BERGISCHE UNIVERSITÄT WUPPERTAL

Doctoral Thesis

Vector length agnostic SIMD parallelism on modern processor architectures with the focus on Arm's SVE

Author:
Bine Brank

Supervisor:
Prof. Dr. Matthias Bolten

Scientific advisor:
Prof. Dr. Dirk Pleiter

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Natural Sciences

in the department
Mathematics & Informatics
School of Mathematics and Natural Sciences

In cooperation with
Jülich Supercomputing Centre

April 2023

"I have a cunning plan, Sir."

Baldrick

UNIVERSITY OF WUPPERTAL

Abstract

School of Mathematics and Natural Sciences
Mathematics & Informatics

Doctor of Natural Sciences

Vector length agnostic SIMD parallelism on modern processor architectures with the focus on Arm's SVE

by Bine BRANK

This thesis analyzes SVE, a novel SIMD extension for Arm architectures. SVE removes the concept of vector size from the ISA, which allows CPU implementations with different vector sizes to execute the same SIMD instructions. In our work, we ask what consequences this has on application developers and computer architects. We select a set of standard HPC benchmarks and applications for the analysis and rely on Gem5, a state-of-the-art simulator for computer architecture research. We first evaluate the SVE ISA by looking at the vectorization of specific loops and searching for new vectorization opportunities. Afterward, we analyze how the VLA concept translates to algorithms and kernels in HPC. Finally, we study how different SVE lengths impact the execution and behavior of components in the microarchitecture. Our results show that the VLA paradigm in many algorithms naturally extends the fixed-width SIMD implementation. A larger SVE size results in better performance, especially in compute-bound kernels. At the same time, we show that different SIMD widths in a CPU can significantly affect the out-of-order execution and influence bottlenecks in various microarchitectural components.

Acknowledgements

I want to express my deepest gratitude to Prof. Dr. Dirk Pleiter for giving me the opportunity to carry out my research in such a fascinating field. This thesis would not be possible without his encouraging support, continuous guidance, and deep technical knowledge, which resulted in great suggestions and well-received criticism.

I am also thankful to other colleagues at Jülich Supercomputing Centre and Huawei Munich Research Center who helped or supported my research. Especially, I would like to thank Dr. Matthias Gries for his excellent mentorship during my PhD internship and Dr. Guido Giuntoli for his help in preparing GPAW benchmarks.

Additionally, I want to thank Prof. Dr. Matthias Bolten for agreeing to supervise my thesis and for his support in the last stages of my work.

Funding for parts of this work has been received from the European Commission H2020 program under Grant Agreement 779877 (Mont-Blanc 2020).

I would also like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the innovative high-performance Ookami computing system, which was made possible by a $5M National Science Foundation grant (#1927880).

Contents

List of Figures

List of Tables

Abbreviations

ACLE	**A**rm **C** **L**anguage **E**xtensions
ACfL	**A**rm **C**ompiler for **L**inux
API	**A**pplication **P**rogramming **I**nterface
AWS	**A**mazon **W**eb **S**ervices
BLAS	**B**asic **L**inear **A**lgebra **S**ubprograms
CG	**C**onjugate **G**radient
CPU	**C**entral **P**rocessing **U**nit
CSR	**C**ompressed **S**parse **R**ow
EU	**E**xecution **U**nit
FCC	**F**ujitsu **C** **C**ompiler
FD	**F**inite **D**ifferencies
FEM	**F**inite **E**lement **M**ethod
FFT	**F**ast **F**ourier **T**ransform
FU	**F**unctional **U**nit
GCC	**G**NU **C**ompiler **C**ollection
GEMM	**GE**nereal **M**atrix-**M**atrix multiplication
GPU	**G**raphics **P**rocessing **U**nit
HBM	**H**igh-**B**andwidth **M**emory
HPC	**H**igh **P**erformance **C**omputing
HPL	**H**igh-**P**erformance **L**inpack
ILP	**I**nstruction-**L**evel **P**arallelism
IPC	**I**nstructions **P**er **C**ycle
IQ	**I**nstruction **Q**ueue
ISA	**I**nstruction **S**et **A**architecture
LCAO	**L**inear **C**ombination of **A**tomic **O**rbitals

LSQ	**L**oad-**S**tore **Q**ueue
MD	**M**olecular **D**ynamics
MPI	**M**essage **P**assing **I**nterface
MSHR	**M**iss **S**tatus **H**olding **R**egister
PW	**P**lanar **W**aves
ROB	**R**e**O**rder **B**uffer
ROI	**R**egion **O**f **I**nterest
SELL	**S**liced **ELL**pack
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SLC	**S**ystem-**L**evel **C**ache
SLP	**S**uperword-**L**evel **P**arallelism
SVE	**S**calable **V**ector **E**xtension
SoC	**S**ystem-**o**n-**C**hip
SpMVM	**Sp**arse **M**atrix-**V**ector **M**ultiplication
TLB	**T**ranslation **L**ookaside **B**uffer
TRMM	**TR**iangular **M**atrix-**M**atrix multiplication
TSVC	**T**est **S**uite for **V**ectorizing **C**ompilers
VLA	**V**ector **L**ength **A**gnostic

Symbols

ν	frequency	Hz
t	execution time	s
t_c	execution time	cycle
lat	instruction latency	cycle
W	workload	Flop (Op)
AI	arithmetic intensity	Flop/cycle
Q	memory bandwidth	Byte/s
b_{SVE}	SVE size	bit
l_{SVE}	SVE lanes	
N_{SVE}	SVE units in the CPU	
η	SIMD vector speed-up	
ϵ	SIMD parallel efficiency	
	Gem5	
I	committed instructions	
I_C	committed instructions of type C	
μ	committed micro-instructions	
l_{ROB}	reorder buffer size	
l_{LSQ}	load-store queue size	
l_{IQ}	instruction queue size	

Chapter 1

Introduction

1.1 Motivation

High-Performance Computing (HPC) has seen a substantial increase in computing power over the recent decade. In June 2008, the first petascale system was introduced, which could compute more than 10^{15} floating-point operations per second (Flop/s). Since then, multiple research projects worldwide have aimed to achieve the exascale (10^{18}) mark. Finally, in June 2022, this goal was accomplished with the supercomputer Frontier at Oak Ridge National Laboratory in the US. The machine reached a performance of 1.1 EFlop/s on the High-Performance Linpack (HPL) benchmark. Many consider this a noteworthy milestone due to estimates that a human brain operates at a similar processing power at the neural level. However, this accomplishment does not change much for HPC users and developers.

We can observe an exponential growth of computing power in the Top500 list, which reports the five hundred fastest supercomputers globally [1]. Figure 1.1 shows the performance of the first, last, and the sum of all machines on the Top500 list over time. Such a long-lasting exponential behavior was first predicted by Gordon E. Moore in 1965 when he observed that the number of transistors in an integrated circuit doubles roughly every two years [2]. Moore's law will end in the not-so-distant future due to the physical constraints of the semiconductors. However, this does not necessarily mean an end to the HPC system performance growth, and the term *zettascale* has already been coined in the HPC community. Therefore, the per-core and per-node capabilities will likely continue increasing despite new challenges. The same holds for increased system-level parallelism of combining nodes into a large-scale system. For example, today's fastest machines include hundreds of thousands of processors that can simultaneously work on a single problem.

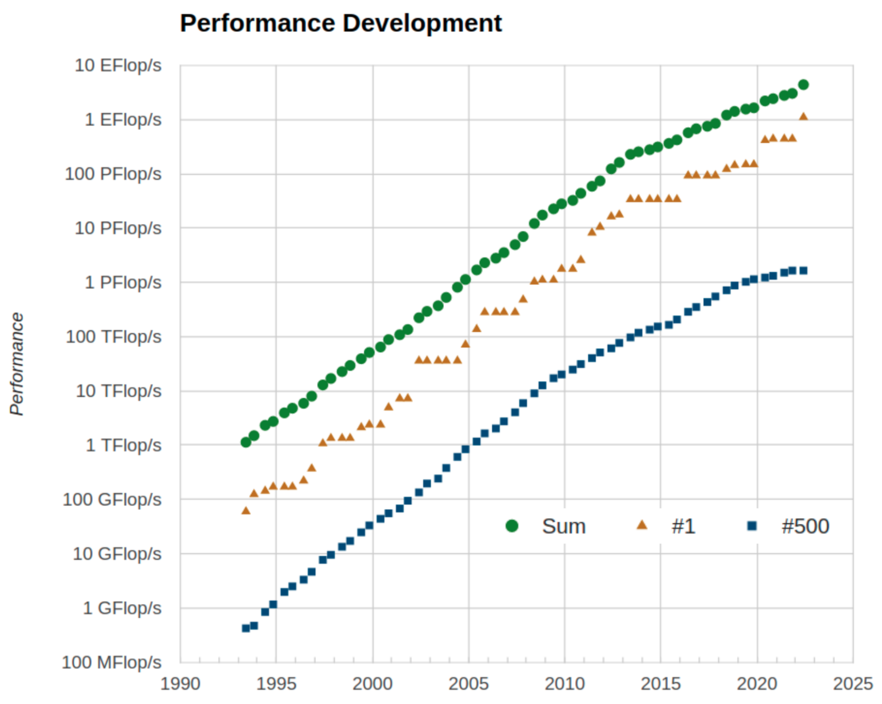


FIGURE 1.1: Top500 performance over time
Taken from `https://www.top500.org/statistics/perfdevel`

In recent years, supercomputing centers based most of their systems on high-end server processors (usually from Intel Xeon, AMD Epyc, or IBM Power processor families). These processors offer a high throughput of operations due to their large number of cores. Furthermore, they have a highly advanced microarchitecture in the Central Processing Unit (CPU) that provides the best out-of-order capabilities. A dedicated Graphics Processing Unit (GPU) often accompanies such machines and acts as a high-performance accelerator. Modern GPUs can output tens of TFlop/s, and many HPC applications offer the option to offload computationally expensive kernels to GPUs. Therefore, a heterogeneous CPU-GPU architecture has become common for machines at the top of the Top500 list.

In June 2020, Fujitsu introduced a new supercomputer Fugaku which implements an Arm-based processor, the A64FX. Fugaku was a novel surprise in the ranking because Arm-based machines were uncommon in the HPC community.[1] The system took first place on the list with an impressive 415 PFlop/s on the HPL benchmark. Another interesting thing that stands out is the machine's power efficiency. Despite having no additional GPU devices, the power consumption relative to computing power (usually measured in Flop/s/Watt) was similar to state-of-the-art machines with GPUs. The A64FX chip features 48 cores and is the first to implement a new vector extension from Arm called Scalable Vector Extension (SVE) [3]. Equipped with two 512-bit SVE

[1]Before Fugaku, the only Arm-based machine on the Top500 list was Astra at Sandia National Laboratories in the US. (Astra implements the ThunderX2 processor.)

units and the high-bandwidth memory (HBM), a single node of the A64FX (running at 2.2 GHz) has a theoretical peak performance of 3379 double-precision GFlop/s and a theoretical memory bandwidth of 1024 GByte/s. With such impressive numbers, the machine remarkably announced the coming of Arm in the HPC market.

1.2 Single Instruction Multiple Data

Many improvements in compute-power technology today originate from a concept of parallelism. In a broad sense, parallelism describes a process in which many calculations are executed simultaneously to solve a common task. This simple idea has become an intrinsic principle in the design of modern processors. Therefore, to achieve maximum performance on today's high-end machines, it is crucial to exploit all kinds of parallelisms that a machine offers. One part involves making an application scale to many cores & nodes. HPC applications often use MPI for inter-node communication and OpenMP or pthreads to utilize all cores inside each node (multithreading). The other side of parallelization refers to the in-core parallelization, which is often underutilized. The main concepts for in-core parallelization are instruction-level parallelism and vectorization.

Instruction-level parallelism (ILP) is a term that encompasses the parallel execution of different instructions within a core. Most instructions on today's CPUs are pipelined.[1] This means that the execution process of instructions accumulates through a pipeline. Each pipeline stage is independent of others, and the execution of multiple instructions within each functional unit can overlap. We use the term functional unit (sometimes also called execution unit) to describe a hardware entity that executes a set of instructions. Another aspect of ILP refers to CPUs having multiple functional units for the same family of instructions. The CPU renames the instruction's operands from architectural state to physical registers, and the CPU scheduler schedules independent instructions to different units to maximize the number of executed instructions per cycle (IPC).

Single Instruction Multiple Data (SIMD) or simply vectorization is an essential optimization principle in today's processors. It builds upon the observation that the same stream of instructions often acts on many individual elements. Consequently, computer architects started combining computation and forcing a Single Instruction to perform operations on Multiple Data, hence the name SIMD. SIMD techniques were first introduced with vector processors in the late 1970s.[2] The central concept behind those machines was an instruction set that operated on vectors rather than scalars. Vector

[1]An exception to this are instructions that require recursive computation, e.g., divide or square root.
[2]The first supercomputer which successfully implemented vector processing was CRAY-1.

instructions took the size of the vector as an input parameter and executed the operation on a vector of elements. Vector processing led the design of supercomputers until the 1990s, when the design shifted to combining multiple machines into a cluster.[1] However, the idea of SIMD parallelization persisted. In the early 2000s, scalar machines started implementing fixed-length SIMD extensions. This means that despite the processor primarily targeting scalar instructions, dedicated execution units that could operate on small vectors extended the instruction set with SIMD operations. We often refer to these functional units simply as SIMD units. Today's SIMD architectures implement 128, 256, or 512-bit wide vector units that operate on floating-point and integer data types. Examples include Streaming SIMD Extensions (SSE, SSE2, SSE3, SSE4) and Advanced Vector Extension (AVX, AVX2, AVX512) for x86 and Advanced SIMD (NEON) for Arm architectures.

SIMD operations offer a significant performance benefit compared to scalar operations. This is most evident in compute-intensive applications, where the floating-point operation throughput limits the performance. Depending on the SIMD vector size and the data type size, the throughput can be even an order of magnitude higher when SIMD instructions are used. However, not all code can be vectorized. Because SIMD instructions operate simultaneously on multiple elements, it is impossible to use them when data dependencies exist between elements.

1.3 Research goals

In this thesis, we perform an architectural exploration of a novel Arm SIMD extension called Scalable Vector Extension. As we explain in Chapter 2, this extension brings some new concepts to the SIMD Instruction Set Architecture (ISA). Most notably, the size of the SIMD vector is not predefined in the architectural state but is restricted to a set of values from 128 to 2048 bits. A processor implements a specific SVE size; however, there is no notion of the vector size in assembly. Therefore, the same binary can run on machines with different hardware vector lengths. We call such SIMD instruction set Vector-Length-Agnostic (VLA). This opens many new questions in terms of programming and hardware implementation.

Firstly, we assess SVE and its capabilities for vectorizing various loops. SVE requires unique loop control to generate VLA binaries, and we want to know if this leads to any limitations in the loop vectorization. We then ask ourselves what kind of loops SVE can vectorize and how this compares to other SIMD ISAs. Besides, we are interested if simple changes in the SVE ISA could open new vectorization opportunities. For every new

[1]The first example of a computer cluster was the Beowulf cluster in 1994.

ISA introduced on the market, some time is needed until the compiler engineers adopt the new ISA technology into the compilers. Like HPC developers, compiler engineers are even harder affected by a vector length agnostic instruction set. Therefore, we want to analyze how compilers deal with the VLA concept and how it impacts auto-vectorization decisions. Finally, we analyze if compilers recognize all SVE opportunities, and where this is not the case, what is the effort of manual vectorization? We hope that such analysis will give some insights into the current state of SVE and the maturity of compilers.

Developers of HPC simulation codes know the potential benefit of a good SIMD utilization. Traditionally, vectors have a predefined size, and when targeting a specific architecture, developers can always make assumptions about the architecture's vector size. Having this information might also result in a specific design choice. For example, a particular algorithm or data layout could better utilize the underlying vectors. In our research, we would like to explore how VLA ISA affects the existing HPC codes and the changes necessary to adapt applications to this new concept. Furthermore, we are also interested in how the underlying algorithm changes when we target a VLA ISA. For this work, we choose a set of HPC applications to study. We know that no two applications are the same, and each application offers unique features to researchers. However, many simulation codes share computational patterns, and choosing the right set of applications can cover a wide range of HPC workloads. Besides, we want to select applications widely used in the scientific community and encourage VLA concepts with open-source contributions.

SVE has many advanced features compared to previous generations of SIMD sets. For many complex kernels, there exist different approaches to loop vectorization, and the developer often has to choose one of several possible implementations. The difference might arise with specific code refactoring or simply due to different (forms of) instructions. Also, specialized gather-load and scatter-store instructions enable effortless vectorization of outer loops (t.i. loops that contain other loops). By vectorizing outer loops, we execute more code with SIMD instructions, but this requires additional work of reassembling data in the SIMD registers. What performance benefits can we obtain for these different types of vectorizations? Additionally, we analyze the SVE's interaction with the hardware by looking at the first implementation of SVE (the A64FX processor). We inspect how different instructions are treated regarding execution latency, throughput, and decoding. In particular, we look at advanced memory operations like gather-load and scatter-store.

Since the SVE size is not predefined, the vendors implementing SVE in their chips must decide on a particular size. Large vector units will lead to better performance but at the cost of higher power consumption and more transistors. Choosing the vector size and the number of SIMD units is a significant design decision. Unfortunately, studies have shown that it is hard to give a unique answer to which vector length is the best. It depends on what kind of workloads the machine will run. With this in mind, we would like to know what performance benefits are possible for applications when increasing the vector size. In our experiments, we rely on Gem5, a simulator for computer-architecture design exploration. We want to point out strongly that our goal is not to correctly simulate real hardware in Gem5. We are aware that the model in Gem5 has significant simplifications, and improving it would exceed the scope of this thesis. Therefore, we would not like to base our findings on the absolute performance numbers achieved in the simulator. Instead, we would like to base our results on relative numbers and see how the performance changes when we change the application or specific parameters in the microarchitecture. To make these conclusions applicable to real hardware, our model must be realistic. We, therefore, want to configure the model similar to a real machine. (In our case, this will be a Graviton 2 processor from Amazon.) Ideally, we want to see how different SVE widths affect the out-of-order execution and occupation of resources like the reorder buffer, reservation stations, or the load-store queue. Hopefully, this will expose bottlenecks that may arise in the microarchitecture due to different SVE implementations.

Chapter 2

Background

2.1 SVE

Arm introduced the concept of SIMD in the ARMv6 architecture in 2002. Initially, SIMD instructions were associated with 32-bit integer SIMD registers. The SIMD ISA was relatively small, with support for approximately sixty instructions. Their primary goal was to implement audio and video encoders more efficiently in multi-media applications. Throughout the years, the developers recognized the potential of SIMD for other use-cases, and Arm expanded the SIMD with more features. The ISA and the register sizes grew to a 128-bit Advanced SIMD. ASIMD is also known as the NEON technology for the Arm Cortex-A and Cortex-R series processors and is implemented in most of Arm's chips today.

In recent years there has been a growing demand for longer SIMD vectors and more advanced SIMD features in the HPC community [4]. For x86 architectures, Intel proposed the AVX-512 SIMD set in 2013, which has longer 512-bit vectors. Additionally, many extensions of the AVX-512 started introducing advanced instructions offering support for masking, permutation, conversion, and other operations. These enhancements pushed SIMD capabilities very far, but not without a cost. As the number of transistors in chips increased, the power consumption grew accordingly. To deal with this, frequency downscaling has become a common phenomenon on some Intel machines. For example, for workloads that rely heavily on AVX-512 instructions, the CPU clock frequency can reduce by 40% to avoid too much heat dissipation [5]. This opens an important debate as to which vector length is the most appropriate. Few studies have analyzed this, and there did not seem to be a unique answer. Different applications utilize the SIMD units differently, so the answer depends on the machine's usage. Therefore, it is crucial to consider the machine's targeted users and demands when designing a new chip. Although vendors try to do that, users usually have pretty different requests, and it is impossible to create a single chip that would accommodate all perfectly.

Scalable Vector Extension [3] is a new vector extension for ARMv8-A, introduced in 2017, that tries to tackle the abovementioned problem. Traditionally, the SIMD extensions had a predefined size of the vector register. In the case of NEON, 128-bit registers can store data for different elements depending on the data type size: two elements for 64-bit types or four (eight) elements for 32-bit (16-bit) data types. The same also holds for SIMD extensions in the x86 architecture. Unlike these, SVE introduces a new Vector-Length-Agnostic (VLA) instruction set. VLA means that the size of the SVE register is not predefined in the ISA. It can range from a minimum of 128 to 2048 bits in 128-bit steps. CPU vendors choose the vector size they want to implement in their chip. Afterward, CPUs with different vector lengths support the same instructions and can run the same binaries. Therefore, the SVE assembly has no notion of the underlying vector size.

To stay concise, we introduce the following terminology. We denote the size of the SVE vector in bits as b_{SVE}:

$$b_{SVE} \in \{128, 256, ..., 2048\} \tag{2.1}$$

The slots that hold individual elements inside the SIMD register are called lanes. The number of lanes depends on b_{SVE} and the data type that fills the register. For a data type d, the number of lanes is equal to $b_{SVE}/sizeof(d)$, if $sizeof(d)$ is the size of data type d in bits. We denote the number of lanes as l_{SVE}. Unless otherwise specified, this symbol refers to the number of lanes for 64-bit data types. Therefore, $l_{SVE} \in \{2, 4, ..., 32\}$. Additionally, we mark the number of SVE units inside the core as N_{SVE}.

To denote the size of the data type, we use a standard notation from the Arm documentation, which also translates to assembly syntax:

size (bits)	name	assembly size specifier
64	doubleword	.d
32	word	.w
16	halfword	.h
8	byte	.b

Table 2.1: SVE data type sizes

2.1.1 Architectural state

SVE defines a new architectural state shown in Figure 2.1. Developers can use thirty-two SVE vector registers named *z0-z31*. These new registers are an extended version of NEON registers, *v0-v31*, which occupy the lowest 128 bits. As explained before, the size of the SVE register b_{SVE} is unknown to the developer and is a multiple of 128 bits. SVE register can hold double-, single- and half-precision floating-point or integer numbers.

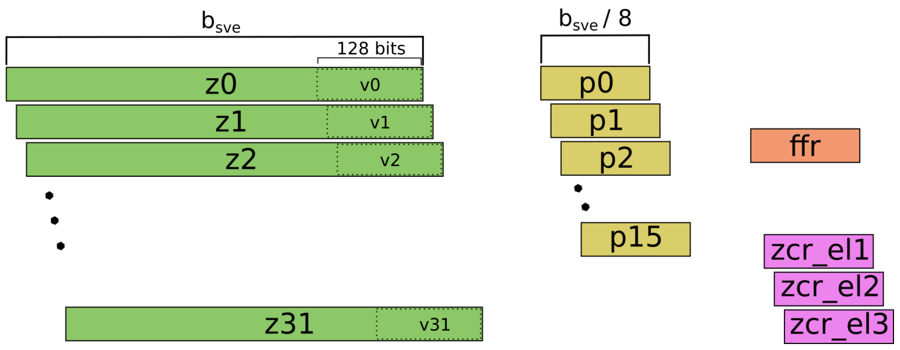


FIGURE 2.1: SVE architectural state

In addition, SVE introduces sixteen predicate registers *p0-p15* that play a crucial role in constructing vector length agnostic loops. Most SVE instructions take a predicate register as an input operand. Predicate register controls which lanes of the output register are updated. The size of the predicate register is equal to $b_{SVE}/8$. SVE sets individual bits to one depending on whether a specific lane is active. For 8-bit data types, each bit is used to construct a predicate. For doublewords, only the lowest bit of each byte is set. Figure 2.2 shows an example of a simple SVE addition instruction *fadd z2.d, p0/z, z0.d, z1.d* for $l_{SVE} = 6$. The predicate's first, third and fourth lane are active (green color), and others are inactive (red color). Only elements corresponding to the active predicate lanes are updated in the output register. Three forms of predicate use are defined: *Z-* predicate form updates the inactive lanes of the output register to zero (example in the figure), *M-* form merges the inactive lanes with those of the input register, and *X-* form leaves them undefined.

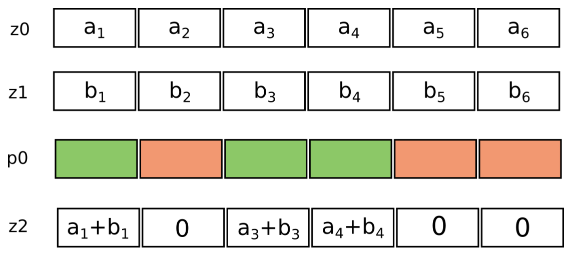


FIGURE 2.2: Predicated *add* operation

Lastly, SVE includes the First-Fault Register (FFR) and three control registers *zcr_el*. The purpose of the FFR register is to suppress the segmentation faults for first-fault vector load instructions. Registers *zcr_el* control the scalable vector system for different Arm privilege levels. As a result, the SVE size can be reduced to a smaller value within the maximum implemented hardware vector size. Therefore, the user can run applications with a smaller vector size if needed.

2.1.2 Main features

This section briefly discusses some of the main SVE features we commonly reference throughout the thesis. Here, we only give an overview of high-level concepts. A detailed explanation with code examples is provided in Appendix A. (A certain familiarity with assembly is needed to understand the examples.)

2.1.2.1 Predication

Predication is the central concept of the SVE's design. Predicate registers drive decisions in a vectorized loop control flow. In traditional SIMD architectures, predication is often implemented by constructing a vector that holds a sequence of incrementing numbers from a loop counter. Such vector is then used as an input to a vector compare operation to check if any lane satisfies the loop's exit condition. If yes, the remaining iterations are computed with scalar instructions (called loop tail). This leads to a wasted register and an additional compare operation. SVE overcomes this problem with a family of *while* instructions. These instructions construct a predicate register whose active bits correspond to iteration decisions in a sequential loop. Because the SVE size is not predefined, the *while* instruction will create a different predicate register depending on the machine on which it is running. Figure 2.3 shows an example of this for a loop with eleven iterations. We see that for an eight-lane SVE register, we execute two iterations. In the first iteration, `whilelt(0, 11)` construct a predicate register with all active bits (green color). In the second iteration, `whilelt(8, 11)` returns a predicate with the first three active bits and others inactive (red color). This leads to correct loading and storing of data without out-of-bound errors. Additionally, it removes the need for treating loop tails as is typical for the NEON vectorization. Similar correct behavior is also shown for two-lane and four-lane SVE.

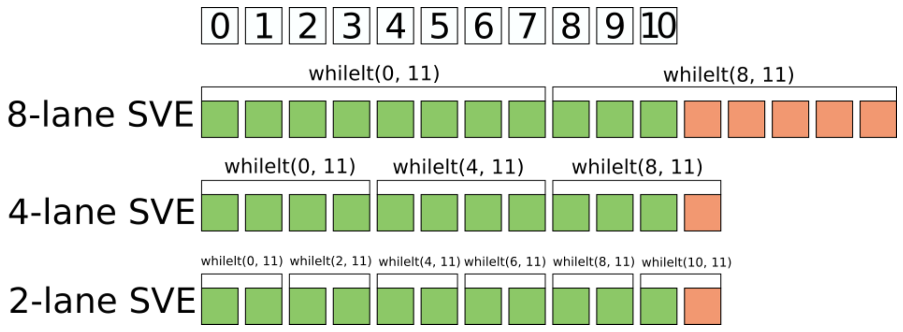


FIGURE 2.3: SVE *whilelt* instruction

The loop counter in vectorized loops is updated with *cnt* instructions. This instruction counts the number of lanes l_{SVE} on the machine where the application is currently running. Different forms are used for different data type sizes: *cntd*, *cntw*, *cnth*, and *cntb*. With these instructions, we can increment the loop counter in a VLA fashion. We show a typical example of the SVE loop in Appendix A.1.

2.1.2.2 Gather-load & Scatter-store

SVE features many instructions for loading and storing data from/to memory. The most basic instructions are the *ld1* instructions that load the data with a unit stride t.i. from contiguous locations in memory. The last letter in the instruction specifies the datatype size of the elements (for example, *ld1d* for 64 bits). To load data with a bigger stride, one can use *ld2*, *ld3*, and *ld4* (stride 2, 3, and 4, respectively) instructions which are especially useful when dealing with arrays of structures or complex numbers. A new feature in SVE that is not available in NEON is gather-load and scatter-store instructions. These instructions enable us to load data from memory with any stride or pattern. The register which holds indices to memory locations for gather-loads is usually called an index register. Instruction with the same name, *index*, is used to create a register that holds integers in an arithmetic sequence. Figure 2.4 shows an example of how to create an index register with stride five and offset two.

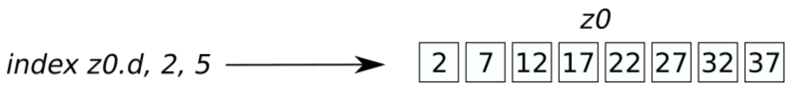


FIGURE 2.4: SVE *index* instruction

However, an index vector can hold arbitrary indices. Figure 2.5 shows an example of a gather-load instruction with random ordering. In this example, values from the array *a* are loaded with an index vector $\{0, 2, 3, 6\}$. Appendix A.2 shows how to use gather-loads and index vectors to vectorize an array permutation.

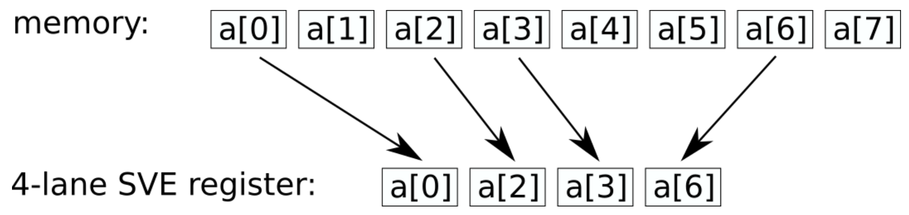


FIGURE 2.5: SVE gather-load

Although gather-load and scatter-store provide a neat way to vectorize specific loops, there is also a drawback in using these instructions. Most importantly, loading the data to individual lanes is not executed fully in parallel. The exact behavior is implementation-dependent. However, hardware almost always splits gather-load and scatter-store into multiple micro-instructions. We, therefore, refer to these as sequential memory instructions, and *ld1* as normal memory instructions. Especially for scatter-store instructions, the SVE ISA mandates that individual lanes must be handled separately in-order. For example, suppose the values in the index register are the same (different lanes writing data to the same memory location). In this case, the memory should be updated by the last element (write-after-write dependency). Therefore, hardware must handle a completely separate memory flow for each lane. Due to no data dependencies, there is more flexibility in hardware implementation for gather-load instructions.

2.1.2.3 Reduction operations

SVE introduces a rich family of reduction operations. Reduction operations operate between elements within the same SVE register. Some examples include *min* (*max*) instruction which calculates the minimum (maximum) element, or *faddv*, which computes a sum of elements in all lanes. In SVE, we distinguish two types of reduction operations[1]:

- Ordered reduction operations (see Figure 2.6 left). These operations perform the calculation across elements in order from left to right. Such order is important for floating-point numbers where one has to comply with the IEEE 754 standard on the non-associativity of floating-point operations. An example of this operation is *fadda*.
- Tree-like reduction operations (see Figure 2.6 right). These operations perform the reduction in a tree-like fashion, which leads to a smaller latency and faster execution (with less accuracy). An example of this operation is *faddv*.

[1]For commutative operations, there exists only one form of reduction operation.

Appendix A.3 shows an example of both approaches in practice. Due to a dependency chain of ordered operations, the execution latency of these instructions is often bigger than tree-like. For example, 512-bit single-precision *fadda* on the A64FX processor takes 31 cycles while *faddv* is executed in 9 cycles.

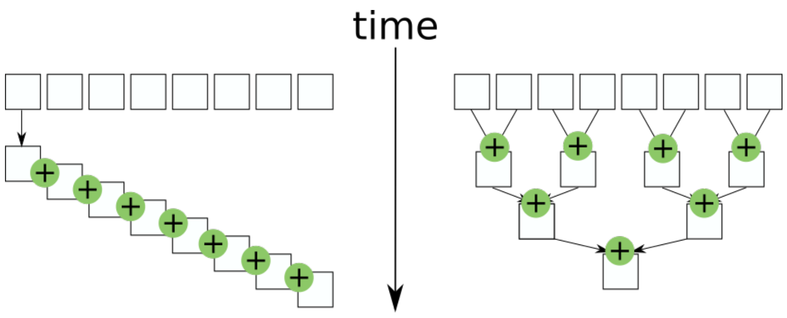


FIGURE 2.6: In-order vs. tree-like reduction

2.1.2.4 Other

Vector compare operations perform a comparison of the entire SVE vectors lane by lane. The output of these instructions is a predicate register. The lanes of the predicate register are active if the comparison operator evaluates to true and inactive otherwise. Compare operations exist both for integer and floating-point data types. Some examples include *greater than*, *less than*, and *is equal.* These instructions enable the vectorization of loops with operations under a specific condition (if-else constructs).

SVE also introduces instructions for complex arithmetics. When loops involve complex numbers, the usual approach combines all real parts of the complex array in one vector and all imaginary parts in the other. This is usually done with *ld2* instructions, which load data with stride two. Complex arithmetics are then constructed using normal multiply-accumulate operations, separately for the real and imaginary parts.[1] However, SVE introduces new *fcmla* operations. These instructions enable us to construct a complex multiplication on SVE vectors with real values in even and imaginary values in odd lanes. (We can load an array of complex numbers with normal *ld1* instructions into SVE registers). For a more detailed explanation, we refer to Appendix A.4.

Speculative vectorization is another advanced feature of SVE. Specific loops do not explicitly define the number of loop iterations but rather stop on a particular condition of the data values they process. A typical example of this is operations on strings, where the loop ends when a NULL character is encountered. Such loops are hard to vectorize because the standard load instructions can access locations outside of allocated memory

[1]Multiplication of two complex numbers requires three multiply-accumulate, and one multiply-subtract operation.

(after the last character) and cause a segmentation fault. SVE introduces a family of first-fault load instructions to tackle this problem. These instructions suppress faults on all but the first active element in the vector. Faults are detected and used to create a predicate register of elements that do not fault. An example of a vectorized *strlen* function, which computes the length of the string, is explained in Appendix A.5.

2.1.3 Code generation

There are different strategies when writing SIMD code. In most software, where performance is not the central aspect of design, developers can just rely on the automatic vectorization by the compiler. Auto-vectorization requires the least effort and is very portable because compilers can target different architectures. However, compilers cannot always vectorize loops. In some instances, auto-vectorization can be improved by a specific algorithm design that is more appropriate for SIMD execution. Otherwise, developers can also resort to external tools. For example, SIMD libraries facilitate SIMD code generation by providing high-level mathematical routines guaranteed to execute using SIMD instructions. These libraries often target various architectures, so portability is not compromised. Some examples include NSIMD [6], Vc [7], and MIPP [8]. When developers require more control over the generated code, they can use intrinsic functions or write assembly. Such code becomes less portable and requires more effort. An overview of different techniques is shown in Figure 2.7.

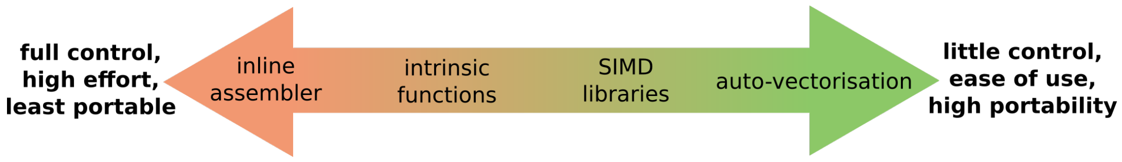


FIGURE 2.7: Application porting

2.1.3.1 Assembly

Today, developers write assembly primarily for two reasons. First, to access specific processor instructions or system calls that the compiler does not support. Secondly, to optimize performance-sensitive parts of code, where they want to have complete control over the generated code. (This is the case for the OpenBLAS library.) Due to a lack of portability, software that leverages assembly usually contains separate implementations for different ISAs.

The syntax of SVE assembly instructions consists of the operation code, destination register, predicate register, and input operands.[1] Figure 2.8 shows an example of a floating-point multiply-accumulate *fmad* instruction. The instruction first sums elements of registers *z1* and *z2* before adding the result to elements of *z0*. This particular instruction acts on 32-bit elements as denoted by the *.s* specifier. Additionally, we use the *m* predication form. This leaves the inactive elements in *z0* as they are.

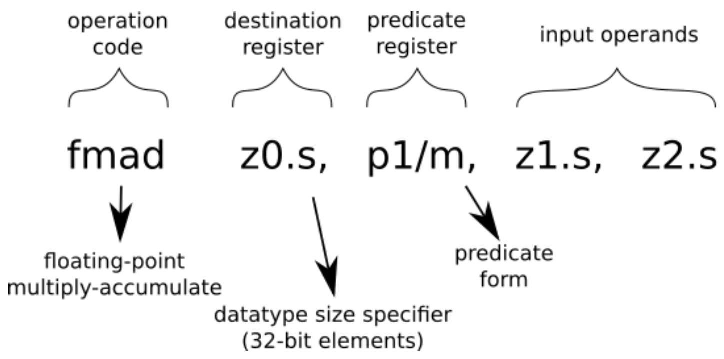

operation code
destination register
predicate register
input operands
fmad z0.s, p1/m, z1.s, z2.s
floating-point multiply-accumulate
datatype size specifier (32-bit elements)
predicate form

FIGURE 2.8: SVE assembly syntax

To directly leverage assembly in code, we either write complete functions in the assembly source or use an inline assembler. Unfortunately, writing functions in assembly is cumbersome and prone to errors because the developer must ensure proper code calling conventions and register saving. On the other hand, inline assembler is a feature of compilers that allows developers to embed low-level assembly code directly inside higher-level languages. Most modern compilers support it. For example, in GCC, we use it through the keyword *asm*. The main problem of inserting assembly directly inside the C code is that it interferes with the register allocation. When the compiler compiles code, it keeps track of the variables and their associated registers. Developers do not have any information on this mapping, which would normally result in unwanted instructions and register spilling. GCC provides the syntax of extended inline assembly to deal with this problem. (See Listing 2.1.)

```
__asm__ <asm-qualifier> ( "assembly code"
                        : output operands
                        : input operands
                        : clobber registers )
```

LISTING 2.1: GCC extended inline assembler

The primary purpose of the extended syntax is to read and write C variables directly inside the assembly. The string of assembly instructions is followed by a list of output and input operands, separated by a colon. We can use these operands as names of the

[1]Some instructions do not have any predicate register or input operands.

registers, and the compiler will correctly allocate registers to individual variables. To use other registers for computation, we can include a list of clobber registers, which are guaranteed to return to the original state at the end. Sometimes, compilers optimize the hand-coded assembly to produce the most efficient code. To prevent this, a qualifier *volatile* must be used to ensure that the hand-coded assembly is injected into code as-is. An empty *asm volatile* statement is often used in benchmarks to prevent compiler's dead code elimination.

2.1.3.2 Intrinsic code

SVE code can also be generated in C and C++ programming languages using Arm C Language Extensions (ACLE) for SVE [9]. We refer to these extensions as intrinsic functions or simply intrinsics. The main goal of intrinsic functions is to provide a set of types and functions that correspond to SVE vectors and predicates. Using these particular types, a programmer can explicitly work with the SVE registers and perform operations on them. An ACLE API defines functions for almost all SVE instructions. The developer can directly access SVE features without touching the assembly. Generally speaking, the ACLE is more general than the ISA, and it is a compiler's job to pick the best instruction for each intrinsic function. Even though the names of intrinsic functions closely resemble the names of SVE instructions, there is no guarantee that these are mapped one to one to underlying hardware operations.

To use intrinsic functions, we include the header `arm_sve.h`. Such a file does not exist because the intrinsic functions are included in the compiler. All intrinsic SVE types and functions are named with the prefix *sv-*. Listing 2.2 shows a function that computes the DAXPY kernel using SVE intrinsic functions. DAXPY computes $y = ax + y$, where y and x are vectors of size n, and a is a scalar. Computation is done on double-precision floating-point numbers.

Here, we give a quick walkthrough of the code, explaining the most important steps. First, a loop counter is created using a 64-bit integer value set to 0 (line 2). A type `svbool_t` in line 3 is a type for a predicate register. Function `svwhilelt_64` returns a predicate register for doublewords, as explained in Figure 2.3. In this case, the variable `pg` can only be used as an operand for operations acting on 64-bit elements. We construct the loop over elements of array `y` using a *do-while* construct. In lines 6 and 7, we use a type `svfloat64_t` which corresponds to an SVE register containing 64-bit floating-point values. SVE ACLE defines different SVE types depending on the data type used. For example, one can use `svint32_t` for integers or `svuint16_t` for unsigned shorts. We use `svld1` instruction to load contiguous data from `x[i]` and `y[i]`. The first argument for this function is the created predicate type `pg`, and the second is the address to memory. Afterward, we use the function `svmla_f64_x`. As the name of the function suggests, this corresponds to a multiply-accumulate instruction of 64-bit floating-point numbers using *-x* form of the predicate. In lines 10 and 11, we update the loop counter and predicate register for vector-length-agnostic execution. The `svcntd` instruction in line 10 returns the number of lanes l_{SVE}, and in line 11, the predicate register is updated with the new active lanes. The exit condition in line 13 means that the loop repeats if any active lanes are present in the predicate register.

```
1  void daxpy(double *x, double *y, double a, int64_t n) {
2      int64_t i = 0;
3      svbool_t pg = svwhilelt_b64(i, n);
4      do
5      {
6          svfloat64_t x_vec = svld1(pg, (const double *) &x[i]);
7          svfloat64_t y_vec = svld1(pg, (const double *) &y[i]);
8          svfloat64_t res = svmla_f64_x(pg, y_vec, x_vec, a);
9          svst1(pg, (double *) &y[i], res);
10         i += svcntd();
11         pg = svwhilelt_b64(i, n);
12     }
13     while (svptest_any(svptrue_b64(), pg));
14 }
```

LISTING 2.2: Example of DAXPY with SVE intrinsics

ACLE defines types for a single SVE vector or a tuple of size 2, 3, or 4. These are named *svBASExN_t*, where *BASE* is the data type, and *N* is the tuple size. The vector-length-agnostic nature of the SVE presents a significant limitation on how compilers treat these types. The intrinsic vector types occupy b_{SVE} bits in memory. Because b_{SVE} is different depending on the machine, the ACLE documentation completely removes the concept of size. It defines a new category of a type called *sizeless* type. (These are not yet addressed in any high-level language standards). *Sizeless* types are more restrictive than complete types but are not incomplete. Most cases of restrictive usage of these types come from the fact that the compiler does not know how to allocate size in memory correctly. Some of the main features of sizeless types that impact the usage in HPC applications are[1]:

- Sizeless types can not be used as an array element or a member of a union, struct, or class. Some HPC applications abstract intrinsic functions for different architectures by implementing a wrapper class for various SIMD targets. Each target is then implemented as a class that overrides the wrapper class by having an intrinsic

[1]For a complete definition of sizeless types, we advise looking in the ACLE SVE documentation.

vector data type as a class field. This approach is not possible for SVE unless we use fixed-size types. (See the next paragraph.)

