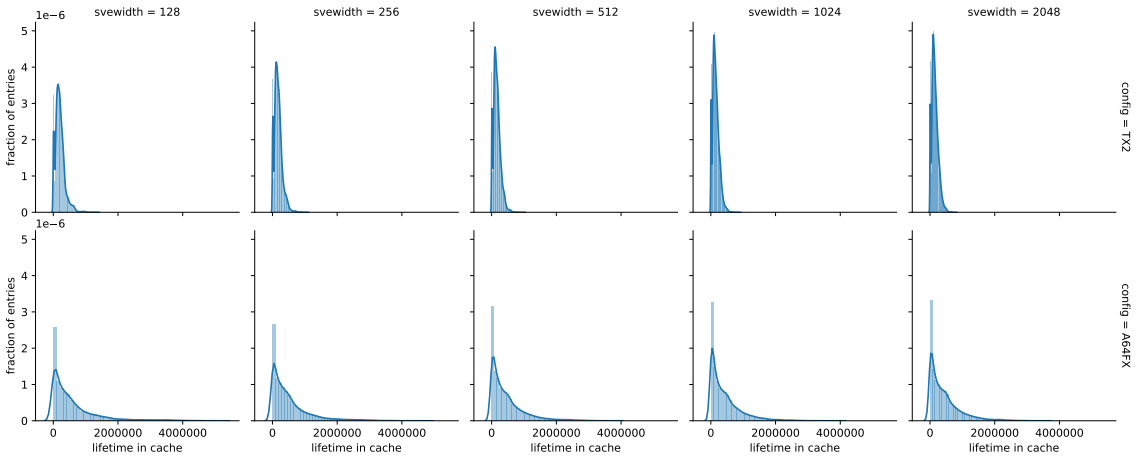


Figure 5.8: Level 1 cache lifetimes for **MiniFMM** at different SVE lengths under the A64FX and TX2 configurations.

ThunderX2 hardware, this can be seen in the poor utilisation of cache when running MegaSweep [23], which is partly caused by an interaction between the cache parameters — its total size, line size, and associativity — and the memory access pattern of the application. On the A64FX configuration, the effect was not observed, even though this configuration has smaller associativity, thus further reinforcing the lack of a correlation between set size and cache performance at small and medium line sizes.

The MiniFMM profile shares some similarities with those for CloverLeaf and MegaSweep, showing the same change to a wider and shorter curve when going from the TX2 configuration to the A64FX configuration. Unlike the other two, MiniFMM did not show more than one identifiable peak. The access patterns of this application are unpredictable, with significant use of non-contiguous access where available, so the main factor contributing to better performance in the A64FX configurations was the higher cache size. Results for this application are shown in Figure 5.8.

5.3.4 Non-Contiguous Accesses

SVE includes non-contiguous memory access operations to perform gather reads and scatter writes. These can interact with cache in different ways from contiguous accesses, because the amount of data accessed does not

always directly correlate with the number of cache lines used. In a hardware implementation, the cost of performing gather or scatter operations is likely to scale with the number of cache lines touched, because these have to be present in cache regardless of how many elements from each are being used.

Out of the three applications studied in this chapter, CloverLeaf and MiniFMM utilise non-contiguous memory accesses. Figure 5.9 shows a distribution of the number of cache lines needed at each SVE width to fulfil the non-contiguous memory accesses in CloverLeaf for the cache configurations of the Marvell ThunderX2 and Fujitsu A64FX. For each SVE width, a thicker line shows that a higher proportion of the memory accesses needed to access the corresponding number of cache lines to collect all the data for the scatter or gather operation.

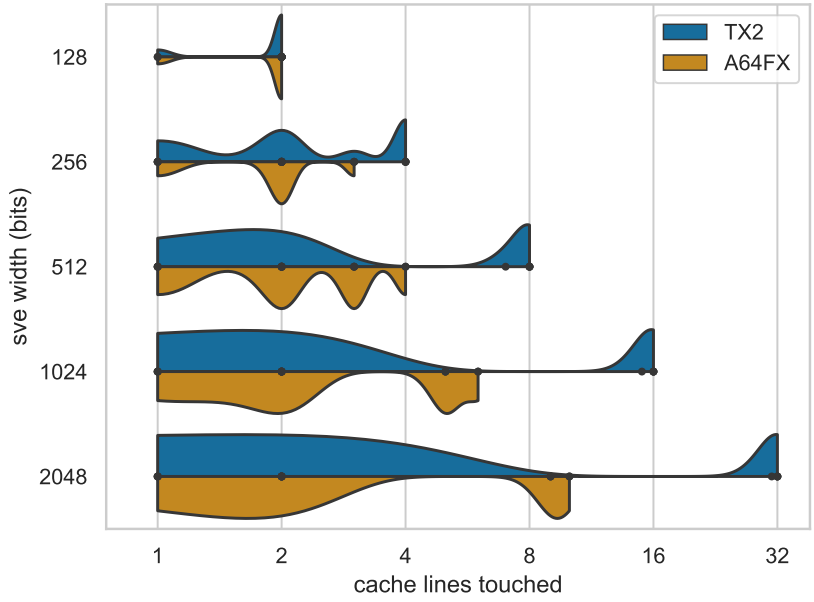


Figure 5.9: Distribution of the numbers of cache lines touched by non-contiguous SVE memory accesses for **CloverLeaf** on the A64FX and TX2 cache configuration. Thicker bars represent more memory accesses.

At 128-bit SVE, the two configurations showed a similar distribution of cache lines touched. Above this width, however, the larger cache lines of the A64FX contributed to a significant reduction in the total number of lines touched: at 1024-bit SVE, for example, most accesses were serviced using 5

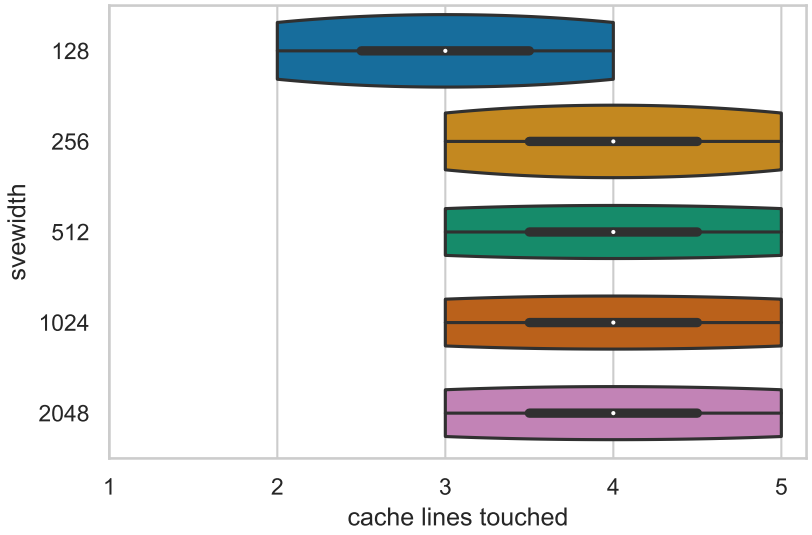


Figure 5.10: Distribution of the numbers of cache lines touched by non-contiguous SVE memory accesses for **MiniFMM** on the TX2 cache configuration. On the A64FX configuration, all requests were services by 2 cache lines.

or 6 cache lines, whereas the TX2 needed 16. When moving up to 2048-bit SVE, the maximum width allowed, the peak is centred around 9 cache lines on A64FX, but on TX2 it is 32.

There were also a significant number of accesses where the cache line size did not impact the total number of lines needed. These accessed pairs of values located very far apart, with more than 15 KB between them, in a single SVE instruction. Because this distance is orders of magnitude of the size of a single cache lines, both caches needed two separate lines to service these requests, which contributes to the local peaks around 2 in the figure.

In the case of MiniFMM, the distribution of widths of the non-contiguous accesses was narrower. Regardless of SVE width, the largest address range spanned by a single gather or scatter instruction, i. e. the distance between the first and the last byte touched, was 320 bytes. Under the A64FX cache configuration, this allowed all non-contiguous requests to be serviced by at most 2 cache lines. On TX2, where cache lines are 64 bytes long, most accesses touched between 3 and 5 cache lines, and all accesses touched at

least 2 cache lines at vector widths above 256 bits. These results are shown in Figure 5.10.

5.4 Implications for Vector Processors

The results presented in Section 5.3 show a clear relation between the choice of cache parameters and the behaviour of HPC applications running on the system, as measured by cache hit and miss metrics. Better results measured this way should correlate with faster performance on real hardware. As caches — and the arithmetic units utilising them — become bigger, the optimal choices are different, and "common" choices may not produce the best results [136]. The introduction of SVE gives computer architects unique freedom in terms of vector processing power, but for a processor to be efficient as a whole, its memory subsystem needs to be designed hand-in-hand with the core if it is to keep up to its demands. I have shown how vector length influences optimal choices for core-private caches, but it is likely that other design decisions, such as the number of vector units employed or the prefetching algorithm, will have impacts of their own. All of these aspects needs to be considered together when designing high-performance hardware.

Beyond the field of HPC, vector units have much wider applications, e. g. media or energy-efficient computing. In other fields, not only constraints will be different, but it is likely that the optimal design decision from a performance point of view will differ, too. Both these factors highlight the need for long-term co-design, a need for processors to be designed hand-in-hand with the applications they will run, and Arm-based architectures offer a unique opportunity to this end.

5.5 Towards Performance-Portable Application Design

The results presented in this chapter suggest that applications which are aware of the hardware configurations they run on can make different perform-

ance decisions based on the capabilities and resources available. However, placing this burden on application developers is infeasible and would lead to a large amount of duplicated work. Instead, such decision could be made by the programming frameworks themselves, the very tools used to develop these applications.

Given access to accurate simulation tools, a future application can make informed decisions about how best to utilise a wide range of hardware, e. g. through careful laying out of data. These decisions are particularly important — and hard to make — on systems with configurable caches, e. g. the sector cache on the A64FX. The sector cache allows application programmers to reserve a portion of the processor's cache for specified data structures by lowering the associativity of the remaining, general-purpose section [38]. This represents a hardware-backed way to identify particular data objects that should be kept in cache, which can greatly benefit applications that perform a mix of streaming memory operations and cache-resident operations [2], such as CloverLeaf.

There already exist high-level programming frameworks that support making automatic changes to the data layout based on the architecture targetted [32], and by integrating further information about the hardware configuration, the potential for performance gains increases. One way to achieve this is to run micro-benchmarks at compile-time, or at the first run of the application, and tune performance based on the results. This approach is a generalisation to any given hardware platform of auto-tuning in OpenCL applications targeting GPUs [110].

In Chapter 7, I explore the efficacy of modern parallel programming frameworks which have the potential to accurately target architectures and adjust application behaviour with minimal requirements from the programmers' perspective. I evaluate a number frameworks that have recently started being adopted by comparing them to established alternatives, on a wide range of hardware that accurately represents today's HPC market.

5.6 Reproducibility

The data used in this chapter can be found online[1]. The raw data is given in CSV format, and scripts are provided to reproduce the visualisations.

5.7 Conclusion

In this chapter, I have investigated the effects of using wide vector instructions on processor caches, focusing on Arm SVE. Using examples from several classes of common HPC applications, I have shown that there is a direct correlation between the (hardware) parameters of the processor's caches and its efficient utilisation. Changing the line size had the biggest impact on performance. Several good and less optimal choices for set size were identified when paired with longer cache lines, but the difference was small when using shorter cache lines. For all applications tests, increasing the size of the cache outweighed a reduction in associativity. This correlation is stronger for some applications — those which implement algorithms that are naturally more cache-friendly — but it affects *all* classes.

I have also presented how changing the (hardware) vector widths interacts with the cache hierarchy. Even though wider vectors can mean more data is processed in parallel, they also change the profile of effects on the cache. When non-contiguous memory operations are used, because of the large variety of access patterns these can cover, the effects of changing cache parameters can be hard to identify *a priori*. I have found that different cache parameters are optimal at different vector lengths, and since applications also tend to prefer some vector widths [107], it is important that decisions in both hardware and software are made using a co-design approach.

[1] `https://github.com/UoB-HPC/cache-effects-reproducibility/tree/eahpc-2020`

CHAPTER 6

Next-Generation Vector Processors

Content from this chapter appears in the following publication:

- Andrei Poenaru, Tom Deakin, Simon McIntosh-Smith, Simon D. Hammond and Andrew J. Younge. 'An Evaluation of the Fujitsu A64FX for HPC Applications'. In: Cray User Group. May 2021. In Press

Arm-based processors have been investigated for use in HPC systems since the early 2010s [112]. Initially based on mobile designs, dedicated high-performance cores have been used in recent years to provide performance similar to high-end Intel and AMD x86-based processors. Likewise, the tools ecosystem is mature, stable, and production-ready, making Arm a first-class citizen in HPC [115]. Due to their flexibility, an increasing number of vendors are integrating Arm-based cores into their upcoming exascale-era products [97].

Since 2017, a number of systems have deployed Arm in production HPC using the Marvell ThunderX2 (TX2), one of the first Arm-based processors designed specifically for HPC [87, 100]. Studies on these systems have helped identify the types of workloads that are suited for these processors, as well as what their weakness are: TX2 offered a large number of cores and high memory bandwidth, but its short 128-bit-wide vectors make it less suited for compute-intensive applications.

In 2020, the Fugaku system in Japan deployed a new Arm-based design: the A64FX built by Fujitsu [119]. The A64FX improved on both aspects compared to the TX2, implementing for the first time in a CPU HBM2 memory that offers a peak of 1 TB/s of bandwidth and 512-bit vectors based on the Arm Scalable Vector Extension (SVE) [128]. This system was ranked #1 in the TOP500 list in June 2020, and since then several other HPC centres have been adopting the A64FX processors for their own deployments.

In this chapter, I evaluate the performance of the Fujitsu A64FX processor on a range of scientific mini-apps and full applications. I compare the A64FX with the other mainstream HPC processors at the time of writing, and I devote special attention to its other Arm-based competitors.

6.1 Background

The A64FX is the new HPC-first processor designed for the Japanese supercomputer Fugaku. Its core design is custom-made by Fujitsu based on the ARMv8.2 architecture with extensions. The chips contain 48 cores running at up to 2.2 GHz, without simultaneous multithreading (SMT). They are used in single-socket configurations, connected to either TofuD or 100 Gbps InfiniBand networking [120].

An A64FX chip houses four stacks of HBM2 memory. It is the first CPU to utilise HBM2 memory, which had only been used on GPUs before. Each stack is directly attached to a subset of 12 cores, known as a *Core–Memory Group* (CMG). Each core has a private Level 1 cache, but Level 2 (the Last-Level Cache) is shared between cores in a CMG. To the operating system, each CMG appears as a separate NUMA node, and in order to achieve high performance the latency between these nodes needs to be carefully considered. Figure 6.1 shows a block diagram of an A64FX chip with four CMGs.

The A64FX is also the first hardware implementation of SVE. SVE is a VLA instruction architecture, allowing each implementation to choose its desired vector length, while ensuring that the same code remains compatible with all implementations. In the A64FX, the native vector width is 512 bits,

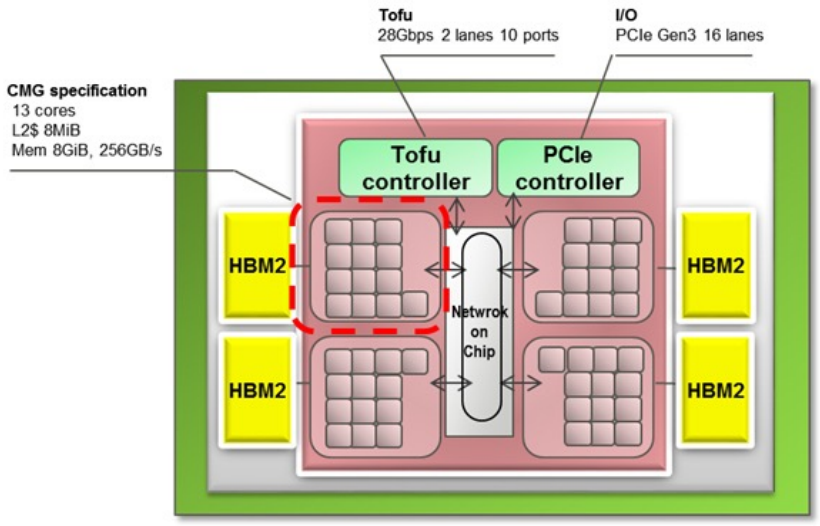


Figure 6.1: A64FX block diagram. Source: Fujitsu [70].

chosen after experiments in simulation have suggested it is efficient for a range of applications important for the Fugaku supercomputer [63]. As a successor to the NEON ASIMD vector instruction set used in previous Arm-based processors, it offers a wider range of instructions, including gather loads and scatter stores, and per-lane predication for all operations. These features are important for the tuning of low-level optimised math libraries [2].

6.2 Performance Evaluation Methodology

To evaluate the performance of the A64FX for HPC workloads, I used mini-apps representative of common classes of HPC applications, as well as full-scale codes that are widely used in supercomputing centres around the world. I chose these benchmarks because their performance profiles closely resemble real workloads, and hence should provide a good indication of the real-world performance achievable by these processors. I split them according to the type of resource they depend on most heavily: memory bandwidth or raw compute performance.

Using these benchmarks, I compared the performance achieved by the A64FX with that of other common HPC processors. The platforms I com-

pare against are the Arm-based Marvell ThunderX2, AWS Graviton 2 (in an `M6g.metal` EC2 instance), and Ampere Altra, and the x86-based Intel Cascade Lake (CLX) and AMD EPYC Rome. At the time of writing, these represent the top offerings from the most widely utilised vendors in HPC. The specifications of these processors are given in Table 6.1. Note that the A64FX and Graviton 2 can only be used in single-socket configurations, but the other processors were used in dual-socket nodes.

On all platforms, I used the latest versions of the common HPC compilers: GCC 11.1 supports all the platforms in this study, Arm Compiler for Linux (ACfL) 21.0 supports all the Arm-based targets, Intel Compiler 19.1 (part of the 2020.4 package) supports all the x86-based processors, Cray Compilation Environment (CCE) 11.0 supports all the platforms except the Graviton 2 and the Altra, and Fujitsu Compiler 4.3 supports the A64FX only. There were two exceptions to the above:

- The latest version of CCE available for the A64FX is a pre-release version based on 10.0. This uses the legacy Cray-proprietary frontend instead of the Clang-based frontend used in CCE 11.0;
- There was a regression in the performance of the TeaLeaf benchmark with CCE 11.0, so 10.0 was used to obtain the fastest results for this application.

6.2.1 Bandwidth-Bound Benchmarks

To evaluate the best-case achievable memory bandwidth, I used **BabelStream** [27], a C++ implementation of the *de facto* memory bandwidth benchmark, STREAM [83]. BabelStream contains implementations in many programming models, and for this work I used the baseline OpenMP version.

I used the mini-apps **TeaLeaf** [90] and **CloverLeaf** [76] as representative bandwidth-bound workloads. These are both written in Fortran, using hybrid MPI and OpenMP, and they solve equations for heat diffusion and hydrodynamics, respectively. I have studied these extensively in the past and found that their performance correlates well with STREAM performance. Of

Table 6.1: Hardware specifications of the processors benchmarked.

CPU	Cores	Clock Speed (GHz)		Compute Peak (DP TFLOP/s)	Bandwidth Peak (GB/s)
		Base	Boost		
AMD Rome 7742	2×64	2.25	3.4	6.9	410
Ampere Altra Q80-30	2×80	3.0	—	3.8	410
AWS Graviton 2 M6g.metal	64	2.5	—	1.3	205
Fujitsu A64FX	48	1.8	—	2.8	1,024
Intel Cascade Lake 6230	2×20	2.1	3.9	2.0	375
Marvell ThunderX2	2×32	2.2	2.5	1.3	320

the two, CloverLeaf is slightly more computationally intensive, as it includes divisions and trigonometry functions.

Finally, I evaluated the performance of **OpenFOAM** [58], a well-known computational fluid dynamics (CFD) application and one of the top 10 most heavily used applications on ARCHER, the UK's national supercomputer. I used version 2006 of the code, the DrivAer open-source test-case [50], and the standard `simpleFoam` solver, applied for 50 time steps. This test case was developed at a hackathon on the Isambard system and later used for the performance results presented in Chapter 3. Because the time reported for the first step includes some initialisation overhead, I excluded it from the final benchmark times.

6.2.2 Compute-Bound Benchmarks

I used **miniBUDE** for a compute-bound mini-app. This is a molecular docking benchmark developed at the University of Bristol which has previously been shown to achieve close to 60% of peak arithmetic performance on contemporary HPC hardware [106]. The code is implemented in several programming models, of which I used the standard OpenMP implementation here. The performance reported for miniBUDE is in the number of poses computed per unit time.

Another benchmark studied is **SPARTA**, a Direct Simulation Monte Carlo (DSMC) mini-app from Sandia National Laboratories designed for large systems [39]. It is implemented in C++ and MPI, with optional support for threading through the Kokkos library [32]. As a Monte Carlo application, this code is challenging to vectorise and its memory access patterns are irregular.

I took **MiniFMM** [7] to represent applications that use task-based parallelism instead of traditional loop-based parallelism. For this benchmark, I used the provided input set based on a Plummer distribution and recorded the total time taken.

A good example of a widely used compute-bound application is **GROMACS**. I used GROMACS 2021.1 and two different benchmarks to evaluate the performance of the systems tested under different conditions:

- The integrated `nonbonded-benchmark`, which runs in flat OpenMP mode and is heavily compute bound. This does not require any input files and runs a Particle Mesh Ewald (PME) [34] simulation; the size parameter for this benchmark was set to 64;
- The `ion_channel_vsites` benchmark, which simulates a membrane protein system comprising around 145,000 atoms. It uses FFTs and represents a realistic use-case for GROMACS in modelling drug molecules. Compared to `nonbonded-benchmark`, PME calculations in `ion_channel_vsites` only take about ⅓ of the total time. I ran this benchmark for 5000 steps of 5 fs.

The 2021.1 release includes initial support for SVE through the GROMACS SIMD abstraction layer [99], although at the time of writing this can only be used with the GNU compiler.

6.3 Results and Performance Analysis

6.3.1 Benchmark Results

BabelStream I was able to achieve 824 GB/s in the Triad run on BabelStream on the A64FX, which represents more than 80% of the platform's peak memory bandwidth. This result is more than double that of the next best platform for memory bandwidth. High memory bandwidth is of course expected due to the use of HBM2 on A64FX compared to traditional DDR-DRAM used by the other platforms.

I achieved this result using the Fujitsu compiler, which utilises zero-fill (`zfill`) instructions to zero cache lines before writing to them. This prevents the hardware from first loading the data from memory, because it will be overwritten anyway; it essentially emulates streaming stores even though the architecture does not support it explicitly. The other compilers do not

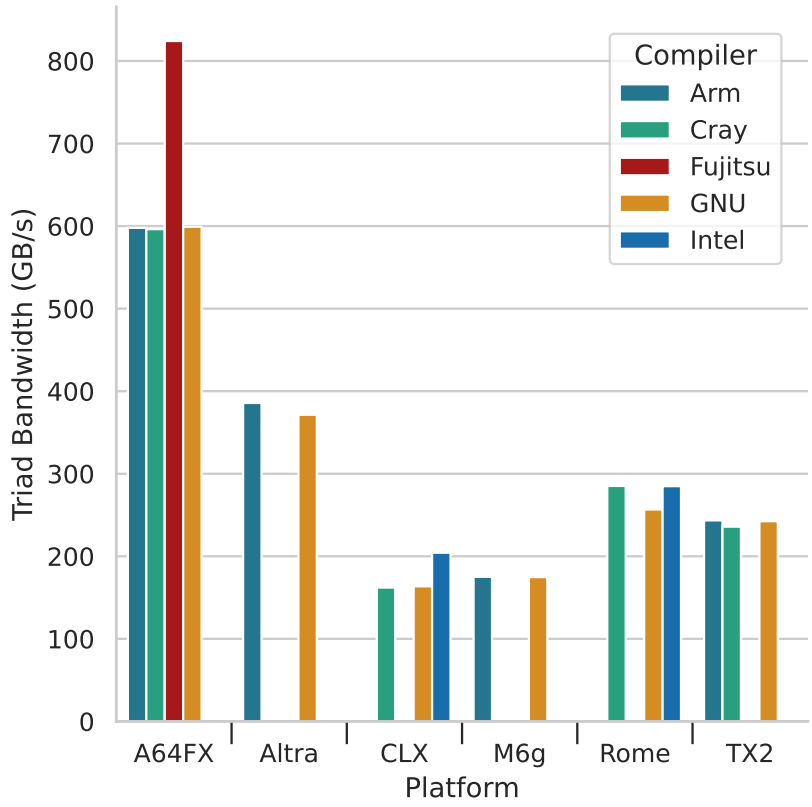


Figure 6.2: Achieved bandwidth in BabelStream Triad. Higher numbers show better results.

use this procedure, and so observed memory bandwidth there was lower at around 600 GB/s.

It was also important for this benchmark to set the `XOS_MMM_L_PAGING_POLICY` environment variable to `demand:demand:demand`. This controls how memory pages are allocated between the four NUMA domains in the A64FX, ensuring they are placed in the same CMG where they are needed, as opposed to that of the core that first started running the program.