- Sizeless types can not be used for static or thread-local storage variables (even if they are just declared and not defined).
- They cannot be used as an argument to the `sizeof()` function.
- It is not possible to perform arithmetic operations (`+`, `-`, `++`, `--`) on pointers to sizeless types.

Such restrictions impact software, in which developers must know the SIMD size at compilation. ACLE also defines a fixed-size SVE type that enables developers to generate code only for specific runtime SVE sizes to avoid this problem. Fixed-size types are compiled only for hardware with a particular SVE length, while the behavior is undefined for others. To create a fixed-size type, we use the attribute `arm_sve_vector_bits`. Additionally, the macro `ARM_FEATURE_SVE_BITS` is set to b_{SVE} and can be used to separate implementations for different sizes. An example of this is shown in Listing 2.3.

```
// Following only works on a 256-bit SVE machine
#if __ARM_FEATURE_SVE_BITS==256
typedef svint32_t vec __attribute__((arm_sve_vector_bits(256)));
typedef svbool_t pred __attribute__((arm_sve_vector_bits(256)));

svbool_t pg = svptrue_b64();
svint32_t x_vec = svld1(pg, (const int *) &x[i]);
// x_vec is guaranteed to be 256 bits

...

#endif
```

Listing 2.3: Fixed-size SVE types

2.1.3.3 Compiler support

Since Arm introduced SVE in 2017, compiler engineers have added various degrees of support. GCC added support for SVE in version 8. This included basic auto-vectorization that was vastly improved in GCC 9. Version 10 introduced support for SVE intrinsic functions and the generation of additional (more complex) SVE instructions. GCC 11 also added targets for the A64FX core for microarchitectural tuning. It also improved intrinsic code generation and auto-vectorization. In this thesis, we rely primarily on GCC 11.1.0. The other major open-source compiler toolchain, LLVM, added support for SVE in version 5 (only assembly and disassembly). Intrinsic functions were added in version 11, whereas version 12 added experimental support for auto-vectorization (improved in version 13). SVE is also supported in commercial compilers (Arm Compiler for Linux, Fujitsu compiler, and Cray compiler).

Most compilers implement two auto-vectorization algorithms: a traditional loop vectorizer and a Superword-Level Parallelism (SLP). These techniques are closely related to other topics in compiler engineering like dependence testing, loop normalization, loop interchange, scalar expansion, and others. Loops are the natural candidate for vectorization because the same instructions are repeated on different data in each iteration. This is the most widely used way of exploiting data-level parallelism and SIMD instructions. However, vectorization can only be performed if there exist no data dependencies between iterations. Figure 2.9 shows a simple example of traditional loop-level vectorization.

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

```
for (int i = 0; i < N; i += vec_len) {
```

$$\begin{bmatrix} \texttt{a[i]} \\ \texttt{a[i+1]} \\ \texttt{...} \\ \texttt{a[i+l-1]} \end{bmatrix} = \begin{bmatrix} \texttt{b[i]} \\ \texttt{b[i+1]} \\ \texttt{...} \\ \texttt{b[i+l-1]} \end{bmatrix} + \begin{bmatrix} \texttt{c[i]} \\ \texttt{c[i+1]} \\ \texttt{...} \\ \texttt{c[i+l-1]} \end{bmatrix}$$

```
}
```

FIGURE 2.9: Traditional loop-based vectorizer

Superword-level parallelism denotes a vectorization algorithm where individual scalar instructions are combined into a vector instruction. This algorithm works by scanning the code and looking for repeated sequences of scalar instructions, regardless if they are in a loop. After scanning, it tries to recognize blocks with isomorphic statements. Statements are then packed with vector instructions instead of traditional scalar instructions. An example is shown in Figure 2.10. Although SLP is not connected to loops, we may still be able to use it in cases where traditional loop vectorization is unsuccessful. In this case, a loop is first unrolled to obtain a single block of instructions inside many iterations. Then an SLP algorithm scans the code for vectorization opportunities. This is also called unroll-and-jam.

```
a[i] = b[i] + c[i];
a[i+1] = c[i] + d[i];
e[i] = f[i] + c[i];
f[i] = b[i] + c[i];
```

$$\begin{bmatrix} \texttt{a[i]} \\ \texttt{a[i+1]} \\ \texttt{e[i]} \\ \texttt{f[i]} \end{bmatrix} = \begin{bmatrix} \texttt{b[i]} \\ \texttt{c[i]} \\ \texttt{f[i]} \\ \texttt{b[i]} \end{bmatrix} + \begin{bmatrix} \texttt{c[i]} \\ \texttt{d[i]} \\ \texttt{c[i]} \\ \texttt{c[i]} \end{bmatrix}$$

FIGURE 2.10: SLP vectorizer

If the compiler identifies the opportunity for vectorization, it can generate the SIMD instructions. However, each SIMD opportunity does not necessarily translate into a

SIMD code. Scalar instructions are less expensive than SIMD, and therefore, a vectorized loop will not necessarily perform faster (especially if the number of iterations is small). For this reason, modern compilers employ a cost model. Cost models determine whether a vectorization of a specific loop will yield better performance. For example, both GCC and LLVM compute the cost of scalar and vectorized code blocks and decide on one with a smaller cost. The block cost is computed as a sum of the costs of individual instructions. Of course, these depend on the underlying hardware and the vector size. Since SVE does not define a vector size, compilers do not have this information at compile time. When compiling VLA binaries, GCC, for example, performs a cost analysis for the smallest possible size - 128 bits. It assumes that a speedup is constant or strictly increasing with vector size, which is not necessarily the case. We can override the decisions originating from the cost model with compilation flags[1].

Developers can also provide hints or suppress vectorization with compiler directives (pragmas). For example, for GCC, developers can use `pragma GCC ivdep`, which tells the compiler to ignore vector dependencies between iterations. (Using it where this is not the case can lead to incorrect behavior.) Vectorization can also be encouraged through the use of OpenMP directives (`omp parallel simd`).

2.2 Applications

This subsection gives a brief overview of the main libraries and applications used in our work. A more detailed explanation is given in Chapter 4.

2.2.1 Benchmarks

STREAM [10] is a de-facto standard benchmark to measure memory bandwidth. Four kernels (*copy*, *scale*, *add*, *triad*) are evaluated over a predefined number of iterations. STREAM reports the average time and the measured bandwidth for each of them. There exist different conventions for reporting memory bandwidth. STREAM computes the amount of data transferred as is perceived by the developer. This means that the number of bytes the user asks to read and write from memory is summed up. In other words, if N bytes of data are copied from one memory location to another, the transferred amount is $2 \times N$ bytes (from memory to CPU and then back). This number can be different from the data transferred in hardware due to the write-allocate cache policy. On most systems today, a cache miss on store operation will first load the cache line before overwriting it. Therefore, the developer-perceived bandwidth (as reported by STREAM) is 75% of the

[1]For GCC, we can use the flag `-fvect-cost-model=`*model*.

actual hardware bandwidth for *add* and *triad* kernels. (67% for *copy* and *scale* kernels.) We use STREAM version 5.10. To reduce the Gem5 simulation time, we set the number of iterations to 3 and array size to $N = 10000000$.

Tinymembench [11] is a benchmark that measures memory throughput and latency. Since we already measure the memory bandwidth with the STREAM benchmark, we focus only on the part that measures the memory latency. The benchmark implements a pointer-chasing kernel written in Arm assembly. The assembly does not include any SIMD instructions, so we are executing a strictly scalar code. Latency is measured for a single and double read at each iteration. When the array size gets large, the measured latency is also affected by TLB misses and page walks. However, this effect is still relatively low for the biggest size in our Graviton 2 run. (In our Gem5 model, memory pages and the TLB are not simulated.) The latency measurement is repeated for different array sizes to observe the cache effects. The benchmark increments the array size as the power of two, and we set the maximum size to 64 MByte.

NAS Parallel Benchmarks [12] (NPB) is an open-source benchmark suite developed by the NASA Advanced Supercomputing division. Eight kernels are extracted from CFD applications to evaluate the performance of parallel supercomputers quickly. The original benchmarks are written in Fortran. However, we rely on an unofficial C version[1]. NPB includes five kernels that implement integer sort, embarrassingly parallel algorithm, conjugate gradient, multi-grid, and Fast Fourier Transform (FFT). Additionally, three pseudo-applications apply block tridiagonal solver, scalar penta-diagonal solver, and lower-upper Gauss-Seidel solver.

Test Suite for Vectorizing Compilers [13] (TSVC) is a benchmark suite for evaluating the compiler's auto-vectorization capabilities. The original version, written by Callahan et al., contained 135 synthetic loops. In 2011, TSVC was extended to 151 loops (TSVC2). Loops test various strategies in vectorization algorithms, like dependence testing, statement reordering, loops interchange, scalar expansion, etc. One important thing to note is that the loops are not extracted from any HPC application but were explicitly written for compiler analysis.

2.2.2 OpenBLAS

OpenBLAS [14] is an open-source package for dense linear algebra. It is an active fork of GotoBLAS, which was developed by the Texas Advanced Computing Center. The library is heavily used in the scientific community and provides a good alternative to industry implementations like Intel MKL or ArmPL. The library implements APIs for

[1]https://github.com/benchmark-subsetting/NPB3.0-omp-C

BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage). This thesis will focus on the BLAS part, which we briefly explain here. We give a more detailed explanation in Chapter 4. There exists an interface for both Fortran and C programming languages (the latter is commonly referred to as CBLAS). BLAS is composed of three different levels.

- Level 1 is a group of functions that perform vector-vector operations. These include, among others, dot product, a sum of two vectors, scalar-vector multiplication, or finding a minimum (maximum) element in a vector.
- Level 2 functions implement matrix-vector operations. Apart from matrix-vector multiplication for different types of matrices (triangular, Hermitian, conjugated, general), this also covers the solution of a linear system of equations for a triangular matrix.
- Level 3 functions consider the computation of matrix-matrix operations. Most widely used is a general matrix-matrix multiplication (GEMM) which computes $C = \alpha AB + \beta C$ for matrices A, B, and C. This function is commonly used as a benchmark that stresses the floating-point performance.

Functions from all three levels are implemented for single (S, C) and double (D, Z) precision floating-point numbers (real and complex numbers). The letter that prepends the function name corresponds to the data type. For example, DGEMM stands for a General Matrix Multiplication for double-precision floating-point numbers. Whereas level 1 and 2 functions are usually memory-bound on today's machines, level 3 kernels are compute-bound. However, achieving high performance (close to the machine's theoretical peak) in functions such as DGEMM is not trivial. Different architectures require diverse techniques to attain high performance, and developers must tailor the code to each machine individually. In this regard, compilers are often not mature enough to generate such high-precision code. OpenBLAS includes specialized assembly kernels (often called macrokernels) targeting specific instruction sets and CPUs. These kernels ensure that the floating-point execution units in the CPU execute one instruction per cycle without any stalls. Additionally, the original data layout is transformed to maximize data reuse and good cache utilization.

2.2.3 GROMACS

GROMACS [15] is one of the most widely used molecular dynamics (MD) codes for simulating a system of particles in biochemistry. The program uses a modified Verlet algorithm which computes forces between the particles and solves the Newtonian equations of motion. In addition, a particle mesh Ewald algorithm is used to approximate long-range forces. (Ewald algorithm mainly relies on the FFT.) GROMACS is primarily used for simulations of proteins and nucleic acids. For state-of-the-art simulations, the number of particles usually exceeds one million.

Compared to other MD simulation codes, GROMACS is known for its high performance, making it particularly popular in the HPC domain [16]. Even though it was primarily designed for complicated bonded interactions inside biochemical molecules, the application is also fast at simulating non-bonded interactions, which is usually more compute-intensive. This is realized with its heterogeneous design that leverages different levels of parallelization in modern machines. For example, intrinsic functions are used to utilize SIMD registers inside the core fully. Parallelism inside spatial domains is configured with OpenMP, while MPI is used for decomposing an entire problem over multiple nodes in the cluster. Additionally, GPUs and other accelerators can handle kernels computing the non-bonded interactions. These paradigms have a dedicated implementation separated from others and targeted to attain high performance.

2.2.4 GPAW

GPAW [17] is a simulation software for calculating materials' electronic structures and atomic properties. It is used across multiple physics, chemistry, and materials science fields. Written with MPI and OpenMP parallelization, the code is specifically targeted at massively parallel supercomputers. The algorithm applies functional density theory (DFT) based on the projector-augmented wave (PAW) method and the atomic simulation environment (ASE). GPAW uses three main numerical methods:

- **Finite differences** (FD) is the default mode in GPAW. Simulation space is discretized on a uniform real space orthorhombic grid, which can expand the pseudo wave function. Two different discretizations are used (coarse and fine). The effective potential is evaluated on the fine grid and then restricted to the coarse grid, acting on the wave functions. Different stencils can be applied for the Laplacian operator and Poisson equations.
- **Linear combination of atomic orbitals** (LCAO) is an alternate mode of computation for calculating wave functions. In this mode, waves are approximated using a basis set of atomic orbital-like functions. This makes the computation cheaper; however, the simulation is less accurate if the chosen basis is not selected correctly. Therefore, GPAW implements an internal basis tool that can generate a set of orbital functions for each element in the system.

- **Plane-wave** (PW) mode is usually used for small systems where good parallel efficiency is unnecessary. In this mode, quantities are represented using Fourier transforms on the periodic supercell with periodic boundary conditions.

2.2.5 MiniFE

MiniFE [18] is a proxy application for the exploration of parallel programming techniques for codes based on the implicit Finite Elements Method (FEM). The application is not used for solving real-life FEM workloads but rather as a mini-application (mini-app), a small but accurate representation of a real-world application. It enables developers to quickly evaluate the performance of different programming techniques without the effort of changing big production codes. MiniFE was developed at Sandia National Labs as part of the Mantevo project. Mantevo project includes several mini-apps used for performance optimization.

The application solves a steady-state conduction equation for heat transfer. Temperature is computed on a box-shaped domain composed of linear 8-node hexagonal elements (a linear hexahedral mesh). Dirichlet boundary conditions are applied on all sides of the box. Despite its small size of code (approximately 8000 lines), the MiniFE mimics all kernels of a full FEM simulation:

- Hexahedral mesh assembly. The box domain is decomposed into regular hexahedrons. The application computes the node coordinates for each cell and maps them to a global array. When MPI is used, each subdomain is assigned to a different MPI rank.
- Generation of the matrix structure. The matrix is first constructed with the appropriate number of rows and columns for non-zero elements depending on the sparse matrix format. Next, each OpenMP thread initializes the memory for its local matrix.
- Computation of equation operators. The mini-app calculates the local FEM operators to assemble a diffusion matrix for each node.
- Assembly of the sparse matrix and RHS vectors. Diffusion matrices of nodes are accumulated into a final matrix. Additionally, Dirichlet boundary conditions are applied to calculate the right-hand side.
- Conjugate Gradient (CG) method for solving a linear system. MiniFE includes a CG solver and separate implementation of the sparse matrix-vector product and other vector kernels (DAXPY, dot product, vector norm).

- Verification of results. Results are compared to an analytical solution.

2.3 Hardware

For the results presented in this thesis, we mostly rely on the Gem5 simulator, which we explain in Section 2.5. However, to compare the simulator with the actual hardware, we also perform experiments on two Arm-based machines: the Graviton 2 and the A64FX. First, we give a more detailed explanation of the Neoverse N1 core, which is implemented in the Graviton 2 processor. We use the Neoverse N1 as a template when configuring the Gem5 core model. Then, for the A64FX, we explain the most critical parameters and differences compared to the Neoverse N1. Vendors use different terminology when describing execution in the CPU. To stay concise, we introduce the following terminology:

- **Instruction** is a fundamental part of the ISA and does not depend on the hardware implementation. A stream of instructions constitutes a program that is executed in a CPU. Instructions are fetched from the instruction cache before being decoded.
- **Micro-instruction** is a hardware-level entity that is decoded from the instruction. A single instruction can be decoded into a single or multiple micro-instructions that are issued to the corresponding execution units. On Arm-based machines and in Gem5, micro-instructions are often called micro-operations.
- **Operation** is a more general term that describes pure mathematical operation (multiplication, addition) or an execution process in the functional unit.

2.3.1 Neoverse N1

The Neoverse N1 (or just N1) is a high-performance CPU designed by Arm [19]. The N1 was introduced in 2019 as a server counterpart to the mobile's Cortex-a76 core. Arm does not manufacture the core itself but instead sells the core's intellectual property (IP). Other semiconductor companies buy the license to implement and manufacture the core in their products. The design of the N1 CPU targets a high-end server-class market. It offers low power consumption and high multithreading capabilities with a scalable design (8 to 128 cores per socket). The microarchitecture of the Neoverse N1 core is loosely based on the Cortex-a76 and is optimized for heavy OS activity and infrastructure applications. It implements the Armv8.2 instruction set architecture and is usually built on a 7 nm lithography process.

Figure 2.11 shows a high-level diagram of the microarchitecture. The N1 core can fetch up to four instructions per cycle from the L1 instruction cache (L1-I). Compared to the previous Arm cores, the biggest change in the core's frontend is a decoupled branch predictor. Separated from the instruction fetch, the predictor can speculatively fetch addresses even if the fetch pipeline to the L1-I stalls. These addresses continue to access the I-cache, which acts as a prefetcher and reduces the number of future L1-I cache misses. Afterward, instructions are forwarded to a 4-wide decoder, where each lane can decode one instruction per cycle. At this stage, instructions are first decoded into macro-operations. Frequently used instructions (ALU and branches) are decoded into a single macro-operation, while more complex instructions are decoded into multiple macro-operations. Additionally, I-cache can store partially decoded instructions to speed up this process.

Decoded macro-operations are then sent to the rename unit, which can also receive up to four operations per cycle. Each macro-operation is renamed into one or two micro-instructions, and architectural registers are replaced with physical registers. Here, simple register-to-register *mov* instructions can be simplified or eliminated through the rename tables. Rename unit dispatches up to eight micro-instructions to the out-of-order backend. When a micro-instructions is dispatched, its status is tracked in the reorder buffer and the issue queue. The reorder buffer can hold up to 128 micro-instructions. The issue queue is responsible for monitoring the instruction's source operands. Once all operands are available, the operations are issued to the corresponding execution unit from the instruction queue. Each execution unit features an additional 16-entry buffer to increase the out-of-order window. However, if the buffer is empty, dispatched instruction can bypass it to minimize latency. The reorder buffer tracks micro-instructions and commits them in order after execution (up to eight per cycle).

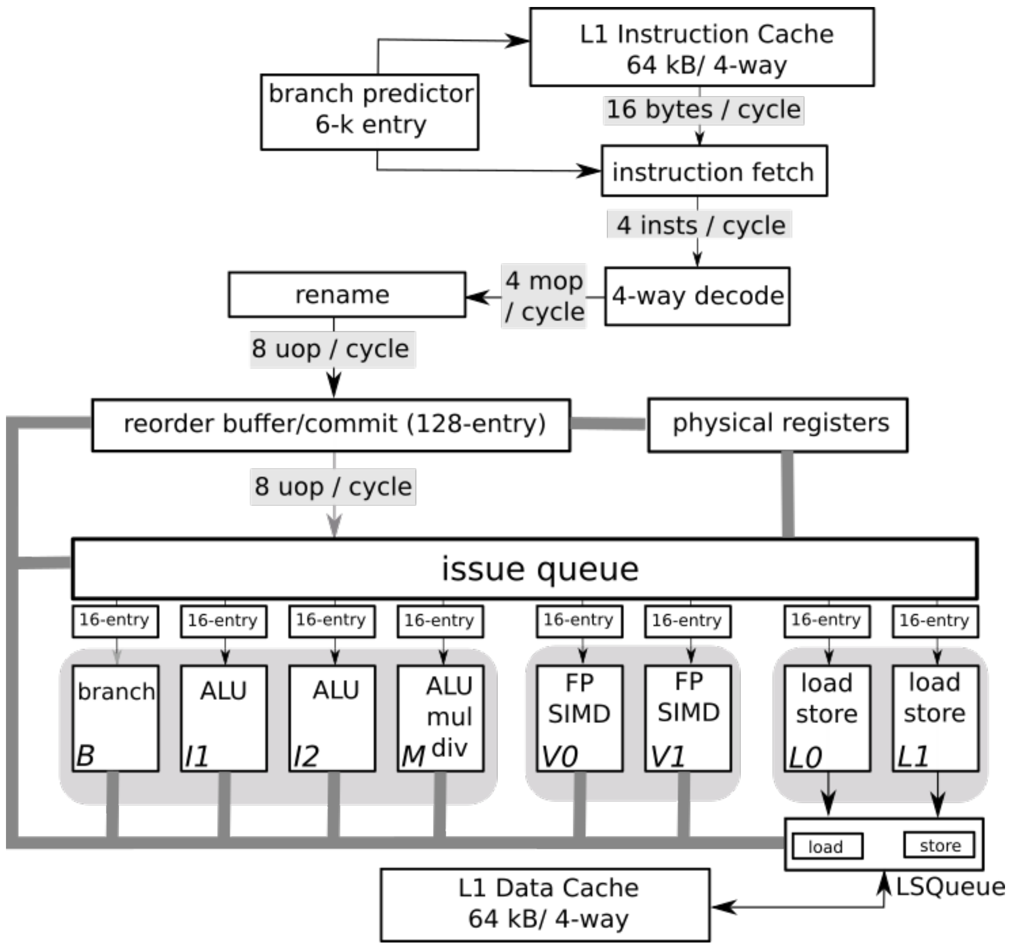


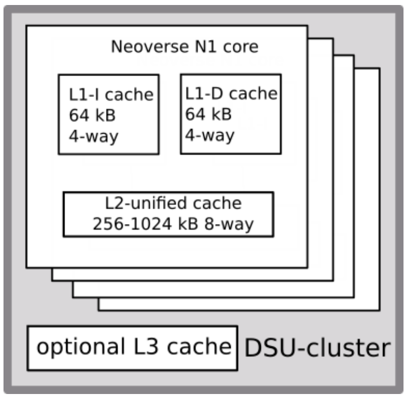
FIGURE 2.11: Neoverse N1 microarchitecture

The execution stage features eight separate functional (execution) units. The first unit *B* executes branch instructions, most of them in one cycle. The following three units *I1, I2, M* target the integer instructions. Basic ALU instructions can be executed in a single cycle on all three FUs, whereas more complex integer instructions[1] are executed only on the *M* unit. The core also includes two SIMD units (*V1, V2*) for floating-point and NEON instructions. Most NEON instructions can be executed on both units, while some complex instructions (fdiv, fsqrt, fconvert) are executed only on the *V1* unit. Lastly, two load/store units (*L1, L2*) are responsible for memory operations and address generation. These are accompanied by a load-store queue (LSQ) with 68 load and 72 store entries.

The Neoverse N1 features private L1 and L2 caches. Furthermore, the L1 cache is split into instruction (L1-I) and data (L1-D) cache. Both L1 caches are 4-way set-associative with a size of 64 kByte. The L1-D has a latency of 4 cycles and a bandwidth of 32 Byte/cycle. The L2 cache can be configured with different sizes with a load-to-use

[1]multiply, divide, multiply-accumulate, saturate, sign/zero extend...

latency of between 9-11 cycles (depending on its size). The Most important parameters are shown in Figure 2.12. All caches are write-back, and the external memory is not updated unless the cache line is evicted from the System Level Cache (SLC) or explicitly requested. Additionally, all caches allocate a cache line on read and write access. The L1-D cache is strongly inclusive to the L2, which means that any cache line in the L1 is also present in the L2. The L1-I, on the other hand, is weakly inclusive. When the cache line is allocated, it is allocated in both L1-I and L2. However, it can later be removed from the L2 and persist in the L1.



cache line size	64 Byte
L1i & L1d size	64 kByte
L1i & l1d associativity	4
L1d clusivity to L2	strongly inclusive
L1i clusivity to L2	weakly inclusive
L2 size	512-1024 kByte
L2 associativity	8
L2 clusivity to L3	exclusive

FIGURE 2.12: Neoverse N1 cache configuration

An interface between the CPU and the external interconnect is called DynamIQ Shared Unit (DSU). DSU contains all external interfaces of the N1 core, including bus, power management, and clock interfaces. It can be configured either as a direct connect or a multiple CPUs cluster. A direct connect has no L3 cache, and memory connections pass directly through the DSU to the interconnect. When multiple CPUs are implemented in the DSU, the cluster contains an optional L3 cluster cache and the Snoop Control Unit (SCU). The cluster cache is shared between the cores in the cluster and has a more complex cache line allocation. Finally, the N1 System-on-Chip (SoC) can implement up to 256 MByte of shared System-Level Cache.

2.3.1.1 Graviton 2

For our experiments, we rely on the Graviton 2 CPU [20]. Graviton 2 is a processor developed by Amazon and Annapurna Labs which implements the Neoverse N1 core. It was designed for usage in the Amazon Web Services (AWS) cloud as an alternative to x86-based systems. The main parameters of the Graviton 2 are shown in Table 2.2. The cores operate at a frequency of 2.5 GHz.

clock frequency	2.5 GHz
#cores	64
DSU configuration	32 x 2 (core-duplex)
L1-I&L1-D cache size	64 kByte
L2 cache size	1 MByte
L3 cache size	none
SLC cache size	32 MByte (1 MByte slice per core-duplex)

Table 2.2: Graviton 2 configuration

The processor features 64 cores in the configuration of 32 DSUs on the SoC. Cores are connected over the Arm's coherent mesh network CMN-600. Each DSU cluster contains two cores without an L3 cluster cache. Therefore, cores connect directly to the mesh interface over the Component Aggregation Layer (CAL). AWS reports that such a design improves latency to the SLC and DRAM because it removes the logic of the optional cache and SCU inside the cluster. Each crosspoint of the mesh interconnect connects to a slice of 1 MByte SLC, giving a total SLC cache size of 32 MByte. Figure 2.13 shows a high-level overview of the cluster configuration on SoC.

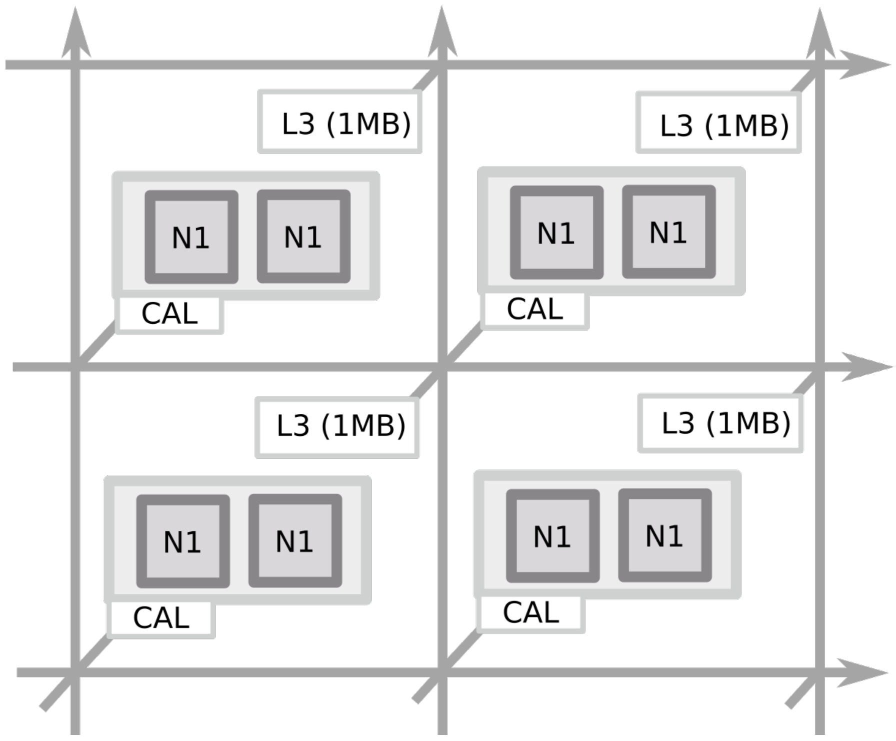


Figure 2.13: Graviton 2 SoC

2.3.2 A64FX

The A64FX [21] is the first processor that implements SVE. The processor was introduced in 2020 with the supercomputer Fugaku which took first place on the Top500 list. The A64FX processor is architecturally very different from the Graviton 2.

The processor is designed for the HPC market and implements the Armv8.2-a[1] architecture profile with support for SVE. Each core features two 512-bit SVE units. Compared to N1, a single core can execute four times more floating-point operations per cycle. The processor consists of 48 cores (plus two or four assistant cores) running at 1.8-2.2 GHz. A private 4-way 64 kByte instruction and data cache accompany a shared L2 cache. The cache-line size is 256 bytes, four times bigger than in Graviton 2. The main parameters of the A64FX CPU are shown in Table 2.3.

clock frequency	1.8-2.2 GHz
#cores	48 (+2/+4)
Core configuration	4 x 12 (4 CMG)
SVE units	2
SVE size	512 bit
Cache-line size	256 Byte
L1-I&L1-D cache size	64 kByte
L2 cache size	8 MByte per CMG
Memory	8 GByte (HBM)

Table 2.3: Main parameters of the A64FX

The A64FX chip has a very different SoC configuration. The SoC comprises four core memory groups (CMG) and a Tofu-D interconnect controller. Each CMG has 12 cores and a unified 8 MByte L2 cache that is shared between all cores in the CMG. The A64FX also uses a second-generation High-Bandwidth Memory (HBM). Each CMG has a memory access controller (MAC) connected to the HBM2 stack. The HBM memory offers high bandwidth but at the cost of a higher latency. The theoretical peak bandwidth of a single CMG is 256 GByte/s, which gives 1024 GByte/s for a full node. However, as reported in the A64FX manual, the load-to-use latency is 131 to 140 ns (depending on the distance to the core). This is roughly 60% slower than in Graviton 2. An overview of the SoC configuration is shown in Figure 2.14.

[1]It also supports complex arithmetics instructions from Armv8.3-a

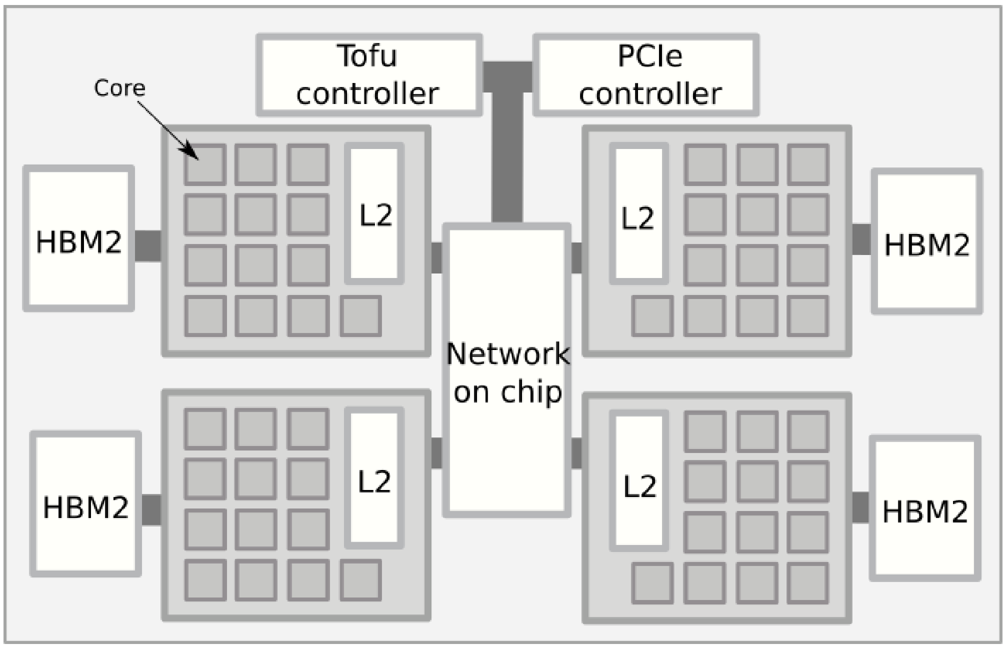


FIGURE 2.14: A64FX SoC

The out-of-order microarchitecture is also quite different from the N1. An overview is shown in Figure 2.15. The core can fetch up to eight instructions per cycle from the L1 instruction cache (two times more than in N1). This is achieved with the help of the branch prediction mechanism that predicts branch directions and target addresses. Fetched instructions are temporarily stored in the instruction buffer, which can hold 48 instructions.

Then, up to six instructions per cycle are sent to a decoder which decodes instructions into micro-instructions. Most instructions are decoded into a single micro-instruction, while complex instructions are split into multiple micro-instructions. After decoding, micro-instructions are dispatched in-order to the reservation stations. The A64FX implements five reservation stations (RS), which issue the operations out of order. Unlike Graviton 2, where reservation stations are unified in an issue queue, the A64FX completely separates different reservation stations. Each RS issues micro-instructions to a subset of execution units (see Figure 2.15). The reorder buffer, also known as the Commit Stack Entry (CSE), has 128 entries, the same as the N1. Up to six micro-instructions can be committed per cycle.

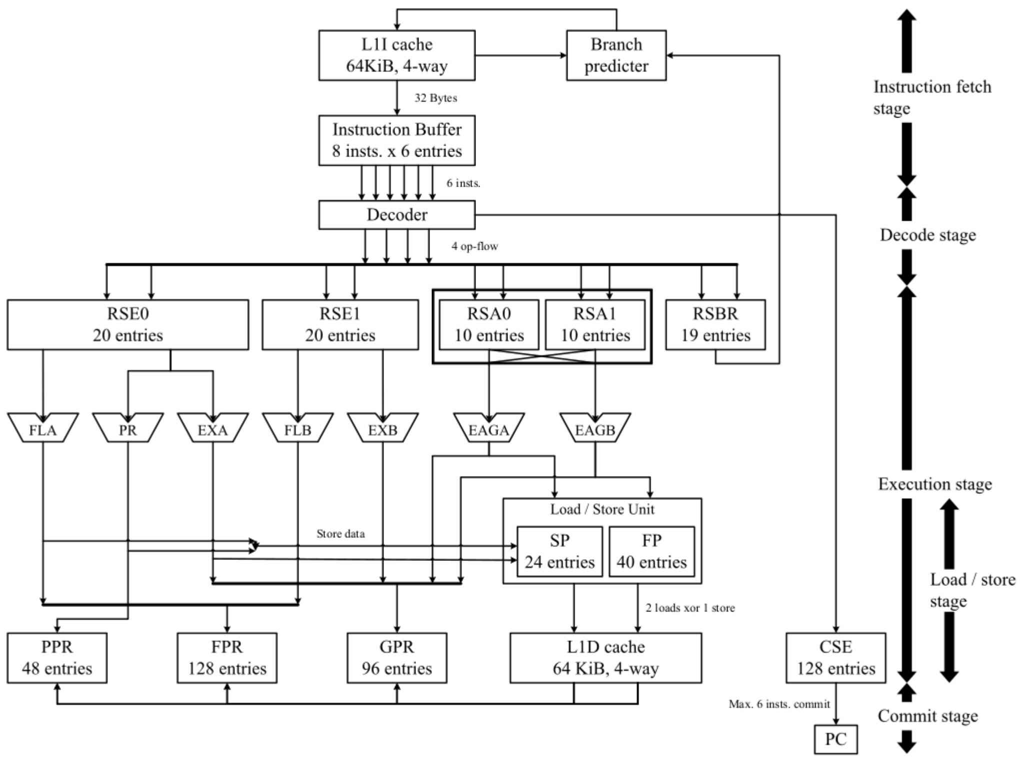


Figure 2.15: A64FX microarchitecture
Taken from the A64FX manual (`https://github.com/fujitsu/A64FX`)

The execution engine is composed of two integer pipelines (EXA/EXB), two pipelines for floating-point operations (FLA/FLB), a predicate pipeline (PR), and two pipelines for load/store and address generation (EAGA/EAGB). Due to importance of the gather-load and scatter-store instructions, we take a look at how these are treated in the A64FX processor. Gather-load instructions are first split into $l_{SVE}/2$ micro-instructions. Each micro-instruction loads a pair of elements in neighboring lanes. If elements in a pair point to the same 128-bit address space, they are treated in one memory flow. If they are not, they are split further into two separate micro-instructions. For scatter-store instructions, the processor issues one memory write per lane. Compared to the N1, the core has a smaller buffer for load (40) and store (24) instructions.

Overall, the A64FX has fewer resources for an out-of-order execution than the N1. Although the architecture of the A64FX is quite different, the five reservation stations have 79 entries, 41 less compared to the issue queue in N1. Additionally, the RS are separated, leading to a bottleneck if execution units of the same RS are overstressed. Also, the number of general-purpose registers is smaller (96 to 128).

SIMD instruction	N1 (NEON)	A64FX (SVE 512-bit)
integer addition (add)	2	4
integer multiplication (mul)	4	9
shift left (shl / lsl)	2	4
integer compare equal (cmeq / cmpeq)	2	4
FP addition (fadd)	2	9
FP multiplication (fmul)	3	9
FP multiply-accumulate (fmla)	4	9
FP square root (fsqrt)	7-17	154
FP divide (fdiv)	7-15	154

Table 2.4: Instruction latencies of Neoverse N1 and A64FX

Another big difference to N1 is that the execution units for SVE implement much deeper pipelines. As a result, the execution latencies of different SIMD instructions are bigger compared to the N1. In table 2.4 we show major execution latencies for double-precision integer and floating-point SIMD operations. Most of the A64FX's instruction latencies are two to three times bigger than in N1. For *fsqrt* and fdiv operations, the execution is up to ten times slower.

2.4 Tools

Arm Instruction Emulator (ArmIE) [22] is an SVE emulator developed by Arm. It enables developers to run SVE binaries on existing Arm platforms that do not support SVE. SVE instructions are emulated with native Armv8-a instructions. The tool can emulate SVE instructions for any supported vector length (128-2048 bits). Listing 2.4 shows an example of an ArmIE command.

```
armie -msve-vector-bits=<128,256,..> ./sve_bin.exe
```

Listing 2.4: ArmIE command

Additionally, the tool can provide some useful insights into the SVE code. ArmIE uses DynamoRIO, a dynamic binary instrumentation tool for x86 and Arm platforms. Users can write new or modify existing binary-level instrumentation clients for their purposes. Multiple clients are preconfigured for use and can be invoked with the options `-i` and `-e`.

HPCToolkit [23] is a set of tools for measuring an application's performance. Due to its ability to measure data for massively parallel applications, HPCToolkit is a common choice for profiling applications in the HPC community. Furthermore, both MPI and OpenMP programming models are supported. The tool works by sampling timers and hardware performance counters during the execution.

2.5 Gem5 simulator

Gem5 [24][25] is an open-source system simulator used to assess the design space of computer architectures. The tool was introduced in 2011 by merging the GEMS and M5 simulator codes. Since then, a large community has actively developed the code and added many new features. Nowadays, Gem5 is a widely used tool for computer architecture research in industry and academic institutions. The simulator provides a highly configurable simulation framework with support for multiple ISAs. This includes, among others, x86, RISC-V, POWER, and ARM architectures. In addition, support for SVE was added in 2018. The Gem5 framework features a modular design, with various components connected in a simulation system. Many of these components are premade, and users can quickly build their system without understanding how each component works internally. Additionally, a few fully premade systems are available out-of-the-box for users who quickly want to test the simulator.

Choosing accuracy vs. performance is a crucial design question for any simulator. For example, a detailed simulator that completely simulates real hardware to cycle-level detail in all architectural components would be very accurate and slow. On the other hand, simplifying certain parts can significantly increase performance, but with simplifications, the accuracy suffers. Gem5 is a flexible tool that allows the user to choose where he would like to perform simulations in this accuracy-speed spectrum.

Gem5 includes different CPU models, ranging from fast and straightforward in-order cores to more detailed out-of-order cores. The simplest core model is an *AtomicSimpleCPU*, a fully functional abstract model without a pipeline. The operations are executed in order, and memory accesses happen instantly. This makes the model super fast but not realistic. It is usually used for testing or the warm-up phase when a detailed model is unnecessary. *TimingSimpleCPU* uses the same core model as *AtomicSimpleCPU* but with more realistic memory accesses that stall the CPU until data is transferred to the core. *MinorCPU* is a more detailed in-order superscalar CPU with a fixed pipeline. This model executes multiple instructions simultaneously if issued to different functional units. However, the execution is still done strictly in order. Additionally, *MinorCPU* model also allows visualization of instruction's pipeline execution process. The most detailed model with the ability for out-of-order execution is the *O3CPU* model. This model resembles today's general-purpose processors. The *O3CPU* is loosely based on the Alpha 21264 CPU but provides a high degree of freedom in its configuration. We provide more details in Section 3.3.1.

Similarly to a CPU configuration, Gem5 also provides different models for simulating a memory system. A wide range of DRAM, HBM, and abstract memory technologies are preconfigured. Additionally, Gem5 provides two cache configuration models. A *classic* memory model is a default model that supports private and shared cache levels, different cache replacement policies, and the MOESI snooping protocol. Memory objects (caches, buses, memory controllers) are connected via unique interfaces called *ports*. These always come in pairs with the *mem_side* sending requests and receiving responses and the *cpu_side* receiving requests and sending responses. Each component can have multiple ports. Such a modular design enables a quick assembly of the memory system. A more detailed *Ruby* model also exists, which can simulate different interconnects and a wide variety of cache coherence protocols. Furthermore, the caches can be arranged flexibly into arbitrary topologies, creating homogeneous and heterogeneous systems.

Gem5 can operate in two modes for simulating devices. A *System-emulation* mode emulates system calls. Whenever a system call is encountered in the application, Gem5 intercepts it, emulates it, or forwards it to a host (machine on which we are running the simulation) operating system. Unfortunately, many calls are still not supported for various architectures, which leads to problems for applications that rely on many system calls. Gem5 can also simulate a complete operating system and devices with a *Full-system* mode. This mode supports different privilege levels, interrupts, and I/O devices. In this case, Gem5 relies on a disk image that contains an installed operating system. Although the full-system mode provides a more configurable environment for a simulation, it also requires more effort to set up correctly. Additionally, the simulation time is increased due to the need for system boot[1]. In this thesis, we exclusively use a system-emulation mode. This requires some application modifications (to remove specific system calls). Still, we feel that this effort outweighs the complexity of the full-system mode. All application code changes are presented in Section 3.2.3, with the code publicly available.

Gem5 is written in C++ and Python and simulates time as a series of discrete events (called ticks). Each Gem5 object schedules events for a specific tick in the future based on the event's elapsed time. These events are placed in the *Eventqueue* and are called at their scheduled tick. Ticks are incremented in chronological order.

[1]This can be mitigated with simulation checkpoints and snapshots.

2.6 Related Work

Despite the first SVE hardware only becoming available in 2020, there were already a few studies about SVE before that relied on simulation and emulation. Kodama et al. [26] performed simulations in the Gem5 simulator for various kernels and showed that increasing vector length could improve performance when the bottleneck arises due to a long chain of operations or instructions issues. However, this only holds if there is a sufficient number of physical registers. If the number of physical registers is too small, the bigger vector size can worsen the performance. For memory-bound applications, the vector length did not affect the performance significantly.

Poenaru et al. [27] analyzed the effectiveness of the SVE instruction set for a set of mini-apps. They observed that compilers successfully generate VLA code and utilize main SVE features. For all workloads, compilers vectorized more code when targetting SVE than when compiling for NEON, confirming that SVE offers more features. They also discussed the benefit of per-lane predication, which allows easy vectorization of loops with heavy control flow. Furthermore, the evaluation of mini-apps with ArmIE concluded that increasing the vector size decreases the total instruction count. However, not all applications can fully utilize all lanes of the SVE register when the vector size becomes big ($\geq$ 512-bit). Also, the memory accesses were efficiently handled by gather-load and scatter-store instructions.

Pohl et al. [28] studied the difference between vector-length-agnostic and vector-length-specific code. They evaluated the Test Suite for Vectorizing Compilers (TSVC) benchmark suite in Gem5 and noticed a better SVE vectorization than NEON. Additionally, on average, vector-length-agnostic code was roughly 10% slower than vector-length-specific due to additional predicate operations. They also observed that the average speedup of loops does not scale perfectly with increasing vector length due to memory limitations. Here, an increasing L1 cache size significantly impacted longer SVE vectors. They conclude that large vectors will push today's applications even more in the memory-bound region. This opens new challenges for the cache hierarchy design required to utilize large vectors fully.

Armejach et al. [29] investigated the potential of SVE for stencil codes. They showed that manual optimizations could speed up the code up to a factor of 1.57 compared to a straightforward compiler-vectorized code. These improvements come from various code-optimization principles like loop unrolling and data reuse. However, specific optimizations can also hurt performance due to the reduced arithmetic intensity and instruction overheads.

Meyer et al. [30] presented the SVE port in the lattice QCD simulation software framework GRID. They reported that a suitable data layout is crucial for good SIMD parallelization. Because the data layout depends on the SIMD width, the SVE vector length was fixed at compile time. They showed various real and complex arithmetics implementations but have not reported any performance evaluations.

Alappat et al. [31] modeled the performance of streaming kernels and sparse matrix-vector multiplication on the A64FX processor. They established an execute-cache-memory (ECM) performance model which accurately predicted (up to 20% error) performance for in-memory data sets. They identified long floating-point instruction latencies and limited out-of-order execution capabilities as the main culprits of poor performance and lack of bandwidth saturation. Furthermore, they showed how CSR is not an appropriate matrix-storage format for Sparse Matrix-Vector Multiplication (SpMVM) and how the SELL-C-σ format improves performance.

Jackson et al. [32] investigated the performance of various applications on the A64FX. They saw that most applications and benchmarks were ported to SVE with minimal effort and no code changes required. In addition, they observed excellent performance on the A64FX, which often outperformed other Arm and top-of-the-range Intel processors. However, they also reported worse performance for a few benchmarks and concluded that further work is needed to optimize math libraries for the A64FX processor.

Chapter 3

Methodology

3.1 Auto-vectorization

Automatic vectorization of compilers has evolved rapidly over the recent decade. Both GCC and LLVM compiler technologies have added new features that recognize more vectorization opportunities and better predict the benefit of potential vectorization. Since new features are gradually released, auto-vectorization can be different for different compiler versions. Compilers also introduce various flags that impact how vectorization is treated. The major optimization levels (`-O2`, `-O3`, `-Ofast`) enable a different set of optimizations, and these are not exactly the same for different compilers. Additionally, optimizations for a particular architecture or a CPU (`-march` or `-mtune`) introduce another degree of freedom. For these reasons, we do not focus on the pros and cons of a particular compiler and how different optimization levels impact vectorization. Instead, we focus on how compilers interact and adopt SVE. We analyze three compilers: GCC 11.1.0, Arm Compiler for Linux (ACfL) 22.0.1, and the Clang version of the Fujitsu compiler 4.7 (FCC).

We evaluate the compiler's ability to vectorize loops with SVE instructions using the TSVC benchmark. Although all compilers can compile fixed-size SVE code, we focus on the generation of VLA binaries. For each compiler, we enable **all** optimizations which we deem relevant for vectorization. Most importantly, we enable the `-Ofast` optimization, which sets the `-ffast-math` flag. This flag enables a set of optimizations that break the IEEE floating-point compliance. In GCC, we also enable `-fivopts` which enables induction variable optimization on trees and use `aarch64-auto-vectorization-preference` to only vectorize loops with SVE. For ACfL and Fujitsu compiler, `-ffp-contract=fast` is used to enable fused multiply-add operations. In GCC, this is enabled by default. In

ACfL, the option `-fsimdmath` allows the compiler to generate calls to vectorized ArmPL library routines. Table 3.1 shows flags used for each compiler.

GCC (gcc)	ACfL (armclang)	FCC (fcc -Nclang)
`-march=armv8.2-a+sve` `-Ofast -fivopts --param` `aarch64-autovec-preference=2` `(-fno-tree-vectorize)`	`-march=armv8.2-a+sve` `-Ofast -fsimdmath` `-ffp-contract=fast` `(-fno-vectorize)`	`-march=armv8.2-a+sve` `-Ofast -ffp-contract=fast` `(-fno-vectorize)`

Table 3.1: TSVC compiler flags

We are interested in how well different features of SVE are exploited and whether a particular family of loops possesses any problems for SVE. We rely on both static and dynamic analysis. First, we check the compiler-generated code by inspecting assembly and reading the compiler optimization reports. For dynamic analysis, we rely on the A64FX processor, where we compare the benchmark execution times. We introduce the vector speed-up

$$\eta = \frac{t_{scalar}}{t_{vec}} \tag{3.1}$$

as the ratio between execution times of scalar and vectorized binaries. Here, t_{scalar} and t_{vec} refer to the time measurement from the beginning of the first iteration until the completion of the last iteration. Scalar binaries are produced using the same flags but adding `-fno-tree-vectorize` and `-fno-vectorize` to disable vectorization. We set up the same framework as Maleki et al. [33], where a speed-up of at least 1.15 is required to call a vectorization beneficial. The array lengths were set to `LEN_1D=8000` and `LEN_2D=80`. This way, the whole data fits in the cache, avoiding any bottleneck arising from the memory bandwidth. Each loop is executed 1000 times.

Afterward, we look closely at loops where all compilers fail to vectorize. In these cases, we try to manually vectorize loops with intrinsic functions. Finally, for loops where we succeed with vectorization, we analyze whether compilers failed specifically due to VLA/SVE or some other optimization technique. In the latter, we try to classify missed opportunities according to optimization techniques. We also ask ourselves if simple changes in the SVE ISA could increase vectorization opportunities.

3.2 Application setup

3.2.1 Computational patterns

To evaluate the potential of SVE in real-life workloads, we select a set of applications and libraries that capture some of the most common computational patterns in the HPC domain. In 2004, Collela identified seven computational dwarfs covering a wide range of applications important in science and engineering [34]. In 2006, this list was expanded to thirteen by Asanovic et al. [35]. Each dwarf describes a common computation and data movement scenario. Here, we give a brief overview of the original seven dwarfs:

1. **Dense Linear Algebra**. Matrices and vectors are dense, and memory accesses are often consecutive (unit-stride).
2. **Sparse Linear Algebra**. Matrices are sparse and usually stored in a compressed format to reduce storage and bandwidth requirements.
3. **Spectral methods**. Data is stored in the frequency domain rather than the spatial/time domain. The most common computation is the calculation of the Fast Fourier Transform (FFT).
4. **N-Body methods**. Such problems compute interactions between many discrete points. Moreover, each point depends on other points, leading to a $O(n^2)$ complexity. Typical examples are a simulation of atomic particles or stars in the galaxy.
5. **Structured grids**. A numerical solution is computed on a regular grid (nested rectangular array). Local stencil operators are applied to each rectangle.
6. **Unstructured grids**. Data is computed on an irregular grid. Therefore, we must first determine a list of neighboring points to update a grid point.
7. **Monte Carlo simulations**. Repeated random trials to compute a statistical result. Monte Carlo is often regarded as embarrassingly parallel with little communication between processes.

For our work, we choose four applications/libraries that cover the different dwarfs listed above. OpenBLAS (see Section 2.2.2) is a package for BLAS functions. We mostly focus on level 3 BLAS routines which are described by the Berkely dwarf 1. Although the performance of OpenBLAS does not always match that of commercial libraries from hardware vendors, it is still actively developed and widely used. The second application we choose is GROMACS, which belongs to dwarf 4. GROMACS (see Section 2.2.3) supports a wide range of molecular dynamics simulations and is heavily used in the pharmaceutical field. Parts of the application (long-range interactions) are also covered in dwarf 3. GPAW (see Section 2.2.4) is an application for materials science and electronic structure calculation. Due to many different modes of computation, GPAW encompasses various computational patterns. However, we focus on kernels belonging to stencil operations on a regular grid (dwarf 5). The last application is MiniFE (see Section 2.2.5), a mini-application for finite element methods. Although the solution is evaluated on a regular grid, the problem is formulated implicitly by assembling a sparse linear system. Therefore, we treat it as dwarf 2.

Another way to classify different HPC kernels is by analyzing the arithmetic intensity. For example, let us denote W as the work done by the CPU, which is typically measured in the number of floating-point operations. In some cases (not for the applications we consider), it may be better to count the number of integer operations. Next, we define the number of bytes transferred from memory as Q. Unlike W, the memory transfer depends on the platform parameters due to different cache hierarchies. Unless the kernel is very simple, one usually estimates Q only asymptotically and must rely on measurements for exact computation. The arithmetic intensity is defined as

$$AI = \frac{W}{Q}. \tag{3.2}$$

Arithmetic intensity AI is often used together with a roofline model [36]. The roofline model combines two platform-specific performance ceilings in a single plot, first due to the processor's peak performance and second due to the memory bandwidth. We refer to this as applications being compute-bound or memory-bound. The arithmetic intensity where the two ceilings connect is known as the ridge point. Despite its simplicity, the roofline model can provide an insightful visualization of a bottleneck in the system. Additionally, the model has been extended to account for caches (Execution-Cache-Memory model) [37].

Our four applications cover both aspects of the roofline model. OpenBLAS and GROMACS are compute-bound, and GPAW and MiniFE are memory-bound on most systems. Consequently, selected applications have different exposures to SIMD parallelism and porting efforts. For compute-bound kernels, good utilization of SIMD is crucial, and OpenBLAS and GROMACS rely on assembly and intrinsic functions. On the other hand, GPAW and MiniFE are less reliant on SIMD, and the SIMD instructions are generated only through compiler auto-vectorization.

3.2.2 Application hot spots

Hot spots are performance-critical parts of code where the application spends the most time. Consequently, these parts of codes are most interesting for optimization. For each of the four applications, we first identify hot spots. In some cases, these are easily identified if we know where the underlying algorithm is implemented. Additionally, some HPC applications provide such information at the output. For others, we have to resort to profiling tools. Application profiling is a standard method for analyzing a program's performance. Developers profile applications to measure various statistics about the application's execution. For our work, we are interested in the execution time spent in different code sections.

In this thesis, we use HPCToolkit (see Section 2.4) to find hot spots of applications. To use HPCToolkit, we first prepare the application for measurement. Overall, HPCToolkit is easy to use and does not require any manual code instrumentation. However, a helpful step is to compile the application with a debug flag `-g`. This way, the compiler records information about inlining and how instructions in code are mapped to assembly. There are no restrictions on the application's build system. For dynamically linked binaries, the `hpcrun` command automatically instruments the code for measurement. However, if the application is statically linked, the command `hpclink` is used to link the tool's monitoring code into the application. Afterward, we use the `hpcviewer` command to analyze the results.

3.2.3 Benchmark preparation

From our selected applications, only GROMACS offers explicit support for SVE. Therefore, a significant effort was made for SVE exploitation. We present these porting efforts in Chapter 4, where we also explain details of applications and their underlying algorithms.

Despite Gem5's design efforts to improve the simulator's performance, long simulation times are still the primary constraint of our work. Gem5's performance depends on the detail of the simulated model and the execution context. Our experience shows that the simulation rate of Gem5 is approximately five orders of magnitude slower than the execution on real hardware. (One simulated second in Gem5 takes approximately one day in real time.[1]) Therefore, we use Gem5 only for final simulations and rely exclusively on ArmIE when testing and modifying the applications. Among HPC simulation codes, long runtimes are standard for production runs in state-of-the-art research. However,

[1]This is only a very rough estimate.

due to a slow simulation rate, running such use cases is not feasible in Gem5. To solve this, we scale all applications to the total runtime of approximately one second on a single core. We do this differently depending on the application.

OpenBLAS DGEMM

For our analysis, we focus on the double-precision matrix-matrix multiplication. We write a simple **DGEMM benchmark** that measures the execution time of OpenBLAS's DGEMM function. The benchmark first defines arrays for matrices and fills them with random values. Afterward, it executes a series of calls to OpenBLAS's DGEMM function in a *for* loop. In each iteration, the execution time is measured using *gettimeofday()*. The benchmark accepts two command-line arguments, specifying the matrix dimensions and the number of iterations. Scaling down the time of DGEMM computation is simple because it is closely related to the matrix size. The best performance across iterations is reported in GFlop/s. The number of floating-point operations is calculated as $mn(2k+2)$, where m, n, and k are dimensions of matrices. The code is publicly available[1].

For our experiments, we link the benchmark to OpenBLAS version 3.20. Release 3.20 includes our SVE changes presented in Chapter 4.1. Additionally, we have removed the *mbind* system call which is not supported in Gem5. The build system in OpenBLAS is very straightforward. For native compilations, the program recognizes the underlying hardware and configures the compilation accordingly. When cross-compiling, the user selects the build target from a list of supported architectures. Because the Gem5 model is not a real machine, OpenBLAS does not have any targets that would correctly set the block sizes based on the Gem5 hardware parameters. Therefore, we modified the pseudo target `ARMV8SVE` to match the architecture of the Gem5 model. This involved correctly setting the size, associativity, and cache line size for different cache levels. Afterward, we cross-compiled the library with:

```
make TARGET=ARMV8SVE CROSS=1 USEOPENMP=1
```

Listing 3.1: OpenBLAS compilation command

We use a GCC compiler, version 11.1.0. For experiments on Graviton 2 and A64FX, we only change the target option to `TARGET=NEOVERSEN1` and `TARGET=A64FX`, respectively. OpenBLAS includes both statically and dynamically linked libraries by default.

[1]https://github.com/binebrank/DGEMM-benchmark

GROMACS

GROMACS offers many standard use cases for users who quickly wish to test or benchmark the application. However, most use cases are too big to be executed in the Gem5 simulator. We, therefore, choose use cases of small molecular systems where we also decrease the number of iterations of the MD simulation. Since the computational pattern of each iteration is the same, we do not lose any information by focusing only on a few iterations. For our simulations, we focus on the simulation of Ribonuclease protein in a water solvent and a box of pure water molecules. The input configurations (*rnase_bench_systems* and *water_GMX50_bare*) were taken from GROMACS's official repository. Additionally, we run simulations using GROMACS's internal nonbonded benchmark. This benchmark is shipped as part of the code and is frequently used to measure performance. The nonbonded benchmark repeatedly executes the *Nbnxm* nonbonded kernel that computes the short-range nonbonded forces between particles. As we explain in Chapter 4.2.2, this is the most compute-intensive part of the simulation. We run the benchmarks for ten iterations. An example of the command line for nonbonded benchmark is shown in Listing 3.2.

```
gmx nonbonded-benchmark -size 1 -simd 4xm -nt 1 -iter 10 -warmup 2 -time
```

Listing 3.2: Nonbonded benchmark execution command

To evaluate GROMACS, we use version 2021.5, to which we added changes explained in Section 4.2.3. Using this version initially failed in Gem5 due to incorrectly aligned addresses in the SVE implementation, which we fixed. Additionally, we experienced problems with the GROMACS's timing measurement. The application records time by reading high accuracy counters in the CPU. Due to a system-emulation mode of Gem5, these counters are not available for Gem5 simulations. We, therefore, modified the code to measure time with *gettimeofday()*. The modified code is available on Gitlab[1]. We compiled the application using GCC 11.1.0 and target SVE with a command-line option `-DGMX_SIMD=ARM_SVE`. Because the SVE size needs to be set at compile-time, we have build three different versions, separately for 128-, 256-, and 512-bit SVE. For Graviton 2 runs, we build GROMACS the same way but with the option `-DGMX_SIMD=ARM_NEON_ASIMD` which targets the NEON architecture. The application features CMake build system and has very few external dependencies. It implements an internal library for Fast Fourier Transformations and BLAS functions. However, these can also be linked to external libraries. For our compilation, we link GROMACS to FFTW version 3.3.9. Although there has been some effort for SVE in FFTW [38], the

[1]`https://gitlab.com/binebrank/gromacs/-/tree/phd_gem5`

official release does yet support it. The full command line for 128-bit SVE is shown in Listing 3.3.

```
module load CMake FFTW
FLAGS="-march=armv8-a+sve -msve-vector-bits=128 -static -fopenmp "
FLAGS+="-DGEM5ROI" # used for Gem5 region of interest

cmake .. -DCMAKE_C_COMPILER="gcc" -DCMAKE_CXX_COMPILER="g++" \
         -DCMAKE_BUILD_TYPE=Release \
         -DCMAKE_C_FLAGS_RELEASE="-O2" -DCMAKE_CXX_FLAGS_RELEASE="-O2" \
         -DCMAKE_C_FLAGS="$FLAGS" -DCMAKE_CXX_FLAGS="$FLAGS" \
         -DGMX_MPI=off -DGMX_OPENMP=on -DGMX_GPU=off -DGMX_DOUBLE=off \
         -DGMX_SIMD=ARM_SVE -DGMX_SIMD_ARM_SVE_LENGTH=128 \
         -DGMX_BUILD_SHARED_EXE=off -DGMX_PREFER_STATIC_LIBS=ON \
         -DGMXAPI=off -DGMX_THREAD_MPI=on -DGMX_IMD=off -DGMX_USE_TNG=off
```

Listing 3.3: GROMACS build command

GPAW

To study GPAW, we rely on use cases from the Unified European Applications Benchmark Suite (UEABS) [39]. However, all use cases are too big for a Gem5 simulation. To solve the problem, we have extracted two of the most computationally expensive functions out of the application. (See Section 4.3.3.) Both functions are therefore executed as standalone benchmarks without any external dependencies. The code is publicly available[1]. For analysis of application hot spots, we use version 22.1.0 and compile the code with Python 3.9.4 and GCC 11.1.0. Additionally, we rely on OpenMPI 4.1.2, OpenBLAS 0.3.18 and FFTW 3.3.9.

MiniFE

In the case of MiniFE, we evaluate the *openmp4.5* implementation of version 2.1. The only input parameter for MiniFE is the size of the grid, where the problem is solved. The default size, $10 \times 10 \times 10$, is very small (even in the context of the Gem5 simulator), so we evaluate cases where the overall runtime on real hardware is approximately one second. As we explain in Section 4.4.4, we added the Sliced ELLpack matrix format and implemented the SpMVM kernel with intrinsic functions for better utilization of SVE. For the Gem5 statistic measurement, we inserted the ROI markers around a single iteration of matrix-vector multiplication in the CG method. The code is publicly available [2].

[1] `https://gitlab.jsc.fz-juelich.de/brank1/gpaw-benchmarks`
[2] `https://gitlab.jsc.fz-juelich.de/epi-wp1-public/minife`

When testing the MiniFE code in the Gem5 simulator, we noticed strange behavior if we compiled the code with the auto-vectorization enabled (both NEON and SVE). For specific input configurations, Gem5 simulations failed with segmentation fault errors. However, we could not reproduce these errors on real hardware, which suggests that the problems occurred because of bugs in the simulator. To avoid this problem, we have disabled auto-vectorization with the `-fno-tree-vectorize` flag and also applied manual vectorization for the CSR matrix format. Additionally, we set the number of cores in the Gem5 model to match the number of OpenMP threads due to the unimplemented *wait* system call. (This call is redundant if the number of cores matches the number of threads.) We limit the CG method to 100 iterations to avoid long simulation times. However, this is relevant only when the grid size $n_{x,y,z} \geq 30$, because the CG method converges in less than 100 iterations for smaller input sizes. The full command line used for the compilation of SVE binaries for Gem5 is shown in Listing 3.4.

```
gcc -O3 -fno-tree-vectorize -fopenmp -march=armv8-a+sve -static
    -DMINIFE_SELL_MATRIX
    -DMINIFE_SCALAR=double
    -DMINIFE_LOCAL_ORDINAL=int
    -DMINIFE_GLOBAL_ORDINAL=int
```

LISTING 3.4: Compilation command for MiniFE

3.2.4 Region Of Interest

In our analysis, we focus on application hot spots. However, because our workloads are significantly reduced in size due to the Gem5 simulator, the hot spot might take a smaller portion of the total runtime due to the initialization phase. Therefore, we only want to measure and collect the data during the hot spot execution. Both Armie and Gem5 support measurement in the Region Of Interest (ROI): a user-defined code section. This way, we only measure data for parts of the code we are interested in.

By default, the ArmIE instrumentation client collects data for the entire program. Therefore, we must instrument the code with special markers (hardware instructions) to measure data only for a specific code section. Listing 3.5) shows an example of this. Additionally, the option `-a -roi` has to be used at the command line; otherwise, the markers are ignored.

```
#define __START_TRACE() { asm volatile (".inst 0x2520e020"); }
#define __STOP_TRACE() { asm volatile (".inst 0x2520e040"); }

int main() {
...
__START_TRACE();
// measurement region
__STOP_TRACE();
...
}
```

Listing 3.5: ArmIE Region of Interest

Without source modification, Gem5 also outputs statistics for the whole application runtime. Similar to ArmIE, ROI markers are provided to specify where Gem5 should reset and dump statistics in the application. Listing 3.6 shows an example of this in practice, where three ROIs are separated. First from the start of execution until line 5, second for the hot spot, and third from line 7 until the end of the program. Additionally, one has to link the application to the Gem5 library.

```
1 #include "gem5/m5ops.h"
2
3 int main() {
4 ...
5 m5_dump_reset_stats();
6 // hot-spot
7 m5_dump_reset_stats();
8 ...
9 }
```

Listing 3.6: Gem5 Region of Interest

3.3 Gem5 model

This section explains how we configure the Gem5 model for our experiments. To make our results applicable to real hardware, we want our model to be a realistic representation of the current design of Arm cores. We achieve this by configuring the model based on the Neoverse N1 core (Graviton 2 CPU). Most of the architectural parameters for N1 are publicly available. With the AWS, the Neoverse N1 has become one of the most widely used Arm processors in the server market.

In our model, we use an out-of-order *O3CPU* core with a *classic* memory system. As observed by some studies, we expect that the SVE ISA and different SVE sizes will significantly impact the core architecture design rather than the memory system. We, therefore, decided to keep our focus on a detailed configuration of the core model and make simplifications in our cache/memory configuration. Furthermore, using a simplified memory reduces the runtime of our simulations. Although our simulations primarily focus on a single core, we build a multi-core system with four cores. Of course, Gem5 supports much bigger systems (up to 64 cores), but we noticed that the increased number of cores makes the simulator less stable and more prone to failures.

3.3.1 *O3CPU*

Before explaining our configuration, we give an overview of the *O3CPU* model in Gem5. We frequently reference some of the Gem5 internal workings for the rest of the thesis, and it is essential to know what exactly we are simulating. *O3CPU* is a detailed out-or-order model composed of seven pipeline stages. One of the main features of the Gem5 model is that the ISA is decoupled from the CPU model. Therefore, none of the CPU models is explicitly limited to a particular architecture. This is implemented by splitting the instruction's static information from its dynamic execution.

The *StaticInst* class describes the static information for each binary instruction. This class contains information about instructions' operation code, the source and destination registers, and into which micro-instructions it decodes. The primary method *execute()* defines all architectural movements and mathematical operations that the instruction performs. In the case of memory instruction, the process is split into address calculation and the actual memory access, depending on the architecture. Additionally, each instruction is assigned its operation class. Gem5 groups similar instructions into operation classes called *OpClass* which simplifies specific steps in the core backend. An example of an operation class is *IntALU* which groups simple integer ALU operations (add, subtract, shift, bit operations, etc.). To increase the simulator performance, a *StaticInst* object for every binary instruction is stored in a hashmap. This way, each machine instruction is decoded only once. Note that all the information in StaticInst is derived exclusively from the ISA, and there is no connection to its dynamic execution.

The *DynInst* class is a class that includes all dynamic information about the instruction. This consists of a program counter (PC), renamed source and destination registers, the result after the execution, information about the thread number, etc. This class is instantiated for each binary instruction together with a *StaticInst.* This enables the *O3CPU* model to work for different ISAs because only the *StaticInst* changes while dynamic execution stays the same. When the simulation is running, the *DynInst* provides an interface to the execution context to modify the CPU state (physical registers, program counter, memory access). Pointers to each *DynInst* object pass through seven pipeline stages. The stages are shown in Figure 3.1. (We also show the most important high-level functions in the calling chain for non-memory instructions.)

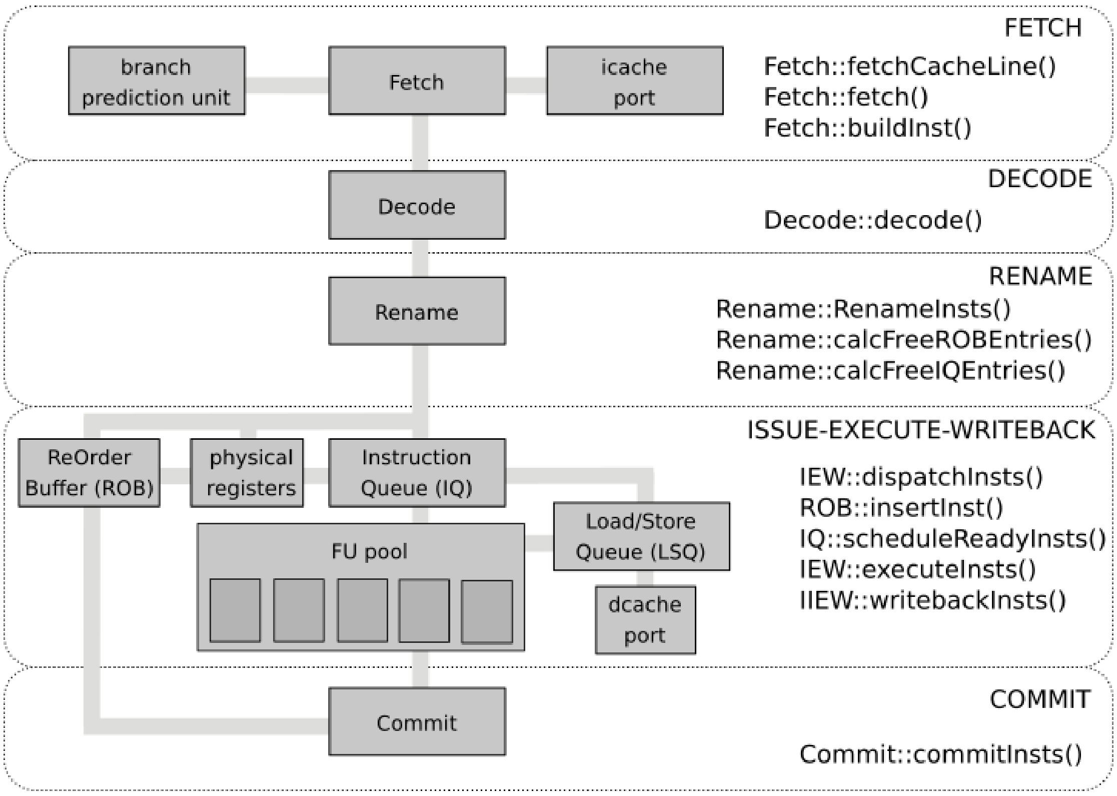


FIGURE 3.1: Out-of-order pipeline in the O3CPU model

1. At **Fetch** stage, instructions are fetched from the instruction cache. Each fetched instruction creates instances of the *DynInst* and *StaticInst* classes.

2. **Decode** stage models instruction decoding and PC resolution of unconditional branches. (The actual decoding of binary instructions is done when the *StaticInst* instance is created.)

3. **Rename** stage handles register renaming and configuration of the out-of-order execution. Each micro-instruction is renamed from architectural registers to physical registers, and the history of renaming is kept. When the branch is falsely predicted, the instructions are squashed, and mappings are reverted to the correct stage of the program. This stage stalls when there are insufficient physical registers or back-end resources.

4. Gem5 combines **Issue**, **Execute**, and **Writeback** into a single stage (**IEW**). This stage first dispatches a micro-instruction to the instruction queue (IQ) and creates an entry in the reorder buffer (ROB). The IQ is a single unified buffer that holds all micro-instructions. In this sense, the Gem5 model is more similar to the N1 than

the A64FX processor, which has multiple reservation stations. (See Section 2.3.2.) The IQ waits until all operands are available before issuing the micro-instruction to one of the units in the *FUPool*. At the same time, the IQ obtains the execution latency i_{lat} from the FU. Then, on the last execution cycle, the write-back stage signals the result to the IQ, which issues dependent micro-instructions. This allows back-to-back scheduling. For a memory operation, the *execute* function creates a new entry in the LSQ. In case of a load, the LSQ issues a cache request to the L1 data cache. For a store instruction, the LSQ waits until the instruction is committed to issue a cache request. An executed instruction is removed from the IQ but stays in the ROB until it is committed.

5. Finally, **Commit** stage commits instructions back in order and handles redirect in case of a branch mispredict. Committed instructions are removed from the ROB.

Each pipeline stage operates in different states depending on the input, available resources, and a possible branch mispredict. Additionally, each stage features a buffer (*skidBuffer*), where current instructions are stored in case of a stall. Table 3.2 shows an overview of the five main execution states.

Idle	The stage receives no input from the previous stage and waits for instructions.
Run	The stage is in a normal operation state. For example, the decode stage receives instructions from the fetch stage and outputs decoded instructions to the rename stage.
Block	The stage stops its execution due to a lack of resources or a signal from another stage. During this state, no instructions are processed. When a block occurs, the stage signals to the previous stage to block. All instructions currently in this stage are put in the *skidBuffer* for processing when the stage unblocks.
Unblock	The stage unblocks itself due to available resources or a signal from another stage. When this happens, the stage first starts processing instructions in the *skidBuffer*. When all instructions from the buffer are processed, the stage enters an idle state and signals the previous stage to unblock.
Squash	Occurs in the case of a branch mispredict. All instructions are squashed, and the *skidBuffer* is emptied. The instruction thread reverts to the last retirement state, and when new instructions are available, the stage enters the run state.

Table 3.2: Gem5 pipeline states

3.3.2 Core configuration

As explained in Section 3.3, our goal is configuring the Gem5 model to resemble the Arm Neoverse N1 core, which is also implemented in the Graviton 2. Our starting point is the *Arm_O3_v7* model, a default Gem5 configuration of the *O3CPU* model for Arm architectures. This core is loosely based on a rather old Cortex A15 core. Therefore, we significantly modify most aspects of its microarchitecture.

The main configuration step of the *O3CPU* is defining a set of functional units in the core backend. The *O3CPU* model includes an object *FUPool* which describes units that execute operations. Gem5 simplifies the configuration of functional units by combining similar instructions into groups. These groups are called operation classes (class *opClass*, see Section 3.3.1). Each unit includes a list of operation classes (instructions) that it can execute. We configure the units of our model to match those in the Neoverse N1 (see Figure 2.11 in Section 2.3.1) with a few exceptions. Firstly, Gem5 groups branch instructions together with arithmetic instructions in the operation class *IntAlu*. Therefore, we could not create a separate FU for branch instructions but have combined the two types. Three FUs for integer and branch instructions are accompanied by an additional FU for complex integer operations. (These are equivalent to *B, I1, I2, M* in the N1). We also add two units for floating-point and SIMD (SVE) operations. Here we make a minor simplification and make both units execute all types of floating-point operations. (This includes *fsqrt* and *fdiv*, which can be executed only on one FU in the N1). Additionally, we add a functional unit for predicate operations, which is not used for NEON binaries. Finally, two units are added that execute the memory operations. Gem5 treats gather-load and scatter-store instructions similarly to the A64FX. All gather-load and scatter-store instructions are decoded into l_{SVE} micro-instructions. The entire FU pool, with main operation classes for each FU, is shown in Figure 3.2. (Compare with 2.11).

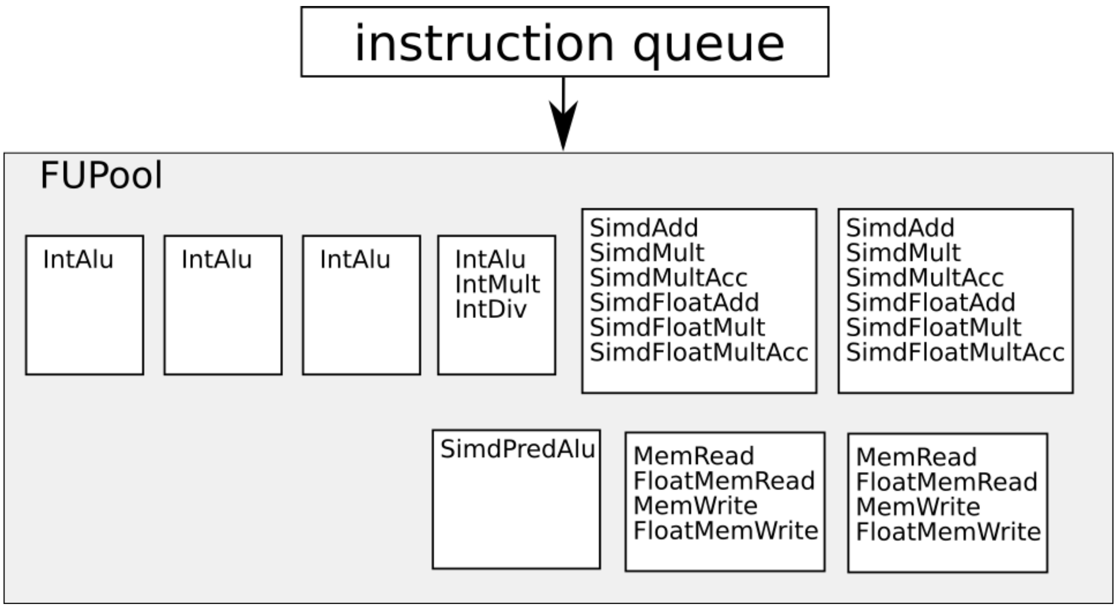

instruction queue
FUPool
IntAlu
IntAlu
IntAlu
IntAlu IntMult IntDiv
SimdAdd SimdMult SimdMultAcc SimdFloatAdd SimdFloatMult SimdFloatMultAcc
SimdAdd SimdMult SimdMultAcc SimdFloatAdd SimdFloatMult SimdFloatMultAcc
SimdPredAlu
MemRead FloatMemRead MemWrite FloatMemWrite
MemRead FloatMemRead MemWrite FloatMemWrite

FIGURE 3.2: Gem5 functional units

Configuration of the *FUPool* also requires setting the execution latencies of different instructions. In Gem5, these are set for each operation class. Our goal was to configure execution latencies to match those on the Neoverse N1. However, we noticed that operation classes for Arm ISA in Gem5 are set too broadly. For multiple cases, instructions with different characteristics and latencies in N1 are put in the same operation class. This makes it impossible to set the latencies of all instructions to match precisely. Additionally, reconfiguring Gem5 operation classes would require a significant source code change, which is not our goal. We solve this problem by setting the latency of each operation class to the latency of what we think is the most commonly executed instruction of that class. See Appendix B for detailed documentation of the *FUPool* and the latencies used in our model.

Here, special care was taken for the SVE in-order reduction instruction *fadda* (see Section 2.1.2.3). Instructions like this one are special because their execution is not done simultaneously for all lanes in the SVE register. To model this correctly, we look at how these instructions are handled on the A64FX. First, we notice that the execution latency of *fadda* in the A64FX depends on the SVE size (latency gets bigger for larger SVE size). Additionally, these instructions are sequentially decoded into l_{SVE} micro-instructions and dispatched in order. We imitate this behavior in Gem5 by splitting the model for three SVE sizes (128, 256, and 512 bits) and setting the latency for each SVE size separately. Additionally, we configure these instructions as not pipelined to mimic the sequential decode. Non-pipelined instructions block the execution unit until the instruction is executed. For example, on real hardware, this is a typical scenario for *fsqrt* and *fdiv* instructions that require multiple iterations of the Newton-Raphson method to compute the result. We know that such a configuration is not ideal, but it is more realistic than a fully pipelined operation.

Parameters of other stages of the core pipeline are also set to realistic values of real hardware. In Table 3.3, we report the main parameters that match the microarchitecture of the Neoverse N1. We found most of these parameters in the Arm public documentation [19]. For other parameters, we relied on the unofficial internet sources.[1] [2]

reorder buffer size	128
load queue size	68
store queue size	72
instruciton queue size	120
# integer registers	120
# floating-point/vector registers	128

fetch width	4
decode width	4
rename width	8
dispatch width	8
issue width	8
write-back width	8

TABLE 3.3: Gem5 pipeline buffer widths

3.3.3 Cache configuration

To configure the memory system of the Gem5 model, we rely on Gem5's *classic* memory model. This introduces many simplifications but makes the simulation faster. Also, our goal is not a detailed simulation of the Graviton 2's memory system.

Caches in our Gem5 model resemble Graviton 2 with a significant change in the system-level cache. Each core features a private L1 instruction and data cache and a unified L2 cache. Latencies of the caches were configured to match those reported in the documentation of Graviton 2. In addition, we use a Least Recently Used replacement policy (LRU) for all cache levels. Configuring a shared SLC cache with multiple slices connected over a mesh-interconnect would require an advanced Ruby memory model. We avoid this and keep a simple architecture with a unified shared L3 cache that is connected to private L2 caches via a coherent crossbar. On the memory side, the L3 is connected to a memory controller. Because our model comprises only four cores, the L3 cache size was scaled down from 32 to 8 MByte. Additionally, we set the number of Miss Status Holding Registers (MSHR) to 20 and 46 for the L1-D and L2 cache, respectively. MSHR (also called miss buffer) tracks the cache line misses that are being fetched from lower cache levels. Main cache parameters are shown in Table 3.4.

[1]`https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_n1`
[2]`https://chipsandcheese.com/2021/10/22/deep-diving-neoverse-n1/`

L1-data		L2		L3	
size	64 kByte	size	1 MByte	size	8 MByte
tag latency	3	tag latency	5	tag latency	48
data latency	3	data latency	5	data latency	48
associativity	4	associativity	8	associativity	16
clusivity	mostly_incl	clusivity	mostly_incl	clusivity	mostly_excl

Table 3.4: Cache parameters of the Gem5 model

The hardware prefetcher is an important part of the cache system in Graviton 2. Unfortunately, both Arm and Amazon do not disclose much information about it. However, some sources [19] suggest that several prefetchers exist that load data in multiple cache levels and TLBs. They support stream, strided, and spatial memory pattern recognition. We use a tagged prefetcher with degree 32 at the L2 cache in our model. This decision was partially based on the results of the STREAM benchmark for a single core.

We are aware that in this aspect, our Gem5 configuration is very different from the reference hardware architecture, Graviton 2. Because the hardware prefetcher can be the main performance factor for single-threaded memory-bound kernels, we point out again that results in the Gem5 simulator in these cases can be quite different than on the Graviton 2. Figure 3.3 shows an overview of the entire cache hierarchy in our Gem5 model.