The Ampere Altra obtained the second-highest result with its two sockets of 8-channel DDR4-3200. Even though the TX2 also has 8 channels of DDR4 and in dual-socket configuration, its slower DDR4-2400 memory put its result closer to that of a single-socket Graviton 2 with DDR-3200. The TX2 achieved a lower fraction of peak bandwidth in the BabelStream bench-

mark, and I observed a regression with CCE 11.0: reverting to version 10.0 produces a result higher by about 15%. The fastest results obtained on each platform are shown in Figure 6.2. Where a result for a compiler is not shown this is due to that compiler not supporting the platform.

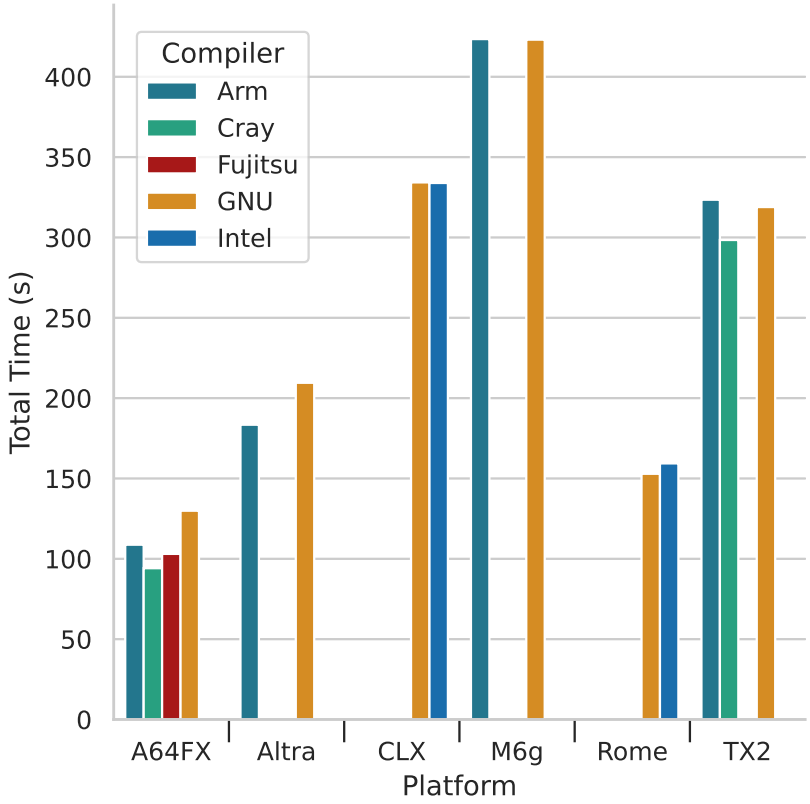


Figure 6.3: TeaLeaf `bm5` benchmark time. Lower numbers show better results.

TeaLeaf and CloverLeaf Due to their memory-bandwidth-bound nature, I expected the CloverLeaf and TeaLeaf results to follow similar distributions between platforms as I saw for BabelStream. I largely observed this behaviour, but there were some important differences.

These two applications can be run in hybrid MPI–OpenMP mode, and I tested all viable combinations. On most platforms, I have previously found that running in flat MPI mode, i. e. setting the number of OpenMP threads to 1 and filling all the cores with MPI ranks, generally provides the best

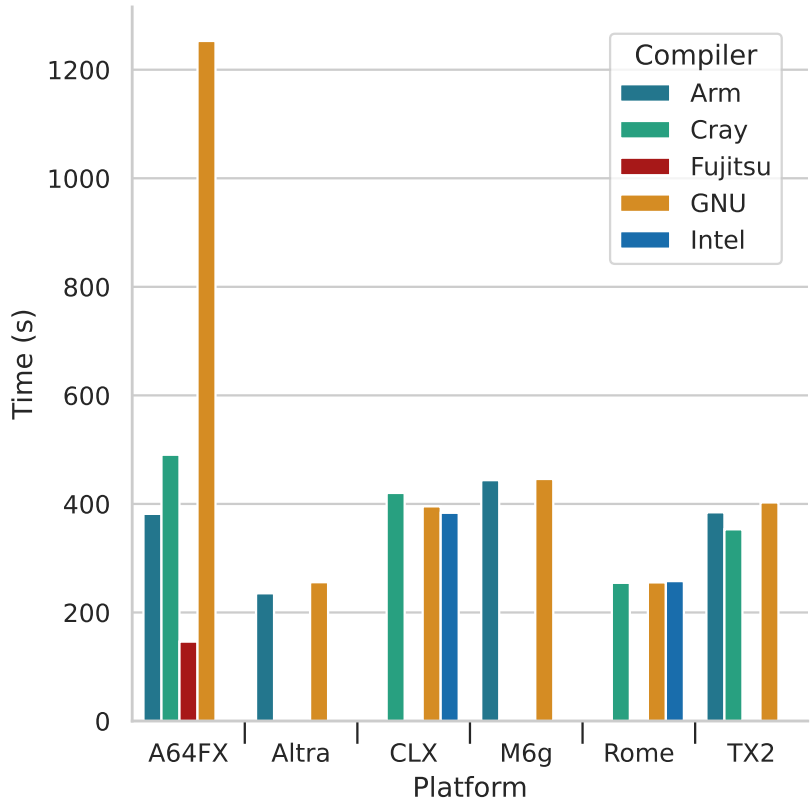


Figure 6.4: CloverLeaf `bm16` benchmark time. Lower numbers show better results.

performance in single-node configurations [29]. Where there was a difference between flat MPI, flat OpenMP, and hybrid MPI–OpenMP, it was below 10%. On the A64FX, however, I have found larger differences between these run configurations. This section discusses the fastest results obtained, regardless of the run configuration, but Section 6.3.2 goes into more details about the differences.

TeaLeaf contains relatively fewer arithmetic operations compared to CloverLeaf, so memory bandwidth is even more important. Of the Arm-based processors, in descending order and starting with the fastest result, first was the A64FX, then the Altra at just under twice the run time, then TX2, closely followed by the Graviton 2. Where available, the Cray compiler produced

the fastest results. On A64FX, the Fujitsu compiler was a close second, and the Arm and GNU compilers performed similarly on all the platforms.

CloverLeaf includes division operations, which on the A64FX have high execution latency. To work around this, some compilers can replace division with an iterative reciprocal approximation, which is much faster at a slight cost of accuracy. The Arm, Cray, and Fujitsu compilers are all able to apply this optimisation — Cray and Fujitsu do it automatically when targeting the A64FX, and with Arm the user can specify the `-fiterative-reciprocal` flag. The GNU compiler does not apply this optimisation, which results in almost 10× slower performance compared to Fujitsu. Fujitsu is further able to optimise this benchmark by using software pipelining of instructions, a technique which carefully schedules operations such that the processor's out-of-order resources are utilised as efficiently as possible.

Due to all the optimisations it applied, the Fujitsu compiler on A64FX produced the fastest time in this benchmark. However, the Ampere Altra and AMD Rome benefited from their large number of cores and obtained results faster than when using the A64FX with other compilers. The Graviton 2 and the TX2 performed almost identically, suggesting that the newer out-of-order architecture in the Graviton 2 was able to make up for the slightly lower overall memory bandwidth.

The results obtained for CloverLeaf and TeaLeaf are shown in Figures 6.4 and 6.3, respectively. These figures show run time, so lower numbers correspond to better performance.

OpenFOAM When run on a single-node, OpenFOAM is generally bound by memory bandwidth and does not benefit greatly from vectorisation [87]. These two effects work for and against the A64FX, respectively: it should see good performance from the HBM2 memory, but the 512-bit SVE may not bring a significant improvement over NEON. The results showed that the fastest processor in this benchmark was the AMD Rome, closely followed by the Ampere Altra, suggesting that the large amount of total L2 cache — 1 MB/core in both the these processors — helped more than HBM2 did on A64FX. The Fujitsu compiler was again the fastest choice on the A64FX,

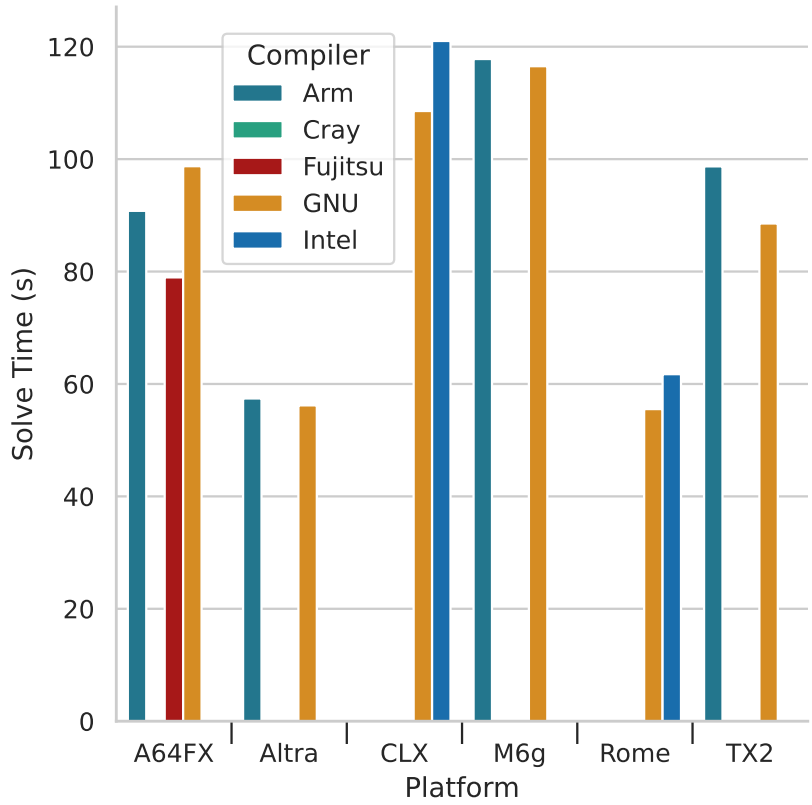


Figure 6.5: OpenFOAM DrivAer solve time after 50 time steps. The time taken for the first step is excluded. Lower numbers show better results.

but this time the differences to the other compilers were smaller; Arm and GNU produced similar results on A64FX and Graviton 2. Figure 6.5 shows the results on all platforms.

miniBUDE miniBUDE scales very well to many-core architectures — the full BUDE application is routinely run on GPUs. As expected, the results for this benchmark followed the peak compute performance of the processors: TX2 and Graviton 2 achieved similar results, Cascade Lake was more than twice as fast, and the Altra obtained the highest result of the Arm processors, only surpassed by the AMD Rome. On the A64FX, the Fujitsu compiler was able produce better optimised code compared to other compilers, reaching almost 3× the performance obtained with ACfL and GCC, and more than 1.5× the performance of the CCE-compiled binary. However, the perform-

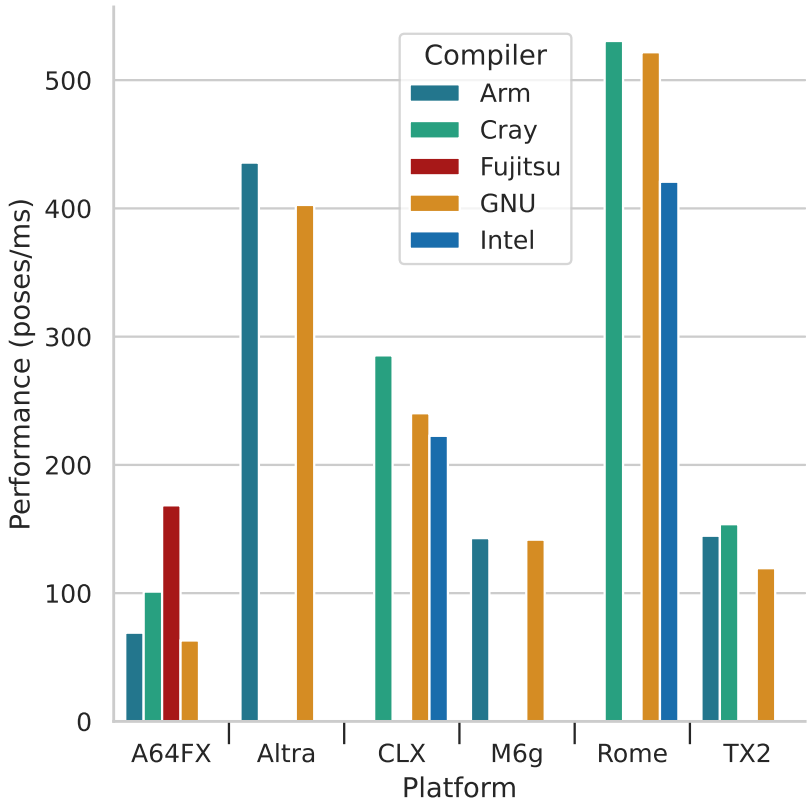


Figure 6.6: Achieved performance in miniBUDE. Higher numbers show better results.

ance achieved by the Altra was over twice that of the A64FX, despite their difference in peak performance being lower than a factor of 2, showing that its high core count can rival higher vector width as long as the application parallelises well. The results for miniBUDE are presented in Figure 6.6.

SPARTA The performance of SPARTA scaled very well with the number of cores available. There was virtually no vectorised code on any of the platforms, and the choice of compiler made little difference towards the final run time on this benchmark. The data access patterns of this application were not cache-friendly, with only 58.5% of the requests hitting L2 cache, so a lot of time was spent fetching data from main memory.

In general, GCC offered the highest performance on most platforms, being only slightly slower than the Intel compiler on Cascade Lake and Rome. The

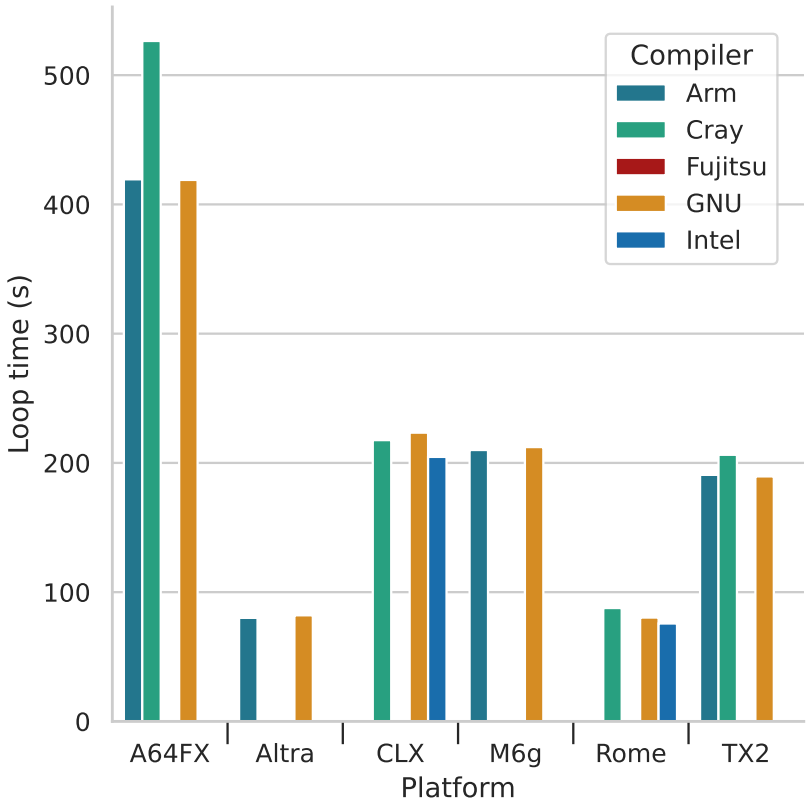


Figure 6.7: SPARTA benchmark time using the collisional flow input, 10M cells, and 5000 iterations. Lower numbers show better results.

TX2, Graviton 2, and Cascade Lake achieved similar results, despite the narrower vectors available on the Arm-based platforms. On the A64FX, the Fujitsu compiler failed to link the benchmark, in either Trad or Clang mode, and without its aggressive optimisations the platform's low out-of-order resources led to a slower benchmark time. Figure 6.7 shows the results on all platforms.

MiniFMM I found the vectorisation efficiency of the MiniFMM benchmark to be low on all the Arm-based platforms. With NEON, a lot of the code was not vectorised, and although SVE was able to address that to an extent, many operations were masked and utilised only a fraction of the available vector width [107]. In addition, it did not scale well to high core counts: beyond 60 cores, the run time stopped decreasing, and above 80 it started

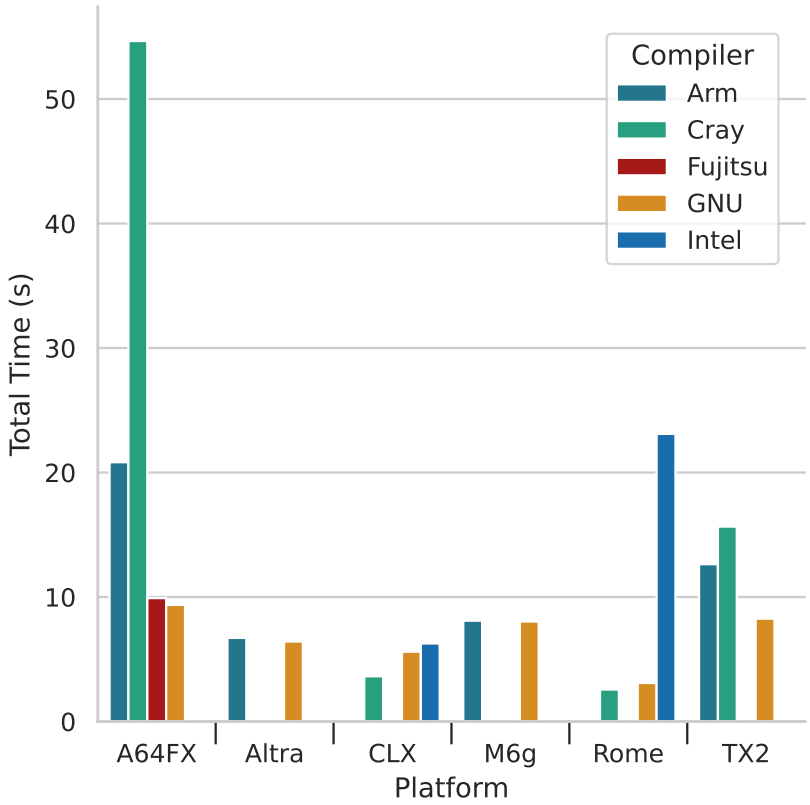
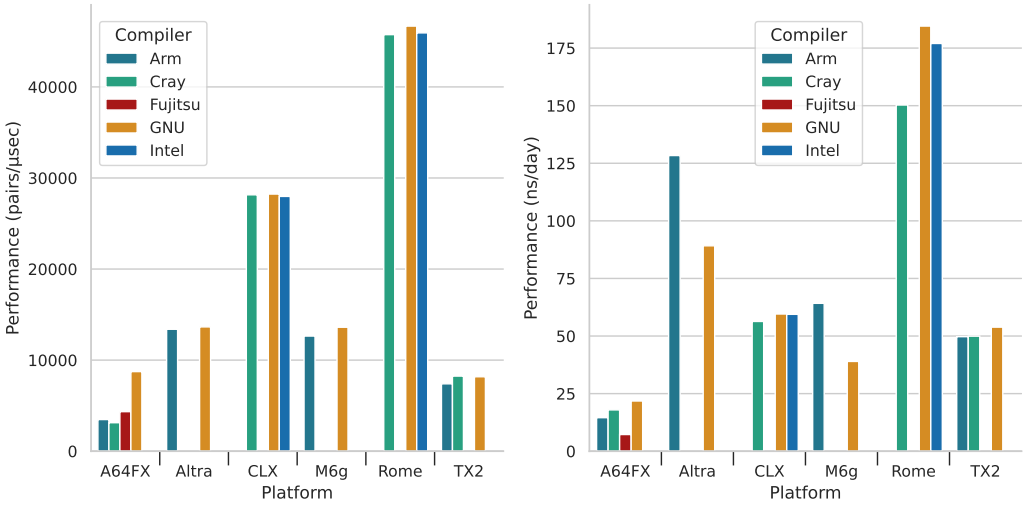


Figure 6.8: MiniFMM benchmark time using a Plummer and the OpenMP tasks implementation. Lower numbers show better results.

increasing. For the Altra, this meant that fewer than half of the available cores were utilised. On the x86-based platforms, there was more benefit from vectorisation, but the high core count of Rome again did not show a tangible benefit in this benchmark.

The best result was similar on all the Arm platforms, with a slight advantage to Altra due to its high clock speed. I found that the Cray and Arm compilers were less efficient at exploiting parallelism in this task-based benchmark compared to GCC, which was the best compiler choice even on the A64FX. The Intel compiler performed well on Cascade Lake, but it was significantly slower compared to Cray and GNU on Rome. The results are presented in Figure 6.8.



(a) `nonbonded-benchmark` (b) `ion_channel_vsites`

Figure 6.9: Achieved performance in two GROMACS benchmarks. The open-source FFTW library was used with GCC and Fujitsu, ArmPL was used with the ACfL, MKL with the Intel compiler, and Cray's optimised build of FFTW was used with CCE. Higher numbers show better results.

GROMACS With `nonbonded-benchmark`, there were significant performance differences between the x86 platforms, where AVX2 and AVX-512 could be used, compared to the Arm platforms. This workload is heavily compute-bound, so the wider vector length constituted a significant advantage. This benchmark cannot be used with MPI and the maximum number of OpenMP threads allowed in GROMACS is 64, which limited the performance achieved by the Altra and the Rome platforms, and resulted in similar performance on Altra and Graviton 2, since both use the same Neoverse N1 cores. Even though the early SVE implementation for A64FX — which was only usable with the GNU compiler — achieved almost twice the performance of the NEON implementation on the Fujitsu platform, it still only produced results similar to a ThunderX2 running NEON; with more optimised code, it should be possible for the A64FX to produce results several times faster than this. On the other platforms, there were virtually no differences between the compilers, because the performance-critical PME kernels in GROMACS are written in hand-tuned intrinsics.

However, the more realistic `ion_channel_vsites` test case revealed different behaviour. On the one hand, the change in performance profile to place more emphasis on the memory system brought the results of a 64-core TX2 node very close to that of a 40-core Cascade Lake node, despite the difference in native vector length between the two processors. On the other hand, the benefit from the early SVE implementation on the A64FX was lower and closed the performance gap to the other compilers, which still used the NEON implementation. With core usage no longer limited to 64, the Altra's performance increased relative to the other platforms with lower core counts. I observed that the optimised FFT implementations in the Arm Performance Libraries performed significantly better on the Neoverse N1, granting a 1.43× speed-up over FFTW; on the other platforms, FFTW built from source, Cray's Optimised FFTW and ArmPL (on TX2) performed similarly. For OpenMP parallelism, the best choice of number of threads was 2 or 4 on all the platforms, with enough MPI ranks run to fill all the available hardware threads, i. e. utilising SMT where available.

The results for the two GROMACS benchmarks on all the platforms are shown in Figure 6.9.

Application Performance Summary Finally, Figure 6.10 aggregates all the benchmark results into a single view. Here, the best result was kept for each platform, for each benchmark. The results are presented relative to the Cascade Lake baseline, so numbers above 1 show higher performance than Cascade Lake.

Overall, the A64FX achieved over 2.5× the performance of Cascade Lake on memory-bandwidth-bound benchmarks. On the other hand, it was weaker on compute-bound benchmarks, where the Ampere Altra and AMD Rome usually produced the fastest results.

6.3.2 Thread Placement on the A64FX

Due to the four NUMA node configuration, placement and binding of MPI ranks and OpenMP threads are particularly important on the A64FX.