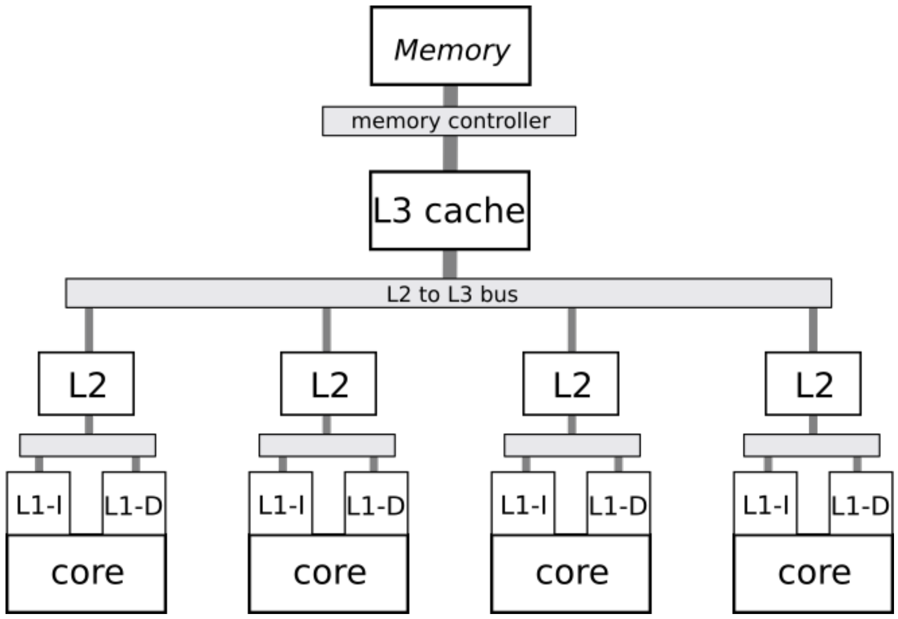


Figure 3.3: Gem5 cache configuration

3.3.4 Memory configuration

Gem5 supports DDR, GDDR, and HBM memory technologies. In our model, the caches and interconnect are so simplified that it makes little sense to try to mimic the memory technology of Graviton 2 with a detailed model. Therefore, we configure the random access memory with the Gem5's *simpleMemory* model. This is an easy-to-configure memory system with two parameters: latency and bandwidth. We set the bandwidth to 25.6 GByte/s and latency to 50ns. Similar to the number of cores, we also scale down the number of memory channels and add a single bank of such memory. Therefore, the total bandwidth (25.6 GByte/s) is eight times lower than on Graviton 2.

3.4 SVE static analysis

SVE enhances the ARMv8-A ISA with many new features compared to NEON. To study how these features translate to the code, we analyze SVE binaries with different ArmIE instrumentation clients. Although one can write custom DynamoRio clients, we use preconfigured clients with ArmIE.

To count the number of SVE instructions in a program, we use ArmIE together with the `libinscount_emulated.so` client. This client outputs the number of emulated instructions, giving a simple overview of SVE utilization. Additionally, we frequently rely on `libopcodes_emulated.so`. This client reports the histogram of decoded SVE instructions in binary form. However, ArmIE provides the python script *enc2instr.py* that leverages the llvm-mc machine code analyzer to decode the operations. For further analysis, we write an additional python script that sums instructions with the same operation code and different input operands.[1] With this, we can count the number of individual SVE instructions. Finally, `libmemtrace_sve<sve>.so` collects simple memory traces of SVE instructions. Most importantly, this gives us insight into the amount of gather-load and scatter-stores instructions. Additionally, we see the number of memory operations with fully and partially active lanes.

We also read the number of instructions in Gem5. Gem5 separates the count for different stages of the *O3CPU* model (number of fetched instructions, decoded instructions, executed instructions, etc.). These mainly depend on the simulated architecture. However, the number of committed instructions I is independent of the architecture. Unlike Armie, Gem5 does not report the number of individual SVE instructions. Instead, most statistics are based on operation classes (see Section 3.3.1). The name of the class *OpClass* is usually self-explanatory. For example, class *SimdFloatMultAcc* refers

[1] `https://github.com/binebrank/phd_scripts/tree/main/armie_scripts`

to different forms of *fmla* and *fmad* instructions. We also read the number of decoded micro-instructions, which are more relevant when analyzing the core's backend. The reason for this are instructions that are decoded into multiple micro-instructions. Table 3.5 shows the most important metrics in our analysis.

Symbol	Gem5 event name	Description
I	committedInsts	Committed instructions
μ	committedOps	Committed micro-instructions
I_C	committedInsts::*OpClass*	Committed instructions for *OpClass C*
μ_C	not available	Committed micro-instructions for *OpClass C*
I_{SVE}	not available	Committed SVE instructions

TABLE 3.5: Gem5 static analysis counters

3.5 Architectural exploration

Hardware simulators are often used to study the impact of hardware parameters on the application's performance. Such simulations also provide computer architects with feedback about how different hardware components work together throughout the execution. An important step for software/hardware codesign is to specify the design and analysis space. The design space includes a list of architectural parameters that are changed during simulations. The analysis space defines parameters for studying the hardware and application's response.

Usually, the design and analysis spaces are tightly connected to the simulator's capabilities and accuracy. Therefore, the full codesign methodology often involves multiple simulation environments. In our work, we only use Gem5, which provides much information for microarchitectural analysis.

3.5.1 Design space

Gem5 allows users to easily repeat experiments with different sets of architectural parameters. However, simultaneously changing many parameters expands the exploration space into many dimensions making analysis more difficult. The modern microarchitecture is very complex, and different components' behavior is often coupled. For example, let us assume that poor performance results from a bottleneck in one microarchitectural component. Increasing the capability of this component will not altogether remove the bottleneck but merely shift it (in a reduced form) to some other part of the system.

Therefore, a usual approach is to analyze behavior by changing only one parameter at a time and keeping others fixed.

Since SVE is a vector-length-agnostic, the first question is how the microarchitecture responds to different SVE sizes. This will be the main topic of our simulations. Although the size of SVE can range up to 2048 bits, the current trends of the SIMD width do not show any plans of going beyond 512 bits. Therefore, we perform experiments with 128, 256, and 512 bits sizes. However, increasing SVE size also increases the number of transistors and power consumption in real machines. To consider this, we also perform experiments where we change the number of SVE units while keeping the parameter Flop/cycle constant. For these experiments, we choose three configurations: 4x128-bit, 2x256-bit, and 1x512-bit SVE. Another point of interest is the A64FX core, which is architecturally very different from the Neoverse N1. The main difference is deep pipelines in execution units which results in higher instruction latencies. Where the performance in the A64FX differs significantly from our N1 model, we analyze this with an alternative Gem5 model, where the latencies are matched to the ones in the A64FX.

In our analysis, we mainly focus on the core backend. This involves studying the occupation of the functional units (see Figure 3.2) and the buffers for out-of-order execution in the core backend (see Figure 3.1). Here, we are limited to the *O3CPU* model, where the main components are ROB, IQ, physical registers, and LSQ. The sizes of these parameters are kept fixed (See Table 3.3).

3.5.2 Microarchitectural analysis

Gem5 simulator outputs an extensive report about the completed simulation. File *stats.txt* is generated alongside the model configuration and executable output. This file contains statistics for events of every object created in the Gem5 model. Analyzing events of architectural components gives us a deep insight into the behavior of the simulated architecture. Firstly, we are interested in the overall performance benefit of a bigger SVE size. Although all selected applications report performance, the overall runtime in cycles t_c can also be read from the number of simulated clock cycles *numCycles* in Gem5. To quantify results, we define the relative vector speed-up as the execution time for longer SVE width (e.g. η_{256}) compared to the 128-bit SVE (t_{128}).

$$\eta_{256} = \frac{t_{128}}{t_{256}}, \quad \eta_{512} = \frac{t_{128}}{t_{512}} \tag{3.3}$$

Additionally, we refer to a parallel efficiency for double-precision as:

$$\epsilon_{256} = \frac{\eta_{256}}{2}, \quad \epsilon_{512} = \frac{\eta_{512}}{4} \tag{3.4}$$

To see how well applications utilize instruction level parallelism, we often refer to the number of committed instructions per cycle *ipc*.

As discussed in the previous section, the second part of our analysis focuses on how SVE length impacts other components in the core's backend. The execution units in a Gem5 model are configured in a *FUPool* (See Section 3.3.2). Gem5 provides information on the occupancy of units with *fuBusy* events. These events are triggered if a micro-instruction can not be issued to the execution unit as soon as it is ready (has all operands available). This usually happens because the unit is stalled or if the CPU tries to issue many micro-instructions in the same cycle. *fuBusy* events are also available for each *OpClass* separately (see Table 3.6).

Symbol	Gem5 event name	Description
t_c	numCycles	Number of CPU cycles simulated
ipc	ipc	Instructions per cycle
B	fuBusy	FU busy when requested (Count)
B_{rate}	fuBusy.rate	B/μ
B_C	statFuBusy::*opClass*	FU busy for *opClass* C

TABLE 3.6: Gem5 execution counters

A large number of *fuBusy* events for a particular class of instructions shows that a core cannot handle all operations efficiently. This indicates a potential bottleneck in the execution stage. In the best case, an execution unit executes one operation per cycle. Therefore, we can estimate the lower bound for the total number of cycles. The minimum number of cycles needed to execute all SVE instructions is

$$t_c \geq \frac{I_{SVE}}{N_{SVE}} \tag{3.5}$$

Here, we can make a tighter bound if we consider that some instructions are not pipelined. As explained in Section 3.3.2, non-pipelined instructions (e.g. *fadda*) block the unit for the execution duration. For example, let I_{fadda} be the number of *fadda* instructions and lat_{fadda} its latency. The minimum time needed to execute all *fadda* operations is

$$t_c \geq \frac{I_{fadda} lat_{fadda}}{N_{SVE}} \tag{3.6}$$

The same analysis can also be applied to memory instructions. We do not have any non-pipelined instructions for units that execute memory operations because requests are automatically put in the LSQ. However, we have to separate normal and sequential memory operations. Gem5 decodes gather-load and scatter-store instructions into l_{SVE}

micro-instructions. In contrast, each normal load/store is decoded into a single micro-instruction. Therefore, if μ_{mem} is a total number of memory micro-instructions, the minimum number of cycles to execute these is

$$t_c \geq \frac{\mu_{mem}}{N_{mem}} \tag{3.7}$$

To evaluate how SVE size impacts out-of-order buffers, we choose the following statistics shown in Table 3.7. As the name suggests, *fullRegistersEvents* counts the number of events where all physical registers (either integer of SIMD) are fully occupied at the rename stage. Another point of interest is the reorder buffer which tracks the micro-instructions that execute out-of-order. A *ROBFullEvents* counts the number of events for a full reorder buffer. Additionally, *iqFullEvents* reports the number of times that micro-instructions could not be dispatched to the IQ because the queue was full. Finally, *lsqFullEvents* counts such events for the LSQ.

Gem5 event name	Description
fullRegistersEvents	All physical registers occupied
vecRegfileReads	Vector regfile reads
vecRegfileWrites	Vector regfile writes
ROBFullEvents	Reorder Buffer full
iqFullEvents	Instruction Queue full
lsqFullEvents	Load-Store Queue full

TABLE 3.7: Gem5 out-of-order counters

We can estimate some our-of-order bottlenecks from the buffer sizes. Let us denote l_{LSQ} and l_{ROB} sizes of the LSQ and ROB, respectively. Each memory operation in the LSQ is also present in the ROB. When the *lsqFullEvent* occurs, l_{LSQ} entries in the ROB are occupied by memory operations. Because the ROB stores operations in-order, the *lsqFullEvents* can only occur when the number of memory operations compared to other operations in the code reaches a critical factor

$$f_{LSQ} = \frac{l_{LSQ}}{l_{ROB} - l_{LSQ}}. \tag{3.8}$$

Otherwise, the ROB fills up before LSQ, and the bottleneck shifts to the *ROBFullEvents*. Our Gem5 model has a factor $f_{LSQ} = \frac{72}{128-72} = 1.29$. For loop kernels, the number of memory operations can be estimated by looking at the assembly and counting operations in one iteration. Predicting *iqFullEvents* is more complex because the IQ holds operations out-of-order. When an instruction is executed, it is removed from the IQ but stays in the ROB until it is committed. Therefore, the IQ can become full only when more than

$$f_{IQ} = \frac{l_{IQ}}{l_{ROB}} \tag{3.9}$$

micro-instructions in the ROB have not yet been executed. In our Gem5 model, $f_{IQ} = 120/128 = 94\%$. In other words, *iqFullEvent* occurs when less than 6% of instructions in the ROB have finished execution. We do not anticipate many *iqFullEvents* since these are likely hidden by the *ROBFullEvents*.

The number of events for a full buffer significantly impacts the out-of-order execution. Although Gem5 does not report the number of stall cycles for individual events, we look at the number of cycles spent in different pipeline states (see Table 3.2). Our main focus is the rename stage, where the pipeline blocks if the physical registers, ROB or IQ, get full. Table 3.8 shows the main counters for measuring the state of the rename stage. For analysis, we define the stall cycles as cycles from both idle and block states. (When unblocking, the stage is already processing instructions.)

Gem5 event name	Description
rename.idleCycles	Cycles in idle
rename.blockedCycles	Blocked cycles
rename.runCycles	Cycles in normal operation
rename.unblockCycles	Processing of the *skidBuffer*
rename.squashCycles	Cycles in squash state

Table 3.8: Main rename stage counters

To gain insight into cache performance, we focus on the LSQ. Table 3.9 shows the most important LSQ statistics. We use the average load-to-use latency *loadToUse::mean* to analyze how well the data is cached. This is reported with a standard deviation and a histogram of latencies for all read requests put in the LSQ. (In the table, we only show events for latencies less than 30 cycles.)

Gem5 event name	Description
lsq0.LoadToUse::samples	All load requests put in LSQ
lsq0.LoadToUse::mean	Average load-to-use latency
lsq0.LoadToUse::stdev	Standard deviation of load-to-use latency
lsq0.LoadToUse::0-9	Requests fetched in 0-9 cycles
lsq0.LoadToUse::10-19	Requests fetched in 10-19 cycles
lsq0.LoadToUse::20-29	Requests fetched in 20-29 cycles

Table 3.9: Main LSQ counters

Additionally, we analyze the cache behavior by inspecting the number of cache refills (replacements) for each cache level. Here, *cache.replacements* counts replacements due to both hardware prefetcher and the core requests. To separate both cases, we inspect the cause of MSHR misses. Table 3.10 shows the most important counters for our analysis.

Gem5 event name	Description
dcache.replacements	L1-D cache refills
l2cache.replacements	L2 cache refills
l2cache.overallMshrMisses::prefetcher	L2 cache misses caused by prefetcher
l2cache.overallMshrMisses::cpu0.data	L2 cache misses caused by the core
l3cache.replacements	L3 cace refills

Table 3.10: Main LSQ and cache counters

Finally, we read the traffic over the memory controller. This enables us to analyze the memory footprint and the utilized bandwidth of all cores. Table 3.11 shows a few Gem5 statistics with the name of Gem5 events being self-explanatory.

Gem5 event name	Description
mem_ctrls.numReads::cpu0.data	Read request from core 0
mem_ctrls.numReads::cpu1.l2cache.prefetcher	Read request from core 1 prefetcher
mem_ctrls.bytesRead::total	Bytes read (all cores)
mem_ctrls.bytesWritten::cpu2	Bytes written from core 2
mem_ctrls.bwRead::total	Total read bandwidth
mem_ctrls.bwWritten::total	Total written bandwidth
mem_ctrls.bwTotal::total	Total badwidth (read and write)

Table 3.11: Gem5 memory controller

3.5.3 Typical workflow

Listing 3.7 shows an example of a command line for Gem5 simulations. We modify the SVE size with parameter `system.cpu[:].isa[:].sve_vl_se`. The number of execution units is configured with different CPU types and selected via `--cpu-type`. Other input parameters (see Section 2.5) are predefined in the model. Additionally, we have modified the default *se.py* script for our needs.

```
/home/brank1/gem5_dir/build/ARM/gem5.opt -d results-gromacs/sve_128
    -r  /home/brank1/gem5_dir/configs/example/se.py --cpu-type=O3_ARM_Neoverse_N1
    --num-cpus=4 --caches --l3_size=8MB --mem_latency=50ns
    --mem_bandwidth=25.6GB/s --mem-size=1GB --cpu-clock=2.5GHz
    --sys-clock=2.5GHz --param system.cpu[:].isa[:].sve_vl_se=128
    --cmd="${BIN}" -o "${OPTIONS}" --mem-type SimpleMemory
    --mem-channels=1 --mem-ranks=1 &
```

Listing 3.7: Gem5 command line

For easier workflow, we write a set of bash scripts that run the simulations for various SVE lengths, input sizes, and a number of threads. We also write a script that collects the statistics for multiple SVE sizes and ROI and parses them for a more straightforward analysis. All scripts are available in the Github repository[1]. All Gem5 simulations were performed on the Juawei cluster (Kunpeng 916 nodes) at the Juelich Supercomputing Centre.

For accessing the Graviton 2 processor, we rely on the AWS cloud service. AWS features many types of servers in a virtual environment called instances. These instances are configured with a different number of virtual cores and memory depending on the user's needs. All instances are running on the AWS Nitro System, a virtualization technology developed by AWS. For our work, we rely on the C6g instances, which are powered by the Graviton 2 CPUs. For most instances, a single node is most likely shared with other users for most instances, which is not the ideal setup for benchmarking. Therefore, we use the special *c6g.metal* instance, which enables access to all cores and guarantees that we have complete control of the node. Despite this, the *c6g.metal* is still running in a virtual environment. We do not know the exact effect of this on the performance, but AWS claims that the Nitro hypervisor delivers performance indistinguishable from the bare-metal environment. To avoid the additional work of compilation in the cloud, the binaries are compiled on the Juawei cluster with a static linking. Afterward, we transfer the binaries to the AWS cloud and run them.

We gained access to the A64FX processor through the Ookami testbed at Stony Brook University; the first A64FX implemented machine outside Japan. The Ookami HPE Apollo 80 system features 174 A64FX processors, providing researchers access to this novel architecture for various projects. For A64FX, we compile applications natively on the A64FX machine. This is because we noticed poor performance in some instances for statically linked binaries. Additionally, when we reduce the default vector length (512 bits), we rely on Linux's *prctl* system call.

[1]`https://github.com/binebrank/phd_scripts`

Chapter 4

Porting of applications

4.1 OpenBLAS

Extending OpenBLAS to support a new microarchitecture is not trivial. The main work involves adding specialized assembly kernels for various BLAS functions. Most important are kernels for general and triangular matrix multiplication (GEMM and TRMM), which are also reused to optimize other level-3 BLAS functions. In this section, we mainly focus on DGEMM. There are many similarities between algorithms for different BLAS routines, and a good understanding of DGEMM is a prerequisite for implementing other functions. We give a detailed explanation of the SVE kernel for DGEMM and only briefly present implementation changes for other functions. All code changes explained here were contributed and merged into the OpenBLAS official repository. Work for single and double precision real values (S, D) was accepted into the 3.19 release. Kernels for complex numbers (C, Z) were introduced with release 3.20.

4.1.1 BLAS3 general algorithm

In this section, we explain the implementation of the DGEMM function which computes $C = \alpha A \times B + \beta C$. In the following paragraphs, matrices A, B and C are of size $m \times k$, $k \times n$ and $m \times n$, respectively. α and β are both scalars. For the remainder of this chapter, we will consider the case where $\alpha = \beta = 1$. Computing a scalar-matrix product is not an expensive part of the computation and can be done at different stages in the algorithm. Therefore, we consider the case $C \mathrel{+}= A \times B$. Although we focus on OpenBLAS, most of the details described here also apply to other libraries that implement high-performance BLAS.

Figure 4.1 shows the pseudocode of the DGEMM algorithm, which is composed of six nested *for* loops.

```
Loop 1 for $j_c$ = 0 to N-1 in steps of $n_c$
Loop 2     for $k_c$ = 0 to K-1 in steps of $k_c$
               // pack kc x nc block of B
Loop 3         for $i_c$ = 0 to M-1 in steps of $m_c$
                   // pack mc x kc block of A

Loop 4             for $j_r$ = 0 to $n_c$-1 in steps of $n_r$
Loop 5                 for $i_r$ = 0 to $m_n$-1 in steps of $m_r$
Loop 6                     for k = 0 to $k_c$-1 in steps of 1
                               // update $m_r$ x $n_r$ block of C
```

FIGURE 4.1: DGEMM algorithm pseudocode

Three outer loops (loop 1, 2, 3) partition the matrices A, B and C into smaller blocks (see Figure 4.2). Loop 1 traverses over the dimension n, splitting matrices C and B into submatrices of width n_c. The next loop goes through dimension k. This partitions matrix A into submatrices of width k_c and current submatrix of B into blocks of size $k_c \times n_c$. Loop 3 iterates over submatrices of C and A and partitions them into blocks of size $m_c \times n_c$ and $m_c \times k_c$, respectively. We name the resulting matrix blocks A_c, B_c and C_c. (Shown in the figure as blocks of green, orange, and blue colors.) In other words, the three outer loops disassemble the matrix multiplication of whole matrices into a set of smaller multiplications. Each block C_c is computed as a sum of matrix-matrix multiplications of the corresponding blocks of A and B. When multithreading is enabled, different block computations are assigned to different threads.

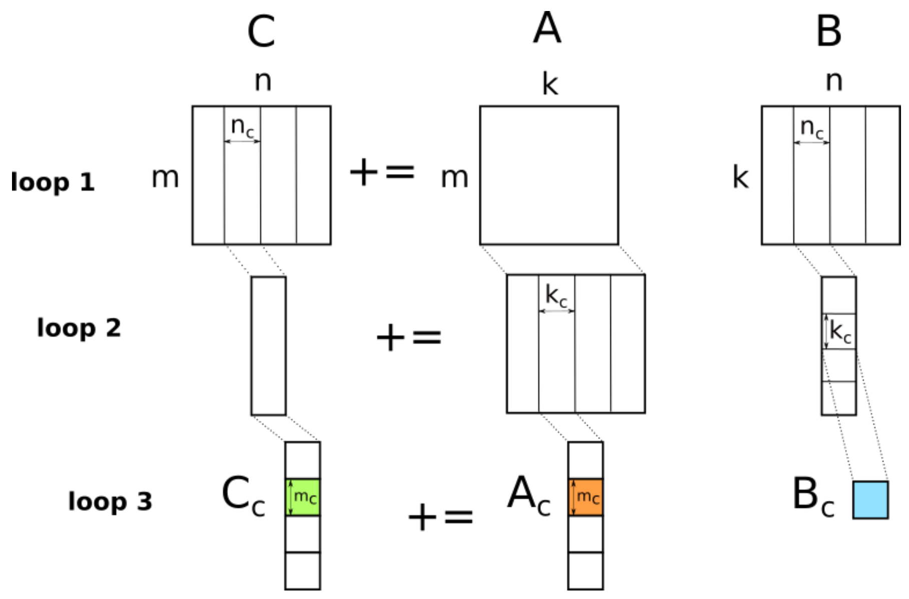


FIGURE 4.2: DGEMM - outer loops

For further discussion, we will focus on a single instance of C_c, A_c and B_c, t.i. when loop counters j_c, k_c and i_c are fixed. We therefore compute

$$C_c \mathrel{+}= A_c \times B_c. \tag{4.1}$$

It is well-known that accessing adjacent elements in memory is faster than accessing elements in separated locations. Therefore, to ensure continuous data access in the inner-most loop (loop 6, which is often called a microkernel), blocks A_c and B_c have a unique data layout. During the execution of outer loops, elements of A and B are reshuffled in the appropriate layout. This is done with *packing* functions. For matrix A_c, the data is organized in row-panels, and within each row-panel, data is stored in column-major order. On the other hand, the matrix B_c holds the data in column-panels, and in each column-panel, data is stored in row-major order. This ordering of elements will become more apparent when explaining the microkernel. Colored matrices in Figure 4.3 show the explained data layout.

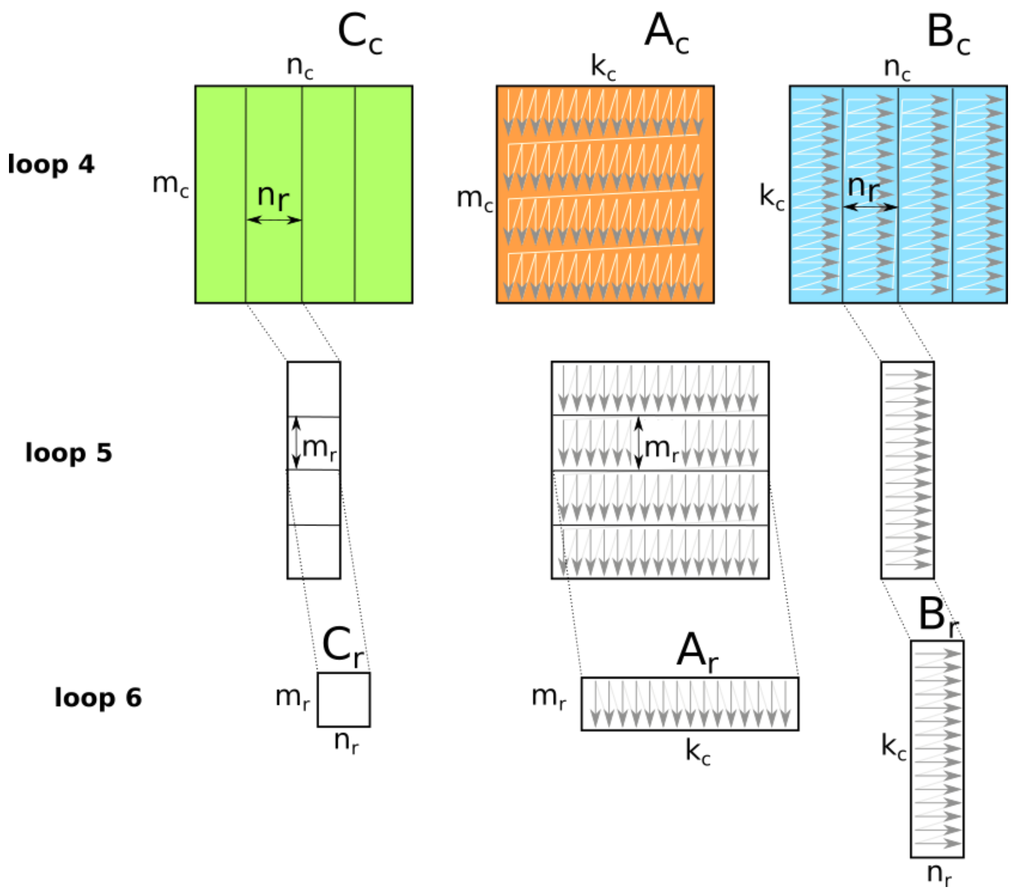


FIGURE 4.3: DGEMM - inner loops

In OpenBLAS, the three inner loops are called a macrokernel and are implemented in assembly. [1] Loop 4 traverses through matrices C_c and B_c, splitting matrices into panels of width n_r (see Figure 4.3). Due to the reordering of elements, each panel B_r has a row-major ordering of elements. Loop 5 traverses in dimension m_c, splitting A_c into panels A_r of height m_r, and panels of C_c into microtiles C_r of size $m_r \times n_r$. Similarly as B_r, the panel A_r is a contiguous block with column-major ordering.

The inner-most loop, loop 6, updates the tile C_r. Each executed loop adds the new product of $A_r \times B_r$ to C_r. The loop runs over k_c, and at each iteration, computes the outer-product of a column of A_r and a row of B_r. The result is accumulated into C_r. For maximum performance, all operations are executed using SIMD instructions. To fully utilize all register lanes, the parameter m_r should be chosen as a multiple of the vector length. In each iteration of the microkernel, values of A_r are loaded into a SIMD register using normal $ld1$ instructions. At the same time, individual elements of B_r are load-replicated into a whole SIMD vector. Then, we compute the matrix multiplication by issuing only multiply-accumulate operations. To maximize data reuse in caches, values of C_r are stored in a set of predefined architectural registers during loop 6.

The parameters of matrix blocks and the size of the microtile are critical and have a significant impact on performance. A study from Meng Low et al. [40] showed that these parameters could be derived analytically from the hardware parameters. Here, we only explain the reasoning behind these but refer to the paper for a complete derivation of parameters. Parameters m_r and n_r determine the size of the microtile. These two parameters depend on the latency of *fmla* operations, the number of SIMD execution units, and the size of the SIMD vectors. Microtile dimensions need to be big enough that no stalls occur in the SIMD pipelines when accumulating the results into registers of C_r. This means that a single *fmla* operation should be completed by the time the same register, holding values of C_r, is again updated. Therefore, the following condition should be satisfied:

$$m_r \times n_r \geq N_{SIMD}\ l_{SIMD}\ lat_{fmla} \tag{4.2}$$

Parameters m_c, k_c and n_c are chosen for the best reuse of data in the cache. Tile C_r, panels A_r and B_r, and matrix blocks are reused different number of times. Different matrix parts should be loaded to different cache levels for best performance and the least number of cache misses. (More reused parts should stay in the lower caches, t.i. closer to the core.) Analysis of the data reuse shows that the most reused part of the matrix is the microtile C_r, followed by a panel B_r, and matrix blocks A_c and B_c. Additionally, the size of parts that are reused more often is smaller as we go from inner loops outwards. This matches the cache hierarchy, where the fastest caches have the smallest size. Therefore,

[1] In the BLIS library, the algorithm is restructured so that only the microkernel is implemented in assembly.

different matrix parts are naturally mapped to different levels of cache. Most reused C_r is stored in the registers, B_r in the L1 cache, A_c in the L2 cache, and B_c in the L3 cache. This is shown on Figure 4.4.

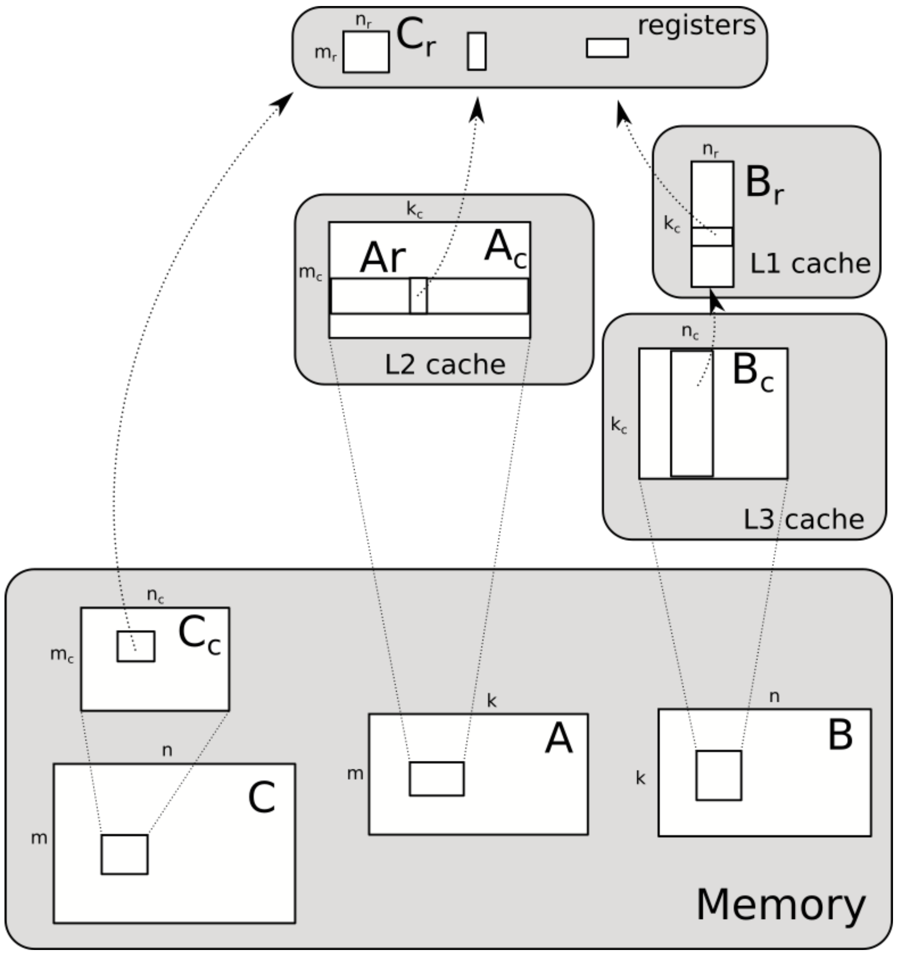


FIGURE 4.4: OpenBLAS GEMM data movement

The parameters k_c, m_c and n_c are chosen so that B_r, A_r, and C_r fit into the caches accordingly. Additionally, one must be careful that loading data into the registers does not evict matrix blocks out of their respective cache. For example, elements of A_r are loaded from the L2 cache into registers during the microkernel. This should not evict elements of B_r from the L1 cache. For a similar reason, one cache line in the L1 cache should be kept free for the update of C_r. Without going into details, we summarize that the parameters k_c, m_c, and n_c depend on the cache line size, the associativity, and the size of L1, L2, and L3 caches, respectively. Again, we refer to [40] for a full analysis.

4.1.2 Preserving the VLA feature

Before our work, OpenBLAS included many macrokernels targeting different architectures. All implemented macrokernels targeted SIMD extensions that had a predefined vector length. Parameters m_r and n_r were therefore hardcoded for each compilation target. Adding support for SVE can be done in two different manners. One is to write an assembly macrokernel for a specific SVE size. Such an approach would be the same as existing target architectures and work only for machines with the same SVE size. Therefore, it would lead to more code and less portable binaries. The second option is to create a vector length agnostic macrokernel. Upon studying the DGEMM algorithm, we discovered that a VLA kernel naturally extends the fixed size. The main idea is to dynamically scale the dimension m_r at runtime to the SVE size while keeping the parameter n_r fixed.[1]. This has one important consequence. Namely, the layout of block A_c should also change depending on the SVE size. Figure 4.5 shows an example of such scalable layout for three different SVE sizes (from smaller l_{SVE} on the left to bigger l_{SVE} on the right).

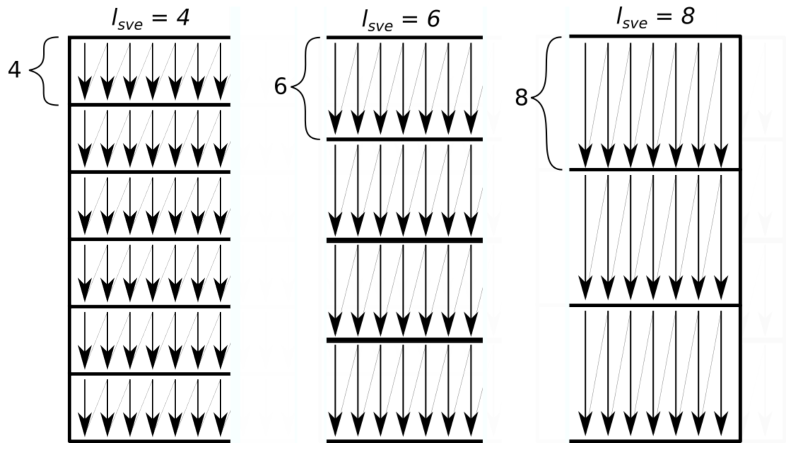


FIGURE 4.5: Scalable layout of A_c

To separate SVE implementation, we added a new compilation target `ARMV8SVE`. This target selects the right SVE macrokernel (see Section 4.1.3) and VLA packing functions (see Section 4.1.4). OpenBLAS defines macros `DGEMM_UNROLL_M` and `DGEMM_UNROLL_N` which are set to m_r and n_r dimensions of the microtile. In the case of SVE, `DGEMM_UNROLL_M` macro has no meaning, and we refactor the code to remove any dependencies on this value. For consistency reasons, this macro is still defined but not used anywhere. On the other hand, `DGMEM_UNROLL_N` was set to eight.

[1]A similar approach was used by Nassyr in the BLIS library, see `https://gitlab.jsc.fz-juelich.de/epi-wp1-public/blis_sve`

The general form of the DGEMM function performs matrix multiplication for matrices of both row-major and column-major order. Additionally, matrices can be transposed, which is specified in the function's input parameters. For each combination of these parameters, packing operations transform the data layout to the A_c and B_c accordingly to use the same macrokernel. Since m_r dimension is scaled to l_{SVE}, and n_r dimension is fixed, packing routines are separated for m_c and n_c dimensions. Whenever code unrolls over dimension m_c, we intercept this by calling the corresponding SVE copy functions with scalable m_r. (See Section 4.1.4) For packing in dimension n_c, we rely on the OpenBLAS generic packing functions. Additionally, this also requires modification of the Automake and CMake build systems.

4.1.3 SVE assembly kernel

We write the SVE macrokernel for DGEMM in assembly. We implement two kernels, one for $m_r = l_{SVE}$ and one for $m_r = 2\ l_{SVE}$. In both cases we set $n_r = 8$. As explained in the previous section, the macrokernel computes the inner three loops of the algorithm (see Figure 4.3). The function begins by pushing the callee-saved registers on the stack. Afterward, we construct loops 4, 5, and 6 by reserving individual general-purpose registers for loop counters and block/panel sizes. Since loop 4 iterates over A_c with steps n_r, special care is taken for remainders in cases where n_c is not a multiple of n_r. (Same for loop 5 when m_c is not a multiple of m_r.) For performance purposes (to decrease the number of compare and branch instructions), we unroll loop 6 over eight iterations. In the implementation, we rely heavily on macro definitions to shorten the long assembly code. Finally, the callee-saved registers are popped from the stack to preserve original values.

A snippet of the macrokernel code for size $2l_{SVE} \times 8$ is shown in Listing 4.1. This example shows a part of macro definition `KERNELv2x8_M1` which computes a single iteration of loop 6. We see that elements of A_r are loaded with a *ld1d* instruction (lines 2 and 3) which loads the data from contiguous locations in memory. Afterwards, the pointers to A_r are updated (lines 4 and 5) to the next column of A_r. The rest of the code shows the *fmla* operations that multiply columns of A_r and elements of B_r and *ld1rd* load instructions that load-replicate elements of B_r. The important aspect of the kernel is how to schedule instructions so that at least N_{SVE} *fmla* instructions are issued every cycle. The load operations take lat_{ld1} cycles to load the data to the SIMD register, if data is in L1 cache. Therefore, between issuing the load instruction and the *fmla* instruction that uses the same register, we must issue at least $lat_{ld} \times N_{SVE}$ other *fmla* instructions. For this reason, we always preload data to registers in advance. In our example, we use registers `z2` and `z3` to hold elements of A_r. However, these registers are not used until the next iteration of loop 6 (after issueing 16 *fmla* instructions). This gives at least eight cycles, until the registers `z2` and `z3` are used for the computation. Similarly, *fmla* operations (after line 7) act on data stored in `z0` and `z1` which were loaded with A_r elements in the previous iteration. The same also holds for registers storing elements of B_r. Consider the register `z8` (used in *fmla* operations in lines 7 and 8). We issue the *ld1rd* instruction immediately after issuing the *fmla* operations, so that `z8` can be used in the next iteration.

```
1   .macro KERNELv2x8_M1
2       ld1d  z2.d, p0/z, [pA1]          // load values for next column of A_r
3       ld1d  z3.d, p0/z, [pA2]          // load values for next column of A_r
4       add pA1, pA1, vec_len, lsl #3    // pA1 = pA1 + vec_len * 8
5       add pA2, pA2, vec_len, lsl #3    // pA1 = pA1 + vec_len * 8
6
7       fmla z16.d, p0/m, z0.d, z8.d     // multiply_accumulate
8       fmla z17.d, p0/m, z1.d, z8.d     // multiply_accumulate
9       ld1rd  z8.d, p0/z,  [pB]         // load replicate next value of B_r
10      fmla z18.d, p0/m, z0.d, z9.d     // multiply_accumulate
11      fmla z19.d, p0/m, z1.d, z9.d     // multiply_accumulate
12      ld1rd  z9.d, p0/z,  [pB, 8]      // load replicate next value of B_r
13      fmla z20.d, p0/m, z0.d, z10.d    // multiply_accumulate
14      fmla z21.d, p0/m, z1.d, z10.d    // multiply_accumulate
15      ld1rd  z10.d, p0/z, [pB, 16]     // load replicate next value of B_r
16      fmla z22.d, p0/m, z0.d, z11.d    // mutliply_accumulate
17      fmla z23.d, p0/m, z1.d, z11.d    // mutliply_accumulate
18      ...
```

LISTING 4.1: DGEMM macrokernel

Another major difference between the SVE kernel and traditional fixed-size SIMD kernels is how loop 5 is implemented. Loop 5 traverses A_c in steps of m_r. We consider the case when m_c is not a multiple of m_r. For traditional architectures, OpenBLAS implementation mandates that m_r must always be a power of two. The remainder in loop 5 is then computed separately for smaller powers of two. Figure 4.6 left illustrates this example for the case when $m_r = 8$ and $m_c = 23$. After the loop over m_r finishes, additional iterations are done in steps 4, 2, and 1 (smaller powers of two). Such an approach is necessary for some architectures with no masking instructions.

In the case of SVE, dealing with a remainder can be done more efficiently with a single sweep across k_c. For this, we create a predicate register with only the first $m_c \mod l_{SVE}$ active bits. Then we use the same approach as for previous row-panels, but only compute the results for the correct number of active lanes. Figure 4.6 right shows what predicate registers are used in our example. Conceptually, this last iteration is not different from previous iterations.

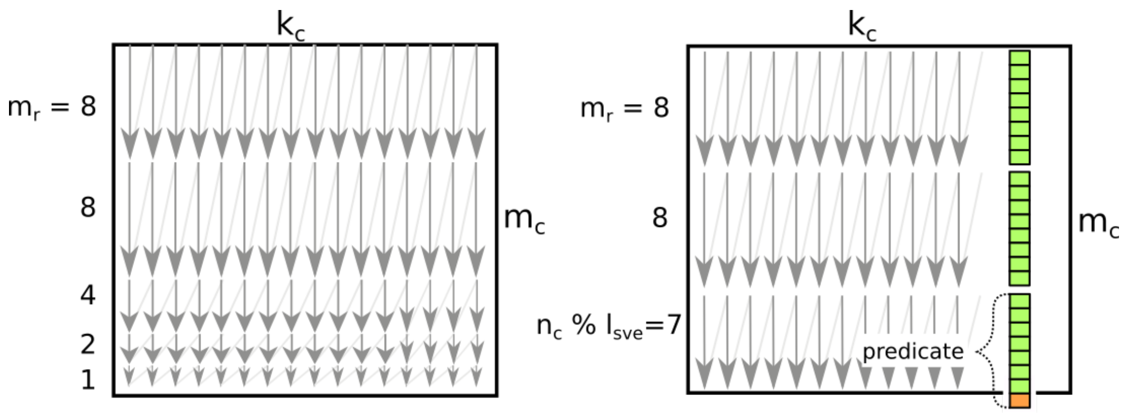


FIGURE 4.6: SVE data layout of A_c

An important consequence of this idea is that the data layout for the SVE target changes. For the m_c dimension, the last few rows of A_c are grouped, whereas, in fixed-size SIMD, they are separated in powers of two. In other words, the data is organized differently even if the SVE size is the same as another SIMD extension (for example, 256-bit SVE and AVX2). This requires specialized packing functions for SVE.

4.1.4 SVE intrinsic packing functions

To reshuffle data correctly into blocks A_c, we implemented vector length agnostic packing functions for SVE architecture. For DGEMM, two different packing functions are necessary for various combinations of input parameters. First, *dgemm_tcopy* is used for cases when A is stored in column-major order and not transposed (or row-major order and transposed). In this case, elements are in the correct order, and we only need to gather different columns in row-panels of height l_{SVE}. *dgemm_tcopy* relies on normal *ld1* instructions. On the other hand, *dgemm_ncopy* is used when ordering is row-major and matrices are not transposed (or column-major and transposed). In this case, the submatrix has to be transposed, which requires the use of gather-load instructions.

Listing 4.2 shows the *dgemm_tcopy* kernel[1]. The kernel consists of two loops. The first loop (line 5) traverses over panels A_r in steps of $m_r = l_{SVE}$. This is done in an SVE fashion, each time incrementing the counter by the number of SVE lanes and updating the predicate register. For each panel, the inner loop traverses in dimension k_c and copies the data from matrix A to the new columns of A_r. We see that each iteration of the inner loop increments the pointer `aoffset1` by the size of the leading dimension of A to ensure that the correct elements of the next row are loaded.

[1]Some variable names have been changed for easier explanation.

```
j = 0;
svbool_t pg = svwhilelt_b64(j, m_c);
active = svcntp_b64(svptrue_b64(), pg);
do {

  aoffset1 = aoffset;

  uint64_t i_cnt = k_c;
  while (i_cnt--) {
    svfloat64_t a_vec = svld1(pg, aoffset1);
    svst1_f64(pg, (double *) boffset, a_vec);
    aoffset1 += lda;
    boffset += active;
  }
  aoffset += sve_size;

  j += svcntd();
  pg = svwhilelt_b64(j, m_c);
  active = svcntp_b64(svptrue_b64(), pg);

} while (svptest_any(svptrue_b64(), pg));
```

LISTING 4.2: Function *dgemm_tcopy*

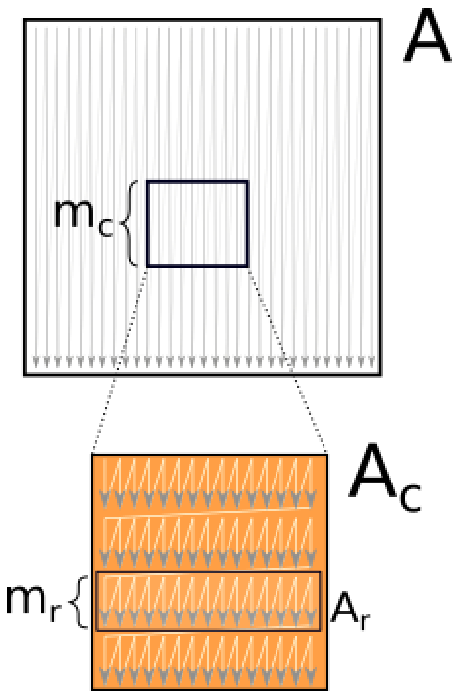


FIGURE 4.7: Function *dgemm_tcopy*

4.1.5 Triangular matrices

Expanding SVE to other forms of GEMM is effortful but does not present any significant challenges. For SGEMM, this involves transforming all instructions in macrokernels and packing functions from doubleword form to word. Macrokernels for complex numbers (CGEMM and ZGEMM) are similar to DGEMM. The main idea is to create a macro for complex multiply-add operation which is composed of three *fmla* and one *fmls* instruction. Afterward, this macro definition is used instead of the typical *fmla* instruction. Of course, data layout has to be modified to account for real and imaginary components of complex numbers. Another part of our work, which is more challenging, is the VLA implementation of the BLAS3 functions for triangular matrices. OpenBLAS includes specialized macrokernels for a TRiangular Matrix-Matrix (TRMM) multiplication.[1] These macrokernels are also used in BLAS3 functions that operate on symmetric matrices (SYMM, SYRK, SYR2K).

TRMM algorithm is similar to a GEMM algorithm explained in previous sections. The main difference is that loop 6 stops at the diagonal. This enables us to quickly transform the GEMM macrokernel to TRMM macrokernel by only changing the loop exit

[1]TRMM computes $B = \alpha op(A) \times B$ for a $m \times n$ matrix B, and a unit or non-unit, lower or upper triangular matrix A. $op(A)$ potentially transposes the matrix A.

conditions and correctly updating pointers to the next row-panels. However, the diagonal matrices for BLAS3 routines are stored in packed format (t.i. without any zeros). To use the TRMM macrokernel, special *trmm_copy* triangular packing functions store the additional zeros above (below) the diagonal. This is shown in Figure 4.8. Due to the modified VLA layout (see Figure 4.6), the generic *trmm_copy* kernels for fixed-size SIMD architectures do not work in our case. Here, we also implemented VLA kernels for SVE. Because of the complexity of SVE implementation (each row would require a different predicate register), we implemented these with scalar instructions. Due to many input parameters of TRMM functions (lower/upper, unit/non-unit diagonal, row/column-major), separate packing functions must be implemented for different combinations of these parameters.

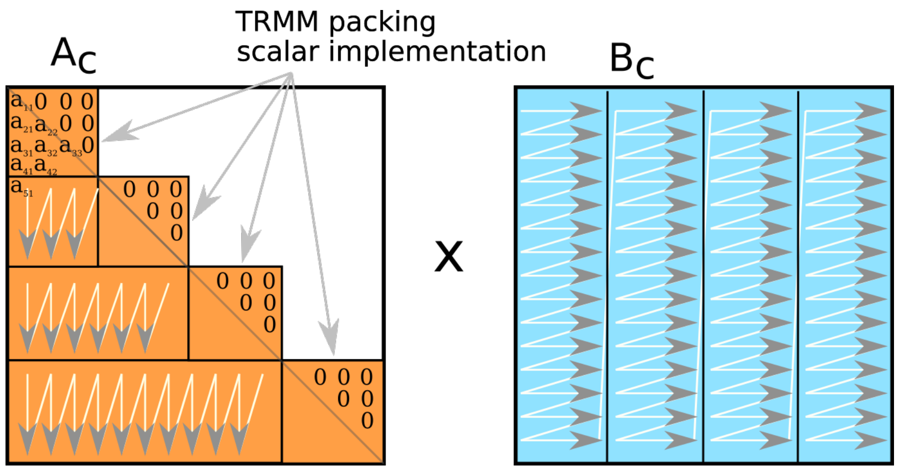


FIGURE 4.8: OpenBLAS TRMM algorithm

Other work for the SVE port included macrokernels and packing functions for TRSM and SYMM. We omit the details of these functions and refer to the source code.

4.2 GROMACS

This section gives an overview of the SIMD/SVE usage in GROMACS. The SVE port was done by the Research Organization for Information Science and Technology (RIST) (Tokyo University of Science). We modified a particular aspect of this implementation which is explained in Section 4.2.3. Changes were contributed to the GROMACS official repository.[1]

[1]`https://gitlab.com/gromacs/gromacs/-/merge_requests/1991`

4.2.1 Compute patterns

The most compute-intensive part of a general MD simulation algorithm is the computation of forces on each particle. Because the simulated systems can include up to millions of particles and a nanosecond time scale, force computation is repeated extensively throughout the simulation. GROMACS splits the total force on a single particle into three main contributions:

- Forces from bonded interactions. Here, we focus on the covalent bond, which groups electrons into shared electron pairs between atoms. Typically, this contribution depends on a few neighboring atoms.
- Forces from non-bonded interactions. This force is caused by other particles in the system and depends on the distance between particle pairs. A Lennard-Jones potential is often a simple yet realistic model for such interactions.
- External and restraining forces. Researchers often use these forces to impose certain constraints on the system. The most common reason for this is to avoid unwanted deviations or lock specific parameters to the experimental data.

Once the forces have been computed, the locations and the velocities of particles are updated by numerically solving Newton's equations of motion. Additionally, we can use the computed data to calculate different statistical properties of the system (for example, the total kinetic energy of particles or the pressure tensor).

The computation of forces between non-bonded atoms usually dominates the simulation time. Each atom exerts a force on other atoms, which results in an $O(n^2)$ n-body problem. Non-bonded forces are further divided into short-range and long-range forces. GROMACS employs a modified Verlet algorithm to compute short-range forces between particles. The key feature of this algorithm is that the pairs of particle-particle interactions are replaced by pairs of tiny clusters of nearby particles. The size of each cluster is determined at compile time to better match the size of the SIMD vectors (or GPU threads) [41]. The interactions are then explicitly computed with SIMD instructions, which removes the need for compiler auto-vectorization.

The main idea is shown in Figure 4.9. On the left, we see a classical Verlet algorithm. A cut-off radius (orange line) and a buffer (dashed orange) determine the list of j-particles used for computing the forces on the central orange i-particle. In practice, this leads to two nested *for* loops, where the outer loop iterates over all i-particles and the inner loop over all neighbors (j-particles). Unfortunately, such an approach leads to poor data reuse because values of all j-particles are loaded for each iteration of the outer loop. In a modified algorithm (right side), the particles are grouped in small clusters. A cluster with at least one particle within the buffered radius is included in the computation. The outer loop traverses over all clusters, while the inner loop computes the forces for all particles in the i-cluster (orange color) simultaneously. This maps well to SIMD instructions and reduces the amount of data transferred from memory. For example, if the cluster size is four, we can compute sixteen interactions by only loading values of four particles. A cluster size of one corresponds to the original Verlet algorithm.

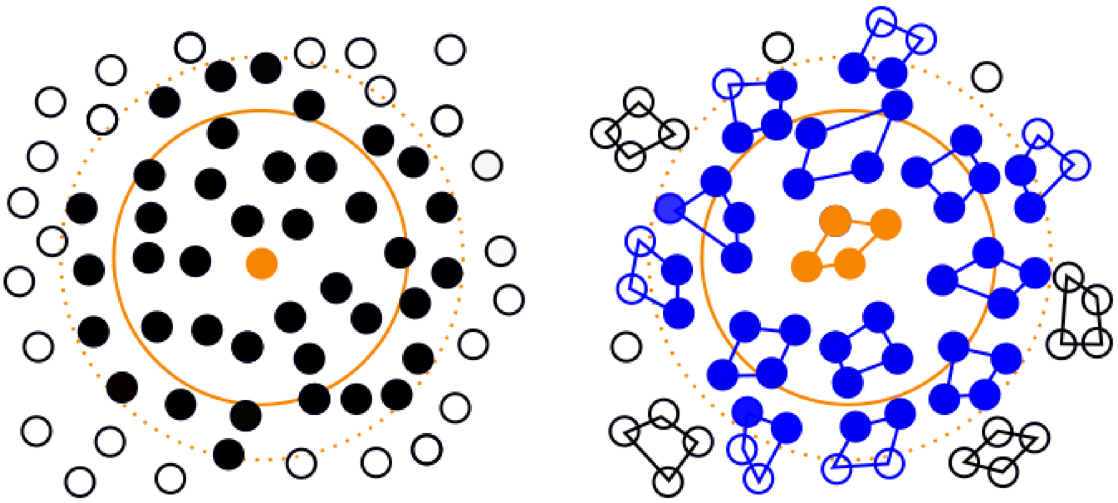


FIGURE 4.9: Classical and modified Verlet algorithm

Long-range forces are computed with a particle mesh Ewald algorithm [42]. The method is based on the interpolation of reciprocal space Ewald sums. The total complexity of the algorithm is $O(Nlog(N))$ and primarily relies on fast Fourier transforms. GROMACS, by default, uses a mixed-precision mode. This means that most computations are performed in single-precision, with only the most critical parts in double-precision. However, for simulations that require complete double-precision computation, this is also possible.

To better understand the application profile, we inspect GROMACS using HPCToolkit. We execute the benchmark on a single node (64 cores) of the Kunpeng 916 CPU. The code was compiled with GCC for the Arm NEON SIMD target. We evaluate three typical use cases:

- **rnase_cubic** use case simulates ribonuclease (also abbreviated as RNAse) in a water solvent in a cubic box domain. RNAse is an important enzyme in organisms used for the degradation of RNA. The whole system is composed of 24000 particles.
- **ion_channel** system simulates a membrane protein GluCL, a chloride channel in a lipid bilayer. This system is an important use case for the pharmaceutical industry and consists of roughly 150000 atoms.

- **benchRIB** is a simulation of an E. coli ribosome in water. E. coli is a bacteria commonly found in warm-blooded organisms, including humans. This use case is the largest, with around two million atoms.

In Figure 4.10, we show a simple profile of GROMACS runs for three cases. We observe that the N-body computation dominates the simulation time. Between 65 and 72% of cycles are spent in the modified Verlet algorithm. Around 10% of cycles are spent in the FFT kernel and between 15 to 25% in other parts of the code.

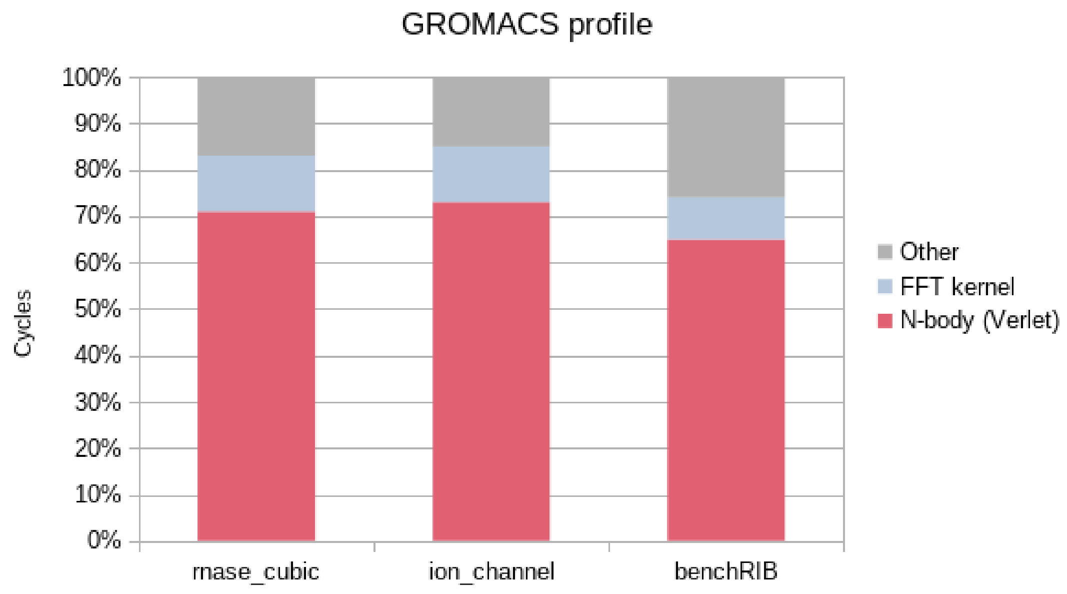


FIGURE 4.10: GROMACS profile

4.2.2 SIMD backend

The modified Verlet algorithm is closely connected to the SIMD architecture. To utilize all lanes of SIMD registers during computation, the SIMD width must be a multiple of the cluster size. In this way, the calculation of forces between particles fully maps to SIMD instructions without any data shuffling in registers. Additionally, vectorization becomes an intrinsic part of the algorithm. GROMACS always groups particles in clusters of four. Additionally, it implements two different kernels of the Verlet algorithm:

- The **4xM** kernel computes interactions between four i-particles and M j-particles where $M = l_{SIMD}$. The data of each i-particle is replicated over an entire SIMD register, while the data from j-particles is loaded in separate lanes. The computation between elements, when $l_{SIMD} = 8$, is shown in Figure 4.11. I-particles are represented in orange color and j-particles in blue color. Black dots show units of computation in the SIMD register.

- The **2xMM** kernel calculates interactions between four i-particles and $M = l_{SIMD}/2$ j-particles. The data from two i-paricles and M j-particles is duplicated in the SIMD registers. This requires some additional instructions to combine values into both register halves. Note that the 2xMM kernel requires at least eight lanes and is not possible for $l_{SIMD} = 4$ ($b_{SIMD} = 128$).

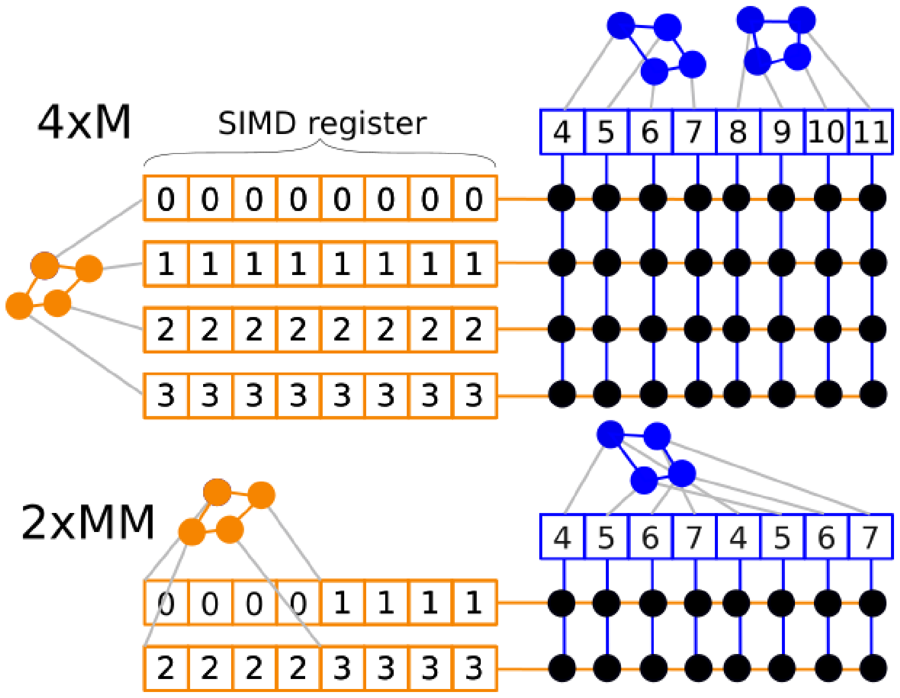


FIGURE 4.11: GROMACS SIMD computation

To map operations for force computation to SIMD instructions, GROMACS implements a specialized SIMD library that leverages intrinsic functions. The library has a modular design with an interface that acts as a wrapper for intrinsic functions. Each supported architecture has a dedicated backend that maps high-level functions to the relevant instructions on the underlying hardware. The backend is completely separated from the non-bonded kernel, which is implemented only with wrapper functions. This enables developers to add new non-bonded kernels without worrying about how the operations translate to a particular target architecture. Similarly, adding new SIMD targets can be done by only implementing the backend module using the intrinsic functions, and the non-bonded kernels can use them immediately.

The target architecture is selected automatically for native builds or via build parameters for cross-compilation. Currently, all major SIMD instruction sets are supported. This includes SSE, AVX2, and AVX512 on x86_64 and NEON and SVE (since version 2021) on Arm.

```
1  #if GMX_DOUBLE
2  typedef SimdDouble SimdReal;
3  #else
4  typedef SimdFloat  SimdReal;
5  #endif
6
7  namespace gmx
8  {
9
10 // Backend class for single precision
11 class SimdFloat
12 {
13 public:
14   SimdFloat() {}
15 ...
16   // constructors
17
18   // Intrinsic NEON SIMD register
19   float32x4_t simdInternal_;
20 };
21 ...
22
23
24 // overloading operator *
25 static inline
26 SimdFloat gmx_simdcall operator*
27   (SimdFloat a, SimdFloat b) {
28   return { vmulq_f32(a.simdInternal_,
29     b.simdInternal_) };
30 }
31
32 ...
```

Listing 4.3: SIMD backend (NEON) in GROMACS

```
SimdReal sir_S0, sir2_S0, sir6_S0;
SimdReal sir_S1, sir2_S1, sir6_S1;
SimdReal sir_S2, sir2_S2, sir6_S2;
SimdReal sir_S3, sir2_S3, sir6_S3;

SimdReal FrLJ6_S0, FrLJ12_S0, frLJ_S0;
SimdReal FrLJ6_S1, FrLJ12_S1, frLJ_S1;
SimdReal FrLJ6_S2, FrLJ12_S2, frLJ_S2;
SimdReal FrLJ6_S3, FrLJ12_S3, frLJ_S3;

...

sir_S0 = sig_S0 * rinv_S0;
sir_S1 = sig_S1 * rinv_S1;
sir_S2 = sig_S2 * rinv_S2;
sir_S3 = sig_S3 * rinv_S3;
sir2_S0 = sir_S0 * sir_S0;
sir2_S1 = sir_S1 * sir_S1;
sir2_S2 = sir_S2 * sir_S2;
sir2_S3 = sir_S3 * sir_S3;
sir6_S0 = sir2_S0 * sir2_S0 * sir2_S0;
sir6_S1 = sir2_S1 * sir2_S1 * sir2_S1;
sir6_S2 = sir2_S2 * sir2_S2 * sir2_S2;
sir6_S3 = sir2_S3 * sir2_S3 * sir2_S3;
FrLJ6_S0 = eps_S0 * sir6_S0;
FrLJ6_S1 = eps_S1 * sir6_S1;
FrLJ6_S2 = eps_S2 * sir6_S2;
FrLJ6_S3 = eps_S3 * sir6_S3;
FrLJ12_S0 = FrLJ6_S0 * sir6_S0;
FrLJ12_S1 = FrLJ6_S1 * sir6_S1;
FrLJ12_S2 = FrLJ6_S2 * sir6_S2;
FrLJ12_S3 = FrLJ6_S3 * sir6_S3;
```

Listing 4.4: 4xM kernel computing Lennard-Jones potential

Listing 4.3[1] shows the most important parts of the code describing the SIMD library. Lines 1-5 define a new type name based on whether the simulation is performed in single or double precision (`GMX_DOUBLE`). In line 11 a class *SimdFloat* is defined for a NEON SIMD extension set. This class has a member *float32x4_t* corresponding to the 128-bit NEON register. In line 26, the `*` operator is overloaded by mapping the multiplication to the SIMD *fmul* operation via *vmulq_f32* intrinsic function. Same is done for all other operations including exponential functions. In Listing 4.4 we see how the SIMD library is used in the 4xM nonbonded-kernel. This code snippet shows a part of the kernel which computes the Lennard-Jones (LJ) force. In the computation, the code uses instances of the `SimdReal` (`SimdFloat`) classes to ensure that SIMD operations are used. LJ force

[1]We exclude certain lines of code for simplification.

is a computationally dominant factor due to coefficients with high powers (r^6 and r^{12}). Lines 13-32 show how the force is computed simultaneously for four i-particles.

4.2.3 SVE specific code

Because SVE is vector length agnostic, the SVE implementation cannot be added similarly to other SIMD ISAs. As discussed in Section 2.1.3.2, SVE SIMD types cannot be used as members of a class due to their sizeless nature. A new CMake option, `DGMX_SIMD_ARM_SVE_LENGTH`, was added to fix the SVE size at compile time. Our code analysis has confirmed that the SVE implementation is of good quality and stable. All instructions needed in the algorithm correctly map to SVE operations. However, we did notice one suboptimal implementation detail when translating kernel functions into intrinsic functions.

Initially, support for SVE was added via the GCC SIMD intrinsic types. When compiled, these are treated as SVE types from Arm C Language Extensions (ACLE) and can be used interchangeably with SVE intrinsic functions. The reason for this is a missed chance from the developers to use SVE fixed-size types directly with the attribute `arm_sve_vector_bits(..)` as defined in ACLE for SVE chapter 3.7.3. Using ACLE types, the code is not limited to only GCC.

```
1  class SimdDouble
2  {
3  private:
4      typedef svfloat64_t simdInternalType_
5              __attribute__((arm_sve_vector_bits(GMX_SIMD_ARM_SVE_LENGTH_VALUE)));
6
7  public:
8      SimdDouble() {}
9      SimdDouble(const double d) { this->simdInternal_ = svdup_n_f64(d); }
10     SimdDouble(svfloat64_t simd) : simdInternal_(simd) {}
11     simdInternalType_ simdInternal_;
12 };
13
14 ...
15 static inline SimdDouble gmx_simdcall operator+(SimdDouble a, SimdDouble b)
16 {
17     svbool_t pg = svptrue_b64();
18     return { svadd_f64_x(pg, a.simdInternal_, b.simdInternal_) };
19 }
20 ...
```

Listing 4.5: GROMACS SVE backend

Listing 4.5 shows an example of a fixed SVE implementation. In lines 4-5, *svfloat64_t* is used as an intrinsic type for the SVE register. However, such type is fixed for a value specified at compile-time (`DGMX_SIMD_ARM_SVE_LENGTH`). In line 15, the + operator is overloaded using *svadd_f64_x* instruction which maps to SVE *add* operation.

4.3 GPAW

GPAW is written primarily in Python (using NumPy and SciPy), with the most performance-critical kernels written in C. In addition, the application relies on several external libraries to speed up the computation, namely a BLAS library, FFTW, and LibXC. It is also highly recommended to use MPI, BLACS, and ScaLAPACK for parallel simulations with many nodes.

In code, there is no notion of explicit SIMD parallelization. GPAW, instead, relies on compiler auto-vectorization. We have noticed two cases where vectorization is encouraged through `#pragma omp simd`. This resulted in a slight performance increase on the Knights Landing architecture [43].

4.3.1 Application hot-spots

For our SIMD analysis, we focus on computationally expensive parts of GPAW, where the application spends the most time. Therefore, we first analyze the application using HPCToolkit. We rely on use cases from the Unified European Applications Benchmark Suite (UEABS) to collect a profile. UEABS aims to provide a benchmark suite of currently relevant HPC applications. We use the smallest listed use case - the Carbon Nanotube (CN) benchmark, which simulates 240 carbon atoms in a 6x6x10 nanotube structure. It calculates the energy of a system's ground state in a vacuum. The default mode of the computation in the CN benchmark is based on a finite differences (FD) algorithm. However, we modified the input configurations to also evaluate the benchmark in the other two modes - PW and LCAO. We name different workloads as CN-FD, CN-PW, and CN-LCAO. The runtime of the selected kernels on a full node of Kunpeng916 (64 cores) is shown in Table 4.1. FD and PW modes calculate the solution in 251 and 291 seconds. LCAO mode is significantly faster at 16 seconds.

CN-FD	CN-PW	CN-LCAO
251	291	16

TABLE 4.1: GPAW runtime (seconds) on Kunpeng 916

Figure 4.12 shows the percentage of cycles spent in external libraries. The profiles of the three use cases are quite different. We observe that most of the time (around 40%)

for FD and PW modes is spent in OpenBLAS. On the other hand, roughly 60% of the LCAO mode is executed in the GPAW internal code. Additionally, LCAO mode relies more on other python code (numpy, scipy), which we show as *Other*. FFTW3 calls are observed only in the plane-waves mode.

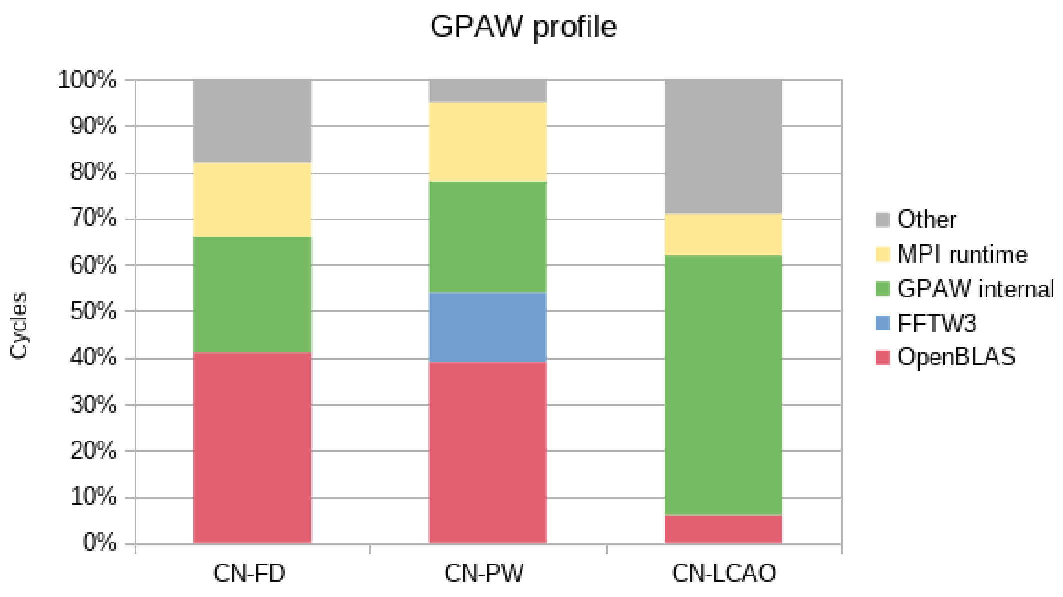


FIGURE 4.12: GPAW profile

4.3.2 Selected GPAW kernels

Because the computation takes several minutes on a full node of Kunpeng 916, the CN benchmark is too big to be evaluated in the Gem5 simulator. We considered reducing the number of atoms, but the runtimes were still not small enough for a feasible simulation. Furthermore, we only want to focus on the GPAW internal code excluding external libraries. (We have already studied OpenBLAS, and MPI & FFTW exceed the scope of application analysis.) To solve this, we inspect the profiles generated by HPCToolkit and select relevant GPAW functions where a sufficient amount of runtime is spent during the computation. Furthermore, we try to choose interesting functions in terms of vectorization. Upon our analysis, we have selected two functions from the GPAW code that we will analyze in this thesis.

- **Bmgs_fd** function (gpaw_root/c/fd.c) consumes around 14% of total cycles in the CN-FD benchmark. The function computes various stencil operations on a grid and is used mostly in finite-difference calculations. In our case, we focus on the Laplace operator.

- **Construct_density** function (gpaw_root/c/lfc.c) is responsible for around 17% of total cycles for the CN-LCAO use case. The function calculates the expected value of the electron density from the density matrix.

4.3.2.1 Laplace operator (bmgs_fd)

In the finite-difference mode of GPAW, the electronic structure calculations are performed on a uniform real-space rectangular grid. Physical quantities, such as wave-functions, electron densities, or potentials, are represented on grid points with linear spacing. Physical coordinates of grid points are $r(i,j,k) = (ih_i, jh_j, kh_k)$ where $h_{i,j,k}$ are distances between the points and $i,j,k = 1, 2....N_{x,y,z}$. Throughout the simulation, GPAW solves the Kohn-Sham and Poisson equations which both involve the Laplace operator:

$$\nabla^2 = (\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}) \tag{4.3}$$

Different discretizations can be used for applying the Laplace operator depending on the unit cell and the order of finite-difference coefficients. GPAW relies on the compact Mehrstellen-type stencil operators of the fourth order [44]. In particular, it uses two stencil operators to solve the Kohn-Sham equation. The first stencil (see Figure 4.13 left) includes 19 points. This corresponds to a central point with six nearest neighbors and twelve next nearest neighbors. The second operator includes only weights of the central and nearest-neighboring points. For our work, we focus on the stencil with 19 points which is computationally more expensive.

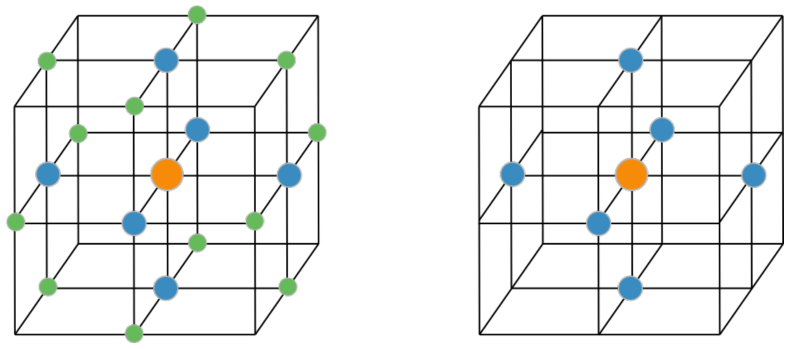
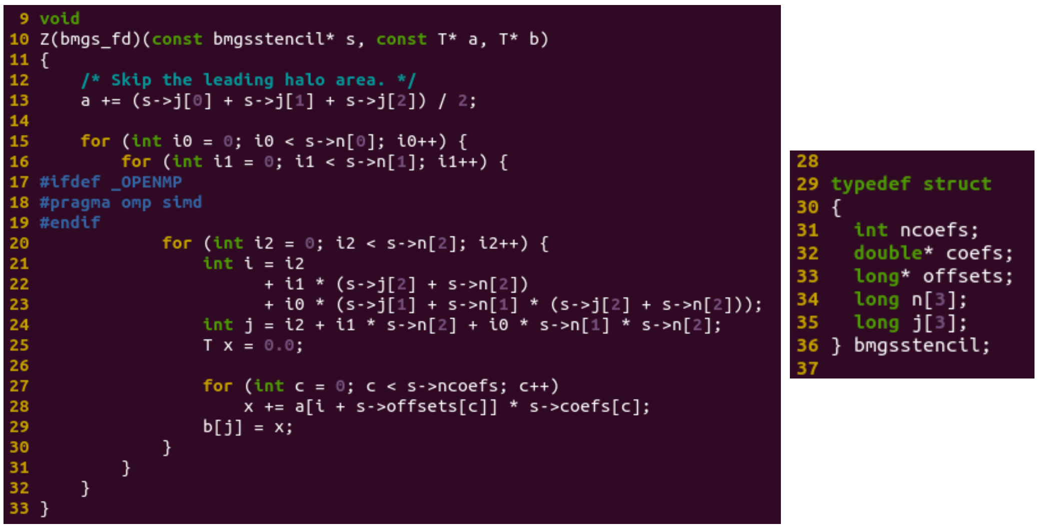


FIGURE 4.13: GPAW 19 and 7-point stencil

Figure 4.14, left, shows the `bmgs_fd`[1] function. This function applies a stencil operator to the points of a three-dimensional grid. A stencil operator is defined as a `bmgsstencil` (Figure 4.14, right) struct. The main field of this struct is the number of stencil points `ncoefs` in line 31. For each stencil point, the weights of the points and the offsets in

[1]A preprocessing macro expands `Z(bmgs_fd)` to `bmgs_fd`

the global array are stored in `coefs` and `offsets`. Lastly, `n[3]` are the dimensions of the grid, and `j[3]` are the strides of the global array in each dimension. In other words, a `bmgsstencil` contains all information about the operator. GPAW includes many stencils with a predefined size and coefficients. The *bmgs_fd* function consists of four nested for-loops. The three outer loops (line 15, 16, 20) traverse over all grid points in three dimensions. For each grid point, the code first computes the indices i and j for the current grid point in the input array `a` and the output array `b`. The innermost loop (line 28) iterates over all stencil points. For our 19-point stencil, this loop has 19 iterations. The weights of stencil points are multiplied with their respective values and accumulated in `x`. Finally, the result is stored in the output array `b[j]`.



```
 9 void
10 Z(bmgs_fd)(const bmgsstencil* s, const T* a, T* b)
11 {
12     /* Skip the leading halo area. */
13     a += (s->j[0] + s->j[1] + s->j[2]) / 2;
14
15     for (int i0 = 0; i0 < s->n[0]; i0++) {
16         for (int i1 = 0; i1 < s->n[1]; i1++) {
17 #ifdef _OPENMP
18 #pragma omp simd
19 #endif
20             for (int i2 = 0; i2 < s->n[2]; i2++) {
21                 int i = i2
22                       + i1 * (s->j[2] + s->n[2])
23                       + i0 * (s->j[1] + s->n[1] * (s->j[2] + s->n[2]));
24                 int j = i2 + i1 * s->n[2] + i0 * s->n[1] * s->n[2];
25                 T x = 0.0;
26
27                 for (int c = 0; c < s->ncoefs; c++)
28                     x += a[i + s->offsets[c]] * s->coefs[c];
29                 b[j] = x;
30             }
31         }
32     }
33 }
```

```
28
29 typedef struct
30 {
31   int ncoefs;
32   double* coefs;
33   long* offsets;
34   long n[3];
35   long j[3];
36 } bmgsstencil;
37
```

FIGURE 4.14: GPAW bmgs_fd function

Next, we estimate the arithmetic intensity of the Bmgs_fd kernel. First, we notice that the computation of indices `i` and `j` is not expensive because the majority of calculations (lines 22-24) can be computed outside of the loop. (Compilers successfully apply such optimization.) Therefore, we focus on the inner-most loop and neglect the calculation of `i` and `j`, and the memory write in line 29. In the inner-most loop, we have three 64-bit memory reads (`a[i + s->offsets[c]]`, `s->offsets[c]`, and `s->coefs[c]`) and three arithmetic operations. There are different methods for counting the number of operations, depending on which operations we count (integer or just floating-point). In our analysis, we count both, but we restrict ourselves to only SVE instructions. All three operations in line 28 are executed in SVE pipelines. (See assembly in Listing 4.6, lines 4, 5, and 8.) This gives a lower bound:

$$AI \geq \frac{3}{3 \times 8} = \frac{1}{8} \frac{op}{Byte} \tag{4.4}$$

This estimate does not consider any caching effects. We can create a tighter lower bound by acknowledging that `s->offsets[c]` and `s->coefs[c]` stay in the L1 cache during the entire kernel. Note that the entire `bmgsstencil` struct (Figure 4.14, right) has a size of 344 bytes when `coefs` and `offsets` are arrays of size 19. This gives:

$$AI \geq \frac{3}{8} \frac{op}{Byte} \tag{4.5}$$

The Bmgs_fd kernel executes 19 memory reads (neglecting reads of struct `s`) per 1 memory write. However, we would like to point out that the elements `a[i + s->offsets[c]]` represent data on the neighboring points of the current point of computation. Since the Laplace operator (inner-most loop) traverses grid points in-order, many elements of array `a` are cached and reused during iterations of outer loops. We analyze this from the Gem5 measurements in Chapter 5.

```
 1  .L6:
 2    ld1d    z0.d, p0/z, [x2, x0, lsl 3]    // load s->offsets[c] into z0
 3    ld1d    z1.d, p0/z, [x3, x0, lsl 3]    // load s->coefs[c] into z1
 4    add z0.d, z3.d, z0.d                   // add i + s->offsets[c] into z0
 5    add x0, x0, x5                         // increment c by the number of lanes
 6    ld1d    z0.d, p0/z, [x4, z0.d, lsl 3]  // gather-load a[i + s->offsets[c]] into z0
 7    fmul    z0.d, z1.d, z0.d               // multiply a[..] and s->coefs[c] into z0
 8    fadda   d2, p0, d2, z0.d               // accumulate all lanes of z0 to d2
 9    whilelo p0.d, w0, w1                   // update predicate p0
10    b.any   .L6                            // branch
```

Listing 4.6: Compiler generated bmgs_fd function

The GCC compiler succesfully vectorizes the inner-most loop (line 27) using SVE instructions. The compiler-generated code (with our comments) is shown in Listing 4.6. Access to the array `a` is non-contiguous, which requires the use of gather-load instructions (line 6). To compute the running sum of products, the reduction instruction `fadda` is used in line 8. This instruction computes the sum of values in an SVE register *in-order*, which is necessary to preserve the order of computation and to comply with the IEEE floating-point standard.[1] The `fadda` instruction usually has high latency due to sequential execution implementation. Typically, two times longer SVE width results in two times fewer iterations. However, due to a small number of iterations (19), the 512-bit SVE executes three iterations while the 256-bit executes five (67% more). Only three of eight lanes are active in the last iteration of 512-bit SVE.

An alternative SIMD approach is a vectorization of the outer loop in line 20. Figure 4.15 shows a visual representation of such approach for $l_{SIMD} = 4$. In this case, the stencil operator simultaneously updates values for four neighboring grid points (each point is

[1]This can be avoided with the use of `-ffast-math` flag, which generates tree-like order reduction. This most likely improves performance, but it has the downside of a bigger floating-point error.

computed in a separate lane). Each iteration loads four values of the same adjacent points in the SIMD register. Then, normal *fmla* operation is used to accumulate products. This approach avoids a reduction operation but requires a scatter-store operation to store `b[j]`. However, this is indifferent to the inner-loop vectorization where a separate memory flow is issued for each write of `b[j]`. In the code, developers try to enforce a vectorization of the outer loop by using a `#pragma omp simd` directive. However, the GCC compiler chooses not to vectorize this loop for SVE. (Optimization report does not clearly explain why this is not possible.)

To analyze outer-loop vectorization's effect, we vectorize the loop manually using intrinsic functions. The central concept of any outer-loop vectorization is executing the work of the inner loop in each lane of the SVE register. Therefore, this is only possible if the number of inner-loop iterations is constant with respect to the outer loop. In other words, each lane of the SVE register must do an equal amount of work. In our example, this is the case because `s->ncoefs` is constant.