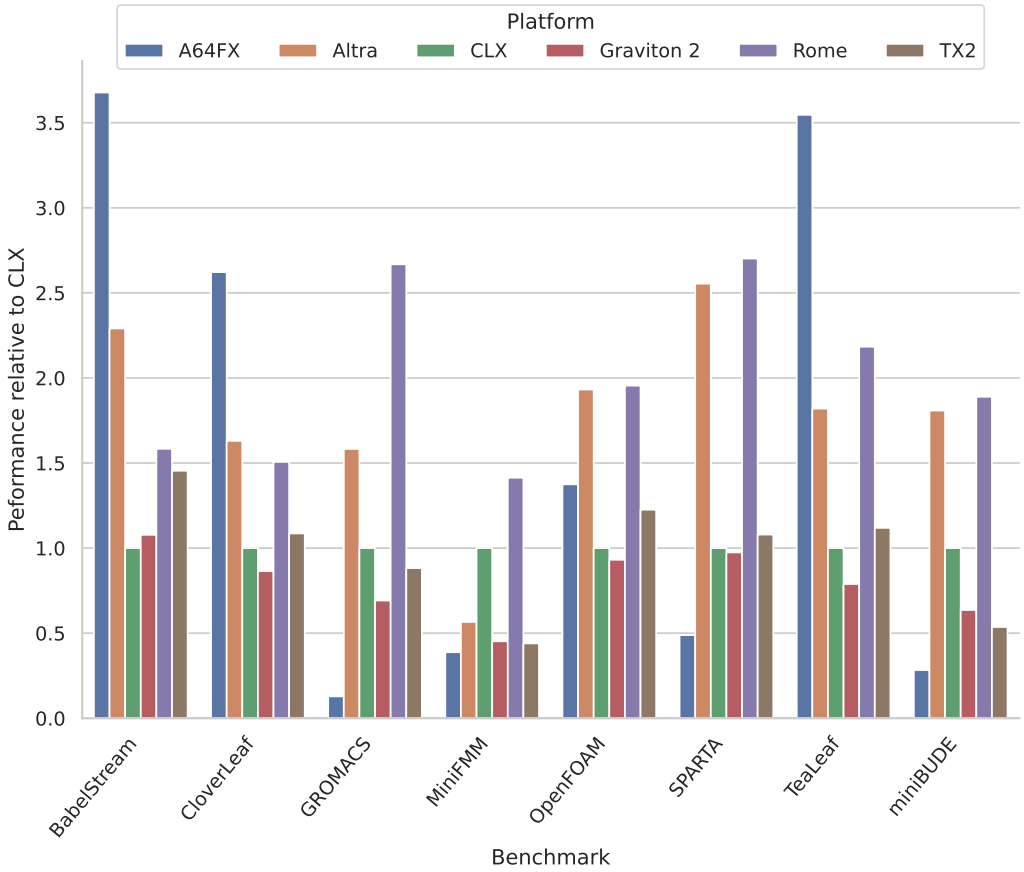


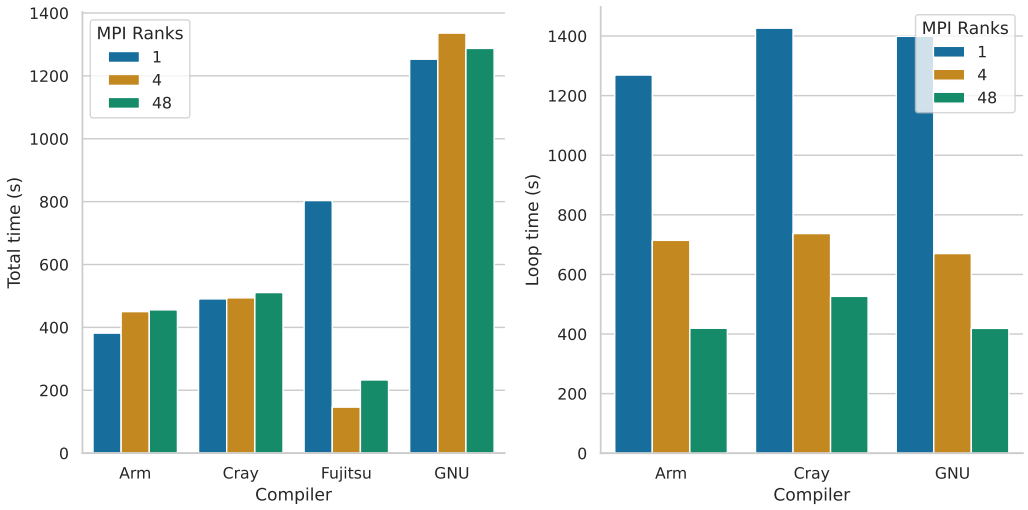
Figure 6.10: Performance across all benchmarks, normalised to Intel Cascade Lake. The best compiler choice was used in each case. Higher numbers represent higher performance.

Three of the benchmarks in this study combine MPI with OpenMP and allow the user to divide parallelism between the two levels: CloverLeaf, TeaLeaf, and SPARTA.

CloverLeaf and TeaLeaf behaved similarly, in that the fastest configuration differed with the compiler used: hybrid MPI–OpenMP, running one rank per CMG and filling all its 12 cores with OpenMP threads, was fastest with all compilers except for Arm, where flat OpenMP was the fastest configuration. The difference between the performance of hybrid MPI and flat MPI was around 5% with GCC and Cray, and around 15% with ACfL. However, the results were very different when using the Fujitsu compiler:

flat OpenMP was the *slowest* configuration, achieving less than 20% the performance of the hybrid configuration, and flat MPI was second, at 62% of the performance of the hybrid run. Placement results for CloverLeaf with all the compilers available on the A64FX are shown in Figure 6.11a.

SPARTA failed to build with the Fujitsu compiler; the other compilers all performed similarly to each other. For this benchmark, flat MPI was the fastest configuration, but here switching some of the parallelism to Kokkos threads *reduced* performance by up to 2×. Even though I used Kokkos 3.4, the latest at the time of writing and which supports the A64FX target, the code it generates may not yet be as optimal as OpenMP produced directly by a compiler. Figure 6.11b shows the run time of SPARTA under the three different run-time configurations.



(a) CloverLeaf (b) SPARTA

Figure 6.11: Comparison of MPI–OpenMP run configurations on A64FX. As many OpenMP threads were used as needed in each case to fill all 48 cores. Lower numbers show better results.

I found that the four compilers that can target the A64FX have different default semantics for binding threads, and sometimes these are different from the optimal configuration. The following settings reliably produced correct rank and thread placement on all the compilers tested:

- For flat MPI, set `OMP_NUM_THREADS` to 1 and bind each rank to a core, e. g. using the `-bind-to core` argument to `mpirun`;
- For flat OpenMP, disable binding of MPI ranks, in order to prevent all threads from being bound to the same object, using `-bind-to none`, then explicitly split threads between all the NUMA nodes using `OMP_PLACES=cores OMP_PROC_BIND=spread`;
- For hybrid OpenMP–MPI, fill all NUMA nodes equally with rank using `-map-by numa`, bind all its threads to the NUMA node with `-bind-to numa`, then spread the OpenMP threads onto the available cores with `OMP_PLACES=cores OMP_PROC_BIND=close`.

6.4 Future Work

In this study I have investigated the performance of the A64FX using single-node benchmarks. I have identified strong and weak points of this processor, but when running at scale these may manifest differently. In particular, compute-bound applications can become network-bound, thus increasing the benefits of using A64FX in a large-scale system.

One of the points for improvement that I have identified is around the compiler support for the A64FX. Because of its relatively lightweight microarchitecture, this processor relies on a good optimising compiler with an accurate cost model to schedule instructions well. There is currently a significant gap between the performance of binaries compiled with the Fujitsu Compiler and open-source alternatives, so there is room for further studies on this architecture to suggest and implement compiler improvements.

Finally, when looking at the next generations of high-performance processors, it is essential to understand how microarchitectural design decisions affect the performance of applications. This process is known as the *co-design* of hardware and applications and it is a way to ensure that future hardware will provide adequate performance for its intended use cases. Such experiments are generally hard, because modelling hypothetical architectures accurately is an involved task that requires specialised tools. Still, it is es-

sential in the co-design process, which is one of the main motivating factors for the upcoming SimEng simulation framework [86]. SimEng aims to enable fast, accurate, flexible simulations through a simple interface for extending existing processor designs with hypothetical additions[1].

6.5 Reproducibility

Instructions on running the benchmarks in this study are available online[2]. The scripts provided obtain the code and any input data, build the applications with the specified compiler, and provide run configurations for the platforms used in this chapter.

6.6 Conclusion

In this chapter, I explored the performance of the Fujitsu A64FX processor on a range of scientific benchmarks. The benchmarks were chosen to cover several important classes of HPC applications, and the results were compared to other common high-performance processors at the time of writing. I gave special attention to Arm-based alternatives, of which I covered the previous-generation Marvell ThunderX2 and the newer AWS Graviton 2 and Ampere Altra. I also compared to the best-in-class x86 processors available at the time of writing.

I found the A64FX to be a competitive processor for HPC. It performed particularly well for memory-bandwidth-bound applications, where its HBM2 with a peak of 1 TB/s was utilised to its full potential. The results on compute-bound benchmarks were mixed: performance was good when using the Fujitu Compiler, which was specifically developed to target the A64FX, but with other compilers that do not apply optimisations such as software pipelining, the relatively lower out-of-order capacity of the A64FX led to reduced performance compared to more heavyweight cores.

[1] `https://uob-hpc.github.io/SimEng-Docs/index.html`
[2] `https://github.com/UoB-HPC/benchmarks`

CHAPTER 7

Programming Models for Modern HPC Architectures

Content from this chapter appears in the following publication:

- Andrei Poenaru, Wei-Chen Lin and Simon McIntosh-Smith. 'A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application'. In: *High Performance Computing. 36th International Conference, ISC High Performance 2021.* Ed. by Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief and Piotr Luszczek. Cham: Springer International Publishing, 2021, pp. 332–350. ISBN: 978-3-030-78713-4. DOI: 10.1007/978-3-030-78713-4_18

This paper has received the Hans Meuer Award for the most outstanding research paper submitted to the ISC 2021 conference.

In HPC, the majority of programs utilise established, long-running parallel programming frameworks. OpenMP and MPI are widely adopted [67], generally on top of the C and Fortran programming languages, and other frameworks are sometimes used in more specific situations, such as task-based parallelism libraries, C++-native applications, or programming models that can target GPUs [10].

Recent developments in parallel programming frameworks — be it frameworks developed from scratch, such as Kokkos, or additions and improvements to existing ones, such as tasking and offload support in OpenMP 4.5 and later — have all shared a number of common goals: performant support for a wide range of hardware platforms, interoperability with modern versions of the C++ language, and a focus on programmers' productivity [80, 103]. These goals have organically arisen as a result of the shortcomings in established programming models, and together contribute to the wider endeavour in the field of HPC towards achieving *performance portability* [29].

In order to support both GPU and CPU platforms, high-level programming frameworks manage underlying data structures automatically: the programmer expresses what data needs to be computed on, and the framework arranges it in a format suitable for the target hardware. Because this process is desirable, but not always optimal from a performance point of view, it is one of the key focuses of previous analyses of high-level parallel programming models [26]. Such previous studies have used mostly memory-bandwidth-bound benchmarks, but the portability and productivity advantages alone brought by these frameworks may be enough to also justify their usage on applications that are not memory-bandwidth-bound, provided that they can deliver performance comparable to established frameworks.

One suitable application for such heterogeneous frameworks is the Bristol University Docking Engine (BUDE), a molecular docking code that is heavily compute-bound [89]. BUDE is routinely used for *in silico* drug discovery, and out of a need to support both CPUs and GPUs, it is comprised of two parallel implementations: an OpenMP version for CPUs and an OpenCL version for GPUs. In this chapter, I used a mini-app created from the core computation kernel for BUDE to analyse the performance of emerging parallel programming models compared to that of traditional models.

7.1 Background

7.1.1 High-Performance Molecular Docking

BUDE is an application for *in silico* molecular docking, a computational technique for predicting the structure of a complex formed between two molecules and estimating the strength of their interaction [89]. Docking is computationally challenging because of the many different ways in which two molecules may be arranged together to form a complex (three translational and three rotational degrees of freedom). Indeed, interacting all patches of the surface of one protein molecule with all patches of a second molecule requires on the order of 10^7 trials, each one of which is a computationally expensive operation [18].

The application includes several modes of operation, of which the most commonly used — and the most computationally intensive — is *virtual screening*. In this mode, molecules of drug candidates, known as *ligands*, are generated using a genetic algorithm and are bonded to a *target* protein molecule. BUDE uses a tuned empirical free-energy forcefield to predict the binding energy of the ligand with the target. There are many ways in which this bonding could occur, so a variety of positions and rotations of the ligand relative to the protein are attempted; these are known as *poses*. For each pose, the energy, i. e. the strength, of the bond is evaluated.

7.1.2 Modern Parallel Programming Models

In the previous decade, low-level programming models that offered the programmer great control over the hardware saw a rise in their usage. Their appeal of low overhead and extensive tuning options made them popular with GPU programmers [37, 131], but over the years the HPC community has learned the cost these frameworks incur: they require extensive knowledge of the hardware, and they steer towards over-optimisations for one target, up to the point where a significant fraction of the code needs to be rewritten when moving to a new system [110, 60, 124]. The latter observation is par-

ticularly relevant in the context of the upcoming exascale systems Frontier, El Capitan, Aurora, and Perlmutter, which together utilise combinations of CPUs from two vendors and GPUs from three vendors [52, 68]. It is, thus, not feasible to use platform- or vendor-specific programming models, and a portable approach is needed.

In moving to new programming models, the C++ language has particular appeal: it can achieve the same zero-overhead performance compared to C and Fortran, but it also offers modern features to write more expressive and safer code. Programmers writing parallel C++ hope to outweigh any lost performance with time gained through easier-to-write and easier-to-debug code. This is the core selling point of modern parallel programming frameworks [95, 41].

Two modern, single-source frameworks with a focus on performance, portability, and productivity have emerged: Kokkos [32] and SYCL [45]. Kokkos is a new framework developed natively for C++, while SYCL builds on previous OpenCL toolchains and integrates them with modern C++ code. Both of these frameworks can generate machine code for both CPUs and GPUs without any change to the high-level source code.

These frameworks solve the same problem in different ways. Kokkos is distributed as source code that needs to be integrated into the application's build process. This means that every application using Kokkos needs to build Kokkos itself — a relatively quick process — but it also avoids the pitfalls of system-wide libraries; a C++ compiler is all that is needed to compile a Kokkos application.

In contrast, SYCL applications rely on a SYCL compiler. At the time of writing, there are three major SYCL compilers: Data-Parallel C++ (DPC++) [55], ComputeCpp [125], and hipSYCL [3]. Each implementation can use different backends: DPC++ can use OpenCL, CUDA, or Intel's Level Zero; ComputeCpp relies on OpenCL; and hipSYCL supports OpenMP to target CPUs, CUDA to target NVIDIA GPUs, and ROCm to target AMD GPUs.

7.1.3 Performance Portability

In recent years, the HPC community has made efforts to understand how to quantify performance portability. Although some formal metrics have been developed and are commonly applied in portability studies [49], the results are not always trivial to interpret correctly [102]. One attempt to solve this challenge relies on carefully designed visualisations [123].

Portability is a common concern for developers — and users — of modern programming models. These make it more feasible to target several kinds of compute devices simultaneously, which has led to a diverse landscape of architectures being investigated in contemporary HPC research. As such, significant attention to portability and programmer productivity is also given in recent studies that evaluate the applicability of novel parallel frameworks [24, 72, 30].

7.2 Evaluation Methodology

7.2.1 A BUDE Mini-App

I have implemented a mini-app for BUDE virtual-screening runs, with kernels written in a range of widely used parallel programming models. The baseline implementation is written in OpenCL and is virtually identical to the core kernel of the full-scale BUDE application. There is a CUDA port with minimal changes, a CPU OpenMP version that restructures the computation in the OpenCL kernel to make it easier for compilers to vectorise, and similar implementations for GPUs using OpenACC and OpenMP `target` offload. I chose SYCL and Kokkos for implementations in novel programming models because of their relative popularity and compatibility with a wide range of platforms, covering both CPUs and GPU.

The focus of the mini-app is on the core computation, and so most of the plumbing around it, such as flexible I/O and custom file formats, has been removed. Instead of using a genetic algorithm to generate ligands, a procedure which takes negligible time in a full-scale BUDE run, the mini-app

uses pre-generated molecules obtained from the full BUDE application. The main advantages of this approach are that it simplifies the mini-app logic, it makes the results easier to reproduce, and it allows for a built-in validation procedure by comparing mini-app output against reference output from the full application. Thus, the mini-app simply reads in a protein and a ligand, computes the bonding energies over a user-defined number of poses, and compares them against a reference set. To enable custom-length benchmarks, the mini-app can run several iterations of the same ligand–protein combination instead of requiring a new ligand each time. The result is a benchmark consisting of a few hundred lines of code for each implementation, which is easy to understand, feasible to profile and analyse, has built-in validation, requires no external libraries, and with a performance profile that maintains the same important characteristics of the full application.

7.2.2 Performance Analysis

I analysed the performance of our mini-app on a range of modern HPC platforms; Table 7.1 shows the systems used and their specifications. Where several compilers could be used for the same programming model, I tested all the options and picked the best-performing one in each case. I used aggressive compiler optimisation flags to the level of `-march=native -Ofast`. Table 7.2 lists the compilers used, the parallel programming frameworks supported by each, and any platform targeting restrictions they have.

I collected performance data using industry-standard tools. On CPU platforms, I accessed hardware counters through the LIKWID framework [135] and the built-in Linux `perf` tool, and collected application-level profiles with Cray Perftools; on GPUs, I used the NVIDIA CUDA profiler and the OpenCL Intercept Layer. I obtained peak memory bandwidth figures using BabelStream [27] and the University of Bristol's HPC Group's cache-bandwidth measurement tool [78].

I used two input decks to benchmark the application: a small input set, consisting of 26 ligands, and a large set, with 2672 ligands. The former takes around 0.5 seconds to run on a contemporary dual-socket-CPU HPC system,

Table 7.1: Hardware platforms used for evaluation.

Platform	Type	Cores	Clock Speed (GHz)	Peak Performance (SP GFLOP/s)
Intel Skylake 8176	CPU	2 × 28	2.1	5,734
Intel Cascade Lake 6230	CPU	2 × 20	2.1	4,096
AMD Rome 7742	CPU	2 × 64	2.25	9,216
Marvell ThunderX2	CPU	2 × 32	2.5	2,560
Fujitsu A64FX	CPU	48	1.8	5,530
NVIDIA V100	GPU	80	1.13	15,700
AMD Radeon VII	GPU	60	1.4	13,800
Intel Iris Pro 580	GPU	72	0.95	1,094

while the latter takes around 1.5 minutes. In both cases, I ran 8 iterations of the algorithm and I computed 2^{16} poses per iteration. I utilised all the available cores on each platform, using a single thread per core on all the CPU platforms; where available, using more than a single thread per core did not improve performance. A warm-up iteration was always run before the timers were started.

There was very little run time variability in miniBUDE. Even on the small input set, when individual iterations take less than 100 ms, variance was only fractions of a percent. This was true for both CPU and GPU implementations, as long as care is taken to bind threads correctly, especially when two interacting systems are present, e. g. OpenMP's `OMP_PROC_BIND` and Cray's `aprun`. There was one exception to this observation, which I address in Section 7.4.

Table 7.2: Compilers used and their programming model and target platform support.

Compiler	CPUs	GPUs	Frameworks
AOCC 2.3	X		m k s
AOMP 11.0		M	m
Arm Compiler 21.0	R		m k s
ComputeCpp 2.1.1	X	I	m k s
Cray Compiler 10.0	R X	N	a[1] m k s
Fujitsu Compiler 4.3	R		m k s
GCC 10.3	R X	M N	a l m k s
Intel ICX 2019	X		m k[2] s
Intel DPC++ 2021.1	X	N	m k s
LLVM 11.0	R X	N	m k s
NVCC 10.2		N	c
PGI 19.10		N	a

CPUs: A**R**M, **X**86; GPUs: A**M**D, **N**VIDIA, **I**NTEL
Frameworks: **c**uda, open**a**cc, openc**l**, open**m**p, **k**okkos, **s**ycl

[1] Version 9.0 only; [2] With the experimental `INTEL_GEN` backend.

7.3 Results and Performance Analysis

7.3.1 CPUs

OpenMP

The OpenMP implementation was written in plain C, without any higher-level framework, and was optimised for CPU platforms. I expected this version to incur the least overhead and thus perform fastest on the CPU. As I will show in this section, OpenMP did offer the best performance on CPUs in most cases, but higher-level implementations were sometimes able to match it.

Parallelism is exposed through OpenMP at two levels: poses are distributed between threads, and the calculations for each pose take advantage of each thread's SIMD lanes. Thread-level parallelism is achieved by dividing the poses into groups and then distributing the groups over threads; this creates an execution model similar to OpenCL workgroups, where each thread iterates over its assigned poses. The size of the group of poses is specified as a compile-time parameter.

I found that the group size had significant impact on the performance of the OpenMP implementation of miniBUDE. On each platform, this parameter should be at least as large as the native vector length, such that all the SIMD lanes are utilised for computation, but I found that most platforms achieved the best performance at group sizes several times larger than the native vector length. This happened because compilers were able to fully unroll the inner thread loops. As such, the group size is not only a vectorisation factor, but also an unroll factor, and higher values allowed platforms to fully exploit their out-of-order resources by interleaving several (unrolled) loop iterations. Furthermore, a small part of the arithmetic can be factored out and computed only once per work group, resulting in additional computation time savings. Figure 7.1 shows the impact of the group size parameter on performance for each platform.

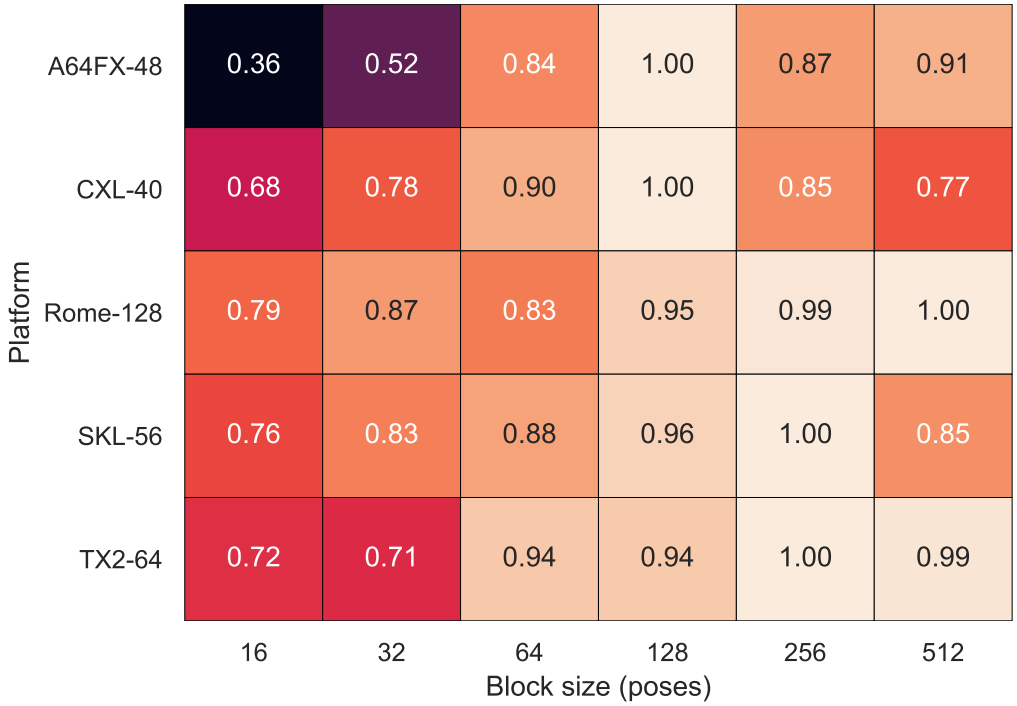


Figure 7.1: Performance of the OpenMP implementation at different group sizes, normalised to the best result on each platform. Platforms are labelled using the abbreviations in Table 7.1 and the number of cores. Higher numbers, shown here in brighter colours, correspond to higher performance.

The other defining factor for the performance of the OpenMP implementation is vectorisation. In order to maintain portability, no architecture-specific intrinsics are used; I rely on compiler auto-vectorisation. The code is structured such that vectorisation is required at the innermost level, which allowed all compilers tested to vectorise the main computation. The Cray and Intel compilers successfully vectorised *all* the loops in the code, while GCC and the Arm compiler did not understand the structure of one `do-while` loop and so did not vectorise it. This last loop, however, is not critical for performance.

The compilers further differed in their instruction choice and scheduling. On the Intel platforms, only the Cray compiler generated 512-bit vector code by default. Because this code is compute-heavy, long vectors greatly benefit performance, and forcing the Intel and GNU compilers to generate 512-bit

operations — instead of their 256-bit default — significantly reduced the run time. In addition, the Intel and Cray compilers automatically interleaved the loop bodies, thus overlapping arithmetic and memory operations from different iterations.

On the other hand, GCC only unrolled the loops, without interleaving, and so instructions for each iteration were scheduled sequentially. This lowered the achieved performance on the platforms with fewer out-of-order resources, such as the A64FX, which performed slower than a ThunderX2, even though the former has 4× the vector width of the latter. The Fujitsu compiler, which has a good cost model of the A64FX and performs aggressive software pipelining and division optimisation, generates the fastest code in this case. Figure 7.2 shows the performance of the OpenMP implementation on the CPU platforms across the compilers tested.

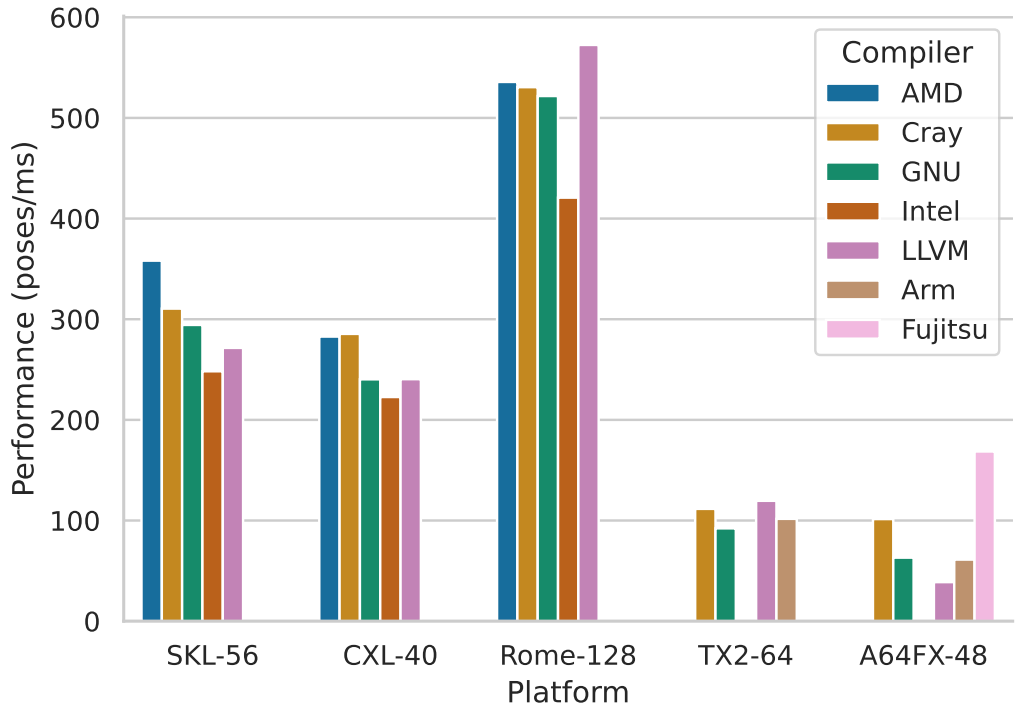


Figure 7.2: Performance of the OpenMP implementation across systems and compilers. Higher numbers represent faster execution.