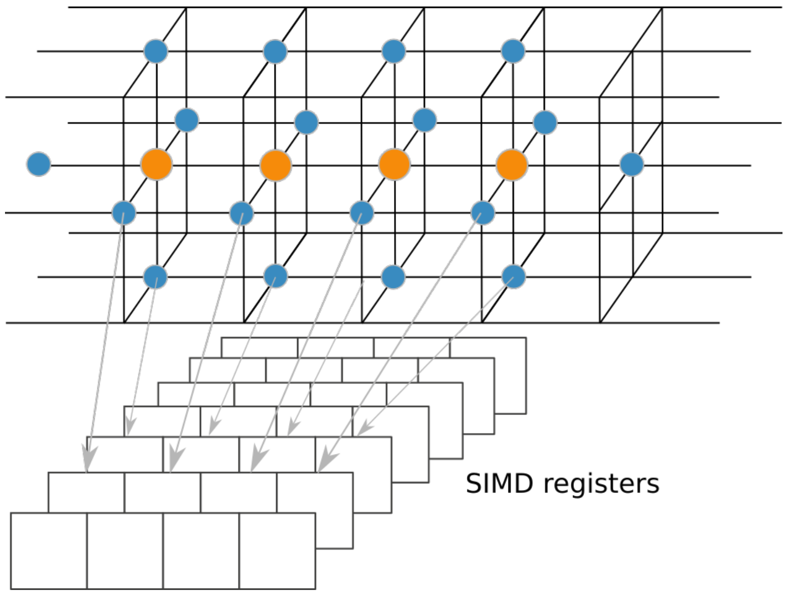


FIGURE 4.15: Visualization of the outer-loop vectorization

```c
void bmgs_fd_sve (const bmgsstencil* s, const double* a, double* b)
{
    /* Skip the leading halo area. */
    a += (s->j[0] + s->j[1] + s->j[2]) / 2;

    for (int i0 = 0; i0 < s->n[0]; i0++) {
        for (int i1 = 0; i1 < s->n[1]; i1++) {
            svint64_t sn1_vec = svdup_s64(s->n[1]);
            svint64_t sn2_vec = svdup_s64(s->n[2]);
            svint64_t sj1_vec = svdup_s64(s->j[1]);
            svint64_t sj2_vec = svdup_s64(s->j[2]);

            svint64_t i1_vec = svdup_s64(i1);
            svint64_t i0_vec = svdup_s64(i0);
            // calculate independent part for i
            svint64_t temp2 = svadd_x(svptrue_b64(), sj2_vec, sn2_vec);
            svint64_t tempi1 = svmul_x(svptrue_b64(), i1_vec, temp2);

            svint64_t tempi_2 = svmul_x(svptrue_b64(), sn1_vec, temp2);
            tempi_2 = svadd_x(svptrue_b64(), sj1_vec, tempi_2);
            svint64_t tempi2 = svmul_x(svptrue_b64(), i0_vec, tempi_2);

            // ipart
            svint64_t tempi = svadd_x(svptrue_b64(), tempi2, tempi1);

            // calculate independent part for j
            svint64_t tempj1 = svmul_x(svptrue_b64(), sn1_vec, sn2_vec);
            svint64_t tempj2 = svmul_x(svptrue_b64(), i1_vec, sn2_vec);
            svint64_t tempj = svmla_x(svptrue_b64(), tempj2, i0_vec, tempj1);

            int64_t i2 = 0;
            svbool_t pg = svwhilelt_b64(i2, sn2);
            do {
                svint64_t i2_vec = svindex_s64(i2, 1);
                svint64_t i_vec = svadd_x(pg, i2_vec, tempi);
                svint64_t j_vec = svadd_x(pg, i2_vec, tempj);

                svfloat64_t x_vec = svdup_f64(0.0);

                for (int c = 0; c < s->ncoefs; c++) {
                    svfloat64_t scoefs_vec = svdup_f64(s->coefs[c]);
                    svint64_t soffsets_vec = svdup_s64(s->offsets[c]);
                    svint64_t load_ind = svadd_x(pg, i_vec, soffsets_vec);
                    svfloat64_t a_vec = svld1_gather_index(pg, &a[0], load_ind);

                    x_vec = svmla_x(pg, x_vec, a_vec, scoefs_vec);
                }
                svst1_scatter_index(pg, &b[0], j_vec, x_vec);

                i2 += svcntd();
                pg = svwhilelt_b64(i2, sn2);
            } while (svptest_any(svptrue_b64(), pg));
        }
    }
}
```

Listing 4.7: Outer-loop vectorization of Bmgs_fd

The implementation is shown in Listing 4.7. The start of the outer loop is in line 32. This loop is constructed with predicate-driven loop control. For each iteration, we update the predicate register with a *svwhilelt_b64* function (lines 32 and 51) and increment the loop counter with `svcntd`. In line 34, we use the `svindex` instruction to fill the SVE register with an arithmetic integer sequence. Each lane holds the loop counter `i2` of the outer loop in a serial execution. Afterward, we first calculate the indices of the input and output arrays `i` and `j`. These are computed for all lanes simultaneously in lines 35 and 36. All parts of the loop-invariant calculation are moved outside of the loop (lines 8 - 29). We then initialize an empty vector `x_vec` which holds the values for array `b[j]` before starting the inner loop, which goes over all stencil points. Constant data (coefficient and the offset of a particular stencil point) is loaded using the `svdup` instruction which duplicates a single value to a whole SVE register. When loading values from array `a`, we use the `svld1_gather_index` in line 28. For that, the indices are first computed with a `svadd_x` operation (line 43). A similar approach is used for storing data with `svst1_scatter_index`. Throughout the implementation, we use the `-x` form of SVE instructions which leaves the inactive lanes undefined. The other options would be to use the *-m* and *-z* forms (which merge the lanes or set them to zero). However, we found that using *-x* forms produces the code with the least instructions in assembly.

4.3.2.2 Electron density (construct_density)

In the LCAO mode of GPAW, the pseudo wavefunctions are represented as a linear combination of localized atom-centered orbital functions

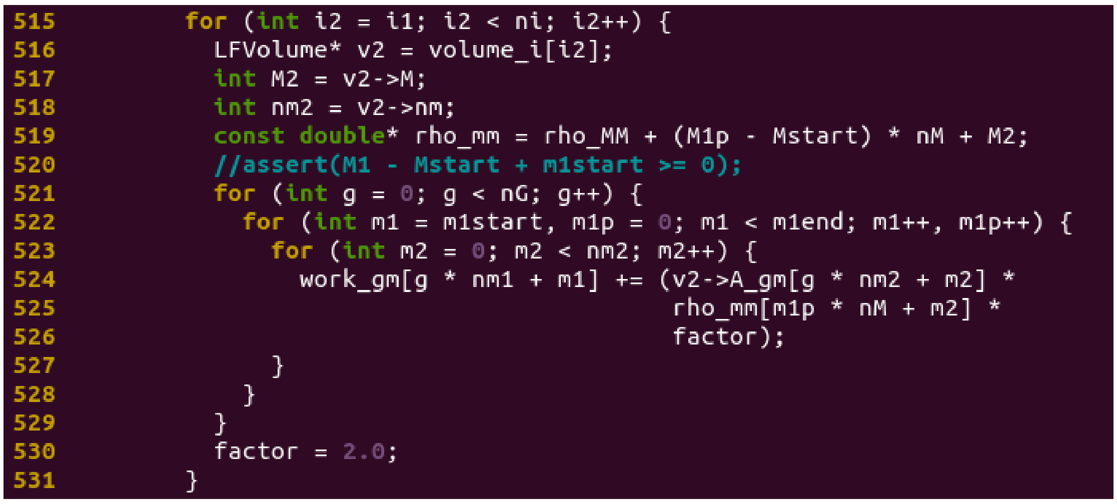
$$\left|\tilde{\psi_n}\right\rangle = \sum_{\mu} c_{\mu n} \left|\Phi_{\mu}\right\rangle . \tag{4.6}$$

Whereas the real-space representation uses wavefunctions as variational parameters, LCAO replaces these with a set of coefficients $c_{\mu n}$. This changes the formulation of Kohn-Sham equations which are solved in a matrix form. In this section, we focus on the computation of pseudo electronic density, which is needed to calculate the expected value of electron density at grid points. Pseudo density in LCAO mode is evaluated as

$$\tilde{n}(\mathbf{r}) = \sum_{\mu\nu} \rho_{\mu\nu} \Phi_{\mu}^{*}(\mathbf{r}) \Phi_{\mu}(\mathbf{r}) + \sum_{a} \tilde{n}_{c}^{a}(\mathbf{r}). \tag{4.7}$$

The second term is a separated contribution of each atom's pseudo core state density in a *frozen core* approximation. We refer to the work from Larsen et al. [45] for a detailed derivation of this equation and an overview of LCAO mode in GPAW.

Function `construct_density` calculates the first part of the equation 4.7. The function is composed of six nested for-loops. Loop 1 traverses over all boundary points in the grid. For each boundary point, new volumes whose pseudo density we are calculating are added to the list. Loops 2 and 3 traverse over all pairs of volumes in the list. Loop 4 traverses again over volumes, while loops 5 and 6 compute the interaction between orbital functions. We omit the details of the algorithm's implementation and only focus on the inner-most loops. Figure 4.16 shows loops 3, 4, 5, and 6 of the algorithm. It is the part of the function where most cycles are spent.



```
515         for (int i2 = i1; i2 < ni; i2++) {
516           LFVolume* v2 = volume_i[i2];
517           int M2 = v2->M;
518           int nm2 = v2->nm;
519           const double* rho_mm = rho_MM + (M1p - Mstart) * nM + M2;
520           //assert(M1 - Mstart + m1start >= 0);
521           for (int g = 0; g < nG; g++) {
522             for (int m1 = m1start, m1p = 0; m1 < m1end; m1++, m1p++) {
523               for (int m2 = 0; m2 < nm2; m2++) {
524                 work_gm[g * nm1 + m1] += (v2->A_gm[g * nm2 + m2] *
525                                           rho_mm[m1p * nM + m2] *
526                                           factor);
527               }
528             }
529           }
530           factor = 2.0;
531         }
```

FIGURE 4.16: GPAW construct_density function

The inner-most loop is composed of two multiplications and one addition. Similarly to `bmgs_fd`, the products from all iterations are accumulated. However, the access to `v2->A_gm` and `rho_mm` is sequential which removes the need for gather-load instructions. The main observation for the `construct_density` function is that the number of iterations in inner loops is minimal. For example, for the carbon-nanotube benchmark, the number of iterations for the inner-most loop (line 523) is either one or three. (One for exactly half executions and three for another half.) The same also holds for the loop in line 522. A small number of iterations was also observed for other UEABS use cases (copper filament and silicon cluster).

GCC 11.1.0 does not vectorize the inner-most loop with SVE instructions. The optimization report (see Listing 4.8) states that vectorization is not possible due to a complicated access pattern.

```
lfc.c:524:51: missed: failed: evolution of base is not affine.
lfc.c:523:35: missed: couldn't vectorize loop
lfc.c:524:39: missed: not vectorized: complicated access pattern.
```

LISTING 4.8: GCC optimization report

Again, we vectorize the function manually with SVE intrinsics. As for the `bmgs_fd` function, we consider different types of vectorization based on how many loops are vectorized. We consider three cases, vectorization of one, two, and three loops. In the case of two loops, our approach is the same as for the `bmgs_fd` kernel. When vectorizing three loops, we extend the idea of outer-loop vectorization one level higher. In this case, code from all three loops is executed with SVE instructions. Each lane of the SVE register corresponds to different loop iterations in line 521. The main problem with such an approach is the construction of indices for gather-load instructions in the lower loop levels.[1]

4.3.3 Benchmark extraction

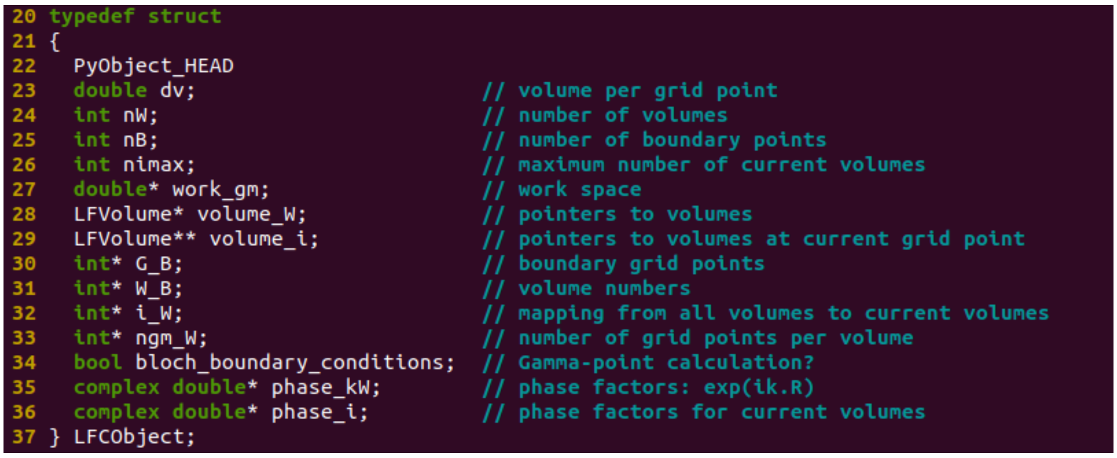
To evaluate `bmgs_fd` and `construct_density` kernels in the Gem5 simulator, we extract both functions into standalone benchmarks. This gives us more control over the code and significantly reduces the simulation time. Additionally, we remove the dependence on external libraries. To retain the realistic nature of CN-FD and CN-LCAO benchmarks, we completely replicate the function's execution profiles. In other words, the benchmarks should exhibit the same memory pattern and number of loop iterations and function calls. Therefore, we first record information about function calls, loop sizes, and all array indices.

For the CN-FD workload, the `bmgs_fd` function is executed 97211 times. We focus on those calls that evaluate the 19-point stencil, which happens in 38880 cases. Since the same work is done in each call, our benchmark can be scaled down by simply executing fewer calls. For our Gem5 simulations, we set the number of iterations (calls to `bmgs_fd`) to ten. For a 19-point stencil, the grid size equals $79 \times 79 \times 124$. However, each rank computes values for a smaller grid due to domain decomposition. Although our Gem5 model comprises four cores, we configure the benchmark for the full grid size. Such a scenario only simulates a bigger use case. Additionally, we note that the offset indices `s->offsets` stay the same during the executions. (The ordering of grid points does not change.)

Similarly, we record information for `construct_density` in CN-LCAO mode. In this case, the function is executed four times throughout the simulation. Compared to `bmgs_fd` the `construct_density` function has a more complicated profile. The number of iterations for loops 2, 3, and 4 depends on the number of volumes added to the list. In GPAW, all information about localized atomic orbitals is stored in `LFCobject` struct. (See Figure 4.17.) To preserve the structure of loops, we extract the values of

[1]`https://gitlab.jsc.fz-juelich.de/brank1/gpaw-benchmarks/-/blob/main/construct_density/lfc.c`

all `LFCObject` fields. Afterward, we use this information to create an identical instance of `LFCobject` in our benchmark. Furthermore, we decrease the number of iterations of loop 1 (`LFCObject->nB`) from 1,593,344 to 100,000 to reduce simulation time. (We compute pseudo density only for one part of the grid.)



```
typedef struct
{
  PyObject_HEAD
  double dv;                         // volume per grid point
  int nW;                            // number of volumes
  int nB;                            // number of boundary points
  int nimax;                         // maximum number of current volumes
  double* work_gm;                   // work space
  LFVolume* volume_W;                // pointers to volumes
  LFVolume** volume_i;               // pointers to volumes at current grid point
  int* G_B;                          // boundary grid points
  int* W_B;                          // volume numbers
  int* i_W;                          // mapping from all volumes to current volumes
  int* ngm_W;                        // number of grid points per volume
  bool bloch_boundary_conditions;    // Gamma-point calculation?
  complex double* phase_kW;          // phase factors: exp(ik.R)
  complex double* phase_i;           // phase factors for current volumes
} LFCObject;
```

FIGURE 4.17: GPAW LFCObject

Both benchmarks first allocate memory and initialize the grid/stencil data and information about localized functions. The actual functions (`bmgs_fd` and `construct_density`) are seperated in different files and compiled separately. This is necessary to avoid optimizations, that are possible due to compile-time data initializiation in `bmgsstencil` and `LFCObject`. Arrays holding physical properties (for example `a` and `b` arrays in `bmgs_fd` or `v2->A_gm` and `rho_mm` in `construc_density`) are filled with random values, because these do not change the execution profiles. Both benchmarks are available on Gitlab[1].

4.4 MiniFE

MiniFE is a proxy application whose primary purpose is to evaluate different parallel programming paradigms. Therefore, the code features many implementations targeting OpenMP, pthreads, MPI, Intel's TTB, CUDA, and Kokkos. However, SIMD was not the focus of any implementation, and code relies on the compiler auto-vectorization to generate efficient SIMD code.

4.4.1 Application chracteristics

The mini-app covers all essential parts of the real-world implicit Finite-Elements (FE) applications. This covers the generation of the mesh, computation of the FEM operators,

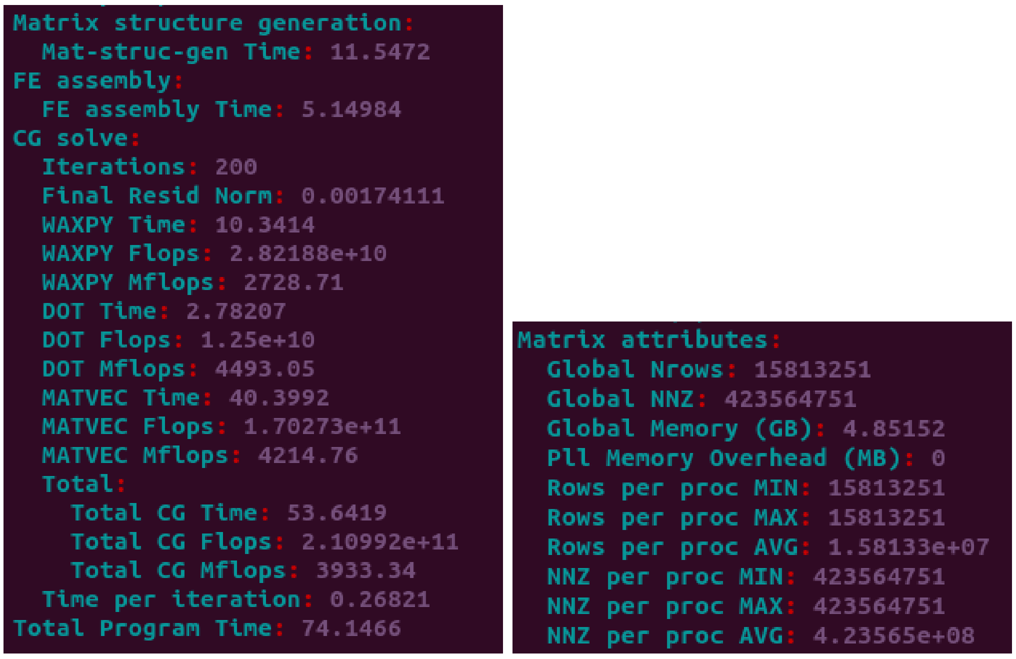
[1]`https://gitlab.jsc.fz-juelich.de/brankl/gpaw-benchmarks`

and assembly of the sparse matrix with the corresponding right-hand side. Afterward, the linear system is solved with a Conjugate Gradient (CG) method. The time needed for FE assembly and CG method depends on the use case. For non-linear problems or problems with time-dependent coefficients, the matrix is frequently reassembled throughout the simulation. In such cases, the FE assembly can take a significant portion of the runtime, and it is worth studying its optimization. However, when solving static problems or problems with very complex geometries, the matrix is usually assembled only once and then solved many times. (For example, with different solvers or preconditioners.) This usually results in a CG solver being predominant in the application. In our case, we consider the latter scenario and study only the CG solver. More specifically, we focus on the most intensive part of the CG solver, the computation of Sparse Matrix-Vector Multiplication (SpMVM).

The only input to MiniFE is the grid size for the mesh. MiniFE solves a steady-state heat conduction problem on a box-shaped domain. Input parameters nx, ny and nz determine the number of cells in the x, y and z directions. The generated mesh is a regular cartesian grid, t.i. all cells are cuboids with the same width, height, and depth. The total number of cells equals $n_x \times n_y \times n_z$. The number of vertices where the solution is sought is

$$N = (n_x + 1) \times (n_y + 1) \times (n_z + 1). \tag{4.8}$$

This number also equals the number of equations in the linear system (degrees of freedom). The resulting sparse matrix is highly regular because of the mesh's simple geometry. For solving the heat equation, MiniFE uses a stencil with 27 points, which results in a matrix with 27 nonzero elements in each row. (Less for vertices on the boundary.) The matrix parameters for $n_x = n_y = n_z = 250$ is shown on in Figure 5.2, right.



```
Matrix structure generation:
  Mat-struc-gen Time: 11.5472
FE assembly:
  FE assembly Time: 5.14984
CG solve:
  Iterations: 200
  Final Resid Norm: 0.00174111
  WAXPY Time: 10.3414
  WAXPY Flops: 2.82188e+10
  WAXPY Mflops: 2728.71
  DOT Time: 2.78207
  DOT Flops: 1.25e+10
  DOT Mflops: 4493.05
  MATVEC Time: 40.3992
  MATVEC Flops: 1.70273e+11
  MATVEC Mflops: 4214.76
  Total:
    Total CG Time: 53.6419
    Total CG Flops: 2.10992e+11
    Total CG Mflops: 3933.34
  Time per iteration: 0.26821
Total Program Time: 74.1466
```

```
Matrix attributes:
  Global Nrows: 15813251
  Global NNZ: 423564751
  Global Memory (GB): 4.85152
  Pll Memory Overhead (MB): 0
  Rows per proc MIN: 15813251
  Rows per proc MAX: 15813251
  Rows per proc AVG: 1.58133e+07
  NNZ per proc MIN: 423564751
  NNZ per proc MAX: 423564751
  NNZ per proc AVG: 4.23565e+08
```

FIGURE 4.18: MiniFE output

Upon execution, MiniFE generates a YAML file containing information about the timings of the three main kernels. These are the generation of the matrix structure, the assembly of finite elements, and the CG method. Additionally, the CG timing separates the times needed for WAXPY, dot product, and matrix-vector multiplication. Figure 5.2 left shows a timing output for the same example run on one node of Kunpeng 916 CPU. The CG method takes 53.6 seconds, 72% of the total runtime. Additionally, the time for matrix-vector multiplication is 40.4 seconds, 75% of the whole CG method.

4.4.2 Sparse Matrix-Vector multiplication

Sparse matrix-vector multiplication is a widely used kernel in scientific applications. To stay concise, we introduce the notation $A \times x = y$, where A is a sparse matrix of size $N \times N$ and x and y are dense vectors of size N. We call the number of nonzero elements of A N_{nz} and the number of nonzero elements per row N_{nzr}. Usually, the SpMVM kernel is executed many times for the same matrix A and different x. Therefore, the matrix A is stored in a compressed format to reduce space and enable a more efficient multiplication. Many algorithms for computing this kernel exist, and each algorithm is tightly connected to how we store the sparse matrix in memory.

All formats for sparse matrices reduce footprint by not storing zeros. The default format in MiniFE is a Compressed Sparse Row (CSR)[1], which is shown in Figure 4.19, in the

[1]Sometimes also called Compressed Row Storage (CRS)

middle. In this format, we store only the nonzero elements. Each nonzero entry in the matrix is stored in array *val* with row-major ordering. Additionally, array *col* stores the respective column indeces of each non-zero element and array *row* stores pointers to where each new row begins. Arrays *val* and *col* have sizes equal to N_{nz} and array *row* is of size $N + 1$. (We store an extra pointer in the end which points beyond the last element.) Algorithm for SpMVM multiplication with CSR matrix format is shown in Listing 4.9. The first loop traverses over all rows of matrix A. With the second loop, we go over all elements of a row, accumulating the sum of products of matrix A and x. Due to the CSR format, the accesses to values of matrix A are contiguous. However, access to array x is non-contiguous are requires the use of gather-load instructions.

```
for ( i = 0; i < N ; ++i ) {
    y[i] = 0.0;
    for ( j = row[i]; j < row[i+1]; ++j ) {
        y[i] += val[j] * x[col[j]];
    }
}
```

Listing 4.9: Matrix vector multiplication for CSR format

CSR matrix format is not very suitable for architectures with long SIMD vectors. The main problem is that the vectorization of the inner loop requires SIMD reduction operations which usually have high latency. We can partially mitigate this problem by accumulating the partial sums of each iteration in a SIMD register and only using a reduction operation at the end of the loop to sum all lanes to `y[i]`. However, when $N_{nzr} \approx l_{SIMD}$, the overhead of a reduction operation still negatively impacts performance. The second problem with the CSR format is that the N_{nzr} is usually not a multiple to l_{SIMD}. For very small N_{nzr}, it leads to SIMD instructions where only a few lanes are active.

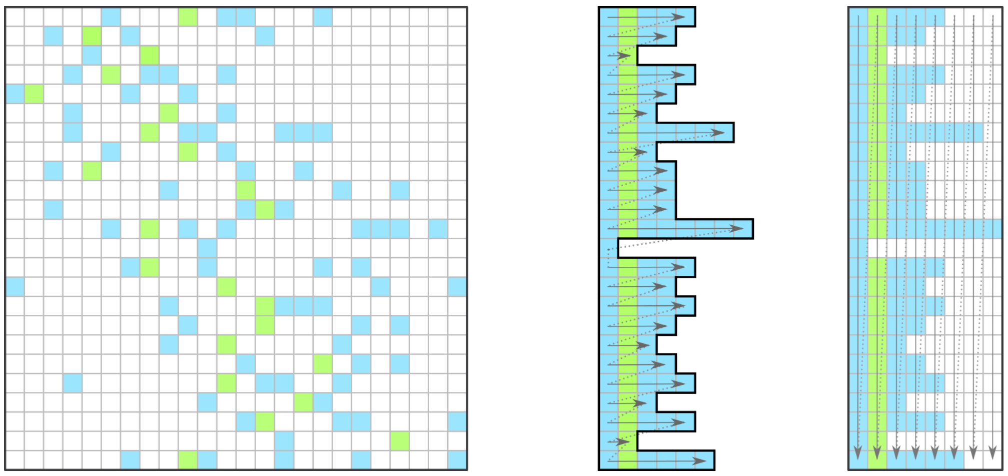


Figure 4.19: Default sparse formats in MiniFE

The other format included in the MiniFE is an ELLPack (or ELL) format. In this format, the values of A are stored in array *val* column-wise (see Figure 4.19, right side). The values are padded with zeros to the maximum number of nonzero elements in a row. The same is done for array *col* storing column indices. Because we store the same number of elements per row, array *row* is redundant. The ELL SpMVM kernel is shown in Listing 4.10. When vectorizing the inner loop, we do not need the reduction operation. However, the sums are accumulated multiple times to the result *y[i]*. Additionally, we compute a lot of products containing zeros for an irregular sparse matrix where N_{nzr} changes frequently. This is further addressed in a Sliced ELLPack format (SELL).

```
// assume y[i] = 0.0 for all i
for ( int j = 0; j < max_row; ++j ) {
    for ( int i = 0; i < N; ++i ) {
        jj = i + N * j;
        y[i] += val[jj] * x[col[jj]]
    }
}
```

LISTING 4.10: Matrix vector multiplication for ELL format

4.4.3 Sliced ELLpack and SELL-C-σ

The Sliced ELLPack (SELL) format improves the ELL format by splitting the matrix A into slices of C rows. Each slice is stored in an ELL format and only padded to the maximum element within each slice. Slices are stored contiguously for arrays *val* and *col*. Additionally, we store the offset of each slice in an array *slice_offsets* for easier implementation. Figure 4.20 shows the sliced ELLPack format for $C = 4$ and $C = 6$. When $C = 1$, SELL is equivalent to a CSR.

The SELL format has several advantages compared to ELL. Most importantly, this format enables a very efficient SIMD vectorization. Usually, C is selected as l_{SIMD} to fully utilize all lanes in the SIMD register. Matrix-vector multiplication for each slice can, therefore, be computed with a single loop, accumulating the products in a SIMD register that holds all $y[i]$ of a slice. Additionally, the number of padded zeros is significantly reduced for matrices with only a few rows with big N_{nzr}. (Consider a case where a single row of a matrix A has many more nonzero elements than all other rows.)

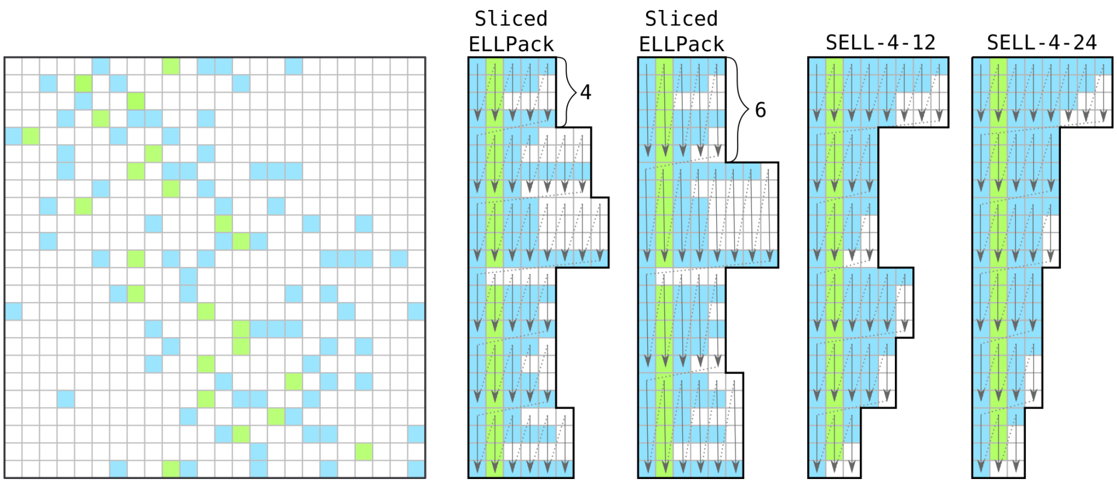


FIGURE 4.20: Sliced ELLpack and SELL-C-σ

To further improve the storage efficiency, we can also apply the SELL-C-σ format. In this format, rows are initially sorted by N_{nzr} in blocks of size σ. This groups rows with a similar number of nonzero elements together, which reduces the zero padding. Usually, σ is a multiple of C. Figure 4.20 right shows example when $\sigma = 12$ and $\sigma = 24$. When $\sigma = 1$, the format is equivalent to SELL.

4.4.4 Intrinsic implementation (VLA SELL)

In MiniFE, an elementary geometry results in a highly regular sparse matrix. Almost all rows (except those corresponding to vertices on the mesh boundary) contain 27 nonzero elements. For the case of $N_x = N_y = N_z = 60$, only 6% of rows contain less than 27 nonzero elements. We conclude that for MiniFE, the additional work of sorting rows does not outweigh the minor benefit of reduced storage. Therefore, we only implemented the SELL format. To efficiently utilize different SVE sizes, we implemented a vector length agnostic SELL, by setting $C = l_{SVE}$.

MiniFE separates different matrix formats by a set of predefined macros. For example, the CSR format is selected at compilation with a flag `-DMINIFE_CSR_MATRIX`. Throughout the code, matrix operations (memory allocation, structure generation, SpMVM kernel, etc.) are split depending on the chosen format. For our work, we have added a new macro `MINIFE_SELL_MATRIX` and implemented the SELL format. The code is publicly available. [1]

MiniFE is written in C++ and heavily uses the C++ standard library. Namely, arrays *val*, *col* and *row* are implemented as `std::vector<T>`. The class implementing the

[1]`https://gitlab.jsc.fz-juelich.de/epi-wp1-public/minife`

SELL matrix is similar to CSR and ELL, but the vector holding row offsets is replaced by a vector of slice offsets. Different slices are initialized with different OpenMP threads, ensuring each thread allocates space in the local memory. To maximize utilization of SVE registers, we set the slice size to the number of SVE lanes, which makes the algorithm vector-length-agnostic. If the code is not compiled for SVE, a default $C = 4$ is used.

Listing 4.11 shows the intrinsic SVE implementation of a matrix-vector multiplication for SELL format. The first loop in line 2 traverses over all slices. This loop is parallelized with OpenMP, each thread computing result for its submatrix. For each slice, we initialize an empty SVE register `sum` to accumulate partial results (line 3). The second loop (line 6) traverses the values of each slice. For each iteration, values of A are loaded contiguously from memory with *ld1* instructions. For x, we first load the column indices of A elements. Afterward, we use a gather-load instruction to gather elements of x into an SVE register. Finally an *fmla* instruction is used to accumulate partial products to *sum*. In line 13, the result is stored back to y.

```
1   #pragma omp parallel for
2   for(int slice_id=0; slice_id < num_slices; slice_id++) {
3       svfloat64_t sum = svdup_f64(0);
4       int C = svlen_f64();
5       svbool_t pg = svwhilelt_b64(0, C);
6       for(int i=Asliceoffsets[slice_id]; i<Asliceoffsets[slice_id+1];i += C){
7           svfloat64_t acofs = svld1(pg, &Acoefs[i]);
8           svuint64_t indices = svld1sw_u64(pg, &Acols[i]);
9           svfloat64_t xcofs = svld1_gather_index(pg, &xcoefs[0], indices);
10
11          sum = svmla_z(pg, sum, acofs, xcofs);
12      }
13      svst1(pg, &ycoefs[slice_id * C], sum);
14  }
```

Listing 4.11: SVE sliced ELLPack SpMVM

Chapter 5

Results & analysis

5.1 SVE ISA exploitation

5.1.1 Auto-vectorization

Table 5.1 reports the number of vectorized loops in TSVC by different compilers. FCC vectorizes most loops (97), followed by GCC (94) and ACfL (93). However, for eleven loops, ACfL recognizes an opportunity for vectorization but does not vectorize due to the cost model predicting no benefit. However, in five of eleven cases, loops are instead vectorized using NEON instructions. On the other hand, GCC and FCC do not report any cases where vectorization is not beneficial. Figure 5.1 shows how vectorized loops overlap between different compilers.[1] A total of 115 loops were vectorized by at least one compiler, and 80 loops were vectorized by all three. Recent studies [46] also report similar numbers (95-111) for x86's AVX-512 SIMD set. Therefore, the VLA paradigm, introduced by SVE, does not introduce any significant drawbacks in terms of vectorization. On top of that, we manually vectorized 21 more loops with intrinsic functions giving 136 vectorizable loops. The code is available in Gitlab.[2]

GCC	ACfL	FCC	Intrinsic
94 (+0)	93 (+11)	97 (+0)	136

TABLE 5.1: Number of vectorized loops

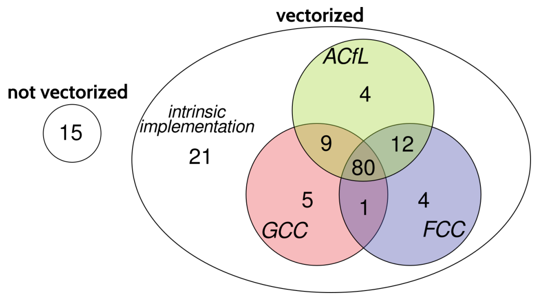


FIGURE 5.1: TSVC vectorization

[1]Figure 5.1 includes eleven loops where SVE vectorization was possible but not applied.
[2]`https://gitlab.jsc.fz-juelich.de/brank1/tsvc_sve`

Table 5.2 shows the total runtime of the entire TSVC suite (151 loops) on the A64FX processor. Here, t_{scalar} and t_{vec} refer to the accumulated time of all loops, with vectorization disabled and enabled, respectively. The fastest binary is produced by the Fujitsu compiler (2.68s), followed by ACfL (2.92s) and GCC (3.73). Although this corresponds with the number of vectorized loops, the result is partly also a consequence of a faster scalar code. We observe that FCC uses aggressive loop-unrolling, and many loops are automatically unrolled over a factor of two or four. This is reflected in the size of the FCC's binary, which is 2.2 and 2.9 times bigger than ACfL and GCC. Also, we note that some loops take considerably longer time than others, so the total time is heavily weighted towards a few computationally more expensive loops. Finally, $t_{intrinsic}$ is the accumulated time for the case where 136 loops were vectorized with intrinsic functions. This produces the fastest code (1.64-1.69s), with compilers showing little difference in performance. Although this code is 40% faster than the code produced by FCC, the difference mostly comes from additional vectorization. (t_{vec} is dominated by loops that are not vectorized.) In Table 5.3, we count the number of loops that achieve a vector speed-up of at least 1.15, 2, 4, and 8. A 512-bit SVE gives for single-precision loops $\eta_{max} = 16$.

	GCC	ACfL	FCC
t_{scalar}	7.58	8.84	6.10
t_{vec}	3.73	2.92	2.68
$t_{instrinsic}$	1.64	1.68	1.69

TABLE 5.2: TSVC total time (s)

	GCC	ACfL	FCC
$\eta \geq 1.15$	86	83	76
$\eta \geq 2$	81	77	73
$\eta \geq 4$	72	67	51
$\eta \geq 8$	42	46	23

TABLE 5.3: TSVC speed-up

Compilation of the TSVC benchmark shows that compilers can exploit most features of SVE. Here, we show four TSVC loops that show different exploitation of the SVE ISA. Consider loop *s111*, shown in Listing 5.1. The loop counter is incremented by 2 in each iteration and requires handling of data with stride 2. All three compilers vectorize this loop. However, ACfL and FCC, based on the LLVM compiler technology, use a different approach to GCC. GCC uses *ld2* instructions to load elements of `a` and `b`, followed by a scatter-store instruction. On the other hand, ACfL and FCC use normal *ld1* load instructions that also load elements that are not needed. Afterward, *zip* and *unzip* instructions are used to separate even and odd elements into different registers. The code produced by GCC is roughly 20% faster. Listing 5.2 shows loop *s4113* where `a[ip[i]]` and `b[ip[i]]` results in a complex memory pattern. (Here, array `ip` is declared with a keyword *restrict* which signals the compiler that no aliasing occurs.) All three compilers successfully vectorize this loop with gather-load and scatter-store instructions.

```
for (int i = 1; i < LEN_1D; i += 2) {
    a[i] = a[i - 1] + b[i];
}
```

Listing 5.1: Loop *s111*

```
for (int i = 0; i < LEN_1D; i++) {
    a[ip[i]] = b[ip[i]] + c[i];
}
```

Listing 5.2: Loop *s4113*

Loop *s314* (see Listing 5.3) requires transformation of *if* statement to a *max* reduction. All compilers recognize this transformation and vectorize the loop with floating-point maximum *fmaxnmv* instruction. Finally, loop *s271* (see Listing 5.4) operates only on elements of array where `b[i] > 0`. All compilers generate SVE *cmpeq* operations to create a predicate register with only those active lanes that satisfy the condition. Such predicate is then used in combination with the *fmla* instruction.

```
x = a[0];
for (int i = 0; i < LEN_1D; i++) {
    if (a[i] > x) {
        x = a[i];
    }
}
```

Listing 5.3: Loop *s314*

```
for (int i = 0; i < LEN_1D; i++) {
    if (b[i] > (real_t)0.) {
        a[i] += b[i] * c[i];
    }
}
```

Listing 5.4: Loop *s271*

Additionally, we have found many cases where compilers generate more complex SVE instructions (*rev*, *zip*, *splice*, *select*, *fnmls*, *ext*, *umulh*). In all 151 loops, we have identified only one case (*s341*), where compilers fail to spot a particular feature of SVE.

Loop *s341* is shown in Listing 5.5. In this loop, all values of `b` which are greater than zero are packed in the array `a`. At first, the loop seems hard to vectorize due to an increment `j++` which makes `j` depend on previous iterations. However, SVE includes a *compact* instruction which concatenates active elements of the register and fills the rest with zero. Therefore, we can first use a *cmpgt* instruction to find all lanes where `b[i] > 0.0f`. Afterward, *cntp* instructions counts the number of active elements in a predicate register. This information is used to store the right number of elements to `a` and correctly increment `j`. The intrinsic implementation is shown in Listing 5.6.

```
j = -1;
for (int i = 0; i < LEN_1D; i++) {
    if (b[i] > 0.0f) {
        j++;
        a[j] = b[i];
    }
}
```

Listing 5.5: Loop *s341*

```
do {
    svfloat32_t bv = svld1_f32(pg, &b[i]);
    cg = svcmpgt(pg, bv, zerov);
    svfloat32_t res = svcompact_f32(cg, bv);
    int inc_j = svcntp_b32(svptrue_b32(), cg);
    tg = svwhilelt_b32(0, inc_j);
    svst1(tg, &a[j], res);
    j += inc_j;
    i += svcntw();
    pg = svwhilelt_b32(i, LEN_1D);
} while (svptest_any(svptrue_b32(), pg));
```

Listing 5.6: Vectorized loop *s341*

As stated before, we identified 21 loops that were not vectorized by any compiler, but we managed to vectorize with SVE intrinsics. However, except for *s341*, missed opportunities do not originate from the SVE ISA. Instead, these loops require a specific code transformation that compilers do not recognize. Here we give a brief overview of the techniques that we used:

- **Loop splitting** is a technique where the loop is split into two separate loops. We use this to vectorize loops where a straightforward vectorization fails due to a specific iteration. This is because other iterations depend on it or the particular iteration has different properties than other iterations. In most cases, vectorization can still be applied if the special iteration is handled with scalar instructions. This applies to loops *s1113* and *s281*.
- **Loop peeling** is a special case of loop splitting where the first (or last) few iterations are split from the loop and performed outside of the loop. Missed opportunity for vectorization involving loop peeling was observed for loops *s244*, *s254*, *s255*, *s291*, *s292*, and *s293*.
- **Preloading** is a technique where data is preloaded into a SIMD vector to store the copy of the original data before it gets overwritten. Missed opportunity for vectorization involving preloading was observed in *s211*, *s1213*, *s241*, *s243*, and *s1244*.
- **Searching loops** are loops that search the first value in an array (and its corresponding index) that satisfies a certain condition. This applies to loops *s332*, *s481*, *s482*, *s3110*, and *s13110*. A way to vectorize this loop is to use the SVE vector compare operation to see if a searched value is in the current iteration. If it is found, then we manually check values one by one and find the first that fits the condition. The performance of such vectorization depends on the values of

the array. If a searched value is not in the first few iterations, vectorized version performs better.

- Some loops involve a combination of multiple techniques for a successful vectorization (for example, *s126*, *s232*, *s2251*).

5.1.2 Potential ISA extension

For cases where vectorization is not possible, we think about small changes in the SVE ISA that could open new vectorization opportunities. One example where SVE is not efficient are loops with a decrementing loop counter. For a VLA loop, *whilelt* and *whilele* instructions are used to correctly set lanes in a predicate register. However, SVE omits *whilegt* that would do this in a reverse order.[1] Compilers still manage to vectorize loops with a workaround. Consider loop `s112` in Listing 5.7. The loop is trivial to vectorize due to a single addition and no dependencies between iterations. However, a decrementing loop counter makes it difficult to produce a VLA code. To solve this problem, compilers rely on *rev* instruction that reverses the contents of the SVE register in each iteration. This approach is shown in Listing 5.8 (four *rev* instructions are required). None of the compilers has reversed the loop to an incremental order, although this is possible.

```
for (int i = LEN_1D - 2; i >= 0; i--)
{
    a[i+1] = a[i] + b[i];
}
```

Listing 5.7: Loop *s122*

```
.LBB154_2:
  rev p1.s, p0.s
  ld1w    { z0.s }, p1/z, [x9, x19, lsl #2]
  ld1w    { z1.s }, p1/z, [x10]
  add x8, x8, x21
  whilelo p0.s, x8, x20
  rev z0.s, z0.s
  rev z1.s, z1.s
  fadd    z0.s, z1.s, z0.s
  rev z0.s, z0.s
  st1w    { z0.s }, p1, [x9]
  addvl   x9, x9, #-1
  addvl   x10, x10, #-1
  b.mi    .LBB154_2
```

Listing 5.8: ACfL compiled *s122*

Another possible improvement to ISA was found in loop *s3112* (Listing 5.9) which computes a prefix sum. Although there exist algorithms that compute a prefix sum using SIMD instructions, they usually do that with partial sums in multiple sweeps. This could be avoided with a hardware implementation of a prefix sum SVE instruction.[2] Since

[1] Instructions *whilegt* and *whilege* were first introduced with SVE2.