Figure 7.3 shows a roofline chart of the Cascade Lake platform. The OpenMP implementation of miniBUDE has an operational intensity of 0.3

and achieved a performance of 2301 GFLOP/s, which represents 56.2% of the platform's peak. The application sits directly below the arithmetic roof and above the memory bandwidth bound, confirming the code is compute-bound. For the purposes of the roofline model, FLOPs and memory traffic (assuming caching as per the cache-aware roofline model) were manually counted in the application's source code and corroborated using hardware counters.

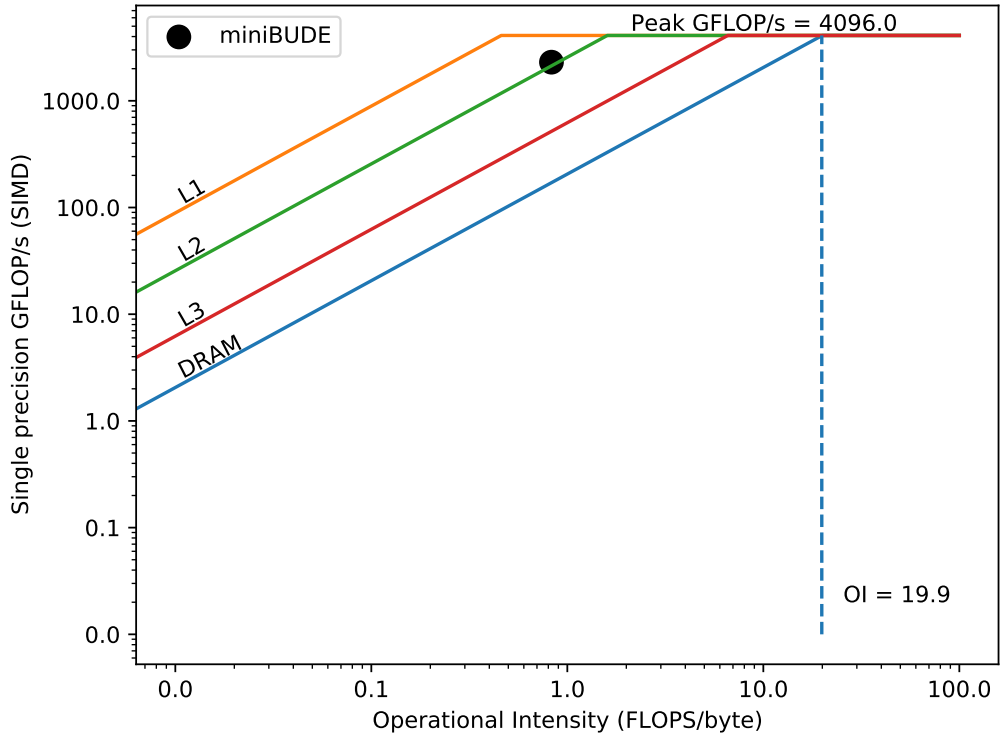


Figure 7.3: Cache-aware roofline for the Cascade Lake platform showing the achieved performance for miniBUDE.

Kokkos

The Kokkos implementation is a direct port of the OpenMP version, with parallelism expressed via the idiomatic `Kokkos::parallel_for` function. I retained the group size parameter to investigate the effects of unrolling, and I found that it had the same effect as in the case of OpenMP, and the same values were optimal on each platform. Like the OpenMP version, Kokkos does

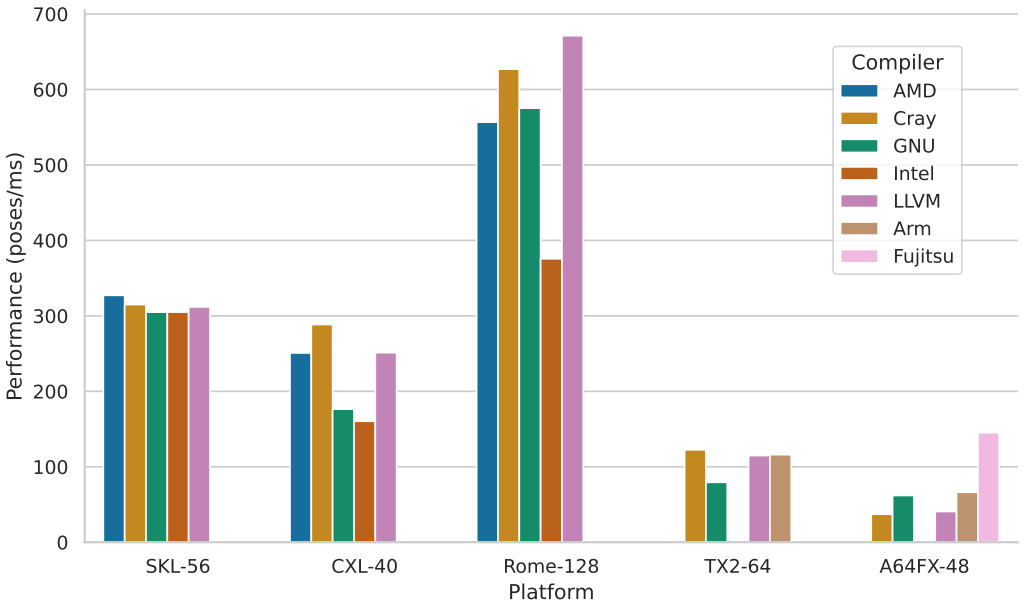


Figure 7.4: Performance of Kokkos with the OpenMP backend on the test platforms. Higher numbers represent faster execution.

not offer built-in types for vectors and functions to use with them. From a productivity standpoint, it may be preferable for the framework and runtime to provide optimised versions of common math types and functions, so that compilers can better optimise code with the correct constraints. This is especially important for parallel frameworks that can target different backends — as Kokkos does — where each platform can have its own unique requirements, e. g. alignment on specific boundaries.

Kokkos was able to provide complete platform support in our study by virtue of being able to utilise many different programming frameworks as backends. Because a C++ compiler is the only requirement to build a Kokkos application, and because Kokkos itself is built as part of the same process, I can compare the relative performance on the platforms studied when using different compilers. Figure 7.4 shows a performance comparison on each CPU platform, where Kokkos uses the OpenMP backend, normalised to the fastest result. The results shows a strong correlation compared to the OpenMP implementation results described in Section 7.3.1, which shows Kokkos is using OpenMP efficiently on all the architectures.

SYCL

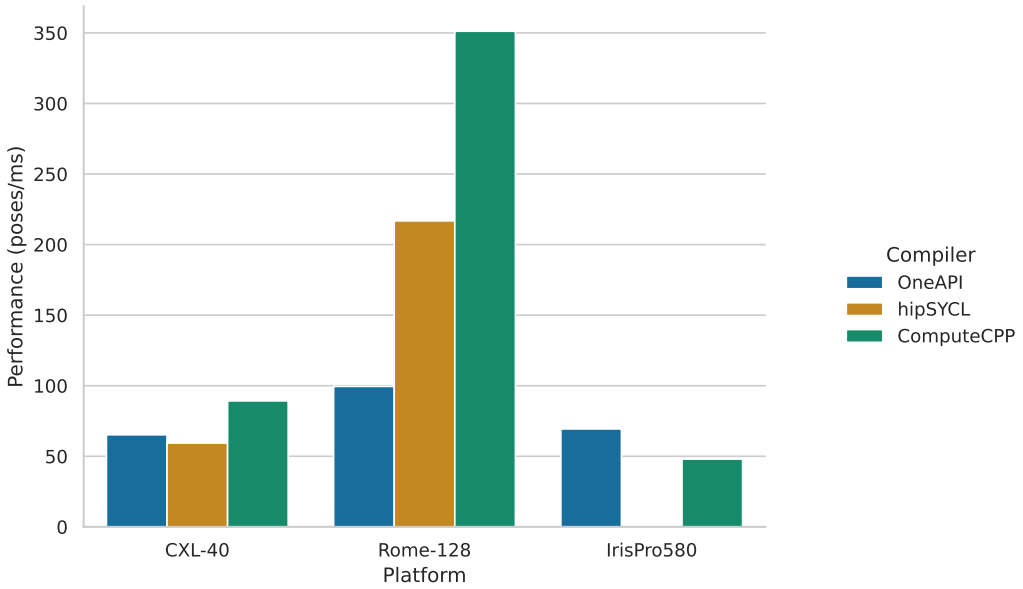


Figure 7.5: Relative performance of SYCL implementations, on the platforms where more than one was available. Higher numbers represent faster execution.

The SYCL implementation was written in idiomatic SYCL 1.2.1. The kernel is a direct port of the OpenCL version, utilising workgroup-based parallelism (`sycl::nd_range`) with few changes required. It retained the existing GPU-friendly optimisations from the OpenCL kernel where data is first copied to local memory via OpenCL's `async_work_group_copy`. Due to SYCL's roots in OpenCL, the APIs used for implementing these operations are identical both in name and semantics. We were even able to retain the use of 3-d vector types which correspond to the `cl_vec3` in OpenCL.

For comparison, we also implemented a separate kernel that is closer to the OpenMP implementation, where parallelism is achieved with flat `parallel_for` calls based on `sycl::range`. Although in theory plain `range` may be easier to map onto the hardware than `nd_range`, I found the performance difference between the two implementations to be negligible (below 2%).

Figure 7.5 shows the performance of all SYCL implementations on the platforms tested where at least two implementations were supported. On

each platform, performance is normalised to the fastest implementation. For hipSYCL on the x86-based platforms, I tried all the compilers available and picked the one that produced the fastest binary, which was Cray on both Cascade Lake and Rome. The Skylake platform is missing from these results because an incorrect interaction between the Intel OpenCL driver installed on the system and the Cray `aprun` launcher resulted in all threads being pinned to a single core, effectively invalidating the results obtained with the two implementations that reply on OpenCL, OneAPI and ComputeCpp. On the V100 and the Radeon VII, hipSYCL is the only usable SYCL implementation.

7.3.2 GPUs

Low-Level: OpenCL and CUDA

The OpenCL implementation is a close representation of the main kernel in the full-scale application, with the modifications presented in Section 7.2.1; the CUDA implementation is a direct port of the OpenCL version. The two versions performed similarly on the NVIDIA V100 GPU: the CUDA implementation was 18% faster than the OpenCL code, on both the small and the large input decks. The performance difference was evenly spread across the execution of the program: all the kernels were slightly slower when using OpenCL. Memory transfers are not timed for the purposes of the benchmark, and they take negligible time ($< 1\%$ of the total run time). All of the benchmarks were run on CUDA Toolkit 10.2 running on NVIDIA driver version 440.64, so the difference likely came from more optimisation on the CUDA side of the NVIDIA library.

Both versions also ran on the AMD Radeon VII, converting the CUDA version through HIP, but OpenCL was 1.6× faster on this platform. Since the kernel code for both implementations was very similar, I attributed the performance difference to inefficiencies in AMD's HIP compiler. CUDA and HIP cannot be used on the Intel GPU.

Directives-Based: OpenMP Offload and OpenACC

The directive-based GPU implementations run the same kernel code in the OpenCL implementation, but expressed in the same C file as the host application and without any of the explicit OpenCL platform set-up and clean-up code. This is a significant advantage for productivity: given host code, only three `pragma` directives are used to transfer the data to the GPU and generate GPU kernel code. The main difference from the OpenCL version is that the global and local sizes are not set by the programmer, but are controlled by the runtime. To control the amount of computation per workgroup, the directives-based implementations include a macro to control loop unrolling, similarly to the CPU OpenMP implementation.

The implementations achieved virtually identical performance on the V100. This was expected, because the same CUDA-based backend is used to generate code for both frameworks. Compiler support, however, differs between the two: the OpenMP code can use the latest versions of the Cray and GNU compilers, but the OpenACC version could only be compiled with an older version of the Cray compiler (9.0). The GNU and PGI compilers produced non-working code for OpenACC, and newer versions of CCE have dropped support for it.

On the V100, the directives-based approach showed about 0.4× the performance of the optimised CUDA code. This is the combined result of inefficiencies I identified in two places: 1) high register usage in the kernels generated by the compiler limits the maximum achievable GPU occupancy; 2) lower performance of library functions. This difference is higher than what has been observed in previous studies [26], and is likely exacerbated by the heavily compute-bound nature of miniBUDE.

On the Radeon, OpenACC can be compiled with GNU, but the resulting code was two orders of magnitude slower than OpenMP, which in turn only reached 0.3× the performance of the fastest model, OpenCL. The low-level nature of OpenCL allowed the code to map very well onto the target hardware, a performance which the GNU offload maths libraries could not match.

On the Intel GPU, OpenMP `target` reached only $0.2-0.3\times$ the performance of the fastest model, which in this case was SYCL. Although SYCL uses the same drivers as OpenCL on this platform, in this case the OneAPI compiler was better able to extract performance from the hardware when starting from higher-level, more expressive programming model. The OpenCL implementation was developed with HPC GPUs in mind, and while with code changes specific to the Intel GPU architecture it should be possible to reach the same performance with a low-level OpenCL implementation, this result highlights the productivity benefit of the higher-level programming model when targeting several platforms simultaneously, as long as the model is well-supported on all the targets.

High-Level: Kokkos and SYCL

Kokkos and SYCL both run on all the GPUs studied, but only one implementation, hipSYCL, runs on AMD and NVIDIA. The code run on the GPU platforms was unchanged from the version run on CPUs, not even to define different parallelism, as was the case when moving from CPU OpenMP to OpenMP `target` offload.

Figure 7.6 shows the results on the GPU platforms for all programming models studied. The three GPUs each target different segments: the V100 is a top-end HPC GPU, the Radeon VII is a high-end consumer GPU, and the Iris Pro is a mobile chip designed for a very constrained power and transistor budget. A direct performance comparison between such different platforms is not useful; instead, I present programming model performance normalised to the fastest result on each platform. In absolute figures, the best result on the V100 (CUDA) was twice as fast than the best on the Radeon VII (OpenCL) and $14\times$ faster than the best Iris Pro 580 result (OneAPI SYCL).

7.4 Towards Portable High-Performance Code

Section 7.3 has analysed the performance of the miniBUDE implementations on the platforms studied, but the implications of these results are

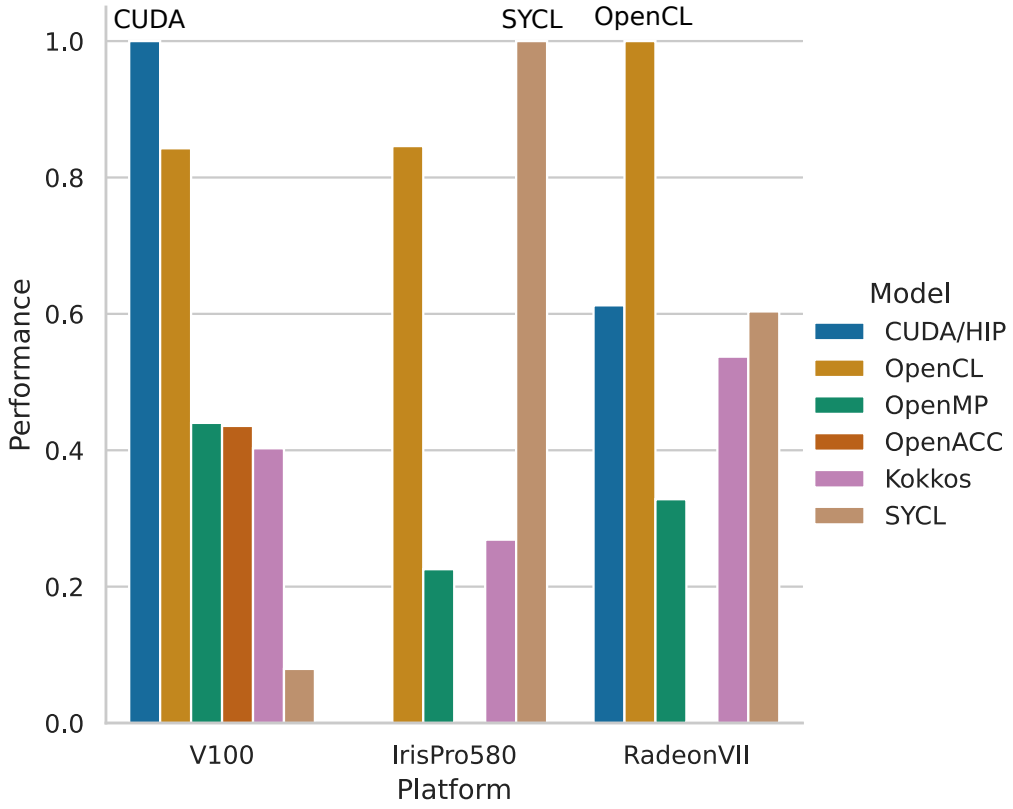


Figure 7.6: Performance of the GPU implementations, normalized to the fastest result on each platform. The fastest model on each platform is labelled explicitly.

further-reaching. Figure 7.7 aggregates the performance results over all the platforms and programming models and highlights that no programming model can currently achieve the best performance on *all* platforms.

This effect is more pronounced on GPUs: each of the three platforms studied achieved the highest performance using a *different* programming model, and they relied on parameter tuning to do so. This immediately imposes a penalty when moving to a new platform, at which point at the very least tuning needs to be redone. In the worse case, low-level frameworks can trap users into code so specific to one platform that a major rewrite is needed when changing targets. However, OpenCL was the fastest model on the Radeon VII and a close second on the other two GPUs studied, suggesting that is may still be the best choice for good performance portability.

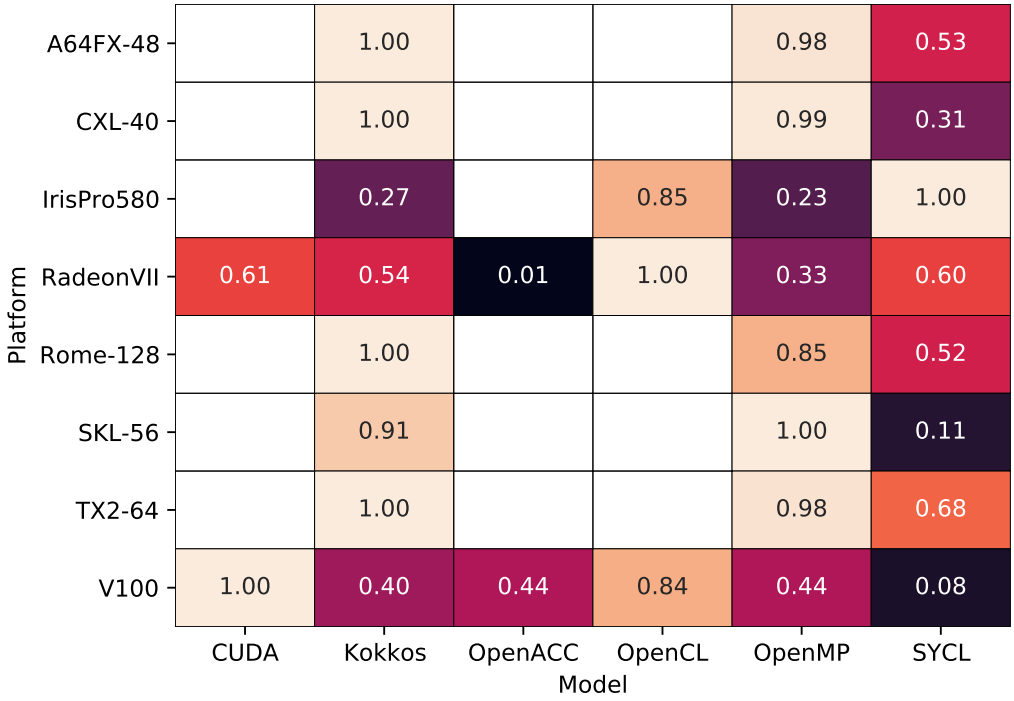


Figure 7.7: Achieved performance across all programming models, normalised to the fastest result on each platform. Lighter colours correspond to higher relative performance; blank cells are impossible results.

Higher-level programming models avoid this issue of over-specialisation of the code, instead relying on being able to translate the high-level code to efficient machine code as part of the framework. Kokkos is a good example of this: on the CPU platforms it achieves performance close to that of OpenMP, and both frameworks require similarly small amounts of framework-specific code, which consists mostly of loop annotations. The same Kokkos code is able to run on both CPUs and GPUs, and on the platforms studied it again achieved performance similar to that of OpenMP, but *without any source changes*; with OpenMP, a *different* version of the code was written for GPUs. Kokkos was the only framework that was able to support all CPU and GPU platforms in one package.

The SYCL landscape is rapidly evolving, and indeed the new SYCL 2020 standard — which is already being adopted by the three main implementations — brings much-needed productivity improvements such as built-in re-

duction support and alignment with the newer C++17 standard [114]. However, at the time of writing there are still rough edges to the current SYCL compilers, mainly around platform support fragmentation. First, support for non-GPU or non-x86 platforms is experimental, or even missing from some implementations. Even for GPUs, there is no single implementation that works across all the hardware from the major vendors.

The open-source hipSYCL implementation is the most portable of the set, being able to run on CPUs, as well as on NVIDIA and AMD GPUs. Both ComputeCpp and OneAPI provide experimental NVIDIA GPU support, but there are still blocking issues such as missing built-in function implementations, which prevent miniBUDE from compiling. Finally, running SYCL on Intel GPUs requires Intel's OpenCL-based ComputeRuntime, but only ComputeCpp and OneAPI support this mode of operation.

The situation on CPUs is similarly complicated. For x86-based platforms, both ComputeCpp and OneAPI run on top of the Intel OpenCL runtime, similar to the situation on Intel GPUs. The OpenCL runtime achieves parallelism via Intel's OneAPI Threading Building Blocks (OneTBB), which provides an optimised abstraction for managing logical threads. Such runtime approaches limit the extent of SYCL implementations to what the underlying runtime supports, from platform coverage to features it can provide; this currently prevents the use of ComputeCpp or OneAPI on Arm-based platforms.

On the other hand, hipSYCL translates SYCL abstractions to OpenMP code, which can then take advantage of existing compiler optimisations natively. This approach results in wide platform support for hipSYCL, but it also means, in principle, that parallelism abstractions are mapped to straightforward OpenMP equivalents. In practice, I found that performance was lower with hipSYCL compared to Kokkos or plain OpenMP, and code changes such as using different parallelism abstractions made little difference for miniBUDE. On platforms not explicitly supported by hipSYCL, as was the case of the A64FX at the time of writing, the additional layer of abstraction also prevented optimal code from being generated, despite having used the correct C++ compiler target flag.

Portability between CPUs and GPUs remains a concern, as SYCL has inherited the same set of problems seen when running OpenCL on the CPU: it is problematic to map workgroup-based parallelism onto a CPU intuitively and efficiently, and it suffers from unexpected setup costs compared to the OpenMP implementation. To work around potentially inefficient mapping, I implemented a compile-time tuning parameter to adjust the amount of work performed by each workgroup, though I found no common setting that provided the best performance on all platforms. On platforms that use Intel's OpenCL runtime, i. e. ComputeCpp and OneAPI, I found the kernel runtime to have large variations, and no functionality was provided to address or mitigate this. In particular, investigations revealed that initialisation of the SYCL context — the `queue` — took upwards of 800 ms in certain cases, even for a simple benchmark that itself ran in half that time.

I also discovered that when running several iterations of a benchmark back-to-back, the first run was usually up to 2× slower than subsequent runs. It was essential to implement a "warm-up" run, which is completely discarded, before starting the timer on the benchmark. Once the warm-up run was completed, the remaining iterations showed consistent run times, as with the other programming models. Both ComputeCpp and OneAPI compile SYCL kernels ahead-of-time, and neither give any indication why initialisation imposes such a large overhead; it is most likely an interaction with the underlying driver. Implementations that do not use the Intel OpenCL runtime, e. g. hipSYCL, did not incur this performance penalty.

7.5 Reproducibility

The source code for all the miniBUDE implementations used in this study, as well as build and run instructions and benchmark input cases, can be found online[1]. A set of scripts is also provided to build and run the benchmark on the platforms used in this study[2].

[1] `https://github.com/UoB-HPC/miniBUDE`

[2] `https://github.com/UoB-HPC/performance-portability/tree/2021-benchmarking/benchmarking/2021/bude`

7.6 Conclusion

In this chapter I have explored performance portability through the lens of a simple, yet realistic, compute-bound benchmark. I have implemented the benchmark in several programming models, including low- and high-level, both well-established and up-and-coming. I have shown that modern programming models can perform on-par with traditional ones, and with constant work done to improve them, their platform support continues to grow.

On the other hand, I have seen that true performance portability is still out of reach: no single version of the code achieved the best performance — or a high fraction of it — on all the platforms studied. Even for a small kernel, platform-specific optimisations and empirical tuning of parameters accounted for more than 30% of the performance and that was enough to differentiate the best-performing implementation from the rest. On GPUs, low-level APIs continue to provide the highest possible performance, and on CPUs, the still-immature driver and implementation ecosystem around SYCL presents an obstacle to the wide adoption of this programming model as a true cross-platform, cross-architecture framework. Of the frameworks studied, Kokkos emerged as a reliable choice, with its lightweight, optimised implementation, and OpenMP remains in a strong position due to it widespread support, although different code paths are still needed for optimal CPU and GPU implementations at the time of writing.

CHAPTER 8

Research for Future HPC Architectures

8.1 Towards Accurate Performance Modelling

In Chapter 4, I analysed SVE applications using emulation tools. However, I found these to be severely limiting in terms of the size of the inputs they can run before emulation time became impractical. These were merely mini-apps, so evaluating full-size HPC applications on real inputs would incur additional performance penalties. With ArmIE, emulation overhead increases by several orders of magnitude when the instrumented application uses system calls, dynamically linked libraries, and file operations. For such experiments, benchmarking real hardware remains the only presently viable option.

For design-space exploration, a fast, flexible simulator built to take advantage of HPC infrastructure could address the performance needs of the tools available today. This class of experiments for microarchitectural design-space exploration with arbitrary hypothetical processor configurations is one of the main goals of the upcoming SimEng simulator developed at the University of Bristol [84].

SimEng could integrate the cache simulator presented in Chapter 5 for its memory hierarchy components. It would be able to extend upon the single-

core work presented in this thesis to more accurately model interactions between cores and their private caches, or contention for shared caches.

The cache simulator itself could be improved in further research. The first extension should be modelling realistic prefetching implementations, which would improve both its accuracy and the relevance of its results. At this point, it would be possible to make direct performance measurements by introducing a cycle model, which would enable data to be collected in terms of cycles spent to fulfil cache requests, not just hits and misses.

After comprehensively covering the single-core cache effects of vector computation, a natural next step is extending the investigation to a multi-core system. At this level, accurate timing of each observation becomes more important, because it may change interactions between private and shared caches, and one core's prefetching may now affect other cores. To address this, a cycle-accurate core simulator could be coupled with the cache simulator, together simulating a processor and its memory hierarchy. This coupled design has the advantage of exposing more parallelism that can be exploited for faster simulations [61]. The upcoming SimEng simulation framework developed at the University of Bristol has such full-system simulations at high performance as one of its main goals.

Another opportunity to investigate and potentially improve performance arises around the design of the applications themselves. Given access to accurate simulation tools, a future application can make informed decisions about how best to utilise a wide range of hardware, e. g. through careful laying out of data. These decisions are particularly important — and hard to make — on systems with configurable caches, e. g. the sector cache on the A64FX. Some high-level programming frameworks already support making automatic changes to the data layout based on the architecture targetted [32], and further insight into memory systems can only improve their decisions.

8.2 Next-Generation Vector Processors

In Chapter 6 I investigated the performance of the A64FX using single-node benchmarks. I have identified strong and weak points of this processor,

but when running at scale these may manifest differently. In particular, compute-bound applications can become network-bound, thus increasing the benefits of using A64FX in a large-scale system.

One of the points for improvement that I have identified is around the compiler support for the A64FX. Because of its relatively lightweight microarchitecture, this processor relies on a good optimising compiler with an accurate cost model to schedule instructions well. There is currently a significant gap between the performance of binaries compiled with the Fujitsu Compiler and open-source alternatives, so there is room for further studies on this architecture to suggest and implement compiler improvements.

As more SVE implementations become available, they are likely to exhibit varying performance characteristics on HPC applications. One of the key differences is the native vector length, and, as I have shown in Chapter 4, each application reacts differently to each choice of vector length. A future study comparing the real-world performance of HPC applications on a range of SVE microarchitectures, and investigating the factors that contribute to their performance, could further the community's understanding of the role of the vector length in achieving high performance and would represent a step forward towards finding the ideal vector length, if such as choice exists.

Finally, when looking at the next generations of high-performance processors, it is essential to understand how microarchitectural design decisions affect the performance of applications. This process is known as the *co-design* of hardware and applications and it is a way to ensure that future hardware will provide adequate performance for its intended use cases. Such experiments are generally hard, because modelling hypothetical architectures accurately is an involved task that requires specialised tools. Still, it is essential in the co-design process, which further motivates SimEng [86].

8.3 Productivity in Modern Programming

The work on miniBUDE presented in Chapter 7 opens the path to additional work on the full-scale BUDE application. Instead of maintaining separate implementations in OpenCL for GPUs and OpenMP for CPUs, the

code could incorporate a framework like SYCL or Kokkos to reduce divergence. Of course, embracing a new programming model for a scientific application is bound to encounter additional challenges, but in solving those the boundary of performance portability will be pushed further. A higher-level language undoubtedly benefits the ease of maintaining an application, but the higher the price that needs to be paid in terms of performance, the less eager developers are be to adopt it. A targeted investigation using the full application, one with more focus on productivity and software development practices, could reveal whether this trade-off would be beneficial for BUDE.

In addition, Kokkos is constantly expanding its support for existing programming models as parallelism backends, thus further increasing its reach on platforms: a SYCL backend is being added, while the existing — but experimental — OpenMP `target` and HIP backends begin to mature. A future study could revisit the performance of hand-tuned, low-level kernels versus implementations using future Kokkos versions.

Some applications, such as GROMACS, use hand-tuned vector intrinsics to maximise performance on supported architectures by manually packing vector registers. However, this comes at the cost of the need for manual changes whenever support for new architectures is to be added. With SVE, where each implementation can have a different native vector length, code changes would be required for each vector length, because even though SVE instructions are VLA, the higher level data structures used are not. One of the challenges of emerging programming models for HPC is to provide a performant abstraction of the underlying vector length, and future studies could investigate their performance across different implementations, comparing it to manual, intrinsic-based vectorisation.

CHAPTER 9

Conclusion

In this thesis I have explored modern high-performance processors by studying design decisions made in their architecture and hardware design, by evaluating how application users and developers can take advantage of this hardware in high-level programming, and by comparing their real-world performance across a set of benchmarks derived from — and representative of — HPC applications used in supercomputing centres around the world.

This work was carried out between 2017 and 2021, a period during which two instructions set extensions became important in mainstream CPUs for HPC: Intel's AVX-512 and Arm's SVE. These two vector instruction sets were key in achieving performance on this generation of processors, and so they set the context for the work presented in this thesis.

Chapter 3 described the context at the beginning of this period. It was the time when the first mainstream Arm-based processor made its way into the field of HPC, and the work undertaken then was among the first in the world that evaluated this new ThunderX2 processor from the perspective of HPC application performance. In doing so, I identified several points for improvement in the design of the TX2, some of which constituted strong points of the upcoming SVE instruction set.

Then, in Chapters 4 and 5 I studied the potential benefits of using SVE in the context of a similar set of scientific workloads. At the time, no SVE hardware had been released, so a combination of static and dynamics analysis on emulated execution was used to identify key areas in which the addition

of SVE could change the execution characteristics of a given processor on these scientific workloads. I found that many of the applications benefited from the increased vector length, as well as from the higher flexibility of the SVE instructions set compared to NEON. For example, the predication in SVE allowed parts of the code to be vectorised when under NEON they were not. On the other hand, I found that other aspects of the processor, such as the cache hierarchy, needs to be carefully designed together with the cores in order to enable performance benefits from these new vector instructions. This was particularly important in instructions such as gather loads and scatter stores, which have the potential to touch a large number of cache lines in a single instruction.

In Chapter 6 I evaluated the first hardware implementation of SVE, the Fujitsu A64FX. This is the same processor used in the Japanese supercomputer Fugaku and in addition to SVE, it utilises high-bandwidth HBM2 memory. Around the same time as the A64FX, two other Arm-based processors were launched for use in HPC: the Ampere Altra and the Amazon Web Services (AWS) Graviton 2, both based on the Neoverse N1 core. The N1-based processors took a different approach to achieving high-performance compared to the A64FX: they offered more cores and higher clock speeds, at the cost of narrower vectors. By comparing these two kinds of processors, as well as their main competitors form Intel and AMD, I obtained valuable insight into the types of workloads that they are best suited for. While the A64FX has impressive peak performance figures, I found that in practice it relies on an advanced optimising compiler with an accurate cost model to reach a high fraction of that peak. This was the case with the Fujitsu compiler, but not always with the other compilers, which sometimes put the A64FX behind the N1-based alternatives, designs for which it was relatively easier to generate optimal code.

Finally, Chapter 7 focused on the software aspects of programming HPC machines rather than the hardware. With the upcoming big exascale systems utilising a varied set of CPUs and GPUs, parallel programming frameworks that are vendor-agnostic and portable between systems are more important than ever. These frameworks generally work at a higher level compared

to traditional choices like MPI and OpenMP, and they promise increased programmer productivity, but they may sacrifice performance in doing so. Such frameworks had previously been evaluated in the context of memory-bandwidth-bound workloads, but my work used a newly developed mini-app to study them in a compute-bound setting. The two main frameworks studied were Kokkos and SYCL, and I analysed their performance on diverse hardware from Intel, AMD, NVIDIA, and several Arm-based vendors. I found that their performance varied with platform, and there was no single implementation that performed best on all the hardware available, but that in some cases it was possible to get performance close to that of proprietary APIs like CUDA. This chapter concluded by highlighting some weaknesses in these modern parallel programming frameworks at the time of writing, which could soon be resolved due to the constantly evolving nature of these projects.

Together, this thesis gives a comprehensive overview of modern HPC processors, the tools used to program them, and typical scientific workloads they often run. The main focus is on vectorisation, which at the time of writing is necessary to obtain high performance from these processors. By studying several contemporary architectures, I was able to identify key strengths and weaknesses of each, and in doing so raise considerations that may improve the future generations of HPC hardware.

Memory bandwidth remains critical for achieving high performance in scientific applications. A lot of the applications that are heavily utilised in large-scale HPC facilities are bound by memory bandwidth at least partly, and tools such as the roofline model have made it easy to expose how far an application is from reaching a platform's peak performance. However, the roofline model does not give any indication about the factors that may be preventing an application from reaching optimal performance, and in many cases the answer is non-trivial.

Recent hardware has focused on offering high memory bandwidth, through a combination of employing fast memory and a high number of channels to access it, as is the case of the Arm-based Marvell ThunderX2 and Fujitsu A64FX processors. Many applications benefit from these improvements, but

they also reach a point where other kinds of resources are needed to improve performance further. For example, the A64FX achieves good results in applications such as CloverLeaf and OpenFOAM, which are generally bound by memory bandwidth, but its weakness is its relatively small out-of-order backend and its high latency for complex mathematical operations. This has led to approaches such as the Ampere Altra — employing high core counts and large caches instead of fast memory and wide vectors — ultimately reaching higher performance compared to the A64FX in OpenFOAM, even though the Altra has less bandwidth to main memory. The same style of architecture can be found in the AMD EPYC Rome, which also performs well in a wide range of benchmarks compared to its contemporary HPC counterparts.

On the Intel platforms studied in this thesis, the wide vectors in AVX-512 come at the cost of reduced clock speed when running vector code, and to some extent the A64FX pays a similar price: while clock speed is not reduced directly, the latency of many operations, such as divisions, is high compared to x86-based alternatives, and the total amount of out-of-order resources is reduced. As a result, while the A64FX is a general-purpose CPU, in reality it is only suitable for selected classes of applications: those which are either heavily bandwidth- or compute-bound, but less so for mixed workloads that rely on the out-of-order backend to hide latency. This raises the question whether it is the very flexibility of SVE that induces a performance penalty on the implementation. Until other SVE microarchitectures become available, this remains an open question.

The VLA nature of SVE is also currently not exploited in the HPC space. While the more general-purpose compilers — GCC and LLVM — are able to general VLA code, the HPC-focused toolchains from Cray and Fujitsu chose to generate fixed-length vector code for maximum optimisation potential and minimal overhead. Those latter two compilers have generally been the best performing choices on the A64FX in my experiments, a result which suggests that VLA code may not bring a direct benefit to HPC users. An implication of this observation is that SVE does not help with portability either: if the best compilers hard-code the vector length, then the code will need to be recompiled when moving to a new target architecture anyway.

SVE offers predicated instructions, which allow more types of loops to be vectorised, in particular those with heavy branching. But on the A64FX, instructions to generate and manage predicates are slow, and many compilers disregard that. With GCC, for example, although loops are vectorised, in many cases the quality of the code generated can be low: a large amount of code is used, including many expensive predicate-related operations, and a good fraction of the vector lanes available can remain unused. In practice, this leads to vector code that does not perform significantly faster than its scalar counterpart.

The same challenge of utilising vector instruction sets efficiently is faced, albeit at a different level, by high-level programming frameworks. Frameworks such as Kokkos and SYCL can help with programmer productivity, but they introduce yet another layer between the raw parallelism that needs to be exposed in application and the compiler's view of the code. This, again, conditions the performance reached on the compiler's ability to understand patterns and generate efficient machine code.

These issues are not specific to SVE, or indeed to any particular instruction set. By comparing modern Arm- and x86-based processors I was able to observe that the instruction set makes little difference to the performance of applications on modern hardware; what is significantly more important is how well a microarchitecture is optimised for a particular workload. Unfortunately, for the HPC community this implies that the details are hidden behind the surface and that it is hard to get to the roots of an application's true performance-limiting factor.

One possible way to address such investigations is through simulation. With access to simulation tools that are flexible enough to model contemporary microarchitecture at near-cycle-accurate levels, and assuming that their performance makes it feasible to run kernels of real workloads through a model, researchers should be able to identify the changes to a given microarchitecture that would allow a given program to run faster.

The other side of such experiments is access to real data. Simulation is useful for attempts to find solutions to a problem, but a separate set of tools is needed to correctly identify problems. Traditional profilers are too

coarse-grained to explain the performance effects seen on modern system, and hardware counters offer too much raw data that often lacks context. In order to provide a starting point for simulation experiments, efficient and structured access to system performance data is needed, a view which is rarely provided by contemporary performance analysis tools.

These challenges are core to the HPC field on the path to exascale and beyond. They are ongoing research topics, and their findings have the potential to redefine performance analysis as we know it, bridging the gap between software and hardware design. In turn, it is this co-design work that will shape tomorrow's high-performance systems and set new boundaries for what it possible in the world of computational science.

APPENDIX A

Data

This chapter lists the raw data used to produce the visualisations in this thesis.

A.1 Chapter 3: Emerging CPU Architectures for HPC

Table A.1: Data from Figure 3.1.

Application	Relative Performance		
	SMT-1	SMT-2	SMT-4
C1	1.000	1.218	1.379
C2	1.000	1.802	3.059
M1	1.000	1.031	1.032
M2	1.000	1.001	0.984

Table A.2: Data from Figure 3.4.

Benchmark	Relative Performance			
	BDW	TX2	SKL20	SKL28
STREAM	1.00	1.93	1.53	1.64
CloverLeaf	1.00	1.71	1.59	1.64
TeaLeaf	1.00	1.85	1.66	1.75
SNAP	1.00	0.71	1.17	1.48
Neutral	1.00	0.89	0.96	1.29

Table A.3: Data from Figure 3.5.

Benchmark	Relative Performance			
	BDW	TX2	SKL20	SKL28
CP2K	1.00	1.15	1.29	1.37
GROMACS	1.00	0.68	1.29	1.45
NAMD	1.00	1.16	0.98	1.21
NEMO	1.00	1.49	1.44	1.65
OpenFOAM	1.00	1.87	1.57	1.66
OpenSBLI	1.00	1.69	1.39	1.72
Unified Model	1.00	0.92	1.06	1.19
VASP	1.00	0.76	1.32	1.42
Geometric Mean	1.00	1.14	1.28	1.45

Table A.4: Data from Figure 3.6.

Benchmark	Relative Performance			
	Arm 18.4	CCE 8.7	GCC 7.2	GCC 8.1
STREAM	1.000	0.991	0.969	0.973
CloverLeaf	0.952	1.000	0.915	0.917
TeaLeaf	0.948	1.000	0.993	0.951
SNAP	0.962	1.000	0.791	0.856
Neutral	0.928	0.852	0.951	1.000

Table A.5: Data from Figure 3.8.

Benchmark	Relative Performance Improvement		
	Arm	CCE	GCC
STREAM	0.077	0.053	0.107
CloverLeaf	-0.022	0.013	-0.030
TeaLeaf	-0.033	-0.004	-0.022
SNAP	0.106	0.231	0.135
Neutral	-0.050	0.133	-0.006

A.2 Chapter 4: Next-Generation Vector Instruction Sets

Table A.6: Data from Figure 4.1: Instruction count, grouped by instruction type, for the STREAM benchmark.

Compiler	Vector Width	Op Group	Op Count
Arm 19.2	no-vec	A64	81790836
		NEON	0
	NEON	A64	13633394
		NEON	29360133
	SVE-128	A64	23070565
		NEON	0
		arithmetic	25165824
		control	12582914
		mem-read	20971526
		mem-write	12582912
		move	4
	SVE-256	A64	8389678
		NEON	0
		arithmetic	14680064
		control	8388612
		mem-read	12582918
		mem-write	8388608
		move	4
	SVE-512	A64	4195374
		NEON	0
		arithmetic	7340032
		control	4194308
		mem-read	6291462
		mem-write	4194304
		move	4
	SVE-1024	A64	2098222
		NEON	0
		arithmetic	3670016
		control	2097156
		mem-read	3145734
		mem-write	2097152
		move	4

	SVE-2048	A64	1049646
		NEON	0
		arithmetic	1835008
		control	1048580
		mem-read	1572870
		mem-write	1048576
		move	4
CCE 9.0a	no-vec	A64	74449203
		NEON	4194308
	NEON	A64	5767474
		NEON	33685513
	SVE-128	A64	46137796
		NEON	0
		arithmetic	12582913
		control	9
		mem-read	25165824
		mem-write	16777216
		move	4
	SVE-256	A64	31457672
		NEON	0
		arithmetic	6291456
		control	8
		mem-read	12582912
		mem-write	8388608
		move	4
	SVE-512	A64	11534780
		NEON	0
		arithmetic	3145728
		control	8
		mem-read	6291456
		mem-write	4194304
		move	4
	SVE-1024	A64	5767616
		NEON	0
		arithmetic	1572864
		control	8
		mem-read	3145728
		mem-write	2097152
		move	4
	SVE-2048	A64	2884026
		NEON	0
		arithmetic	786432
		control	8
		mem-read	1572864
		mem-write	1048576
		move	4
GCC 8.2	no-vec	A64	203425701
		NEON	0
	NEON	A64	60819356
		NEON	46137348
	SVE-128	A64	23070454
		NEON	0
		arithmetic	25165824
		control	12582924
		mem-read	20971520
		mem-write	12582912
		move	4
	SVE-256	A64	16778998
		NEON	0
		arithmetic	12582912
		control	6291468
		mem-read	10485760
		mem-write	6291456
		move	4
	SVE-512	A64	13633270
		NEON	0
		arithmetic	6291456

		control	3145740
		mem-read	5242880
		mem-write	3145728
		move	4
	SVE-1024	A64	12060406
		NEON	0
		arithmetic	3145728
		control	1572876
		mem-read	2621440
		mem-write	1572864
		move	4
	SVE-2048	A64	11273974
		NEON	0
		arithmetic	1572864
		control	786444
		mem-read	1310720
		mem-write	786432
		move	4

Table A.7: Data from Figure 4.2: Instruction count, grouped by instruction type, for the BUDE benchmark.

Compiler	SVE width	Op Group	Count
Arm 19.2	no-vec	A64	872858411
		NEON	2942608
	NEON	A64	32552425
		NEON	243800448
	SVE-128	A64	33564265
		NEON	272
		arithmetic	132066752
		control	31218736
		mem-read	26226848
		mem-write	6266624
		move	53492784
	SVE-256	A64	24186121
		NEON	0
		arithmetic	66047808
		control	15609536
		mem-read	13699360
		mem-write	3133440
		move	28529984
	SVE-512	A64	19497033
		NEON	0
		arithmetic	33038208
		control	7804800
		mem-read	7435616
		mem-write	1566720
		move	16048576
	SVE-1024	A64	17152489
		NEON	0
		arithmetic	16533408
		control	3902432
		mem-read	4303744
		mem-write	783360
		move	9807872
	SVE-2048	A64	15980217
		NEON	0
		arithmetic	8281008
		control	1951248
		mem-read	2737808
		mem-write	391680
		move	6687520
CCE 9.0a	no-vec	A64	841032266

		NEON	1836816
	NEON	A64	31343436
		NEON	237898882
	SVE-128	A64	38502189
		NEON	2342912
		arithmetic	175263552
		control	23802736
		mem-read	25054688
		mem-write	6263552
		move	39801472
	SVE-256	A64	38093901
		NEON	2342912
		arithmetic	87826880
		control	14437744
		mem-read	12527968
		mem-write	3131776
		move	19510496
	SVE-512	A64	21264669
		NEON	1562496
		arithmetic	44108768
		control	7023792
		mem-read	4704592
		mem-write	5888
		move	13267184
	SVE-1024	A64	26287869
		NEON	2342912
		arithmetic	22249488
		control	4292336
		mem-read	3133312
		mem-write	783328
		move	8584640
	SVE-2048	A64	18483213
		NEON	2342912
		arithmetic	11319904
		control	1951088
		mem-read	2344784
		mem-write	1360
		move	7024656
GCC 8.2	no-vec	A64	894082928
		NEON	32
	NEON	A64	31624341
		NEON	235867024
	SVE-128	A64	25429749
		NEON	16
		arithmetic	139631984
		control	48526176
		mem-read	25835104
		mem-write	6263552
		move	18543584
	SVE-256	A64	22304517
		NEON	16
		arithmetic	69999520
		control	24458224
		mem-read	13308384
		mem-write	3131776
		move	11613664
	SVE-512	A64	20741893
		NEON	16
		arithmetic	35183280
		control	12424240
		mem-read	7045024
		mem-write	1565888
		move	8148704
	SVE-1024	A64	19960581
		NEON	16
		arithmetic	17775168
		control	6407248

		mem-read	3913344
		mem-write	782944
		move	6416224
	SVE-2048	A64	19569893
		NEON	16
		arithmetic	9071120
		control	3398752
		mem-read	2347504
		mem-write	391472
		move	5549984

Table A.8: Data from Figure 4.3: Instruction count, grouped by instruction type, for the TeaLeaf benchmark.

Compiler	SVE width	Op Group	Count
Arm 19.2	no-vec	A64	566705
		NEON	809
	NEON	A64	608015
		NEON	32529
	SVE-128	A64	431901
		NEON	809
		arithmetic	90502
		control	24062
		mem-read	52900
		mem-write	18840
		move	6060
	SVE-256	A64	411471
		NEON	809
		arithmetic	55232
		control	18772
		mem-read	32830
		mem-write	12460
		move	6060
	SVE-512	A64	401701
		NEON	809
		arithmetic	38262
		control	16242
		mem-read	23020
		mem-write	9440
		move	6060
	SVE-1024	A64	391931
		NEON	809
		arithmetic	21292
		control	13712
		mem-read	13210
		mem-write	6420
		move	6060
	SVE-2048	A64	391931
		NEON	809
		arithmetic	21292
		control	13712
		mem-read	13210
		mem-write	6420
		move	6060
CCE 9.0a	no-vec	A64	538609
		NEON	809
	NEON	A64	535119
		NEON	32959
	SVE-128	A64	487281
		NEON	395
		arithmetic	4470
		control	472
		mem-read	10260

		mem-write	6350
		move	692
	SVE-256	A64	495313
		NEON	4375
		arithmetic	1950
		control	602
		mem-read	4280
		mem-write	2520
		move	412
	SVE-512	A64	505681
		NEON	4315
		arithmetic	980
		control	432
		mem-read	1980
		mem-write	1200
		move	242
	SVE-1024	A64	510987
		NEON	17715
		control	420
		move	50
	SVE-2048	A64	510987
		NEON	17715
		control	420
		move	50
GCC 8.2	no-vec	A64	471989
		NEON	448
	NEON	A64	361169
		NEON	113948
	SVE-128	A64	298531
		NEON	458
		arithmetic	70030
		control	16064
		mem-read	49540
		mem-write	12320
		move	2172
	SVE-256	A64	294561
		NEON	458
		arithmetic	42680
		control	9734
		mem-read	29460
		mem-write	7260
		move	2172
	SVE-512	A64	292631
		NEON	458
		arithmetic	29610
		control	6624
		mem-read	19640
		mem-write	4840
		move	2172
	SVE-1024	A64	290701
		NEON	458
		arithmetic	16540
		control	3514
		mem-read	9820
		mem-write	2420
		move	2172
	SVE-2048	A64	290701
		NEON	458
		arithmetic	16540
		control	3514
		mem-read	9820
		mem-write	2420
		move	2172

Table A.9: Data from Figure 4.4: Instruction count, grouped by instruction type, for the CloverLeaf benchmark.