[2] The idea of prefix sum hardware implementation was first introduced for vector computers in 1990 by Chatterjee et al.

the *fadda* operation already exists, which computes a sum-reduction in-order, prefix sums already appear naturally during the execution of this operation. Therefore, a modification to a prefix sum does not seem far-fetched.

```
sum = (real_t)0.0;
for (int i = 0; i < LEN_1D; i++) {
    sum += a[i];
    b[i] = sum;
}
```

LISTING 5.9: Loop *s3112*

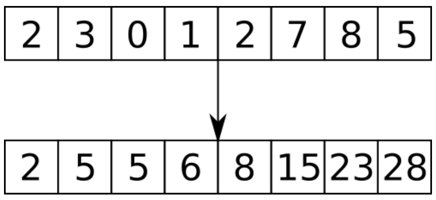


FIGURE 5.2: Possible SVE *prefix_sum*

Loop *s342* (Listing 5.10) is opposite to *s341* (Listing 5.5). Array `b[j]` is unpacked to only positive values of `a[i]`. SVE does not include instructions that would unpack the vector under predicate control (see a possible solution in Figure 5.3). Such instruction, which would have an opposite effect than *compact* instruction, would be needed for vectorizing the loop *s342*.

```
for (int i = 0; i < LEN_1D; i++) {
    if (a[i] > (real_t)0.) {
        j++;
        a[i] = b[j];
    }
}
```

LISTING 5.10: Loop *s342*

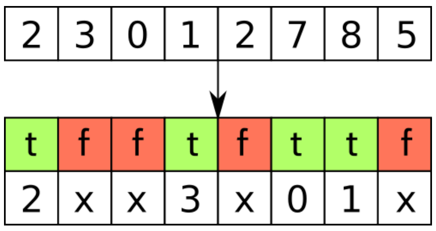


FIGURE 5.3: Possible SVE *unpack*

Loop *s258* is shown in Listing 5.11. The temporary variable `s` only changes value on certain iterations. Such a loop could be vectorized if instructions existed that would copy the last active element under predicate control. An example of such potential instruction is presented in Figure 5.4.

```
for (int i = 0; i < LEN_2D; ++i) {
    if (a[i] > 0.) {
        s = d[i] * d[i];
    }
    b[i] = s * c[i] + d[i];
    e[i] = (s + (real_t)1.) * aa[0][i];
}
```

LISTING 5.11: Loop *s258*

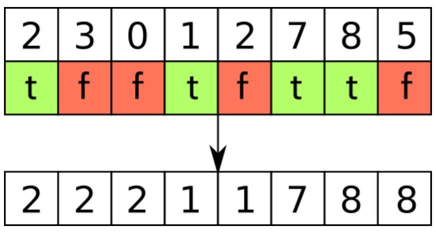


FIGURE 5.4: Possible SVE *last_active*

5.2 Validation of the Gem5 model

To ensure that the Gem5 model operates in a realistic CPU architectural space, we test it using a set of standard benchmarks. (See Section 2.2.1 for a more detailed description.) First, we analyze the memory behavior by running STREAM and Tinymembench. As explained in Chapter 3.3, the memory system of our Gem5 model (in particular, the hardware prefetcher and the interconnect) is highly simplified. Therefore, even if the measured memory bandwidth and latency resemble Graviton 2, many aspects of the cache hierarchy are simulated differently than in the Neoverse N1. Afterward, we show the results for the NAS Parallel Benchmark suite. For the compilation of three benchmarks, we use GCC version 11.1.0. We compile the code on the Kunpeng 916 machine for a generic ARMv8 target. For NEON binaries, we use the `-O3` optimization without any Kunpeng 916 specific optimizations. (The same binaries are used for Graviton 2 as for the Gem5 model when targeting NEON.) In the case of SVE, we added an extra compilation flag `-march=armv8.2-a+sve`.

5.2.1 STREAM

First, we evaluate the STREAM benchmark compiled for NEON (without SVE). Single-core results for both systems are shown in Figure 5.5. For the Gem5 model, we also report the memory bandwidth if we disable the hardware prefetcher. Here, we report the application-perceived bandwidth, which the STREAM benchmark writes on the output. (See Section 2.2.1 for a detailed explanation.)

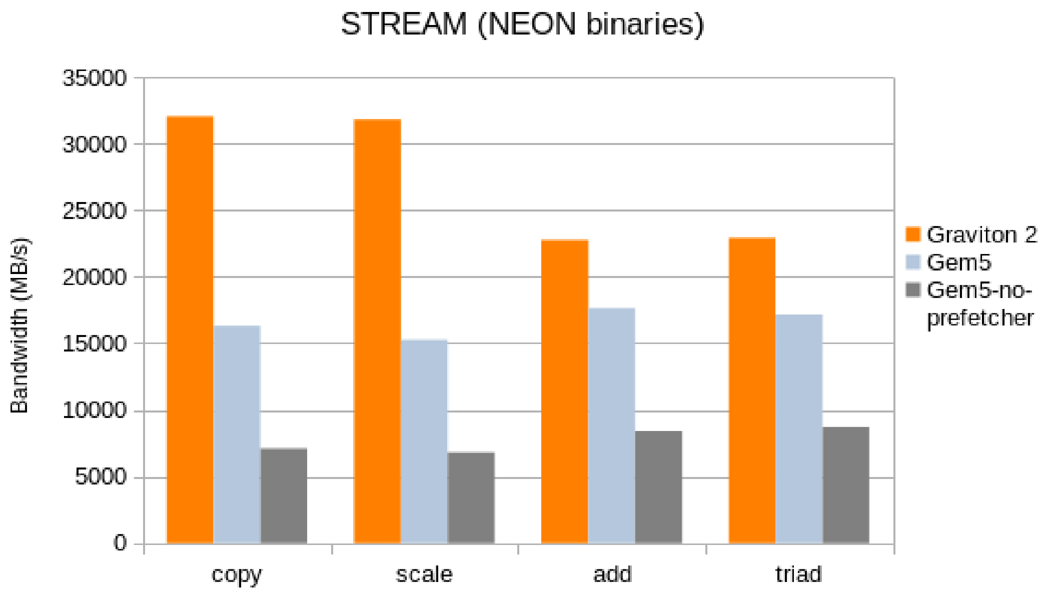


FIGURE 5.5: STREAM results (NEON binary)

Graviton 2 obtains a bandwidth of around 32 GByte/s for *copy* and *scale* kernels and 23 GByte/s for *add* and *triad* kernels. Therefore, a single core is able to saturate more than one memory channel (25.6 GByte/s). Gem5 model shows more consistent results, ranging from 15.2 GByte/s for the *scale* kernel up to 17.6 GByte/s for the *add* kernel. Disabling the hardware prefetcher reduces the bandwidth significantly. In this case, the maximum observed bandwidth is 8.7 GByte/s for the *triad*. The results show that the hardware prefetcher heavily influences the memory bandwidth for a single core. To inspect this further, we look at the Gem5 reported sources of memory accesses over the memory controller. Table 5.4 shows the number of read bytes and written bytes by different sources for a single iteration of the *triad* kernel. Due to the write-allocate cache policy, the total number of read bytes is close to $3 \times 10,000,000 \times 8 = 240$ MByte, which is expected. The hardware prefetcher reads 231.2 MByte, roughly 96.2% of the data.

mem_ctrls.bytesRead::cpu0.inst	2,752
mem_ctrls.bytesRead::cpu0.data	8,811,968
mem_ctrls.bytesRead::cpu0.prefetcher	231,202,368
mem_ctrls.bytesRead::total	240,014,336
mem_ctrls.bytesWritten::total	80,001,664

Table 5.4: Gem5 memory controller (STREAM *triad*)

Unfortunately, the Neoverse N1 core does not include any performance counters for measuring the effectiveness of the hardware prefetcher. This includes the counters for cache refills, which exclude those caused by the hardware prefetcher. However, considering a very regular access pattern of STREAM kernels, there is a strong indication that hardware prefetching is also the leading cause of high single-core bandwidth on Graviton 2.

Table 5.5 shows the results when running STREAM on a full node of Graviton 2 and across all four cores of the Gem5 model. In Graviton 2, we observe a maximum reported bandwidth of 175 GByte/s, which is roughly 85% of the theoretical peak (204.8 GByte/s)[1]. This exceeds the 75% limit, which is predicted by the write-allocate mechanism (See Section 2.2.1). To obtain a better memory bandwidth result, Graviton 2 applies the *write streaming* mode [47]. In this mode, the cache hierarchy does not allocate the cache line when the entire cache line is overwritten. The Gem5 model's maximum measured bandwidth is 19.2 GByte/s (*add* and *triad*), 75% of the peak. For the *copy* and *scale* kernels, the measured bandwidth is 17.8 GByte/s (67%) which is in line with the write-allocate limit.

[1]Configuration of 8 DDR4-3200 channels gives a total of 8 * 1600 * 2 * 8 = 204.8 GByte/s

	Graviton 2 - 64 cores	Gem5 - 4 cores
copy	170	17.8
scale	175	17.7
add	166	19.2
triad	174	19.2

TABLE 5.5: STREAM bandwidth for the full node

Finally, we report the effect of SVE size on the STREAM benchmark. In today's multi-core processor architectures, a single core usually cannot fully saturate the available memory bandwidth. The reason for this is that the core cannot handle enough in-flight memory requests (for example, due to a full LSQ). In this case, a bigger SIMD size can increase the observed bandwidth because a single SIMD memory instruction can transfer more data. Figure 5.6 shows the results for a binary compiled for an SVE target. For the *triad* kernel, the 512-bit model performs 3% better than the 128-bit. This is unsurprising because the 128-bit model can almost fully saturate the memory bandwidth due to the hardware prefetcher. If we disable the hardware prefetcher, the results change significantly. In this case, a bigger SVE improves the bandwidth. For example, for the *copy* kernel, the performance improved from 5.6 to 11.0 to 14.7 GByte/s for 128, 256, and 512-bit SVE lengths, respectively. The effect of a greater single-core bandwidth for larger SVE size was also observed on the A64FX.

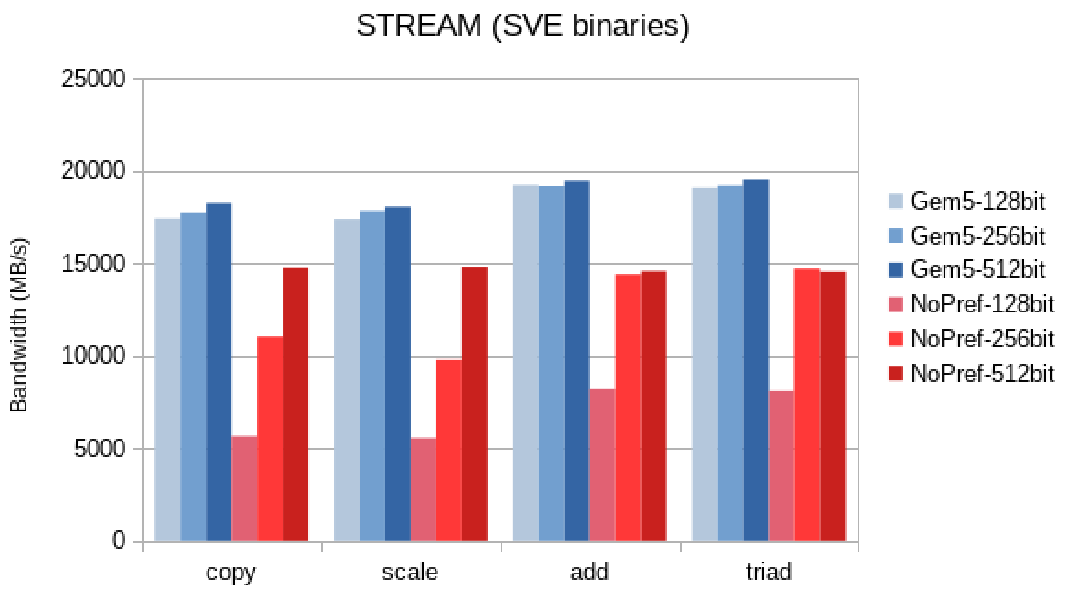


FIGURE 5.6: STREAM results (SVE binary)

Using the results of the STREAM triad, we estimate the ridge point (see Section 3.2.1) of our single-core Gem5 architecture for different SVE sizes. The peak performance of a 128-bit SVE model is 20 GFlop/s[1], which results in $AI \approx 1.05$ Flop/Byte. When the SVE size is 256 and 512 bits, the ridge point is approximately 2.1 and 4.2, respectively. These numbers appear very low compared to real-life machines because of the high single-core bandwidth. Usually, the ridge point is computed for a full node because applications utilize all cores. For example, Graviton 2 obtains 174 GByte/s, roughly 2.7 GByte/s per core. This results in a ridge point of 7.4 Flop/Byte. When we take into account four cores of the Gem5 model, the ridge point stands at 4.2, 8.3, and 16.7 Flop/Byte.

We also observed that the 128-bit SVE produces slightly better results than NEON. For the case of the *add* kernel, the SVE binary reaches roughly 8% better bandwidth. When analyzing the Gem5 reported counters, we noticed a strange artifact in the Gem5 model which simulates NEON store instructions differently than SVE store instructions. The number of committed instructions from the *MemWrite* operation class is twice as big as SVE, see Table 5.6. (NEON store instructions are decoded into two scalar-store micro-instructions.) As a result, the functional unit for memory operations is more occupied, and there are more *FuBusy* events. To confirm that this behaviour is purely due to the Gem5 and not the ISA, we have also inspected the assembly and ran the benchmark on the A64FX. There, the difference between a NEON binary and an SVE binary executed with 128-bit vectors is much smaller (less than 2%). The number of SVE committed memory instructions in Gem5 follows the expected values.

	NEON	128-bit SVE	256-bit SVE	512-bit SVE
commit::MemRead	10,000,170	10,000,170	5,000,170	2,500,170
commit::MemWrite	10,000,221	5,000,221	2,500,221	1,250,221
statFuBusy::MemWrite	1,075,700	64	34,447	28,367

TABLE 5.6: Memory instructions (STREAM *triad*)

5.2.2 Tinymembench

Figure 5.7 shows the memory latency measured with Tinymembench (compiled for the Arm NEON target). We only report the number for dual memory access without huge pages enabled. Overall, the Gem5 model behaves similarly to Graviton 2. When the data fits in the L1 cache (1-64 kByte), the measured time is too low to be seen on the plot. When the data is inside the L2 cache range (128 kByte - 1 MByte), the latency increases to 4.5 ns for Graviton 2 and 3.5 ns for the Gem5 model. This result coincides

[1]Theoretical peak performance in Flop/s is calculated as $\nu \times N_{SIMD} \times l_{SIMD} \times 2$, where ν is the clock frequency, N_{SIMD} is the number of SIMD units, l_{SIMD} is the number of 64-bit lanes and a factor 2 for fused multiply-add.

nicely with the nine cycles for retrieving the data from the L2 cache, which Arm reports. The figure also shows the effect of different L3/SLC cache sizes for the Graviton 2 and the Gem5 model. We see a jump at 64 MByte for Graviton 2 because the total SLC size is 32 MByte. For Gem5, the jump occurs at 16 MByte. Additionally, the latency is increasing for a bigger array size that fits in the SLC for Graviton 2. This is due to the mesh interconnect where the SLC is distributed among slices at each crosspoint. Finally, caching effects are minimal for the 64 MByte array size, leading to a latency between 80 and 90 ns when fetching data from the memory.

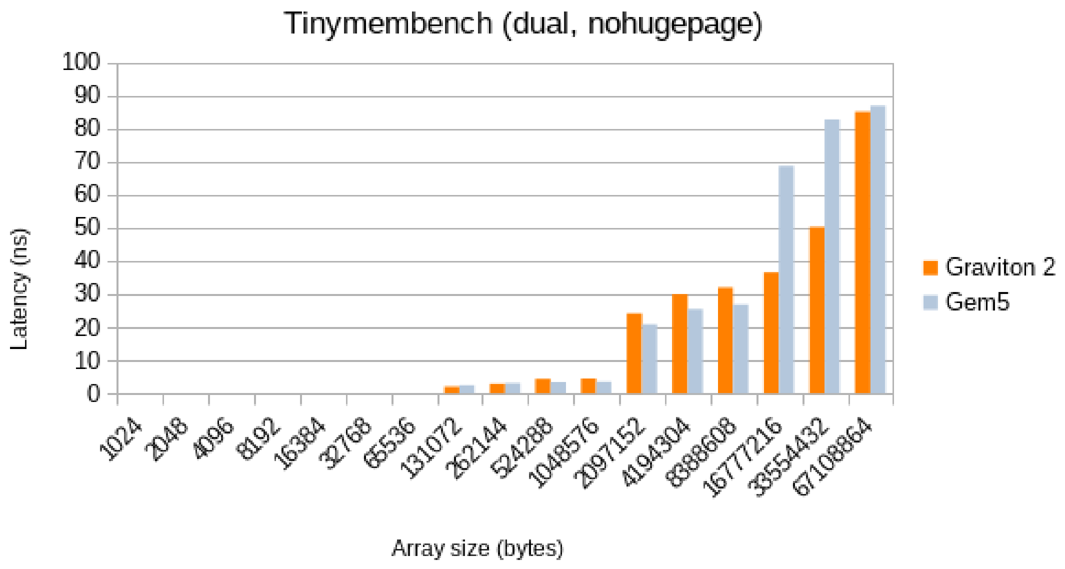


FIGURE 5.7: Tinymembench results

5.2.3 NAS parallel benchmarks

Figure 5.8 shows the results for all eight benchmarks of the NPB (compiled for the Arm NEON target). The performance is reported in a million operations per second (MOp/s) which is part of the NPB output. Overall, results from the Gem5 model largely resemble results on the Graviton 2. We see a good correlation between the two systems for most kernels (bt, ep, ft, lu, mg, sp). The difference between performance does not exceed 15% for these cases. A somehow more significant difference is observed for CG and IS kernels. Conjugate Gradient is a memory-bound kernel, and the lower performance in the Gem5 model is most likely the result of a smaller memory bandwidth. The biggest difference is observed for the IS kernel (972 MOp/s on the Graviton 2 and 378 MOp/s in the Gem5). We are aware that the model is behaving very differently than the Graviton 2 for this specific kernel. However, unlike most selected applications, this kernel is performing integer computations with a random memory access. We decided not to investigate this further and instead focus on selected applications and SVE.

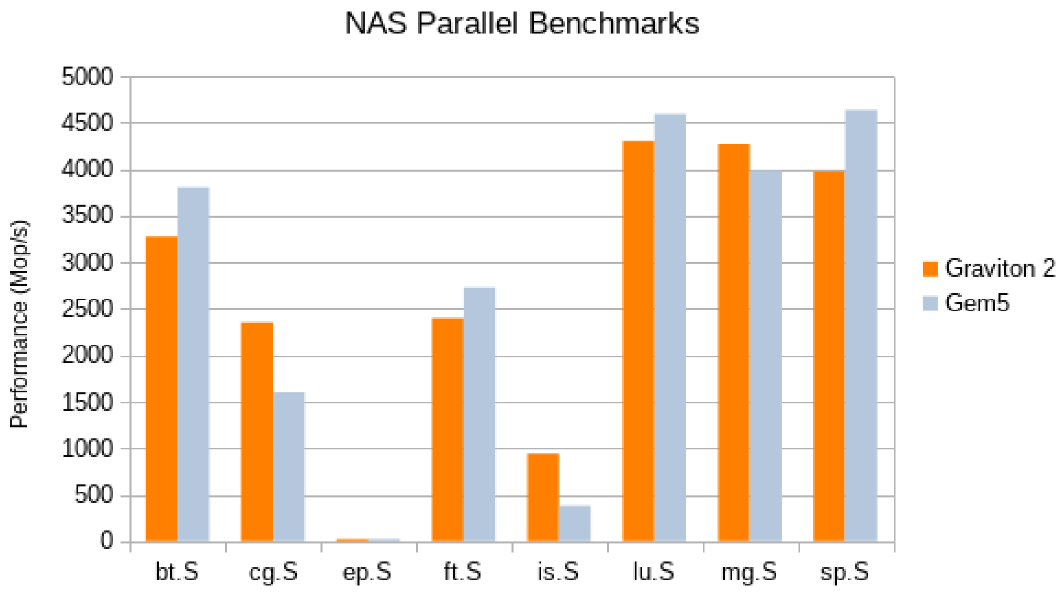


FIGURE 5.8: NPB results

Apart from the performance of the NPB kernels, we also analyze the cache behavior. As explained in Section 3.3.4, the cache configuration between the two systems is pretty different. Therefore, rather than comparing the absolute results of two systems, we would like to compare relative differences between different kernels on the same system. Figure 5.9 shows the number of cache accesses for different cache levels. Due to different environments, the number of cache accesses is measured differently on both systems. In Gem5, we read the counters from the statistics output file. For Graviton 2, we measure the selected hardware counters with the *perf* tool. (See [48] for a more detailed description of the hardware counters on the Neoverse N1.) Table 5.7 shows the names of specific counters that we measured. The number of L1 cache accesses is similar for both systems. The number of L2 cache accesses is roughly 2 to 2.5 times bigger for Graviton 2. However, we see an evident correlation between different NPB benchmarks for both systems. The last level cache shows a different picture. Graviton 2 behaves completely differently than our Gem5 model and we see no similarities between the two systems. (Some kernels exhibit up to 10 times more cache accesses.) Unfortunately, we could not identify the exact reason for such a large difference. We suspect that the hardware prefetcher in Graviton 2 prefetches significantly more cache lines than the Gem5 prefetcher. This further expresses the fact that simplications in the memory system can lead to different results. However, considering the results of the NPB kernels, the L3 cache accesses do not seem to massively affect the overall performance.

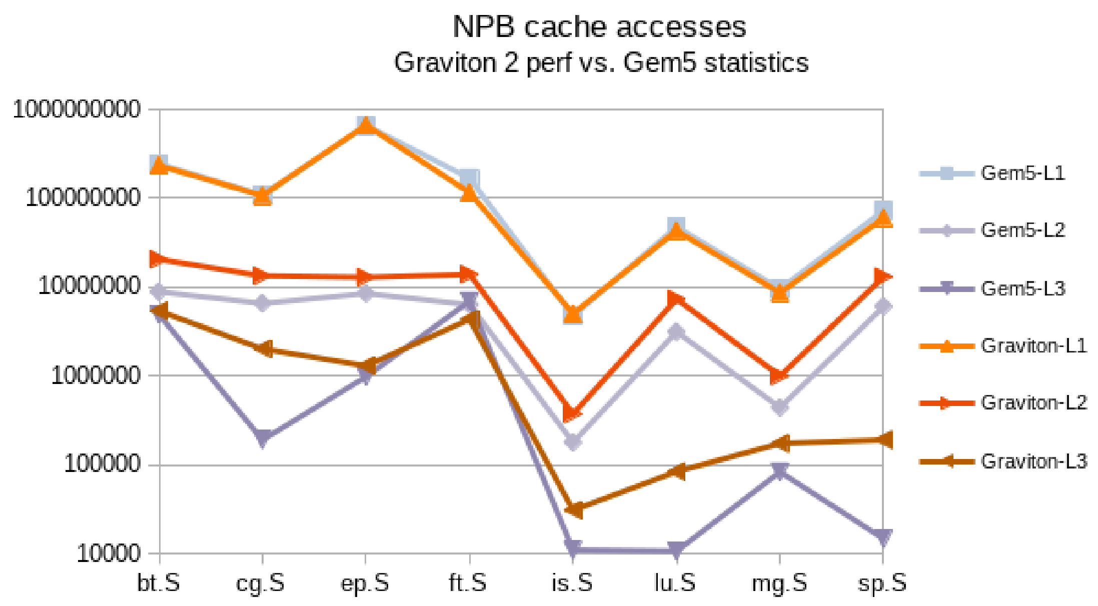


FIGURE 5.9: NPB cache accesses

	Neoverse N1 hardware counter	Gem5 statistic
L1	0x04, L1 data cache access	dcache.overallAccesses::total
L2	0x16, L2 cache access	l2cache.overallAccesses::total
L3	0x36, Last level cache access, read	l3cache.overallAccesses::total

TABLE 5.7: Measured cache-access statistics

5.3 Selected applications

In this section, we report and analyze the performance of selected applications. We evaluate applications for three different SVE sizes in the Gem5 simulator: 128, 256, and 512 bits. Apart from the SVE size, the model is architecturally completely the same for all SVE sizes[1]. Therefore, the difference in results is purely a consequence of the different SVE lengths. For most figures, we also include the results for Graviton 2 and A64FX processors for comparison.

[1]The only difference is the configuration of latencies for in-order reduction operations based on the A64FX.

5.3.1 OpenBLAS

5.3.1.1 DGEMM benchmark

Figure 5.10 shows the results of the DGEMM benchmark (see Section 3.2.3) for the three systems. To account for different clock frequencies of Gem5/Graviton 2 and the A64FX processor, we report the performance in Flop/cycle. The theoretical peak performance for all systems is 8, 16, and 32 Flop/cycle for 128, 256, 512-bit SIMD, respectively.[1] We see that the performance scales well with the increasing SVE size in the Gem5 model. For example, at 128-bit SVE, the performance reaches 7.58 Flop/cycle for $N = 800$, roughly 95% of the peak performance. This results is almost the same as the result on Graviton 2, despite it using NEON instructions. (On the figure, Gem5 plot coincides with the Graviton 2 plot.) When we increase the SVE size to 256 bits, the maximum observed performance is 14.75 Flop/cycle (92%). At 512 bits, the performance is slightly worse at 82% of the theoretical peak. This leads to a relative vector speed-up of $\eta_{256} = 1.95$ ($\epsilon_{256} = 0.98$) and $\eta_{512} = 3.45$ ($\epsilon_{512} = 0.86$). For all cases, the best performance is observed for the largest matrix size when $N = 800$. This suggests that the performance would be even higher for bigger matrix sizes, but we did not evaluate it due to long Gem5 simulation times.

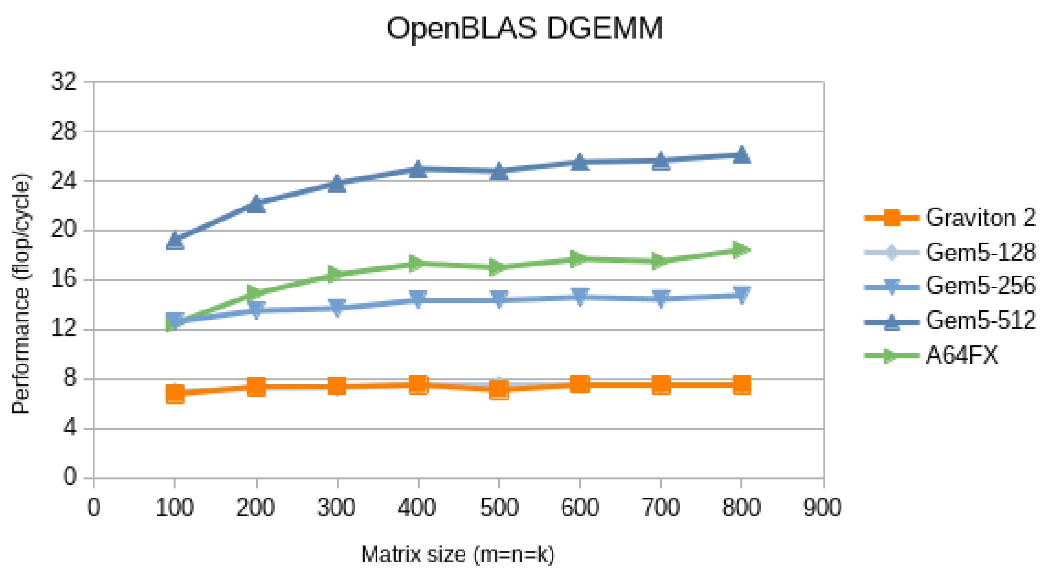


FIGURE 5.10: OpenBLAS DGEMM results

[1]Theoretical peak performance in Flop/cycle is calculated as $N_{SIMD} \times l_{SIMD} \times 2$, where N_{SIMD} is the number of SIMD units, l_{SIMD} number of 64-bit lanes and 2 for fused multiply-add. All three machines have two SIMD units.

For the A64FX processor, the performance is far from the theoretical peak (57.5%). This is mainly because our SVE macrokernel is not well suited for such a processor. As explained in Section 4.1, the size of the microkernel (m_r and n_r) depends on the number of SVE units and the latency of the *fmla* instruction. For most Arm server processors, the *fmla* latency is usually between 4-6 cycles, whereas for A64FX, it is 10 cycles[1]. Our microkernel has a size $2v \times 8$, or 16×8 for a 512-bit SVE, so the condition from Eq. 4.2 is not satisfied. A simple analysis of the microkernel shows that at peak performance, the SVE register is updated once every eight cycles. Therefore, the SVE unit must write back the result of the *fmla* operation in at least eight cycles to not introduce stalls in the pipeline. The minimum microkernel size for the A64FX, which would not stall the SVE unit, is 16×10. Additionally, related work from Nassyr[2] shows that other advanced optimizations must be used for a good performance on the A64FX (for example, a sector cache).

To better understand the difference between SVE sizes, we look at main performance counters reported by Gem5. Table 5.8 shows the main counters for one iteration of DGEMM for $N = 800$. When doubling the SVE size, we observe that the number of committed instructions is reduced by almost two. Therefore, SVE instructions dominate the execution, confirming that the macrokernel is a central part of the DGEMM execution. The majority of operations are multiply-accumulate and memory reads. We also see that the number of committed instructions per cycle is 3.87 for 128-bit and 3.76 for 256-bit. This is close to the theoretical maximum of the Neoverse N1 because the core's frontend can only fetch and decode four instructions per cycle. However, the IPC drops to 3.37 when increasing the SVE size to 512 bits.

SVE length	128	256	512
numCycles	135,698,359	69,962,138	39,409,718
committedInsts	525,166,248	263,609,448	132,831,048
ipc	3.87	3.77	3.37
committedInstType::SimdFloatMultAcc	258,240,000	129,120,000	64,560,000
committedInstType::MemRead	205,111,911	102,921,911	51,826,911
committedInstType::MemWrite	3,210,775	1,930,775	1,290,775

Table 5.8: DGEMM Gem5 counters

Next, we analyze the stalls in the rename stage. Table 5.9 shows the number of events for full buffers and the execution of the rename stage. We observe a major increase

[1]The A64FX manual reports a latency of 9 cycles for the *fmla* instruction, but our experiments shows it raises to 10 if both SVE units are fully occupied.

[2]https://www.youtube.com/watch?v=IwLm0Zt0E00

in *ROBFullEvents* and *fullRegistersEvents* for the 512-bit SVE. This results in roughly four million more stall (idle + block) cycles compared to the 256-bit SVE. On average, each full-buffer event results in 5.7 stalled cycles.

iqFullEvents	0	0	0
lsqFullEvents	60	72	149
ROBFullEvents	69,964	63,836	187,894
fullRegistersEvents	308,887	457,957	1,197,679
rename.idleCycles	154,607	396,280	1,925,533
rename.blockCycles	2,252,847	2,531,461	4,982,041
rename.runCycles	132,864,490	66,445,929	30,052,108
rename.unblockCycles	416,999	580,452	2,442,690
rename.squashCycles	8,246	6,846	6,154

Table 5.9: DGEMM Gem5 rename stalls

To understand the reason for a different behavior of the 512-bit SVE, we inspect the occupation of different functional units in the backend. Table 5.10 shows the relevant counters from the model's backend execution. We observe that the total number of *fuBusy* events is proportional to the execution time ($B_{rate} \approx 0.26$ for all b_{SVE}). For *fmla* operations, *fuBusy* is well correlated with the number of committed *fmla* instructions. However, the same is not true for memory read operations. When $b_{SVE} = 512$, the $B_{MemRead}$ increases from 7,443,836 to 8,471,654, despite the number of committed load instructions decreasing by almost a factor of two.

SVE length	128	256	512
fuBusy	143,539,638	70,256,579	37,686,723
fuBusyRate	0.27	0.26	0.27
statFuBusy::SimdFloatMultAcc	127,245,555	61,951,084	28,625,298
statFuBusy::MemRead	14,760,146	7,443,836	8,471,654
statFuBusy::MemWrite	148,059	152,497	148,245

Table 5.10: DGEMM Gem5 backend counters

A poor DGEMM performance usually occurs because the memory subsystem does not feed data to the core sufficiently fast. When waiting for data, the out-of-order window grows, which explains an increased number of *fullRegisterEvents.* To confirm this for the 512-bit SVE case, we analyze the performance of caches by looking at statistics of entries in the LSQ. Table 5.11 shows relevant load-to-use latencies of the requests put in the LSQ. A bigger SVE size increases the average latency of memory load instructions.

For example, when $b_{SVE} = 128$, the average latency to fetch data is 4.85 cycles, but for 256 and 512 bits, it rises to 5.49 and 7.47 cycles, respectively. Additionally, the standard deviation also increases. The reason for increased load latency lies in the SIMD load instructions. In the case of ideal scaling, the number of loads per cycle remains constant, and a 512-bit unit would load twice as much data as a 256-bit unit. However, since the cache line size remains constant, the number of cache lines to be loaded in the same time would also have to double. Generally speaking, with the increasing SIMD width, the core becomes more powerful but the bandwidth from L1 to L2 and beyond remains unchanged.

SVE length	128	256	512
lsq0.blockedByCache	16,405	127,654	207,503
lsq0.loadToUse::samples	163,207,879	81,927,879	41,287,879
lsq0.loadToUse::mean	4.85	5.49	7.47
lsq0.loadToUse::stdev	4.81	7.49	13.79
lsq0.loadToUse::0-9	96.90%	93.34%	90.57%
lsq0.loadToUse::10-19	2.88%	5.27%	2.29%
lsq0.loadToUse::20-29	0.08%	0.85%	2.61%
lsq0.loadToUse::30-39	0.02%	0.22%	2.18%
lsq0.loadToUse::40-49	0.01%	0.02%	0.65%
lsq0.loadToUse::50-59	0.01%	0.02%	0.47%

Table 5.11: DGEMM Gem5 LSQ counters

For our $2V \times 8$ macrokernels in OpenBLAS, the maximum load latency for A_r and B_r, which does not stall the pipeline, is eight cycles. We see that for the 512-bit case, the latency is close to this limit, and many load instructions most likely do not satisfy this condition. We point out that some read requests refer to the loading of C_r, which can be fulfilled with a higher latency. The table also shows the histogram of latencies. The number of load requests that are completed in less than 10 cycles drops from 96.6% for 128-bit to 90.95% for the 512-bit case. We conclude that the pipeline stalls caused by the load instructions are the leading cause of the poorer performance for 512-bit. Another thing we notice in the LSQ behavior is the increased number of events where the load instruction is not put in the LSQ due to blocked cache access. We have inspected the behavior of the L1 data cache and found that the reason for blocked accesses is not enough entries in the Miss Status Holding Register (MSHR).

The total utilized memory bandwidth is well below the maximum (around 1.5 GByte/s for 512-bit SVE). Therefore, we believe that the poor performance for 512 bits could be mitigated with a better prefetching (both in software and hardware). This is confirmed by looking at the number of useful hardware prefetches (Table 5.12). The coverage of the hardware prefetcher drops from 84%, 82% to 75% for an increasing SVE size. We see that for the L2 cache, more MSHR misses are caused by the core and less by the hardware prefetcher. Additionally, Table 5.13 shows the number of cache refills for each level. The SVE size does not majorly impact the overall data movement.

SVE length	128	256	512
mem_ctrls.bwTotal::total	427,020,050	827,581,341	1,461,207,106
l2cache.prefetcher.pfUseful	983,875	959,370	854,980
l2cache.prefetcher.coverage	0.84	0.82	0.75
l2cache.overallMshrMisses::cpu0.inst	1	1	1
l2cache.overallMshrMisses::cpu0.data	186,533	208,872	238,938
l2cache.overallMshrMisses::cpu0.prefetcher	1,639,286	1,628,409	1,592,834
l2cache.overallMshrMisses::total	1,825,820	1,837,282	1,831,773

Table 5.12: DGEMM Gem5 hardware prefetcher

SVE length	128	256	512
dcache.replacements	9,492,264	9,493,410	9,468,353
l2cache.replacements	1,895,217	1,907,249	1,910,344
l3cache.replacements	294,065	294,294	295,387

Table 5.13: DGEMM Gem5 cache refills

Gem5 N_{SVE} configuration

Lastly, we change the Gem5 model to study the effect of SVE units on the DGEMM performance. We vary N_{SVE} and b_{SVE} while keeping the same theoretical peak performance. Table 5.14 shows the results for three different configurations. The configuration of four 128-bit SVE units is significantly slower. The main reason is the bottleneck in the core's frontend, which can only fetch and decode four instructions per cycle. At peak performance, the kernel 2Vx8 would execute roughly six operations per cycle (four *fmla* and two *ld1rd*, see Listing 4.1). Additionally, the $2v \times 8$ macrokernel is too small for four SVE units. (The condition in equation 4.2 is not satisfied for our kernel.) Configurations 2×256 and 1×512 perform almost identically. We also see that the *fuBusyRate* is significantly lower for 128-bit, because of many SVE units. Finally, the load-to-use latency increases for bigger SVE size, which we also observed in the previous simulations.

SVE configuration	4 × 128 bits	2 × 256 bits	1 × 512 bits
Flop/cycle	7.6 (47%)	14.75 (92%)	14.92 (93%)
ipc	3.87	3.76	1.93
fuBusyRate	0.04	0.26	0.52
lsq0.loadToUse::mean	4.85	5.49	6.38

Table 5.14: DGEMM Gem5 performance (different N_{SVE})

5.3.2 GROMACS

5.3.2.1 Nonbonded benchmark

GROMACS implements two different kernels for computing nonbonded interactions (4xM and 2xMM). The 4xM kernel can be used for a SIMD size of 128 or 256 bits, whereas the 2xMM kernel is used for 256 and 512-bit SIMD. We run all possible combinations of kernels and SVE sizes. To better understand the SVE utilization, we first perform a static analysis using Armie. Armie is more appropriate for such analysis because we can count individual SVE instructions. (Gem5 only reports the number of instructions per operation class.) Table 5.15 shows the number of SVE and other instructions for four different SVE kernels (4000 particles). On average, 80% of the executed instructions are SVE instructions. We see that the number of scalar and SVE instructions reduces with increasing SVE size for both kernels. The 2xMM kernel commits fewer instructions than the 4xM for the same SVE size.

	128-4xm	256-4xm	256-2xmm	512-2xmm
scalar	110,244,188	63,365,134	44,508,210	31,109,804
SVE	387,875,180	225,222,284	203,290,290	118,602,638

Table 5.15: GROMACS instructions

Table 5.16 shows the number of committed instructions for most important SVE instructions (4000 particles). The most executed instruction is the floating-point multiplication *fmul*. This confirms our inspection of the *Nbnxm* kernel (see Listing 4.4 in Chapter 4.2.2) where *fmul* instructions are heavily used for the computation of the Lennard-Jones potential. In addition, the 4xM kernel uses significantly more *ext* operations for extracting a vector from a pair of vectors and the *faddv* reduction operation. On the other hand, 2xMM uses more *fadd* operations. We also observe specific instructions executed only for the 2xMM kernel and not for the 4xM. These instructions (*insr*, *sel* and *splice*) are used to duplicate and merge data of two particle clusters into a single SVE register.

Most used instruction, *splice*, copies all active elements of the first input operand together with the remaining lowest indexed elements of the second operand to the output vector.

	128-4xm	256-4xm	256-2xmm	512-2xmm
ext	34,690,460	20,145,370	2,083,088	1,198,400
fadd	427,980	416,430	17,908,800	10,413,928
faddv	4,166,232	2,396,912	427,980	416,430
fmad	36,532,818	21,014,826	18,640,860	10,742,032
fmla	17,627,470	10,143,574	8,439,284	4,837,168
fmul	120,059,800	69,194,580	60,086,964	34,652,814
insr	0	0	342,384	333,144
sel	0	0	1,041,544	599,200
splice	0	0	6,587,224	3,925,880

TABLE 5.16: GROMACS individual SVE instructions

In Figure 5.11, we see the results of the nonbonded benchmark for the Gem5 model and selected hardware. Performance is reported in the number of computed particle pairs per microsecond. We observe that results do not change when we increase the number of particles. The performance scales well with the increasing SVE size for each kernel. For the 4xM kernel, the performance rises from 111 to 189 *pairs/usec* ($\eta_{256} = 1.70$ and $\epsilon_{256} = 0.85$) when increasing the SVE size from 128 to 256 bits. The 2xMM kernel also scales well from 256 (175 *pairs/usec*) to 512-bit SVE (302 *pairs/usec*). This gives a speed-up of $\eta_{512} = 2.72$ and $\epsilon_{512} = 0.68$ compared to a 128-bit SVE; however, we are comparing two different kernels. For the same SVE size (256 bits), the 4xM kernel performs 8% faster than the 2xMM, despite executing more instructions. Comparing these results to real hardware, we observe a similar performance on the Graviton 2 (105 *pairs/usec*) as in our 128-bit model. The A64FX performance, on the other hand, does not reflect the performance of the 512-bit SVE case. Although the core operates at a lower frequency (1.8 GHz to 2.5 GHz), the 95 *pairs/usec* is approximately three times worse.