Compiler	SVE width	Op Group	Count
Arm 19.2	no-vec	A64	3346509743
		NEON	106
	NEON	A64	1464401200
		NEON	697897545
	SVE-128	A64	1119020917
		NEON	22119370
		arithmetic	566967920
		control	92218501
		mem-read	363577720
		mem-write	90067160
		move	27142489
	SVE-256	A64	943654985
		NEON	22119370
		arithmetic	284270492
		control	46507037
		mem-read	182118080
		mem-write	45239756
		move	14003929
	SVE-512	A64	856085705
		NEON	22119370
		arithmetic	142996892
		control	23666717
		mem-read	91461440
		mem-write	22854956
		move	7434649
	SVE-1024	A64	812301065
		NEON	22119370
		arithmetic	72360092
		control	12246557
		mem-read	46133120
		mem-write	11662556
		move	4150009
	SVE-2048	A64	790408745
		NEON	22119370
		arithmetic	37041692
		control	6536477
		mem-read	23468960
		mem-write	6066356
		move	2507689
CCE 9.0a	no-vec	A64	3664596437
		NEON	18484786
	NEON	A64	1610031731
		NEON	850984289
	SVE-128	A64	1611152950
		NEON	15478
		arithmetic	506356722
		control	3694790
		mem-read	376452725
		mem-write	84383753
		move	30240525
	SVE-256	A64	369700555
		NEON	104477
		arithmetic	427372755
		control	25855090
		mem-read	225134139
		mem-write	49509382
		move	37500345
	SVE-512	A64	157778898
		NEON	97860
		arithmetic	213630311

		control	12929074
		mem-read	113241865
		mem-write	25248274
		move	19261742
	SVE-1024	A64	66812444
		NEON	97932
		arithmetic	106837089
		control	6466988
		mem-read	56628083
		mem-write	12624428
		move	9719678
	SVE-2048	A64	44585521
		NEON	97932
		arithmetic	53440478
		control	3235957
		mem-read	28312552
		mem-write	6303877
		move	4953458
GCC 8.2	no-vec	A64	2768395189
		NEON	0
	NEON	A64	1452149717
		NEON	719359700
		other	4291888
	SVE-128	A64	771576273
		NEON	0
		arithmetic	739129116
		control	126761328
		mem-read	396585294
		mem-write	78637904
		mem-write	5027212
		move	2657266
		other	2215168
	SVE-256	A64	748004329
		NEON	0
		arithmetic	370180484
		control	63519304
		mem-read	198519574
		mem-write	39355492
		move	19002826
		other	16752208
	SVE-512	A64	736233769
		NEON	0
		arithmetic	185783204
	SVE-1024	A64	730348489
		NEON	0
		arithmetic	93584564
		control	16110904
		mem-read	50080054
		mem-write	9931252
		move	4992346
		move	37683466
		other	33365968
	SVE-2048	A64	727405849
		NEON	0
		arithmetic	47485244
		control	8209504
		mem-read	25340134
		control	31913704
		mem-read	99559894
		mem-write	19739332
		move	9662506
		other	8445328

Table A.10: Data from Figure 4.5: Instruction count, grouped by instruction type, for the MegaSweep benchmark.

Compiler	SVE width	Op Group	Count
Arm 19.2	no-vec	A64	44499487
		NEON	63
	NEON	A64	41316671
		NEON	278621
	SVE-128	A64	40381887
		NEON	93
		arithmetic	278536
		control	278552
		mem-read	278528
		mem-write	278528
		move	14352
	SVE-256	A64	40537023
		NEON	93
		arithmetic	139272
		control	845848
		mem-read	139264
		mem-write	487424
		move	4112
	SVE-512	A64	40293311
		NEON	93
		arithmetic	69640
		control	428056
		mem-read	69632
		mem-write	243712
		move	4112
	SVE-1024	A64	40178623
		NEON	93
		arithmetic	36872
		control	231448
		mem-read	36864
		mem-write	129024
		move	4112
	SVE-2048	A64	40121279
		NEON	93
		arithmetic	20488
		control	133144
		mem-read	20480
		mem-write	71680
		move	4112
CCE 9.0a	no-vec	A64	38287040
		NEON	1079432
	NEON	A64	2560241
		NEON	15065722
	SVE-128	A64	8831127
		NEON	84
		arithmetic	9478177
		control	39
		mem-read	5292088
		mem-write	3203080
		move	8235
	SVE-256	A64	4991871
		NEON	2762
		arithmetic	3907636
		control	68
		mem-read	2367581
		mem-write	1601598
		move	581650
	SVE-512	A64	3737146
		NEON	110
		arithmetic	2375726

		control	51
		mem-read	1323073
		mem-write	800784
		move	8239
	SVE-1024	A64	1675608
		NEON	1024110
		arithmetic	1122347
		control	48
		mem-read	624702
		mem-write	384013
		move	8239
	SVE-2048	A64	3014447
		NEON	2778
		arithmetic	466981
		control	53
		mem-read	279630
		mem-write	192047
		move	90138
GCC 8.2	no-vec	A64	42789673
		NEON	0
	NEON	A64	42789673
		NEON	0
	SVE-128	A64	42789673
		NEON	0
	SVE-256	A64	42789673
		NEON	0
	SVE-512	A64	42789673
		NEON	0
	SVE-1024	A64	42789673
		NEON	0
	SVE-2048	A64	42789673
		NEON	0

Table A.11: Data from Figure 4.6: Instruction count, grouped by instruction type, for the MiniFMM benchmark.

Compiler	SVE width	Op Group	Count
Arm 19.2	0	A64	1121089228
		NEON	3229585
	1	A64	613615202
		NEON	191393417
	128	A64	219799000
		NEON	3649331
		arithmetic	153764054
		control	30615324
		mem-read	33465596
		move	36132385
		other	11802479
	256	A64	204162239
		NEON	3649331
		arithmetic	95671567
		control	24635506
		mem-read	21047816
		mem-write	343608
		move	29555567
		other	8468962
	512	A64	196984708
		NEON	3649331
		arithmetic	68820746
		control	21581236
		mem-read	15397420
		mem-write	229072
		move	26718929

		other	7056363
	SVE-1024	A64	192161992
		NEON	3649331
		arithmetic	50921600
		control	19468892
		mem-read	11630876
		mem-write	114536
		move	24827595
		other	6114727
	SVE-2048	A64	192161992
		NEON	3649331
		arithmetic	50921600
		control	19468892
		mem-read	11630876
		mem-write	114536
		move	24827595
		other	6114727
GCC 8.2	no-vec	A64	917137860
		NEON	2678647
	NEON	A64	276657511
		NEON	180071076
	SVE-128	A64	195512059
		NEON	12369888
		arithmetic	164777476
		control	13542568
		mem-read	27918198
		move	10527192
		other	6702181
	SVE-256	A64	192522150
		NEON	12369888
		arithmetic	93019660
		control	7514110
		mem-read	15958562
		move	10502872
		other	3687952
	SVE-512	A64	191109551
		NEON	12369888
		arithmetic	59117284
		control	4666032
		mem-read	10308166
		move	10491432
		other	2263913
	SVE-1024	A64	190167915
		NEON	12369888
		arithmetic	36518020
		control	2766636
		mem-read	6541622
		move	10483370
		other	1314215
	SVE-2048	A64	190167915
		NEON	12369888
		arithmetic	36518020
		control	2766636
		mem-read	6541622
		move	10483370
		other	1314215

Table A.12: Data from Figure 4.7.

Compiler	SVE width	Active Bits	Accesses
Arm 19.2	128	32	4800000
Arm 19.2	128	64	4840000
Arm 19.2	128	96	5040000
Arm 19.2	128	128	92600000

GCC 8.2	128	32	4800000
GCC 8.2	128	64	4840000
GCC 8.2	128	96	5040000
GCC 8.2	128	128	92600000
Arm 19.2	256	32	2440000
Arm 19.2	256	64	2440000
Arm 19.2	256	96	2840000
Arm 19.2	256	128	2600000
Arm 19.2	256	160	2360000
Arm 19.2	256	192	2400000
Arm 19.2	256	224	2200000
Arm 19.2	256	256	41520000
GCC 8.2	256	32	2440000
GCC 8.2	256	64	2440000
GCC 8.2	256	96	2840000
GCC 8.2	256	128	2600000
GCC 8.2	256	160	2360000
GCC 8.2	256	192	2400000
GCC 8.2	256	224	2200000
GCC 8.2	256	256	41520000
Arm 19.2	512	32	1640000
Arm 19.2	512	64	1680000
Arm 19.2	512	96	2000000
Arm 19.2	512	128	2080000
Arm 19.2	512	160	1720000
Arm 19.2	512	192	1120000
Arm 19.2	512	224	1120000
Arm 19.2	512	256	1280000
Arm 19.2	512	288	800000
Arm 19.2	512	320	760000
Arm 19.2	512	352	840000
Arm 19.2	512	384	520000
Arm 19.2	512	416	640000
Arm 19.2	512	448	1280000
Arm 19.2	512	480	1080000
Arm 19.2	512	512	17160000
GCC 8.2	512	32	1640000
GCC 8.2	512	64	1680000
GCC 8.2	512	96	2000000
GCC 8.2	512	128	2080000
GCC 8.2	512	160	1720000
GCC 8.2	512	192	1120000
GCC 8.2	512	224	1120000
GCC 8.2	512	256	1280000
GCC 8.2	512	288	800000
GCC 8.2	512	320	760000
GCC 8.2	512	352	840000
GCC 8.2	512	384	520000
GCC 8.2	512	416	640000
GCC 8.2	512	448	1280000
GCC 8.2	512	480	1080000
GCC 8.2	512	512	17160000
Arm 19.2	1024	32	80000
Arm 19.2	1024	64	40000
Arm 19.2	1024	128	40000
Arm 19.2	1024	160	40000
Arm 19.2	1024	288	160000
Arm 19.2	1024	320	120000
Arm 19.2	1024	352	360000
Arm 19.2	1024	384	200000
Arm 19.2	1024	416	400000
Arm 19.2	1024	448	1280000
Arm 19.2	1024	480	920000
Arm 19.2	1024	512	1040000
Arm 19.2	1024	544	1560000
Arm 19.2	1024	576	1640000
Arm 19.2	1024	608	2000000

Arm 19.2	1024	640	2040000
Arm 19.2	1024	672	1680000
Arm 19.2	1024	704	1120000
Arm 19.2	1024	736	1120000
Arm 19.2	1024	768	1280000
Arm 19.2	1024	800	640000
Arm 19.2	1024	832	640000
Arm 19.2	1024	864	480000
Arm 19.2	1024	896	320000
Arm 19.2	1024	928	240000
Arm 19.2	1024	992	160000
Arm 19.2	1024	1024	600000
GCC 8.2	1024	32	80000
GCC 8.2	1024	64	40000
GCC 8.2	1024	128	40000
GCC 8.2	1024	160	40000
GCC 8.2	1024	288	160000
GCC 8.2	1024	320	120000
GCC 8.2	1024	352	360000
GCC 8.2	1024	384	200000
GCC 8.2	1024	416	400000
GCC 8.2	1024	448	1280000
GCC 8.2	1024	480	920000
GCC 8.2	1024	512	1040000
GCC 8.2	1024	544	1560000
GCC 8.2	1024	576	1640000
GCC 8.2	1024	608	2000000
GCC 8.2	1024	640	2040000
GCC 8.2	1024	672	1680000
GCC 8.2	1024	704	1120000
GCC 8.2	1024	736	1120000
GCC 8.2	1024	768	1280000
GCC 8.2	1024	800	640000
GCC 8.2	1024	832	640000
GCC 8.2	1024	864	480000
GCC 8.2	1024	896	320000
GCC 8.2	1024	928	240000
GCC 8.2	1024	992	160000
GCC 8.2	1024	1024	600000
Arm 19.2	2048	288	160000
Arm 19.2	2048	320	120000
Arm 19.2	2048	352	360000
Arm 19.2	2048	384	200000
Arm 19.2	2048	416	400000
Arm 19.2	2048	448	1280000
Arm 19.2	2048	480	920000
Arm 19.2	2048	512	1040000
Arm 19.2	2048	544	1560000
Arm 19.2	2048	576	1640000
Arm 19.2	2048	608	2000000
Arm 19.2	2048	640	2040000
Arm 19.2	2048	672	1680000
Arm 19.2	2048	704	1120000
Arm 19.2	2048	736	1120000
Arm 19.2	2048	768	1280000
Arm 19.2	2048	800	640000
Arm 19.2	2048	832	640000
Arm 19.2	2048	864	480000
Arm 19.2	2048	896	280000
Arm 19.2	2048	928	240000
Arm 19.2	2048	992	80000
Arm 19.2	2048	1024	40000
Arm 19.2	2048	1056	80000
Arm 19.2	2048	1088	40000
Arm 19.2	2048	1152	40000
Arm 19.2	2048	1184	40000
Arm 19.2	2048	1920	40000

Arm 19.2	2048	2016	80000
Arm 19.2	2048	2048	120000
GCC 8.2	2048	288	160000
GCC 8.2	2048	320	120000
GCC 8.2	2048	352	360000
GCC 8.2	2048	384	200000
GCC 8.2	2048	416	400000
GCC 8.2	2048	448	1280000
GCC 8.2	2048	480	920000
GCC 8.2	2048	512	1040000
GCC 8.2	2048	544	1560000
GCC 8.2	2048	576	1640000
GCC 8.2	2048	608	2000000
GCC 8.2	2048	640	2040000
GCC 8.2	2048	672	1680000
GCC 8.2	2048	704	1120000
GCC 8.2	2048	736	1120000
GCC 8.2	2048	768	1280000
GCC 8.2	2048	800	640000
GCC 8.2	2048	832	640000
GCC 8.2	2048	864	480000
GCC 8.2	2048	896	280000
GCC 8.2	2048	928	240000
GCC 8.2	2048	992	80000
GCC 8.2	2048	1024	40000
GCC 8.2	2048	1056	80000
GCC 8.2	2048	1088	40000
GCC 8.2	2048	1152	40000
GCC 8.2	2048	1184	40000
GCC 8.2	2048	1920	40000
GCC 8.2	2048	2016	80000
GCC 8.2	2048	2048	120000

A.3 Chapter 5: The Effects on Cache of Wide Vector Operations

The data from this chapter can also be found online[1].

[1] `https://github.com/UoB-HPC/cache-effects-reproducibility`

Table A.13: Data from Figures 5.6, 5.7, and 5.8: Cache miss rates at different SVE widths for the CloverLeaf, MegaSweep, and MiniFMM benchmarks, respectively, on the ThunderX2 and A64FX processors.

Benchmark	SVE Width (bits)				
	128	256	512	1024	2048
CloverLeaf TX2	16.28%	23.90%	32.68%	40.03%	45.11%
MegaSweep TX2	5.22%	6.32%	6.44%	6.51%	6.48%
MiniFMM TX2	0.277%	0.046%	0.052%	0.057%	0.351%
CloverLeaf A64FX	4.18 %	7.07%	10.98%	15.17%	18.74%
MegaSeep A64FX	1.16%	1.45%	1.48%	1.49%	1.51%
MiniFMM A64FX	0.046%	0.052%	0.057%	0.061%	0.060%

Table A.14: Data from Figure 5.9: Total number of non-contiguous accesses, grouped by the number of cache lines touched at each SVE width for the CloverLeaf benchmark.

SVE Width (bits)	Lines touched	Count TX2	Count A64FX
128	1	926785	926785
	2	13965014	13965014
256	1	464639	464639
	2	127964	7046099
	3	387	579
	4	6918327	0
512	1	233759	233759
	2	127964	127964
	3	0	3458774
	4	0	579
	7	386	0
	8	3458967	0
1024	1	118319	118319
	2	127964	127964
	5	0	1729094
	6	0	579
	15	386	0
	16	1729287	0
2048	1	60599	60599
	2	127964	127964
	9	0	864254
	10	0	579
	31	386	0
	32	864447	0

Table A.15: Data from Figure 5.10: Total number of non-contiguous accesses, grouped by the number of cache lines touched at each SVE width for the MiniFMM benchmark.

SVE Width (bits)	Lines touched	Count TX2	Count A64FX
128	2	1374432	1947112
	3	458144	0
	4	114536	0
256	2	0	1718040
	3	1374432	0
	4	229072	0
	5	114536	0
512	2	0	1718040
	3	1374432	0
	5	343608	0
1024	2	0	1718040
	3	1374432	0
	5	343608	0
2048	2	0	1718040
	3	1374432	0
	5	343608	0

A.4 Chapter 6: Next-Generation Vector Processors

Table A.16: Benchmark results for each compiler on each platform covered.

Benchmark	Platform	Compiler	Result	Result Type
BabelStream	A64FX	Arm	597.855	Triad GB/s
		Cray	596.297	Triad GB/s
		Fujitsu	824.222	Triad GB/s
		GNU	599.131	Triad GB/s
	Altra	Arm	385.748	Triad GB/s
		GNU	371.443	Triad GB/s
	CLX	Cray	162.142	Triad GB/s
		GNU	163.644	Triad GB/s
		Intel	204.347	Triad GB/s
	Graviton 2	Arm	175.177	Triad GB/s
		GNU	174.804	Triad GB/s
	Rome	Cray	285.240	Triad GB/s
		GNU	256.757	Triad GB/s
		Intel	284.947	Triad GB/s
	TX2	Arm	243.510	Triad GB/s
		Cray	235.888	Triad GB/s
		GNU	242.515	Triad GB/s
miniBUDE	A64FX	Arm	69.161	poses/ms
		Cray	101.203	poses/ms
		Fujitsu	168.469	poses/ms
		GNU	63.100	poses/ms
	Altra	Arm	435.709	poses/ms
		GNU	402.691	poses/ms
	CLX	Cray	285.405	poses/ms
		GNU	240.267	poses/ms

		Intel	222.685	poses/ms
	Graviton 2	Arm	142.902	poses/ms
		GNU	141.699	poses/ms
	Rome	Cray	530.539	poses/ms
		GNU	521.746	poses/ms
		Intel	420.776	poses/ms
	TX2	Arm	144.761	poses/ms
		Cray	153.757	poses/ms
		GNU	119.453	poses/ms
CloverLeaf	A64FX	Arm	381.719	bm16 best time
		Cray	490.598	bm16 best time
		Fujitsu	146.308	bm16 best time
		GNU	1252.991	bm16 best time
	Altra	Arm	235.330	bm16 best time
		GNU	255.764	bm16 best time
	CLX	Cray	420.255	bm16 best time
		GNU	395.506	bm16 best time
		Intel	383.650	bm16 best time
	Graviton 2	Arm	443.768	bm16 best time
		GNU	445.980	bm16 best time
	Rome	Cray	254.626	bm16 best time
		GNU	255.414	bm16 best time
		Intel	257.806	bm16 best time
	TX2	Arm	384.516	bm16 best time
		Cray	353.200	bm16 best time
		GNU	402.733	bm16 best time
GROMACS	A64FX	Arm	3473.007	nonbonded-benchmark pairs/usec
		Cray	3126.500	nonbonded-benchmark pairs/usec
		Fujitsu	4339.852	nonbonded-benchmark pairs/usec
		GNU	8738.046	nonbonded-benchmark pairs/usec
	Altra	Arm	13387.540	nonbonded-benchmark pairs/usec
		GNU	13647.000	nonbonded-benchmark pairs/usec

	CLX	Cray	28152.190	nonbonded-benchmark pairs/usec
		GNU	28233.700	nonbonded-benchmark pairs/usec
		Intel	27976.160	nonbonded-benchmark pairs/usec
	Graviton 2	Arm	12641.010	nonbonded-benchmark pairs/usec
		GNU	13610.000	nonbonded-benchmark pairs/usec
	Rome	Cray	45740.350	nonbonded-benchmark pairs/usec
		GNU	46678.090	nonbonded-benchmark pairs/usec
		Intel	45932.590	nonbonded-benchmark pairs/usec
	TX2	Arm	7396.400	nonbonded-benchmark pairs/usec
		Cray	8245.378	nonbonded-benchmark pairs/usec
		GNU	8172.720	nonbonded-benchmark pairs/usec
	A64FX	Arm	14.555	ion_channel_vsites
		Cray	17.936	ion_channel_vsites
		Fujitsu	7.240	ion_channel_vsites
		GNU	21.800	ion_channel_vsites
	Altra	Arm	128.354	ion_channel_vsites
		GNU	89.165	ion_channel_vsites
	CLX	Cray	56.328	ion_channel_vsites
		GNU	59.559	ion_channel_vsites
		Intel	59.425	ion_channel_vsites
	Graviton 2	Arm	64.201	ion_channel_vsites
		GNU	38.972	ion_channel_vsites
	Rome	Cray	150.284	ion_channel_vsites
		GNU	184.554	ion_channel_vsites
		Intel	177.004	ion_channel_vsites
	TX2	Arm	49.716	ion_channel_vsites
		Cray	49.940	ion_channel_vsites
		GNU	53.856	ion_channel_vsites
MiniFMM	A64FX	Arm	20.828	omp-task plummer.in total time
		Cray	54.659	omp-task plummer.in total time
		Fujitsu	9.913	omp-task plummer.in total time
		GNU	9.368	omp-task plummer.in total time

	Altra	Arm	6.718	omp-task plummer.in total time
		GNU	6.424	omp-task plummer.in total time
	CLX	Cray	3.635	omp-task plummer.in total time
		GNU	5.606	omp-task plummer.in total time
		Intel	6.271	omp-task plummer.in total time
	Graviton 2	Arm	8.099	omp-task plummer.in total time
		GNU	8.037	omp-task plummer.in total time
	Rome	Cray	2.571	omp-task plummer.in total time
		GNU	3.094	omp-task plummer.in total time
		Intel	23.105	omp-task plummer.in total time
	TX2	Arm	12.638	omp-task plummer.in total time
		Cray	15.657	omp-task plummer.in total time
		GNU	8.261	omp-task plummer.in total time
OpenFOAM	A64FX	Arm	90.800	block_drivAer_small last-first
		Cray	—	block_drivAer_small last-first
		Fujitsu	78.960	block_drivAer_small last-first
		GNU	98.740	block_drivAer_small last-first
	Altra	Arm	57.420	block_drivAer_small last-first
		GNU	56.190	block_drivAer_small last-first
	CLX	Cray	—	block_drivAer_small last-first
		GNU	108.560	block_drivAer_small last-first
		Intel	121.020	block_drivAer_small last-first
	Graviton 2	Arm	117.810	block_drivAer_small last-first
		GNU	116.530	block_drivAer_small last-first
	Rome	Cray	—	block_drivAer_small last-first
		GNU	55.530	block_drivAer_small last-first
		Intel	61.740	block_drivAer_small last-first
	TX2	Arm	98.710	block_drivAer_small last-first
		Cray	—	block_drivAer_small last-first
		GNU	88.550	block_drivAer_small last-first
SPARTA	A64FX	Arm	419.405	in.collision
		Cray	526.440	in.collision

		Fujitsu	—	in.collision
		GNU	418.935	in.collision
	Altra	Arm	80.114	in.collision
		GNU	82.073	in.collision
	CLX	Cray	217.611	in.collision
		GNU	223.390	in.collision
		Intel	204.644	in.collision
	Graviton 2	Arm	209.973	in.collision
		GNU	212.241	in.collision
	Rome	Cray	87.713	in.collision
		GNU	80.342	in.collision
		Intel	75.748	in.collision
	TX2	Arm	190.748	in.collision
		Cray	206.272	in.collision
		GNU	189.551	in.collision
TeaLeaf	A64FX	Arm	108.781	bm5 best time
		Cray	94.155	bm5 best time
		Fujitsu	103.073	bm5 best time
		GNU	129.939	bm5 best time
	Altra	Arm	183.454	bm5 best time
		GNU	209.559	bm5 best time
	CLX	Cray	—	bm5 best time
		GNU	334.268	bm5 best time
		Intel	333.914	bm5 best time
	Graviton 2	Arm	423.467	bm5 best time
		GNU	423.107	bm5 best time
	Rome	Cray	—	bm5 best time
		GNU	152.921	bm5 best time
		Intel	159.383	bm5 best time
	TX2	Arm	323.483	bm5 best time
		Cray	298.428	bm5 best time
		GNU	318.932	bm5 best time

Table A.17: Data for Figure 6.11: Benchmark performance for different runtime configurations on the A64FX.

Benchmark	Compiler	MPI Ranks	Threads per Rank	Result	Result Type
CloverLeaf	Arm	48	1	455.68	Total time (s)
		4	12	450.10	Total time (s)
		1	48	381.71	Total time (s)
	Cray	48	1	510.71	Total time (s)
		4	12	493.89	Total time (s)
		1	48	490.59	Total time (s)
	Fujitsu	48	1	232.74	Total time (s)
		4	12	146.30	Total time (s)
		1	48	803.32	Total time (s)
	GNU	48	1	1287.34	Total time (s)
		4	12	1335.98	Total time (s)
		1	48	1252.99	Total time (s)
SPARTA	Arm	48	1	419.40	in.collision (s)
		8	6	677.59	in.collision (s)
		4	12	714.36	in.collision (s)
		1	48	1268.95	in.collision (s)
	Cray	48	1	526.44	in.collision (s)
		8	6	651.46	in.collision (s)
		4	12	737.09	in.collision (s)
		1	48	1426.21	in.collision (s)
	Fujitsu	48	1	—	in.collision (s)
		8	6	—	in.collision (s)
		4	12	—	in.collision (s)
		1	48	—	in.collision (s)
	GNU	48	1	418.93	in.collision (s)
		8	6	589.58	in.collision (s)
		4	12	669.97	in.collision (s)

		1	48	1398.97 in.collision (s)

A.5 Chapter 7: Programming Models for Modern HPC Architectures

Table A.18: MiniBUDE performance data on all platforms studied, grouped by programming model.

Platform (Type)	Model	Compiler	Time (s)
SKL-56 (CPU)	OpenMP	AMD	182.970
		Cray	211.044
		GNU	222.678
		Intel	263.990
		LLVM	241.380
	SYCL	hipSYCL	1728.783
		OneAPI	12841.087
		ComputeCPP	12963.504
	Kokkos	AMD	200.266
		Cray	208.063
		GNU	214.869
		Intel	214.833
		LLVM	210.120
CXL-40 (CPU)	OpenMP	AMD	231.790
		Cray	229.624
		GNU	272.763
		Intel	294.298
		LLVM	272.596
	SYCL	OneAPI	1004.624
		hipSYCL	1104.361
		ComputeCPP	734.199

	Kokkos	AMD	261.220
		Cray	227.071
		GNU	371.300
		Intel	408.479
		LLVM	260.832
Rome-128 (CPU)	OpenMP	AMD	122.329
		Cray	123.527
		GNU	125.609
		Intel	155.750
		LLVM	114.473
	SYCL	OneAPI	658.549
		hipSYCL	302.281
		ComputeCPP	186.615
	Kokkos	AMD	117.710
		Cray	104.537
		GNU	113.931
		Intel	174.453
		LLVM	97.632
TX2-64 (CPU)	OpenMP	Cray	587.478
		GNU	710.848
		Arm	644.690
		LLVM	548.110
	SYCL	hipSYCL	790.625
	Kokkos	Cray	534.943
		GNU	824.509
		Arm	564.525
		LLVM	570.137
A64FX-48 (CPU)	OpenMP	Cray	646.219
		GNU	1038.713
		Arm	947.355
		Fujitsu	388.728
		LLVM	1690.439

	SYCL	hipSYCL	3114.744
		hipSYCL	1168.000
	Kokkos	Cray	1761.795
		GNU	1056.854
		Arm	987.884
		Fujitsu	451.178
		LLVM	1605.578
V100	CUDA	NVCC	66.210
(GPU)	OpenCL	GNU	78.560
	OpenMP	Cray	150.430
	OpenACC	Cray	151.970
	Kokkos	GNU	164.348
	SYCL	hipSYCL	834.242
IrisPro580	SYCL	ComputeCPP	1364.646
(GPU)	SYCL	OneAPI	944.615
	Kokkos	Intel	3512.905
	OpenMP	Intel	4187.920
	OpenCL	GNU	1116.500
RadeonVII	OpenCL	GNU	140.310
(GPU)	CUDA	HIP	229.080
	OpenMP	AMD	427.530
		GNU	12780.590
	Kokkos	AMD	261.167
	SYCL	hipSYCL	232.463
	OpenACC	GNU	89743.520

APPENDIX B

Cache Simulator Design

This chapter presents the design and implementation of the cache simulator used in Chapter 5.

The simulator is written in C++17 and built using the Meson build system[1]. It runs on any platform for which a modern C++ compiler is available and has no external dependencies. The code, alongside simple build and run instructions, can be obtained from a git repository online[2].

B.1 The Main Loop

The cache simulator is used to investigate the behaviour of a given cache configuration on a given workload. In order to understand the strengths and weaknesses of each configuration, a common usage pattern is running several configurations side-by-side using the same workload and comparing their performance. The simulator was optimised for this use case: it assumes that a single application trace will generally be run on several cache configurations.

The main simulator program follows the following steps for each invocations:

1. Read an application trace;
2. Read one or more cache configurations;

[1] `https://mesonbuild.com/`
[2] `https://gitlab.com/andreipoe/sve-cache-simulator`

3. Run a simulation for each of the cache configurations using the provided application trace;
4. Output simulation data.

A diagram of the architecture

The following sections describe each of those steps.

Since the simulator is built on the assumption that a single trace will be run on several models, the simulation step is parallelised. Thus, if the cache configuration file contains more than one definition, all the configurations will be run in parallel, using an OpenMP work-sharing loop.

Each of the simulator's components are tested using unit tests implemented in the Catch2 framework[3]. Section B.6 discusses how the tests are implemented.

B.2 Reading Execution Traces

Each run of the simulator reads a single trace file. Traces can be read from text files directly obtained from ArmIE and are converted into an internal representation in the form of `MemoryTrace` objects.

An execution trace is a sequence of memory access requests. These are stored as `MemoryRequest` objects, simple data structures that represent the data obtained from ArmIE. The following parameters are recorded for each request:

- The ID of the thread that performed the request;
- The size of the request, in bytes;
- Whether the request is a read or a write;
- The base memory address for the request;
- The value of the program counter when the access was performed;
- Whether the request is part of an SVE *bundle*, a set of memory requests that together fulfil an SVE gather read or scatter store.

[3] `https://github.com/catchorg/Catch2`

The following is an example of memory trace:

```
214, 0, 0, 1, 64, 0xfffff48c5c88, 0x401e40
215, 0, 0, 0, 64, 0x428330, 0x401e84
```

For performance reasons, a `MemoryTrace` object stores the sequence of memory addresses that appear in the trace in a separate list, in addition to the list of `MemoryRequest`s. This allows fast iteration through addresses when the additional information is not needed.

B.2.1 Efficient Reading of Traces

Memory trace files can reach very large sizes even for small runs if the application performs many memory operations. For the workloads used in this thesis, the sizes of the trace files ranged from 500 MB for the compute-intensive applications to more than 10 GB for the memory-bandwidth-bound workloads, even when small input cases were used. Reading these large text files naively can be more expensive than the simulation part of the process, which is undesirable.

In order to optimise the trace reading process, the simulator can convert text trace files to binary files. The binary files hold the same information, but they occupy less space and are faster to parse, because they can be directly unpacked into `MemoryRequest` objects.

The simulator codebase contains a standalone executable that can convert ArmIE traces from text format to binary. Using this trace converter, the slow reading cost is incurred only once — when converting the trace — because subsequent executions of the simulator can directly use the binary trace.

One significant advantage of the binary format is that the extents of each `MemoryRequest` object are known, so the trace reading process can be parallelised. This is not easily achieved with a text format, because the length of each request depends on the addresses and sizes it uses. Using standard C++ `std::thread`s, binary trace files are split into chunks, taking care to only split on object boundaries. After all the threads have finished reading their part of the trace, the final sequence of memory accesses is reassembled.

In my experiments, this simple binary approach was 5× faster than any other parsing method, whether using the C++ STL, optimised regular expression libraries, plain C parsing using `strtok`, or even protocol buffers. Depending on the size of the trace file and the storage speed of the host used to run the simulator, the optimal number of I/O threads was between 4 and 16.

The simulator includes logic to automatically distinguish between text and binary trace files, so this process is completely transparent after the initial conversion step. The internal representation generated is the same regardless of the type of the input file, and it is ready to be run through cache models.

B.3 Cache Models

The simulator currently supports three types of caches:

- Idealised, infinite caches;
- Direct-mapped caches;
- Set-associative caches.

Most of the functionality is common to all three types, so each type is implemented as a subclass of the base abstract class `Cache`. All `Cache` objects track what memory addresses are currently cached and their locations, and whenever addresses are inserted or evicted, they collect counters on these events. The difference between the three types is in how new elements are inserted and old elements evicted: each type of cache has a different implementation of the `touch` method, which contains the logic executed with each memory access.

The basic block of a cache's internal structure is the `CacheEntry`, which represents the smallest element a cache can hold, a cache line. Each `CacheEntry` keeps a flags that shows whether it is currently active, its tag, and a timestamp showing when it was last set. Because programs run through ArmIE are not executed in a cycle-accurate environment, the timestamps are not cycle numbers, but rather counters of memory accesses: every memory operation

performed increments this counter by one. The actual values of the elements are not held, because the simulator is only concerned with data movement and never uses it in operations, so they are never used.

Modern processors have several levels of cache, and their interactions are key to understanding the whole system. To represent this, each configuration to be simulated is represented as a `CacheHierarchy` object. A `CacheHierarchy` holds a `Cache` object for each of its levels, plus counters for the traffic between consecutive pairs of levels and between the last level and main memory. `Cache` objects are not directly accessed from the main simulation loop; instead, the requests are given to the `CacheHierarchy` representing the configuration to be simulated, which will `touch` all the relevant levels of cache depending on whether they are involved in the request. For example, if a request hits the second level of cache, the third level will not receive a request.

The simulator's entry point constructs a `CacheHierarchy` for each configuration it is given, then in parallel runs through the sequence of memory accesses in the input trace and passes them to the hierarchies. Each `CacheHierarchy` start with its first level of cache, on which it calls `touch` using the request's address and size. The `Cache` object uses this information to compute the location to be accessed in the cache: a tag, index, and block, together bundled in a `CacheAddress` structure. Then, counters are recorded for the access, and if the operation is a write, the corresponding `CacheEntry` is updated. If the access was a hit, processing for the current request stops here; if it was a miss, it is repeated for the next level of cache, until one of the levels hits or main memory is reached. The `CacheHierarchy` records traffic whenever data is moved between levels, such as a read from a higher level of cache, or a write through.

It is possible that a single memory request touches multiple cache lines, e. g. when performing larger vector loads. In these cases, several `CacheAddresses` can be generated for a single request, and each is handled separately, as described above.

B.3.1 Capturing Simulation Data

Each level of cache, represented by a `Cache` object within a `CacheHierarchy`, records several counters as it simulates requests. The counters are stored in a structure called `CacheEvent`, which is updated every time a cache line is `touch`ed. The counters recorded are:

- Number of hits;
- Number of misses;
- Number of evictions;
- The *lifetime* of each cache line, i. e. the number of memory requests served between when the line was loaded and when it was evicted.

In addition to the counters recorded at each level, the `CacheHierarchy` records further counters that apply to the whole cache configuration, not just to an individual levels. These additional counters are:

- The amount of traffic, in bytes, between each two consecutive levels of cache, as well as between the last level and main memory;
- For each SVE bundle encountered: the number of times it was encountered, the number of individual cache operations required to service it, and the total amount of useful data it touches.

At the end of the main simulation loop, the `CacheHierarchy` collects the counters for each of its contained levels and outputs them in addition to its own counters.

B.4 Configuration Files

To describe the cache configurations that will be instantiated into `CacheHierarchy` objects, the simulator uses ini files. The configurations files have two components:

1. The key `levels`, describing how many levels this configuration is comprised of;

2. A section for each of the `levels`, conventionally named `L<n>` at level `n`, containing the parameters `type` and, where appropriate, `cache_size`, `line_size`, and `set_size`.

The following is an example of a configuration file:

```
[hierarchy]
levels = 2

[L1]
type = set_associative
cache_size = 4096
line_size = 64
set_size = 4

[L2]
type = set_associative
cache_size = 32768
line_size = 64
set_size = 4
```

A command-line flag can be used to run several configurations in parallel, in which case a separate ini file will be given for each configuration. This allows the user to define all their desired configurations in separate files, then choose which will be used for every run.

B.5 Simulator Output

When all the requests in a trace have been run through the cache model, the simulator requests all the counters from the `CacheHierarchy` and prints them to the screen. The `CacheHierarchy` will return both the counters it keeps itself, e. g. for traffic between levels, and the counters from each of its contained levels. This data can be output either in a "pretty" readable format or in CSV form; the former is the default when a single configuration is used,

while the latter is the default when two or more configurations are executed in parallel. Using the CSV output format and a batch of configuration files, the simulator can quickly produce data that can then be imported into other tools for post-processing or graphing.

B.6 Testing

The tests work by instantiating each main component, feeding it known input, and comparing its output against good known output:

- The trace reader is given a trace fragments, then the internal representation is compared against a known good state;
- The configuration file reader is given configurations in text format and the cache objects it instantiates are tested to have the same parameters as specified in the configurations;
- Each of the cache types is tested on known trace fragments to ensure all the metrics are counted correctly;

Then, randomised testing is employed to test the behaviour of the caches and to ensure that errors in trace files are dealt with gracefully. These work by generating workloads according to constraints, running them through the simulator, then checking that no unexpected behaviour has occurred in the simulation. For example, a pair of tests generates (randomised) trace fragments that should produce only cache hits or misses, respectively, which is checked in the output of the simulation. The Catch2 framework supports randomised testing that is also easy to reproduce in case of a failure, by reporting the full state of the tests that failed, so running a large number of such tests is a good way to increase the confidence in the simulator's correctness without manually producing trace files.

Finally, integration testing is employed to test the simulator as a whole. Rather than checking the internal state or output of individual components, the whole simulator is run over known input and its (text) output is checked for correctness.

Acronyms

ACfL Arm Compiler for Linux

ArmIE Arm Instruction Emulator

ArmPL Arm Performance Libraries

ASIMD Advanced SIMD

AVX Advanced Vector Extensions

AWS Amazon Web Services

BDW Broadwell

BLAS basic linear algebra subprograms

BUDE Bristol University Docking Engine

CCE Cray Compilation Environment

CFD Computational Fluid Dynamics

CG conjugate gradient

CLX Cascade Lake

CMG Core–Memory Group

CPU central processing unit

DPC++ Data-Parallel C++

DRAM dyamic random access memory

DSL domain-specific language

FFT fast Fourier transform

FFTW The Fastest Fourier Transform in the West

FLOP floating-point operation

FMA fused multiply–add

FPGA field-programmable gate arrays

GCC GNU Compiler Collection

GPGPU general-purpose graphics processing unit

GPU graphics processing unit

HPC High-Performance Computing

ISA instruction set architecture

ICC Intel C/C++ Compiler

LRU least-recently used

ML Machine Learning

MPI Message Passing Interface

NIC network interface card

NUMA non-uniform memory access

OneTBB OneAPI Threading Building Blocks

OPS Oxford Parallel library for Structured mesh solvers

PME Particle Mesh Ewald

SIMD single instruction, multiple data

SKL Skylake

SKU stock keeping unit

SLP superword-level parallelism

SMT simultaneous multithreading

SSE Streaming SIMD Extensions

SVE Scalable Vector Extension

TX2 ThunderX2

VLA vector-length-agnostic

References

[1] Mark James Abraham et al. 'GROMACS: High Performance Molecular Simulations Through Multi-Level Parallelism from Laptops to Supercomputers'. In: *SoftwareX* 1-2 (September 2015), pp. 19–25. ISSN: 23527110. DOI: 10.1016/j.softx.2015.06.001.

[2] Christie L. Alappat et al. 'ECM Modeling and Performance Tuning of SpMV and Lattice QCD on A64FX'. In: *CoRR* abs/2103.03013 (2021). arXiv: 2103.03013.

[3] Aksel Alpay and Vincent Heuveline. 'SYCL beyond OpenCL: The Architecture, Current State and Future Direction of HipSYCL'. In: *Proceedings of the International Workshop on OpenCL*. IWOCL '20. Munich, Germany: Association for Computing Machinery, 2020. ISBN: 9781450375313. DOI: 10.1145/3388333.3388658.

[4] Bob Alverson, Edwin Froese, Larry Kaplan and Duncan Roweth. *Cray XC Series Network*. White Paper WP-Aries01-1112. Cray Inc., 2012.

[5] Adrià Armejach et al. 'Using Arm's Scalable Vector Extension on Stencil Codes'. In: *The Journal of Supercomputing* (8 April 2019). ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-019-02842-5.

[6] Krste Asanović et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, December 2006.

[7] Patrick Atkinson and Simon McIntosh-Smith. 'On the Performance of Parallel Tasking Runtimes for an Irregular Fast Multipole Method Application'. In: *Scaling OpenMP for Exascale Performance and Portability*. Springer International Publishing, 2017, pp. 92–106. ISBN: 978-3-319-65578-9.

[8] Gergö Barany. 'Finding Missed Compiler Optimizations by Differential Testing'. In: ACM Press, 2018, pp. 82–92. ISBN: 978-1-4503-5644-2. DOI: 10.1145/3178372.3179521.

[9] R. F. Barrett et al. 'On the Role of Co-Design in High Performance Computing'. In: *Advances in Parallel Computing* 24 (January 2013), pp. 141–155. DOI: 10.3233/978-1-61499-324-7-141.

[10] David E. Bernholdt et al. 'A Survey of MPI Usage in the US Exascale Computing Project'. In: *Concurrency and Computation: Practice and Experience* 32.3 (2020), e4851.

[11] Iain Bethune, Fiona Reid and A. Lazzaro. 'CP2K Performance from Cray XT3 to XC30'. Presentation at the Cray User Group. Cray User Group. 2014.

[12] Robert F. Bird, Patrick Gillies, Michael R. Bareford, Andy Herdman and Stephen Jarvis. 'Performance Optimisation of Inertial Confinement Fusion Codes using Mini-Applications'. In: *The International Journal of High Performance Computing Applications* 32.4 (2018), pp. 570–581. DOI: 10.1177/1094342016670225.

[13] Alexandru Calotoiu et al. 'Lightweight Requirements Engineering for Exascale Co-Design'. In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2018, pp. 201–211.

[14] Richard Catlow, Scott Woodley, Nora De Leeuw and Andrew Turner. *Optimising the Performance of the VASP 5.2.2 Code on HECToR*. Tech. rep. HECToR, 2010.

[15] Aurélien Cavelan, Rubén M. Cabezón, Michal Grabarczyk and Florina M. Ciorba. 'A Smoothed Particle Hydrodynamics Mini-App for Exascale'. In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '20. Geneva, Switzerland: Association for Computing Machinery, 2020. ISBN: 9781450379939. DOI: 10.1145/3394277.3401855.

[16] Cavium. *ThunderX2 Block Diagram*. June 2018. URL: https://fuse.wikichip.org/wp-content/uploads/2018/06/cavium-thnderx2-block.png (visited on 21/03/2022).

[17] Cris Cecka. 'Low Communication FMM-Accelerated FFT on GPUs'. In: ACM Press, 2017, pp. 1–11. ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126919.

[18] Jacqueline Cherfils and Joël Janin. 'Protein Docking Algorithms: Simulating Molecular Recognition'. In: *Current Opinion in Structural Biology* 3.2 (1993), pp. 265–269. ISSN: 0959-440X. DOI: 10.1016/S0959-440X(05)80162-9.

[19] Patrick Crowley and Jean-Loup Baer. 'On the Use of Trace Sampling for Architectural Studies of Desktop Applications'. In: *Workload Characterization: Methodology and Case Studies. Based on the First Workshop on Workload Characterization*. 1998, pp. 15–24. DOI: 10.1109/WWC.1998.809355.

[20] Paul Stewart Crozier et al. *Improving Performance via Mini-Applications*. Tech. rep. SAND2009-5574, 993908. 1 September 2009. DOI: 10.2172/993908.

[21] M. T. Cruz, D. Ruiz and R. Rusitoru. 'Asvie: A Timing-Agnostic SVE Optimization Methodology'. In: *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. Denver, CO, USA, 2019, pp. 9–16. DOI: 10.1109/ProTools49597.2019.0000.

[22] Ian Cutress. *Managing 8 Rome CPUs in 1U: Cray's Shasta Direct Liquid Cooling*. HPC Wire. URL: https://www.anandtech.com/show/13616/managing-16-rome-cpus-in-1u-crays-shasta-direct-liquid-cooling (visited on 09/02/2021).

[23] Tom Deakin, Wayne Gaudin and Simon McIntosh-Smith. 'On the Mitigation of Cache-Hostile Memory Access Patterns on Many-Core CPU Architectures'. In: *High Performance Computing*. Springer International Publishing, 2017, pp. 348–362. ISBN: 978-3-319-67630-2.

[24] Tom Deakin and Simon McIntosh-Smith. 'Evaluating the Performance of HPC-Style SYCL Applications'. In: *Proceedings of the International Workshop on OpenCL*. IWOCL '20. Munich, Germany: Association for Computing Machinery, 2020. ISBN: 9781450375313. DOI: 10.1145/3388333.3388643.

[25] Tom Deakin, Simon McIntosh-Smith and Wayne Gaudin. 'Expressing Parallelism on Many-Core for Deterministic Discrete Ordinates Transport'. In: *2015 IEEE International Conference on Cluster Computing*. September 2015, pp. 729–737. DOI: 10.1109/CLUSTER.2015.127.

[26] Tom Deakin, Andrei Poenaru, Tom Lin and Simon McIntosh-Smith. 'Tracking Performance Portability on the Yellow Brick Road to Exascale'. In: 2020 IEEE/ACM International Workshop on Performance,

Portability and Productivity in HPC (P3HPC). Atlanta, GA, USA, 2020. In Press.

[27] Tom Deakin, James Price, Matthew Martineau and Simon McIntosh-Smith. 'GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors across Diverse Parallel Programming Models'. In: *International Conference on High Performance Computing*. Springer. 2016, pp. 489–507. ISBN: 978-3-319-46079-6. DOI: 10.1007/978-3-319-46079-6_34.

[28] Tom Deakin, James Price and Simon McIntosh-Smith. *Portable Methods for Measuring Cache Hierarchy Performance*. Poster presented at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC17). November 2017.

[29] Tom Deakin et al. 'Performance Portability across Diverse Computer Architectures'. In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Denver, CO, USA, 2019, pp. 1–13. DOI: 10.1109/P3HPC49587.2019.00006.

[30] Irina Demeshko et al. 'Toward Performance Portability of the Albany Finite Element Analysis Code Using the Kokkos Library'. In: *The International Journal of High Performance Computing Applications* (5 February 2018). ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342017749957.

[31] Jens Domke et al. 'Matrix Engines for High Performance Computing: A Paragon of Performance or Grasping at straws?' In: *arXiv preprint arXiv:2010.14373* (2020).

[32] H. Carter Edwards and Christian R. Trott. 'Kokkos: Enabling Performance Portability Across Manycore Architectures'. In: *2013 Extreme Scaling Workshop (XSW 2013)*. IEEE. 2013, pp. 18–24.

[33] Roger Espasa, Mateo Valero and James E. Smith. 'Vector Architectures: Past, Present and Future'. In: *Proceedings of the 12th International Conference on Supercomputing*. ICS '98. Melbourne, Australia: Association for Computing Machinery, 1998, pp. 425–432. ISBN: 089791998X. DOI: 10.1145/277830.277935.

[34] Ulrich Essmann et al. 'A Smooth Particle Mesh Ewald Method'. In: *The Journal of Chemical Physics* 103.19 (15 November 1995), pp. 8577–8593. ISSN: 0021-9606, 1089-7690. DOI: 10.1063/1.470117.

[35] Stijn Eyerman, Wim Heirman, Kristof Du Bois, Joshua B. Fryman and Ibrahim Hur. 'Many-Core Graph Workload Analysis'. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018, pp. 282–292. DOI: 10.1109/SC.2018.00025.

[36] Michael J. Flynn. 'Some Computer Organizations and Their Effectiveness'. In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[37] A. Fuchs and D. Wentzlaff. 'The Accelerator Wall: Limits of Chip Specialization'. In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 1–14. DOI: 10.1109/HPCA.2019.00023.

[38] Fujitsu. *A64FX Microarchitecture Manual*. Version 1.1. 28 April 2020.

[39] M. A. Gallis, J. R. Torczynski, S. J. Plimpton, D. J. Rader and T. Koehler. 'Direct Simulation Monte Carlo: The Quest for Speed'. In: *29th Intl Symposium on Rarefied Gas Dynamics*. Vol. 1628. 27. Xi'an, China: AIP Conference Proceedings, 2014.

[40] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman and Assefaw H. Gebremedhin. 'MiniVite: A Graph Analytics Benchmarking Tool for Massively Parallel Systems'. In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. Dallas, TX, USA: IEEE, November 2018, pp. 51–56. ISBN: 978-1-72810-182-8. DOI: 10.1109/PMBS.2018.8641631.

[41] Cosmin Gorgovan, Guillermo Callaghan and Mikel Luján. 'Balancing Performance and Productivity for the Development of Dynamic Binary Instrumentation Tools: A Case Study on Arm Systems'. In: *Proceedings of the 29th International Conference on Compiler Construction*. San Diego, CA, USA: ACM, 22 February 2020, pp. 132–142. ISBN: 978-1-4503-7120-9. DOI: 10.1145/3377555.3377895.

[42] William Gropp. 'MPICH2: A New Start for MPI Implementations'. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Dieter Kranzlmüller, Jens Volkert, Peter Kacsuk and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 7–7. ISBN: 978-3-540-45825-8.

[43] Gwen Voskuilen, Clay Hughes and Mengchi Zhang. 'Structural Simulation Toolkit (SST) Tutorial'. Presentation at PACT 2019. Almaty, Kazakhstan, August 2019.

[44] C.D. Hall. 'The UK Meteorological Office Climate Model: The AMIP Run and Recent Changes to Reduce the Systematic Errors'. In: *World Meteorological Organization* (1995), pp. 301–306.

[45] Jeff R. Hammond, Michael Kinsner and James Brodman. 'A Comparative Analysis of Kokkos and SYCL as Heterogeneous, Parallel Programming Models for C++ Applications'. In: *Proceedings of the International Workshop on OpenCL*. IWOCL'19. Boston, MA, USA: Association for Computing Machinery, 2019. ISBN: 9781450362306. DOI: 10.1145/3318170.3318193.

[46] S. Hammond, C. Vaughan and C. Hughes. 'Evaluating the Intel Skylake Xeon Processor for HPC Workloads'. In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. July 2018, pp. 342–349. DOI: 10.1109/HPCS.2018.00064.

[47] Simon D. Hammond. *Towards Accurate Application Characterization for Exascale (APEX)*. SAND2015–8051, 1221578. 1 September 2015. DOI: 10.2172/1221578.

[48] Simon D. Hammond et al. 'Evaluating the Marvell ThunderX2 Server Processor for HPC Workloads'. In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2019, pp. 416–423. DOI: 10.1109/HPCS48598.2019.9188171.

[49] S. L. Harrell et al. 'Effective Performance Portability'. In: *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2018, pp. 24–36. DOI: 10.1109/P3HPC.2018.00006.

[50] Angelina I. Heft, Thomas Indinger and Nikolaus A. Adams. *Introduction of a New Realistic Generic Car Model for Aerodynamic Investigations*. SAE Technical Paper, 2012. DOI: 10.4271/2012-01-0168.

[51] John L. Hennessy and David A. Patterson. 'A New Golden Age for Computer Architecture'. In: *Commun. ACM* 62.2 (January 2019), pp. 48–60. ISSN: 0001-0782. DOI: 10.1145/3282307.

[52] Michael A. Heroux et al. *ECP Software Technology Capability Assessment Report–Public*. Tech. rep. NNSA, 2020.

[53] Torsten Hoefler and Roberto Belli. 'Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236. DOI: 10.1145/2807591.2807644.

[54] Dan Andrei Iliescu and Francesco Petrogalli. *Arm Scalable Vector Extension and Application to Machine Learning*. White Paper.

[55] Intel. *Intel® oneAPI: A Unified X-Architecture Programming Model*. 2020. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi.html (visited on 16/12/2020).

[56] Christian T. Jacobs, Satya P. Jammy and Neil D. Sandham. 'OpenSBLI: A Framework for the Automated Derivation and Parallel Execution of Finite Difference Solvers on a Range of Computer Architectures'. In: *Journal of Computational Science* 18 (2017), pp. 12–23. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2016.11.001.

[57] Heike Jagode, Anthony Danalis, Hartwig Anzt and Jack Dongarra. 'PAPI Software-Defined Events for In-Depth Performance Analysis'. In: *The International Journal of High Performance Computing Applications* 33.6 (2019), pp. 1113–1127. DOI: 10.1177/1094342019846287.

[58] Hrvoje Jasak, Aleksandar Jemcov, Zeljko Tukovic et al. 'OpenFOAM: A C++ Library for Complex Physics Simulations'. In: *International workshop on coupled methods in numerical dynamics* (September 2007), pp. 1–20.

[59] J. Kalyanasundaram and Y. Simmhan. 'ARM Wrestling with Big Data: A Study of Commodity ARM64 Server for Big Data Workloads'. In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. Los Alamitos, CA, USA: IEEE Computer Society, December 2017, pp. 203–212. DOI: 10.1109/HiPC.2017.00032.

[60] Max P. Katz et al. 'Preparing Nuclear Astrophysics for Exascale'. In: *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2020)*. Atlanta, GA, USA, November 2020. In Press.

[61] Georgios Keramidas, Nikolaos Strikos and Stefanos Kaxiras. 'Multicore Cache Simulations Using Heterogeneous Computing on General Purpose and Graphics Processors'. In: *2011 14th Euromicro Conference on Digital System Design*. 2011, pp. 270–273. DOI: 10.1109/DSD.2011.38.

[62] V. V. Kindratenko et al. 'GPU Clusters for High-Performance Computing'. In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009, pp. 1–8. DOI: 10.1109/CLUSTR.2009.5289128.

[63] Yuetsu Kodama et al. 'Preliminary Performance Evaluation of Application Kernels Using ARM SVE with Multiple Vector Lengths'. In: *2017 IEEE International Conference on Cluster Computing*. IEEE. 2017, pp. 677–684.

[64] P. M. Kogge and T. J. Dysart. 'Using the TOP500 to Trace and Project Technology and Architecture Trends'. In: *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–11. DOI: 10.1145/2063384.2063421.

[65] Kazuhiko Komatsu et al. 'Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA'. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018, pp. 685–696. DOI: 10.1109/SC.2018.00057.

[66] JaeHyuk Kwack, Galen Arnold, Celso Mendes and Gregory H. Bauer. 'Roofline Analysis with Cray Performance Analysis Tools (CrayPat) and Roofline-based Performance Projections for a Future Architecture'. In: Cray User Group. Stockholm, 24 May 2018.

[67] Ignacio Laguna et al. 'A Large-Scale Study of MPI Usage in Open-Source HPC Applications'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356176.

[68] Jacob Lambert, Seyong Lee, Jeffrey S. Vetter and Allen Malony. 'CCAMP: An Integrated Translation and Optimization Framework for OpenACC and OpenMP'. In: *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2020)*. Atlanta, GA, USA, November 2020. In Press.

[69] Michael Larabel. *Benchmarking An ARM 96-Core Cavium ThunderX System*. Phronix. 28 February 2018. URL: https://www.phoronix.com/scan.php?page=article&item=cavium-thunderx-96core (visited on 19/05/2021).

[70] Fujitsu Limited. *Fujitsu Presents Post-K CPU Specifications*. 22 August 2018. URL: https://www.fujitsu.com/global/about/resources/news/press-releases/2018/0822-02.html (visited on 15/06/2021).

[71] Heng Lin et al. 'Shentu: Processing Multi-Trillion Edge Graphs on Millions of Cores in Seconds'. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 706–716. DOI: 10.1109/SC.2018.00059.

[72] Wei-Chen Lin, Tom Deakin and Simon McIntosh-Smith. 'On Measuring the Maturity of SYCL Implementations by Tracking Historical Performance Improvements'. In: *Proceedings of the International Workshop on OpenCL*. IWOCL '20. Association for Computing Machinery, 2021. In Press.

[73] David Raymond Lutz and Christopher Neal Hinds. 'High-Precision Anchored Accumulators for Reproducible Floating-Point Summation'. In: IEEE, July 2017, pp. 98–105. ISBN: 978-1-5386-1965-0. DOI: 10.1109/ARITH.2017.20.

[74] Gurvan Madec. 'NEMO Reference Manual, Ocean Dynamics Component: NEMO-OPA'. In: *Preliminary version. Note du Pole de modélisation, Institut Pierre-Simon Laplace (IPSL), France* 27 (2008), pp. 1288–161.

[75] Saeed Maleki, Yaoqing Gao, Maria J. Garzar'n, Tommy Wong and David A. Padua. 'An Evaluation of Vectorizing Compilers'. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. Galveston, TX, USA: IEEE, October 2011, pp. 372–382. ISBN: 978-1-4577-1794-9. DOI: 10.1109/PACT.2011.68.

[76] A. C. Mallinson et al. 'CloverLeaf: Preparing Hydrodynamics Codes for Exascale'. In: *Cray User Group*. Napa Valley, California, USA, May 2013.

[77] Jahanzeb Maqbool, Sangyoon Oh and Geoffrey C. Fox. 'Evaluating ARM HPC Clusters for Scientific Workloads'. In: *Concurrency and Computation: Practice and Experience* 27.17 (2015), pp. 5390–5410.

[78] Matthew Martineau, Patrick Atkinson and Simon McIntosh-Smith. 'Benchmarking the NVIDIA V100 GPU and Tensor Cores'. In: *Euro-Par 2018: Parallel Processing Workshops: Euro-Par 2018 International Workshops*. HeteroPar 2018. Springer. Turin, Italy, 2018, pp. 444–456.

[79] Matthew Martineau and Simon McIntosh-Smith. 'Exploring On-Node Parallelism with Neutral, a Monte Carlo Neutral Particle Transport Mini-App'. In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. September 2017, pp. 498–508. DOI: 10.1109/CLUSTER.2017.83.

[80] Matthew Martineau, James Price, Simon McIntosh-Smith and Wayne Gaudin. 'Pragmatic Performance Portability with OpenMP 4.x'. In: *OpenMP: Memory, Devices, and Tasks*. Ed. by Naoya Maruyama, Bronis R. de Supinski and Mohamed Wahib. Vol. 9903. Cham: Springer International Publishing, 2016, pp. 253–267. ISBN: 978-3-319-45550-1.

[81] Timothy G. Mattson, Beverly Sanders and Berna Massingill. *Patterns for parallel programming*. Pearson Education, 2004. ISBN: 0321228111.

[82] John D. McCalpin. 'HPL and DGEMM Performance Variability on the Xeon Platinum 8160 Processor'. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018, pp. 225–237. DOI: 10.1109/SC.2018.00021.

[83] John D. McCalpin. 'Memory Bandwidth and Machine Balance in Current High Performance Computers'. In: *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2.19–25 (1995).

[84] Simon McIntosh-Smith. 'Enabling Processor Design Space Exploration with SimEng'. Presentation at ModSim: Workshop on Modeling and Simulation of Systems and Applications. ModSim: Workshop on Modeling and Simulation of Systems and Applications (2019). Seattle, WA, August 2019.

[85] Simon McIntosh-Smith, Michael Boulton, Dan Curran and James Price. 'On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures'. In: *Supercomputing*. Lecture Notes in Computer Science 8488 (2014). Ed. by Julian Martin Kunkel, Thomas Ludwig and Hans Werner Meuer, pp. 53–75. DOI: 10.1007/978-3-319-07518-1.

[86] Simon McIntosh-Smith, Jack Jones, Harry Waugh and Andrei Poenaru. 'SimEng: A Fast, Easy-To-Use, Open-Source Processor Simulation Framework'. In: ModSim 2021: Workshop on Modeling and Simulation of Systems and Applications. Seattle, WA, USA, 2021. In Review.

[87] Simon McIntosh-Smith, James Price, Tom Deakin and Andrei Poenaru. 'A Performance Analysis of the First Generation of HPC-Optimized Arm Processors'. In: *Concurrency and Computation: Practice and Experience* 31.16 (2019), e5110. DOI: 10.1002/cpe.5110.

[88] Simon McIntosh-Smith, James Price, Andrei Poenaru and Tom Deakin. 'Benchmarking the First Generation of Production-Quality Arm-Based Supercomputers'. In: *Concurrency and Computation: Practice and Experience* (2019), e5569. DOI: 10.1002/cpe.5569.

[89] Simon McIntosh-Smith, James Price, Richard B. Sessions and Amaurys A. Ibarra. 'High Performance In Silico Virtual Drug Screening on Many-core Processors'. In: *The International Journal of High Performance Computing Applications* 29.2 (2015), pp. 119–134. DOI: 10.1177/1094342014528252.

[90] Simon McIntosh-Smith et al. 'TeaLeaf: A Mini-Application to Enable Design-Space Explorations for Iterative Sparse Linear Solvers'. In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, September 2017, pp. 842–849. ISBN: 978-1-5386-2326-8. DOI: 10.1109/CLUSTER.2017.105.

[91] OE Bronson Messer et al. 'MiniApps Derived from Production HPC Applications Using Multiple Programing Models'. In: *The International Journal of High Performance Computing Applications* 32.4 (2018), pp. 582–593. DOI: 10.1177/1094342016668241.

[92] P. Messina. 'The Exascale Computing Project'. In: *Computing in Science Engineering* 19.3 (2017), pp. 63–67. DOI: 10.1109/MCSE.2017.57.

[93] Nils Meyer, Peter Georg, Dirk Pleiter, Stefan Solbrig and Tilo Wettig. 'SVE-Enabling Lattice QCD Codes'. In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 2018, pp. 623–628. DOI: 10.1109/CLUSTER.2018.00079.

[94] Leo A. Meyerovich and Ariel S. Rabkin. 'Empirical Analysis of Programming Language Adoption'. In: *SIGPLAN Not.* 48.10 (October 2013), pp. 1–18. ISSN: 0362-1340. DOI: 10.1145/2544173.2509515.

[95] Richard Tran Mills et al. 'Toward Performance-Portable PETSc for GPU-based Exascale Systems'. In: *arXiv preprint arXiv:2011.00715* (2020).

[96] Sparsh Mittal and Shraiysh Vaishay. 'A Survey of Techniques for Optimizing Deep Learning on GPUs'. In: *Journal of Systems Architecture* 99 (2019). ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2019.101635.

[97] NVIDIA. *NVIDIA Announces CPU for Giant AI and High Performance Computing Workloads*. 12 April 2021. URL: https://nvidianews.nvidia.com/news/nvidia-announces-cpu-for-giant-ai-and-high-performance-computing-workloads (visited on 09/05/2021).

[98] Tetsuya Odajima, Yuetsu Kodama and Mitsuhisa Sato. 'Performance and Power Consumption Analysis of Arm Scalable Vector Extension'. In: *The Journal of Supercomputing* (10 November 2020). ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-020-03495-5.

[99] Szilárd Páll and Berk Hess. 'A Flexible Algorithm for Calculating Pair Interactions on SIMD Architectures'. In: *Computer Physics Communications* 184.12 (2013), pp. 2641–2650. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2013.06.003.

[100] Kevin Pedretti et al. 'Chronicles of Astra: Challenges and Lessons from the First Petascale Arm Supercomputer'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[101] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy and S. A. Jarvis. 'Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors'. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. May 2013, pp. 1085–1097. DOI: 10.1109/IPDPS.2013.44.

[102] S. J. Pennycook, J. D. Sewall and V. W. Lee. 'Implications of a Metric for Performance Portability'. In: *Future Generation Computer Systems* 92 (2019), pp. 947–958. ISSN: 0167-739X. DOI: 10.1016/j.future.2017.08.007.

[103] Simon J. Pennycook et al. 'Evaluating the Impact of Proposed OpenMP 5.0 Features on Performance, Portability and Productivity'. In: (2018), pp. 37–46. DOI: 10.1109/P3HPC.2018.00007.

[104] Andrei Poenaru. 'A Comprehensive Study of the Effectiveness of Contemporary Vector Instruction Sets on Modern Scientic Codes'. Master's Thesis. University of Bristol, UK, 2017.

[105] Andrei Poenaru, Tom Deakin, Simon McIntosh-Smith, Simon D. Hammond and Andrew J. Younge. 'An Evaluation of the Fujitsu A64FX for HPC Applications'. In: Cray User Group. May 2021. In Press.

[106] Andrei Poenaru, Wei-Chen Lin and Simon McIntosh-Smith. 'A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application'. In: *High Performance Computing. 36th International Conference, ISC High Performance 2021*. Ed. by Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief and Piotr Luszczek. Cham: Springer International Publishing, 2021, pp. 332–350. ISBN: 978-3-030-78713-4. DOI: 10.1007/978-3-030-78713-4_18.

[107] Andrei Poenaru and Simon McIntosh-Smith. 'Evaluating the Effectiveness of a Vector-Length-Agnostic Instruction Set'. In: *Euro-Par 2020: Parallel Processing*. Euro-Par 2020 (Warsaw, Poland, 24–28 August 2020). Ed. by Maciej Malawski and Krzysztof Rzadca. Cham: Springer International Publishing, 2020, pp. 98–114.

[108] Andrei Poenaru and Simon McIntosh-Smith. 'The Effects of Wide Vector Operations on Processor Caches'. In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 2020, pp. 531–539. DOI: 10.1109/CLUSTER49012.2020.00076.

[109] Angela Pohl, Biagio Cosenza and Ben Juurlink. 'Portable Cost Modeling for Auto-Vectorizers'. In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2019, pp. 359–369.

[110] James Price and Simon McIntosh-Smith. 'Exploiting Auto-tuning to Analyze and Improve Performance Portability on Many-Core Architectures'. In: *High Performance Computing*. Ed. by Julian M. Kunkel, Rio Yokota, Michela Taufer and John Shalf. Cham: Springer International Publishing, 2017, pp. 538–556. ISBN: 978-3-319-67630-2.

[111] James Price and Simon McIntosh-Smith. 'Improving Auto-Tuning Convergence Times with Dynamically Generated Predictive Performance Models'. In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip*. IEEE. 2015, pp. 211–218. DOI: 10.1109/MCSoC.2015.31.

[112] Nikola Rajovic, Pall Carpenter, Isaac Gelado, Nikola Puzovic and Alex Ramirez. 'Are Mobile Processors Ready for HPC?' In: *IEEE/ACM Supercomputing Conference*. 2013.

[113] Karthik Raman, Tom Deakin, James Price and Simon McIntosh-Smith. 'Improving Achieved Memory Bandwidth from C++ Codes on Intel Xeon Phi Processor (Knights Landing)'. Presentation at the International Xeon Phi User Group Spring Meeting. Cambridge, UK, April 2017.

[114] Ruyman Reyes, Gordon Brown, Rod Burns and Michael Wong. 'SYCL 2020: More than Meets the Eye'. In: *Proceedings of the International Workshop on OpenCL*. IWOCL '20. Munich, Germany: Association for Computing Machinery, 2020. ISBN: 9781450375313. DOI: 10.1145/3388333.3388649.

[115] Alejandro Rico, José A. Joao, Chris Adeniyi-Jones and Eric Van Hensbergen. 'ARM HPC Ecosystem and the Reemergence of Vectors'. In: *Proceedings of the Computing Frontiers Conference*. CF'17. Siena, Italy: Association for Computing Machinery, 2017, pp. 329–334. ISBN: 9781450344876. DOI: 10.1145/3075564.3095086.

[116] Alejandro Rico et al. 'Trace-Driven Simulation of Multithreaded Applications'. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Austin, TX, USA: IEEE, April 2011, pp. 87–96. ISBN: 978-1-61284-367-4. DOI: 10.1109/ISPASS.2011.5762718.

[117] Richard M. Russell. 'The CRAY-1 Computer System'. In: *Communications of the ACM* 21.1 (1978), pp. 63–72. ISSN: 0001-0782. DOI: 10.1145/359327.359336.

[118] Karthik Sangaiah et al. 'SynchroTrace: Synchronization-Aware Architecture-Agnostic Traces for Lightweight Multicore Simulation of CMP and HPC Workloads'. In: *ACM Transactions on Architecture and Code Optimization* 15.1 (2 April 2018), pp. 1–26. ISSN: 1544-3566, 1544-3973. DOI: 10.1145/3158642.

[119] M. Sato. 'The Supercomputer "Fugaku" and Arm-SVE Enabled A64FX Processor for Energy Efficiency and Sustained Application Performance'. In: *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*. 2020, pp. 1–5. DOI: 10.1109/ISPDC51135.2020.00009.

[120] Mitsuhisa Sato et al. 'Co-Design for A64FX Manycore Processor and "Fugaku"'. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020, pp. 1–15. DOI: 10.1109/SC41405.2020.00051.

[121] David Schor. *Marvell Lays Out ARM Server Roadmap*. WikiChip Fuse. 9 November 2019. URL: https://fuse.wikichip.org/news/2956/marvell-lays-out-arm-server-roadmap (visited on 10/12/2019).

[122] David Schor. *The x86 Advanced Matrix Extension (AMX) Brings Matrix Operations*. WikiChip Fuse. 29 April 2020. URL: https://fuse.wikichip.org/news/3600 (visited on 14/04/2021).

[123] Jason Sewall, John S. Pennycook, Douglas Jacobsen, Tom Deakin and Simon McIntosh-Smith. 'Interpreting and Visualizing Performance Portability Metrics'. In: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). Atlanta, GA, USA, 2020. In Press.

[124] Andrew Siegel. 'ECP: Lessons Learned in Porting Complex Applications to Accelerator-based Systems'. Presentation at the 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). Atlanta, GA, USA, November 2020.

[125] Codeplay Software. *ComputeCPP*. URL: https://developer.codeplay.com/products/computecpp/ce/home (visited on 16/12/2020).

[126] StackOverflow. *2020 Developer Survey*. 2020. URL: https://insights.stackoverflow.com/survey/2020 (visited on 10/06/2021).

[127] Nigel Stephens. 'ARMv8-A Next-Generation Vector Architecture for HPC'. Presentation at Hot Chips 28. Hot Chips 28. Cupertino, CA, 22 August 2016.

[128] Nigel Stephens et al. 'The ARM Scalable Vector Extension'. In: *IEEE Micro* 37.2 (2017), pp. 26–39. DOI: 10.1109/MM.2017.35.

[129] Lynd Stringer. *Vectors: How the Old Became New Again in Supercomputing*. HPC Wire. 26 September 2016. URL: https://www.hpcwire.com/2016/09/26/vectors (visited on 11/12/2019).

[130] Tianjiao Sun et al. 'A Study of Vectorization for Matrix-Free Finite Element Methods'. In: *The International Journal of High Performance Computing Applications* (31 July 2020). ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342020945005.

[131] Neil Thompson and Svenja Spanuth. 'The Decline of Computers As a General Purpose Technology: Why Deep Learning and the End of Moore's Law are Fragmenting Computing'. In: *SSRN Electronic Journal* (20 November 2018). ISSN: 1556-5068. DOI: 10.2139/ssrn.3287769.

[132] TOP500, ed. *TOP500 Historical Charts: Development Over Time*. 2021. URL: https://www.top500.org/statistics/overtime (visited on 14/04/2021).

[133] TOP500, ed. *TOP500 List Statistics*. 2021. URL: https://www.top500.org/statistics/list (visited on 10/06/2021).

[134] Tiffany Trader. *Cray, Fujitsu Both Bringing Fujitsu A64FX-based Supercomputers to Market in 2020*. HPC Wire. 12 November 2019. URL: https://www.hpcwire.com/2019/11/12/cray (visited on 10/12/2019).

[135] Jan Treibig, Georg Hager and Gerhard Wellein. 'LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments'. In: *2010 39th International Conference on Parallel Processing Workshops*. 2010, pp. 207–216. DOI: 10.1109/ICPPW.2010.38.

[136] James D. Trotter, Johannes Langguth and Xing Cai. 'Cache Simulation for Irregular Memory Traffic on Multi-Core CPUs: Case Study on Performance Models for Sparse Matrix–Vector Multiplication'. In: *Journal of Parallel and Distributed Computing* 144 (2020), pp. 189–205. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.05.020.

[137] Andy Turner and Simon McIntosh-Smith. 'A Survey of Application Memory Usage on a National supercomputer: An Analysis of Memory Requirements on ARCHER'. In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Ed. by Stephen Jarvis, Steven Wright and Simon Hammond. Cham: Springer International Publishing, 2018, pp. 250–260. ISBN: 978-3-319-72971-8.

[138] Joost VandeVondele et al. 'Quickstep: Fast and Accurate Density Functional Calculations Using a Mixed Gaussian and Plane Waves Approach'. In: *Computer Physics Communications* 167.2 (2005), pp. 103–128. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2004.12.014.

[139] C. T. Vaughan et al. 'On the Use of Vectorization in Production Engineering Workloads'. In: Cray User Group. Stockholm, 24 May 2018.

[140] Pepe Vila, Pierre Ganty, Marco Guarnieri and Boris Köpf. 'CacheQuery: Learning Replacement Policies from Hardware Caches'. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 519–532. ISBN: 9781450376136. DOI: 10.1145/3385412.3386008.

[141] D. Walters et al. 'The Met Office Unified Model Global Atmosphere 6.0/6.1 and JULES Global Land 6.0/6.1 configurations'. In: *Geoscientific Model Development* 10.4 (2017), pp. 1487–1520. DOI: 10.5194/gmd-10-1487-2017.

[142] Hannes Weisbach, Balazs Gerofi, Brian Kocoloski, Hermann Härtig and Yutaka Ishikawa. 'Hardware Performance Variation: A Comparative Study Using Lightweight Kernels'. In: *High Performance Computing*. Ed. by Rio Yokota, Michèle Weiland, David Keyes and Carsten Trinitis. Vol. 10876. Cham: Springer International Publishing, 2018, pp. 246–265. ISBN: 978-3-319-92040-5. DOI: 10.1007/978-3-319-92040-5_13.

[143] Shasha Wen, Xu Liu, John Byrne and Milind Chabbi. 'Watching for Software Inefficiencies with Witch'. In: ACM Press, 2018, pp. 332–347. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3177159.

[144] WikiChip, ed. *The Vulcan Microarchitecture*. 4 October 2019. URL: https://en.wikichip.org/wiki/cavium/microarchitectures/vulcan (visited on 13/07/2020).

[145] Samuel Williams, Andrew Waterman and David Patterson. 'Roofline: An Insightful Visual Performance Model for Multicore Architectures'. In: *Communications of the ACM* 52.4 (1 April 2009). ISSN: 00010782. DOI: 10.1145/1498765.1498785.

[146] Charlene Yang, Brian Friesen, Thorsten Kurth, Brandon Cook and Samuel Williams. 'Toward Automated Application Profiling on Cray Systems'. In: Cray User Group. 24 May 2018.

[147] Toshio Yoshida. 'Fujitsu High Performance CPU for the Post-K Computer'. In: *Hot Chips 30 Symposium (HCS)*. Vol. 18. 2018.

[148] Robert J. Zerr and Randal S. Baker. *SNAP: S_N (Discrete Ordinates) Application Proxy - Proxy Description*. Tech. rep. LA-UR-13-21070, Los Alamos National Labratory, 2013.

[149] Bo Zhao et al. 'Performance Evaluation of NPB and SPEC CPU2006 on Various SIMD Extensions'. In: *Big Data Computing and Communications*. Springer International Publishing, 2015, pp. 257–272. ISBN: 978-3-319-22047-5.

[150] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall and Hans Johansen. 'Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver, Colorado: ACM Press, 2019, pp. 1–44. ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3356210.

[151] Zhengji Zhao and Martijn Marsman. 'Estimating the Performance Impact of the MCDRAM on KNL using Dual-Socket Ivy Bridge Nodes on Cray XC30'. In: *Cray User Group* (2016).