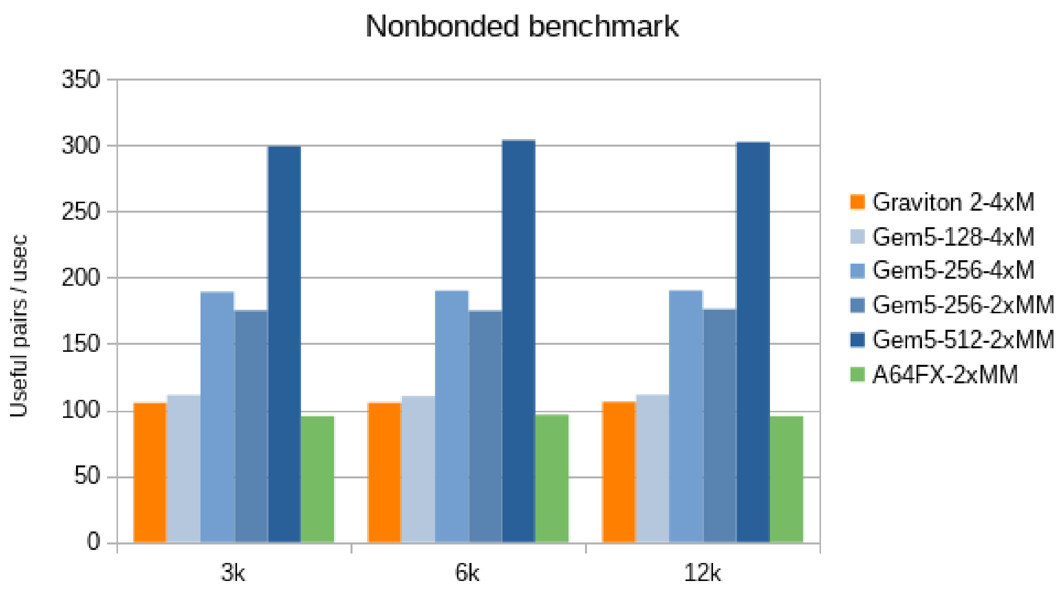


FIGURE 5.11: GROMACS results (nonbonded benchmark)

We try to better understand the performance of different SVE sizes by looking at the simulation of 12000 particles. Table 5.17 shows the main Gem5 counters for four cases. Overall, the 2xMM kernel exhibits less pressure on the SVE units ($B_{rate} = 0.30$) compared to the 4xM kernel ($B_{rate} = 0.54$-0.59).

	128-4xm	256-4xm	256-2xmm	512-2xmm
numCycles	563,818,836	330,754,181	357,226,320	208,065,513
committedInsts	1,416,203,491	818,442,221	702,378,351	422,129,231
fuBusy	831,618,075	446,318,355	211,787,091	129,621,491
fuBusyRate	0.59	0.54	0.30	0.30

TABLE 5.17: GROMACS Gem5 counters

Table 5.18 reports the number of relevant full-buffer events and the renaming statistics. The number of events for a full reorder buffer decreases for the 2xMM kernel but stays roughly the same for different SVE lengths. This suggests that *ROBFullEvents* are caused by the scalar instructions. Additionally, the number of *fullRegisterEvents* is reduced by approximately 40% when we double the SVE size. This number is well correlated with the number of committed instructions. However, it is significantly smaller for the 4XM kernel than the 2xMM when $b_{SVE} = 256$. These results are confirmed when looking at the stall cycles in the rename stage. The number of blocked and idled cycles decreases for a bigger SVE size but increases when we switch from the 4xM to the 2xMM kernel. Other backend queues (IQ, LSQ) showed no bottlenecks.

	128-4xm	256-4xm	256-2xmm	512-2xmm
ROBFullEvents	9,007,535	9,365,214	1,467,094	1,531,062
fullRegistersEvents	42,741,116	25,093,946	45,993,720	26,929,854
rename.squashCycles	120,307	133,020	115,878	128,115
rename.idleCycles	41,238,533	22,445,194	36,376,003	21,129,170
rename.blockCycles	134,757,980	79,619,833	109,637,559	56,014,107
rename.runCycles	321,860,525	188,165,779	149,337,153	92,082,463
rename.unblockCycles	65,829,029	40,377,963	61,747,712	38,698,786

Table 5.18: GROMACS Gem5 rename stalls

Table 5.19 shows the number of *fuBusy* events for major operation classes. We see that all dispatched SVE instructions exhibit the problem of unavailable functional units. (Instructions can not be issued immediately when all operands are available.) This is most evident for the *fmul* instruction. We conclude that insufficient execution units are the main bottleneck of the nonbonded kernel.

	128-4xm	256-4xm	256-2xmm	512-2xmm
statFuBusy::SimdAlu	13.24%	14.37%	14.77%	12.96%
statFuBusy::SimdMisc	14.86%	14.28%	18.41%	21.67%
statFuBusy::SimdFloatAdd	9.23%	10.71%	15.14%	14.31%
statFuBusy::SimdFloatAlu	8.74%	8.72%	6.82%	9.63%
statFuBusy::SimdFloatCmp	10.69%	7.97%	3.42%	5.75%
statFuBusy::SimdFloatMult	27.07%	28.16%	25.87%	23.05%
statFuBusy::SimdFloatMultAcc	10.60%	9.15%	7.34%	7.57%
statFuBusy::SimdFloatSqrt	0.65%	0.92%	1.62%	0.98%
statFuBusy::SimdFloatReduceAdd	0.28%	0.53%	1.34%	2.31%
statFuBusy::SimdTotal	95.37%	94.82%	94.74%	98.25%

Table 5.19: GROMACS Gem5 FuBusy events

We also look at the cache behavior in the Gem5 model. Major Gem5 cache counters are shown in Table 5.20. Looking at the LSQ, we observe a small load-to-use latency (4.31 cycles) which raises to 4.95 for the 512-bit 2xMM execution. Such small latency means that most memory requests hit the L1 cache. This is confirmed by the number of cache accesses, where more than 99% of accesses are in the L1. The SVE size and kernel selection (4xM or 2xMM) also significantly impact the number of cache line refills. For 256-bit SVE, we notice an increased number of L1 and L2 cache replacements for the 2xMM kernel (42% more in L1 and 53% more in L2). When looking at individual kernels

(4xM and 2xMM), longer SVE vectors reduce cache refills for roughly 25-35%. This is expected because the kernel can compute more particle forces with less data transfer. The number of L3 cache replacements is low, because the entire data fits in the L3 cache.

	128-4xm	256-4xm	256-2xmm	512-2xmm
dcache.replacements	1,275,216	883,630	1,250,481	934,839
dcache.overallAccesses::total	298,344,195	173,975,370	123,444,661	122,708,715
l2cache.replacements	462,490	301,893	463,053	302,241
l2cache.overallAccesses::total	1,275,227	883,051	1,250,491	934,795
lsq0.loadToUse::mean	4.31	4.31	4.32	4.95
lsq0.loadToUse::stdev	1.31	1.25	1.44	1.59

TABLE 5.20: GROMACS Gem5 cache counters

Gem5 execution latency configuration

Next, we try to understand the poor performance of the A64FX processor. Since GROMACS scales well with the SVE size, it is surprising that the performance of the A64FX is similar to Graviton 2, which features four times shorter SIMD vectors. While inspecting the SIMD implementation of the kernel, we noticed that large parts of code contain chained SIMD operations. In these operations, the input operands depend on the result of previous operations. A clear example of this is in Listing 4.4 in Section 4.2.2. A chain of *fmul* operations is needed to calculate the high powers (r^6 and r^{12}) for the computation of the Lennard-Jones potential. In such kernels, performance is limited by the dependencies on the critical path [49], where the latency of instructions plays a crucial role. One major difference between the A64FX and Graviton 2 lies in the execution latencies caused by long execution pipelines in the A64FX. To confirm this bottleneck, we change the latencies of the FU pool in the Gem5 model to those of A64FX (see Table 2.4 on page 33 as an example). We repeat the simulation and observe how the performance changes. Results are shown in Table 5.21. We see that longer execution latencies have a huge effect on the result. The performance decreases by a factor of three, and B_{rate} rises from 0.30 to 1.03. Also, the number of *ROBFullEvents* and *fullRegisterEvents* increases significantly.

Instruction latencies	Neoverse N1 backend	A64FX backend
Useful *pairs/usec*	302	102
fuBusy	129,621,491	439,430,748
fuBusyRate	0.30	1.03
ROBFullEvents	1,531,062	2,184,763
fullRegistersEvents	26,929,854	40,481,520

Table 5.21: GROMACS 512-bit results (latency configuration)

Gem5 N_{SVE} configuration

Lastly, we analyze how the execution is impacted by the number of SVE units. We repeat the simulations with different combinations of SVE units and sizes. Table 5.22 shows the results for three different configurations.[1] The best result is observed for a single SVE unit with 512 bits (198 useful *pairs/usec*). For both nonbonded kernels, longer vectors perform better, despite the configurations having the same output of Flop/cycle. The worst performance is seen for four 128-bit SVE units (123 *pairs/usec*). As discussed in the previous paragraph, the performance of nonbonded kernels is dominated by the execution along the critical path. Such kernels do not benefit much from more SIMD units because multiple critical path executions can overlap on the same execution unit. However, a longer SIMD width benefits because the same path computes calculations over more lanes. Unfortunately, the current 4xM kernel is not optimized for architecture with four execution units. This would require a new kernel that would better utilize instruction-level parallelism. Alternatively, it could also be done with a new SIMD backend where a `SIMDReal` class would include two register fields, and two SVE instructions would be issued for each overloaded operation (see Listing 4.3 on page 80). Similarly to OpenBLAS, we see significantly fewer *fuBusy* events for the 4x128 configuration.

SVE units	4x128 (4xM)	2x256 (4xM)	2x256 (2xMM)	1x512 (2xMM)
Useful *pairs/usec*	123	189	175	198
fuBusy	24,654,081	110,168,356	52,403,711	215,371,783
fuBusyRate	0.07	0.54	0.30	1.99
ROBFullEvents	2,086,002	2,184,736	384,969	840,548
fullRegistersEvents	7,908,093	6,305,978	11,763,431	12,739,242

Table 5.22: GROMACS results (N_{SVE} configuration)

[1]Numbers for 256-bit SVE are different than in Table 5.17 because of a smaller input size.

5.3.2.2 Ribonuclease

Our second use case is a ribonuclease protein simulation (RNAse) in a water solvent. RNAse is an essential enzyme for a degradation of RNA in organisms. The whole system comprises around 24000 atoms.

Figure 5.12 shows the performance in Gem5 and selected hardware for one, two, and four cores. Performance of GROMACS is measured in the number of nanoseconds simulated in one day (ns/day). Overall, the SIMD speed-up is smaller than for the nonbonded benchmark. For single-threaded runs, doubling the SVE size from 128 to 256 and 256 to 512 increases the performance by 38% and 26%, respectively. (When running the 256-bit SVE, the application selects the 4xM nonbonded kernel.) This leads to a parallel efficiency of $\epsilon_{256} = 0.69$ and $\epsilon_{512} = 0.43$ compared to 128-bit SVE. 128-bit Gem5 model performs similarly to Graviton 2, showing that SVE does not bring any considerable performance changes compared to NEON if the vector length is the same. Performance is also increased when we increase the number of cores. We see a 67-80% speed-up when going from one to two cores and an additional 51-65% improvement when using four cores.

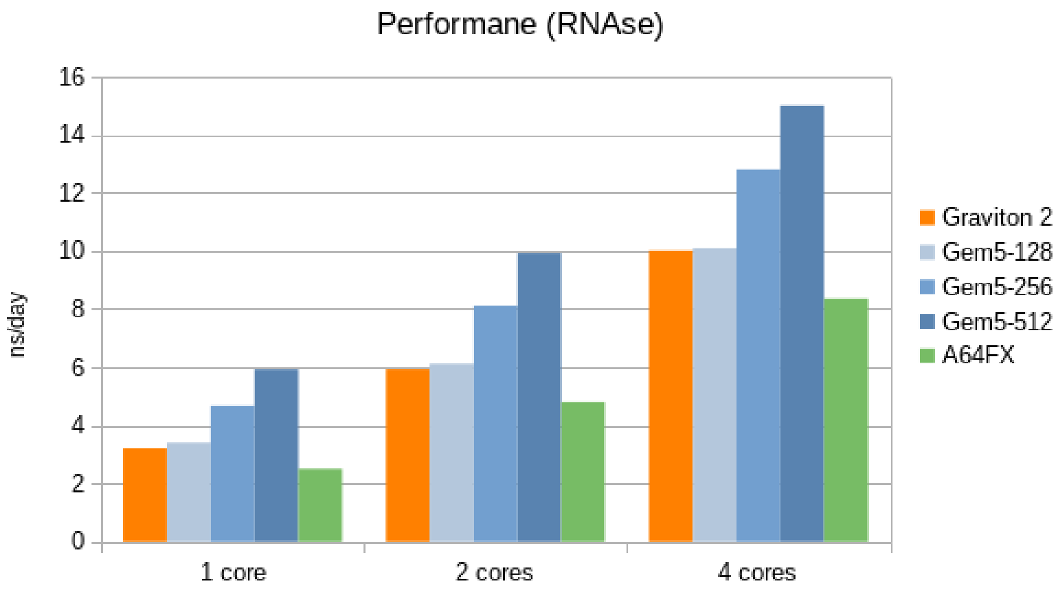


Figure 5.12: GROMACS results (ribonuclease)

The main reason for a lower vector speed-up compared to the nonbonded benchmark is a bigger portion of scalar code. Table 5.23 shows the number of scalar and SVE instructions measured with ArmIE. The proportion of SVE instructions ranges from 60.5% to 56.0% and 46.7% for SVE sizes of 128, 256, and 512 bits, respectively. Additionally, we also observe fewer scalar instructions when the SVE length is increased. The number of scalar instructions decreases by 37% and 26%.

	128	256	512
all	856,683,938	562,660,105	367,836,737
sve	517,873,934	315,340,191	171,939,465
scalar	338,810,004	247,319,914	195,897,272
% of sve	60.5%	56.0%	46.7%

Table 5.23: GROMCAS Ribonuclease instructions

Table 5.24 shows the main Gem5 counters. First, we compare the execution time (*numCycles*) with the nonbonded benchmark for 12000 atoms (see Table 5.17). The total simulation runtime is increased by a factor of 2.5, 3.1, and 3.9 for 128-, 256, and 512-bit SVE (RNAse workload simulates 24000 atoms). We also observe that the bigger SVE size reduces the stress on functional units. For example, when increasing SVE size, the percentage of instructions waiting for the available FU drops from 46% to 39%, then to 29%. ROB is less occupied for bigger SVE sizes, as shown by *ROBFullEvents*. Additionally, the number of *fullRegisterEvents* decreases by approximately 33% when increasing SVE from 128 to 256 bits (41% in the nonbonded benchmark).

SVE length	128	256	512
numCycles	1,413,702,279	1,024,219,224	809,014,973
fuBusy	1,720,625,757	1,059,068,295	580,634,082
fuBusyRate	0.46	0.39	0.29
iqFullEvents	1	2	4
lsqFullEvents	116,402	105,689	201,644
ROBFullEvents	36,733,836	29,642,558	9,149,616
fullRegistersEvents	52,680,514	35,554,755	48,441,066

Table 5.24: GROMCAS Ribonuclease Gem5 backend

In Figure 5.13, we show the single-threaded performance for a box of water molecules (without RNAse) of different sizes. Similar to earlier results, doubling the SVE size results in roughly 40% and 30% performance improvement. The overall runtime is inversely proportional to the number of atoms which means that the performance stays constant for all input sizes.

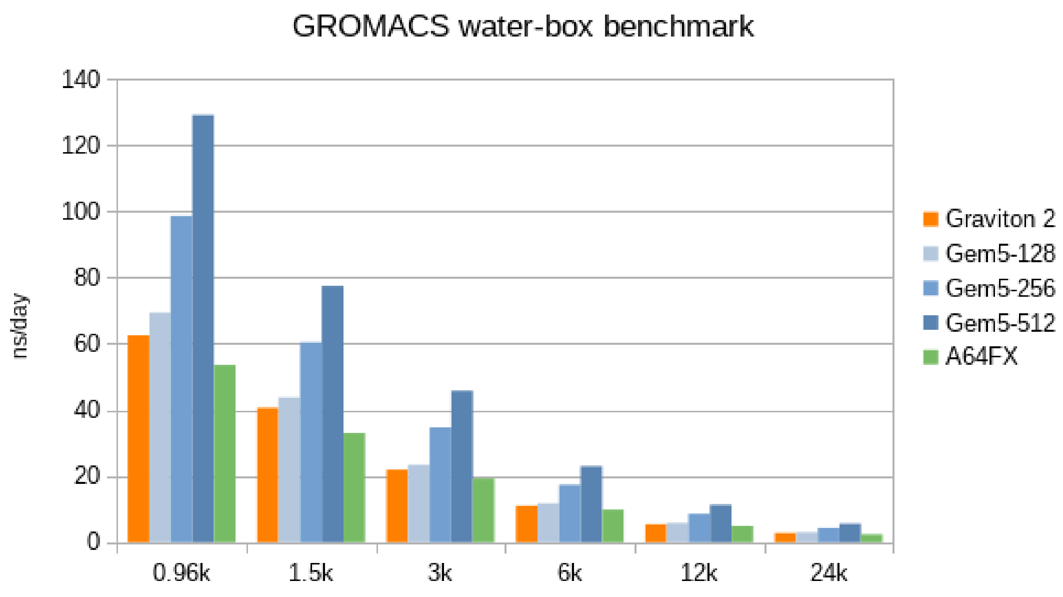


FIGURE 5.13: GROMACS results (water-box)

5.3.3 GPAW

In addition to studying the effect of different SVE lengths, GPAW also presents an opportunity to study different types of vectorization. For the Bmgs_fd function, we split the kernel into two cases, depending on which loop we vectorize (see Section 4.3.2.1). To stay concise, we introduce the names Bf-1 and Bf-2. Bf-1 refers to the vectorization of the innermost loop, while the Bf-2 applies vectorization of the outer loop (the loop that surrounds the innermost loop). Similarly, for the kernel Construct_density (see Section 4.3.2.2), we introduce the notations Cd-1, Cd-2, and Cd-3, where the number denotes the level of vectorization. For example, Cd-3 refers to the vectorization of the inner-three loops. We want to remind the reader that the code performs the same algorithm/computation in all cases. Different vectorizations only change how the calculation is mapped to scalar and SIMD instructions. In all cases, we have manually vectorized the loops with intrinsic functions and compiled the code with GCC version 11.1.0. Different vectorization cases are separated as different functions and chosen via a command line argument.

5.3.3.1 Function Bmgs_fd

The results for the Bmgs_fd function are shown in Figure 5.14. We obtain better performance for the outer loop vectorization for all SVE sizes. Bf-2 outperforms Bf-1 by 15%, 16%, and 37% for 128, 256, and 512-bit SVE, respectively. We also observe better performance for a larger SVE size for both cases. For the Bf-1, we have $t_{128} = 0.164s$ and $t_{256} = 0.102s$, giving $\eta_{256} = 1.61$. At $b_{SVE} = 512$ the time further reduces to $0.085s$ which leads to $\eta_{512} = 1.93$. A good speed-up is also observed for the outer loop vectorization. In this case, the runtime is improved from 0.14 (128 bits) to 0.054 (512 bits) seconds. The vector speed-up equals $\eta_{256} = 1.63$ $\eta_{256} = 2.59$ in this case. Because the outer loop can not be vectorized with NEON instructions, the Graviton 2 results are shown only for the Bf-1, where the time is 7% lower than in the 128-bit Gem5 model.

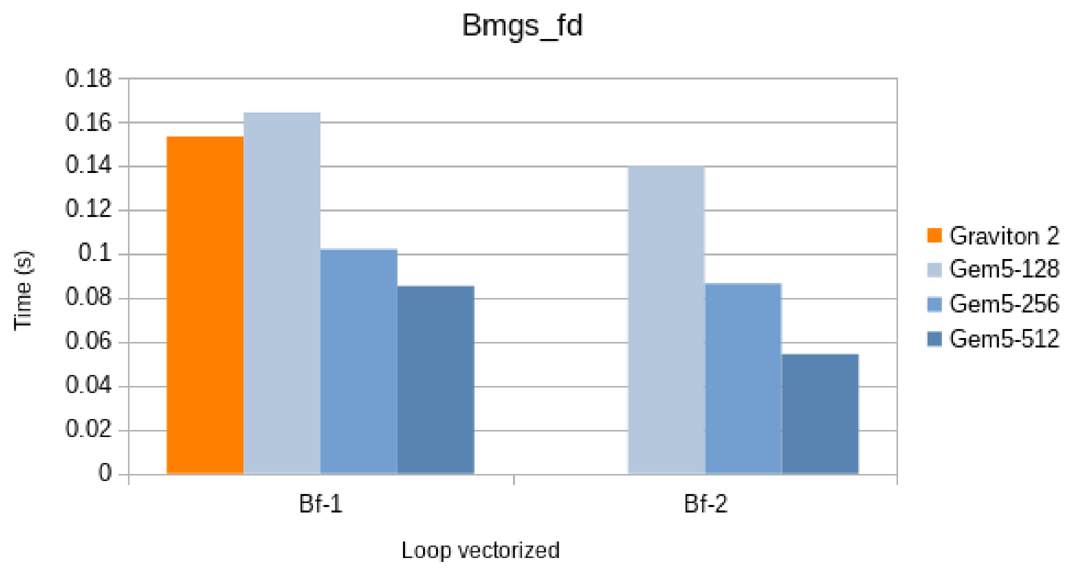


FIGURE 5.14: GPAW results (Bmgs_fd)

Table 5.25 shows the main Gem5 statistics for six cases. For both vectorizations, a bigger SVE size reduces the number of committed instructions. We see a significant discrepancy between committed instructions and micro-instructions for all cases. This is a result of gather-load instructions, which are decoded into l_{SVE} separate micro-instructions (one memory access per lane). For the Bf-1, the number of committed micro-instructions per cycle ranges between 2.40 and 2.49. For the outer loop vectorization (Bf-2), it is slightly higher, averaging at 2.82.

	Bf-1			Bf-2		
SVE size	128	256	512	128	256	512
numCycles	410,458,778	254,751,418	212,896,148	349,412,874	215,876,830	135,534,916
committedInsts	867,572,637	480,630,637	325,853,837	860,269,047	430,763,427	222,938,127
committedOps	1,022,412,294	635,470,294	511,648,854	1,015,046,224	585,540,604	382,708,104
insts/cycle	2.11	1.89	1.53	2.46	2.00	1.64
ops/cycle	2.49	2.49	2.40	2.91	2.71	2.82

TABLE 5.25: GPAW Bmgs_fd Gem5 counters

Despite the kernel in the Bmgs_fd appearing memory-bound (see arithmetic intensity in Section 4.3.2.1), we see that the SVE size greatly impacts performance. To inspect this further, we first analyze the cache behavior and the memory footprint. Table 5.26 shows the most important counters for the LSQ, cache and memory-controller behavior for a 512-bit SVE execution. 128-bit and 256-bit SVE produce a similar memory footprint with a smaller load-to-use latency and memory bandwidth. The main observation is a small load-to-use latency of 4.34 and 5.95 cycles for different vectorizations. This indicates that most cache accesses hit the L1 and L2 cache. This is confirmed when looking at the LSQ, which completes 98% of load requests in less than ten cycles. Most of the remaining requests are returned in less than twenty. The bottom half of the table shows different causes of memory traffic over the memory controller. The hardware prefetcher reads roughly 96-97% of data. This is likely due to the stencil operator's regular access pattern, which traverses grid points in order. Both types of vectorizations exhibit the same memory access pattern. The overall memory bandwidth for different SVE sizes ranges from 1.21 to 2.3 GByte/s for Bf-1 and from 1.42 to 3.67 GByte/s for Bf-2. This is far below what we measured with the STREAM benchmark.

Vectorization	Bf-1	Bf-2
lsq0.loadToUse::samples	232,291,491	191,974,601
lsq0.loadToUse::mean	4.34	5.95
lsq0.loadToUse::0-9	225,068,332	163,377,838
lsq0.loadToUse::10-19	7,117,072	28,404,291
dcache.replacements	8,146,005	8,145,213
l2cache.replacements	2,149,474	2,149,349
l3cache.replacements	2,138,443	2,138,389
mem_ctrls.bytesRead::total	136,871,296	136,867,904
mem_ctrls.bytesWritten::total	62,202,112	62,202,944
mem_ctrls.bwTotal::total (GByte/s)	2.34	3.67

Table 5.26: GPAW Bmgs_fd cache behaviour

Next, we estimate the arithmetic intensity of two kernels. We observe that the number of bytes read from memory is only 2.2 times bigger than the number of written bytes. (In the analysis in Section 4.3.2.1 we saw that the kernel loads 19 elements of the array `a` per one stored element of the array `b`.) To count the number of operations, we look at the number of committed instructions for each Gem5 operation class when $b_{SVE} = 512$. (See Table 5.27.) For analysis, we focus only on the SVE instructions (SIMD op. classes). We count the number of *add*, *fmul*, *fmla*[1], and *fadda* instructions multiplied by $l_{SVE} = 8$.

[1]In case of a fused multiply-add, we multiply the number of instructions by two.

This results in the following arithmetic intensity for both kernels:

$$AI_{\text{Bf-1}} = \frac{3 \times 23,216,520 \times 8}{136,871,296 + 62,202,112} \approx 2.8 \ \frac{op}{Byte} \tag{5.1}$$

$$AI_{\text{Bf-2}} = \frac{(21,032,170 + 2 \times 18,972,640) \times 8}{136,867,904 + 62,202,944} \approx 2.4 \ \frac{op}{Byte} \tag{5.2}$$

The reason for a higher arithmetic intensity of the Bf-1 kernel lies in the last iteration of the inner-loop where not all lanes of the SVE register are active. (In our analysis, we count operations for all lanes of the SVE register even though some of these are not active. For a 512-bit SVE, the last iteration of the inner-loop has only three active lanes.) When $b_{SVE} = 128$ and $b_{SVE} = 256$ the arithmetic intensity of the Bf-1 is 2.4 and 2.5, respectively. For the Bf-2 kernel, the arithmetic intensity stays the same for different SVE sizes since SVE instructions always operate on all lanes. (See Listing 4.7.) Considering a ridge point for different SVE-size models (see Section 5.2.1), we see that unlike the 512-bit model, the 128 and 256-bit models are compute bound.

OpClass (SVE inst.)	Bf-1	Bf-2
SimdAdd (add)	23,216,520	21,032,170
SimdAlu (mov)	30,955,370	21,157,790
SimdCmp (b.any)	23,278,930	1,060,970
SimdMisc (mov (from scalar))	7,738,840	62,410
SimdFloatMult (fmul)	23,216,520	0
SimdFloatMultAcc (fmla)	0	18,972,640
SimdFloatReduceAdd (fadda)	23,216,520	0
SimdPredAlu (whilelo)	30,955,360	10
MemRead	232,291,491	191,974,601
MemWrite	7,739,331	7,988,921

Table 5.27: GPAW Bmgs_fd committed micro-instructions

Next, we inspect the number of *FuBusy* events for different operation classes (see Table 5.28). We notice that the outer-loop vectorization (Bf-2) exhibits significantly less stress on the SVE units. In the case of Bf-1, roughly 42% of *FuBusy* events happen for SIMD operations and 58% due to *MemRead* operations. However, for Bf-2, almost all events occur exclusively for load and store instructions. (94% for *MemRead* and 3% for *MemWrite*.) A high number of *MemRead* operations results from gather-load instructions. For a 512-bit SVE, each gather-load instruction causes eight different memory read operations (due to double-precision). Seeing a large number of FuBusy events, we analyze the occupancy of SVE and load-store pipelines for both kernels.

	Bf-1	Bf-2
fuBusy	155,745,684	93,811,199
statFuBusy::IntAlu	1,628	1,123,562
statFuBusy::SimdAdd	9,251,765	808,851
statFuBusy::SimdAlu	12,807,543	1,075,235
statFuBusy::SimdCmp	10,586,539	2,713
statFuBusy::SimdMisc	17,729,352	558
statFuBusy::SimdFloatMult	14,650,685	0
statFuBusy::SimdFloatMultAcc	0	0
statFuBusy::SimdFloatReduceAdd	7,674	0
statFuBusy::SimdPredAlu	44,561	0
statFuBusy::MemRead	90,604,063	87,742,651
statFuBusy::MemWrite	66	2,995,764

TABLE 5.28: GPAW Bmgs_fd Gem5 *FuBusy* counters

Bf-1 backend analysis

For the Bf-1 kernel, we focus on the occupancy of the SVE units. As explained in Section 3.3.2, certain instructions are not pipelined and block the pipeline for the time of execution. In Bf-1, the only such instruction is a reduction instruction *fadda* (part of Op. class *SimdFloatReduceAdd*) which computes the sum of elements in the SVE register. In the Gem5 model, these instructions stall the SVE unit by lat_{fadda} cycles, depending on the SVE size. Combining Eq. 3.5 and 3.6 [1], we get a lower bound as

$$t_c \geq \frac{\mu_{SVE} + \mu_{SimdFloatReduceAdd} \times (lat_{fadda} - 1)}{2}. \tag{5.3}$$

Using this formula, we can calculate the minimum time for the Bf-1 kernel for a 512-bit SVE: $t_c \geq 147,069,170$. This means that the model executes at 69% of the peak performance (maximum throughput of SVE instructions). Similarly, we get 59% and 64% for 128 and 256 bits, respectively. Additionally, we observe that the Bf-1 kernel is experiencing too few physical registers, which introduces many stall cycles in the rename stage. Table 5.29 shows the number of *FullRegistersEvents* and rename statistics for different SVE sizes. We see that the rename stage operates normally only in 60-68% of the time.

[1] Term $lat_{fadda} - 1$ is due to *fadda* operations already being counted once in μ_{SVE}.

SVE size	128	256	512
rename.fullRegistersEvents	38,979,769	8,117,884	5,504,907
rename.idleCycles	59,202,811	27,545,870	24,500,289
rename.blockCycles	73,296,649	68,128,957	61,385,997
rename.runCycles	216,461,856	131,829,038	105,504,189
rename.unblockCycles	61,424,975	27,175,110	21,433,082

TABLE 5.29: GPAW Bmgs_fd Gem5 rename stage

Bf-2 backend analysis

Seeing that a high number of memory-read operations is the main bottleneck for Bf-2, we analyze the occupation of load-store pipelines. We consider the case when the throughput of memory operations limits the performance. In such a case, the lower limit for the number of cycles is the execution where both memory units execute one operation per cycle. From Eq. 3.7, we get

$$t_c \geq \frac{\mu_{MemRead} + \mu_{MemWrite}}{2}. \tag{5.4}$$

Using this formula for the 512-bit SVE Bf-2 gives $t_c \geq 99,981,761$. Therefore, the Gem5 model executes at 74% of the maximum throughput. For 128 and 256 bits, we get 44% and 54%, respectively. Another interesting observation of the Bf-2 kernel was the poor performance of the branch prediction mechanism. Table 5.30 shows the number of branch mispredicts and the number of executed instructions. We see that roughly one of every twenty branches is mispredicted. This might indicate that the misprediction occurs on the last iteration of the inner-most loop (19 iterations). For the Bf-1 kernel, the number of mispredicts was significantly smaller (less than 0.1%). However, we see that the misprediction resulted only in between 2 and 4 squashed instructions in the IEW stage.

SVE size	128	256	512
numBranches	81,322,845	40,693,937	21,034,846
branchMispredicts	3,932,853	1,998,143	1,061,988
decodedInsts	1,078,307,523	616,137,760	399,953,987
iew.dispatchedInsts	1,058,444,833	600,156,108	391,549,211
iew.writebackCount	1,030,794,701	589,976,783	385,382,610
committedOps	1,015,046,224	585,540,604	382,708,104

TABLE 5.30: GPAW Bf-2 branch prediction

5.3.3.2 Function Construct_density

The results for the Construct_density function are shown in Figure 5.15. We present results for three different loop vectorizations and SVE sizes. Additionally, due to a small number of loop iterations in the function, we also show results for a scalar binary (where we disabled vectorization). Unlike the Bmgs_fd kernel, a bigger SVE does not necessarily perform better. For the Cd-1, 256-bit SVE results in roughly 21% better performance than 128 or 512 bits. A similar effect is observed for the Cd-2, where the 256-bit case outperforms 128-bit and 512-bit by 18% and 9%. For the Cd-3, the biggest SVE size results in worse performance. Additionally, the Cd-1 and Cd-2 show similar performance, while the Cd-3 is slower. Also, the scalar code achieves the same performance, showing that SIMD vectorization does not bring any benefit for such short loops.

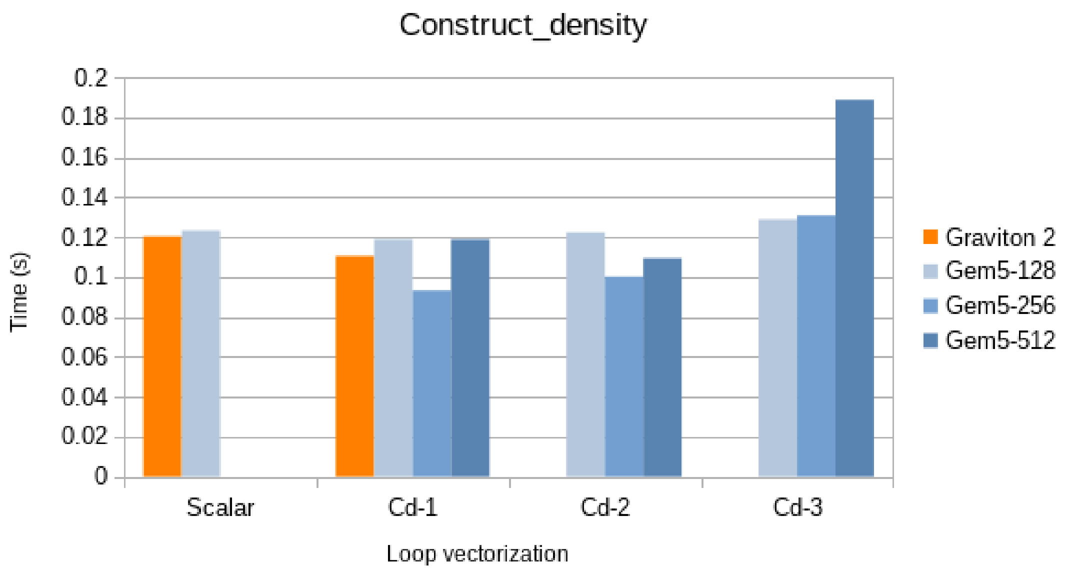


FIGURE 5.15: GPAW results (Construct_density)

As explained in Section 4.3.2.2, the Construct_density kernel is unsuitable for SIMD vectorization due to the low number of iterations in the inner loops. The number of iterations for the inner-most and the inner-second loop is either one or three (each occurs in 50% of the call paths). In the case of three scalar loop iterations, this leads to two SVE iterations for 128-bit and one SVE iteration for both 256 and 512-bit SVE. Therefore, increasing the SVE size to 512 bits does not bring any benefit over the 256 bits. We confirm this by inspecting the number of committed SIMD instructions for the Cd-1 kernel, which is shown in Table 5.31. The SVE execution profiles for 256 and 512-bit SVE are exactly the same (except for a few extra instructions in classes

IntAlu, SimdAlu and MemRead). Comparing the number of committed instructions to 128-bit SVE, most SVE instructions are reduced by one-third. We also notice that SVE instructions represent 16-19% of all instructions.

SVE size	128	256	512
Total	844,575,550	726,498,826	726,497,773
IntAlu	496,233,786	453,565,710	453,565,467
IntMult	21,872,148	21,872,148	21,872,148
FloatMisc	6,415,976	6,415,976	6,415,976
FloatAdd	984,994	984,994	984,994
FloatMisc	6,415,976	6,415,976	6,415,976
SimdAlu	5,881,108	5,880,460	5,880,136
SimdCmp	38,528,068	27,614,760	27,614,679
SimdMisc	1,976,484	1,976,484	1,976,484
SimdFloatMult	62,746,043	41,412,248	41,412,248
SimdFloatReduceAdd	32,111,767	21,198,621	21,198,621
SimdPredAlu	21,198,621	21,198,621	21,198,621
MemRead	119,955,342	98,128,402	98,128,078
MemWrite	36,651,401	26,230,752	26,230,752

Table 5.31: GPAW Cd-1 instructions

Despite the same number of committed instructions, the 512-bit Cd-1 performs roughly 20% slower than the 256-bit model. This is strange because the 512-bit model is architecturally almost the same as the 256-bit and all loops are executed with the same number of iterations. The only part, where the 512-bit Gem5 model differs from the 256-bit one is in the configuration of the *fadda* instruction latency. As described in Section 3.3.2, we set the latencies of *fadda* instructions to reflect the SVE size (similar to the A64FX). Despite only one or three lanes being active, the *fadda* latency is two times bigger for a 512-bit SVE than 256-bit. Table 5.32 shows a number of *FuBusy* events per operation class. We observe a big relative difference for the SimdFloatReduceAdd class which counts the *fadda* instructions. Since this is the only difference between architectures that impacts execution, we conclude that it is the main factor for a 20% slower performance.

SVE length	128	256	512
statFuBusy::SimdAlu	134,300	599,188	414,347
statFuBusy::SimdCmp	3,066,991	3,015,519	2,657,105
statFuBusy::SimdMisc	42,891	122,111	305,055
statFuBusy::SimdFloatMult	64,066	542,588	633,467
statFuBusy::SimdFloatReduceAdd	196	77,409	606,056
statFuBusy::SimdPredAlu	1,629,198	531,122	529,601
statFuBusy::MemRead	11,136,311	11,284,925	9,255,450
statFuBusy::MemWrite	512,470	660,245	714,173

Table 5.32: GPAW Cd-1 *FuBusy* events

We now focus on the effect of different vectorizations for the 256-bit SVE. Table 5.33 shows the most relevant Gem5 counters for the out-of-order execution. Comparing the number of committed instructions, the Cd-2 and Cd-3 execute 10% fewer instructions than Cd-1. However, the number of micro-instructions does not reflect that. For example, for Cd-3, the number of micro-instructions is roughly 30% higher than for the other two cases. Cd-2 vectorization also causes less pressure on the ROB.

256-bit	Cd-1	Cd-2	Cd-3
numCycles	233,285,031	250,442,233	327,416,245
committedInsts	692,896,276	629,276,794	632,562,328
committedOps	696,144,592	712,847,979	1,006,042,148
fuBusy	36,812,713	84,790,051	182,810,014
fuBusyRate	0.5	0.11	0.18
iqFullEvents	0	0	0
lsqFullEvents	0	0	0
ROBFullEvents	6,013,365	622,587	11,218,974
fullRegistersEvents	16,672	16,090	286,467

Table 5.33: GPAW Construct_density backend counters

Table 5.34 shows the number of committed instructions by operation class, reported by Gem5. The first thing we notice is a big increase of *SimdAlu* and *MemRead*/*MemWrite* operations for Cd-3. Integer arithmetic operations (*SimdAlu*) are used to create index vectors for properly loading data in the SVE vector. In the case of Cd-3 vectorization, we have three gather-load instructions in each iteration. Each gather-load requires a different index vector assembled with SIMD integer instructions. This is a negative consequence when vectorizing outer levels of nested loops. Like the Bf-1 kernel in the previous section, many memory operations occur due to gather-loads and scatter-stores. However, here, we noticed that memory operations are issued for each lane of the SVE register, regardless of whether the lane is *active* or *inactive*. This means that Gem5 issues eight operations for 512-bit SVE, despite only three of those required to fill the SVE register. The total number of memory operations is 2.6 times higher for Cd-3 than for Cd-2. Applying the same analysis as for the Bf-1 kernel, we calculate that the number of committed memory operations per cycle for Cd-3 is 0.92, 1.38, and 1.74 for increasing SVE size. This means that the load-store pipelines are executing at 46, 69, and 87 % of the maximum throughput for 128, 256, and 512 bits, respectively. For the 512-bit model, this is most likely the main performance bottleneck. Different type of vectorization also impacts the floating-point operations. For Cd-1, the code relies on *fmul* and reduction operation *fadda*. For Cd-2 and Cd-3, *fadda* is replaced by *fmla* but at the expense of more gather-load operations.

	Cd-2			Cd-3		
	128	256	512	128	256	512
total	865,031,294	712,847,979	794,242,130	950,649,443	1,024,068,776	1,385,305,193
IntAlu	476,177,604	383,134,734	383,134,491	344,187,693	306,982,778	302,381,452
SimdAlu	52,832,423	37,442,203	37,441,879	177,537,007	152,131,074	148,897,335
SimdMult	15,177,117	10,140,607	10,140,607	0	0	0
SimdMultAcc	0	0	0	42,969,764	36,367,376	35,528,030
SimdFloatMult	32,179,354	21,333,795	21,333,795	30,266,792	25,346,596	24,782,717
SimdFloatMultAcc	30,701,863	20,348,801	20,348,801	28,789,301	24,361,602	23,797,723
SimdFloatReduceAdd	1,477,491	984,994	984,994	1,477,491	984,994	984,994
MemRead	160,428,598	159,008,352	240,403,232	232,833,468	347,023,545	625,265,470
MemWrite	19,617,520	14,581,010	14,581,010	65,004,231	104,872,037	197,807,413

Table 5.34: GPAW Cd-2 and Cd-3 committed instructions

5.3.3.3 A64FX behavior

To confirm results in the simulator, we run the Bmgs_fd and Construct_density benchmarks on the A64FX processor. We modified the code and added the option to set the SVE length of the current process through the `prctl` function. This enables us to run the benchmark for different SVE sizes.

Figure 5.16 shows the results for the Bmgs_fd function. Although the overall runtime is significantly slower, we see a similar behavior as the Gem5 model. Overall, the Bf-2 obtains much better performance for all SVE sizes. In the 512-bit SVE case, the Bf-2 is roughly seven times faster than the Bf-1. We also observe a better vector speed-up when vectorizing the outer loop. In the Bf-1, the runtime reduces by 10% (7%) when increasing the SVE size to 256 (512) bits. For the Bf-2, the performance rises by 97% and

91%, respectively. We see that reduction operation (*fadda*) which is decomposed into l_{SVE} sequential micro-instructions presents a significant bottleneck. When we remove it with the outer loop vectorization, it results in a much better performance. Figure 5.17 shows results for the Construct_density function. Again, we see a substantially worse performance for the Cd-3 kernel for all SVE sizes. On average, Cd-3 is 83% slower than Cd-2, which obtains the best performance. When vectorizing the inner-most loop, the 512-bit SVE is 55% slower than the 256-bit.

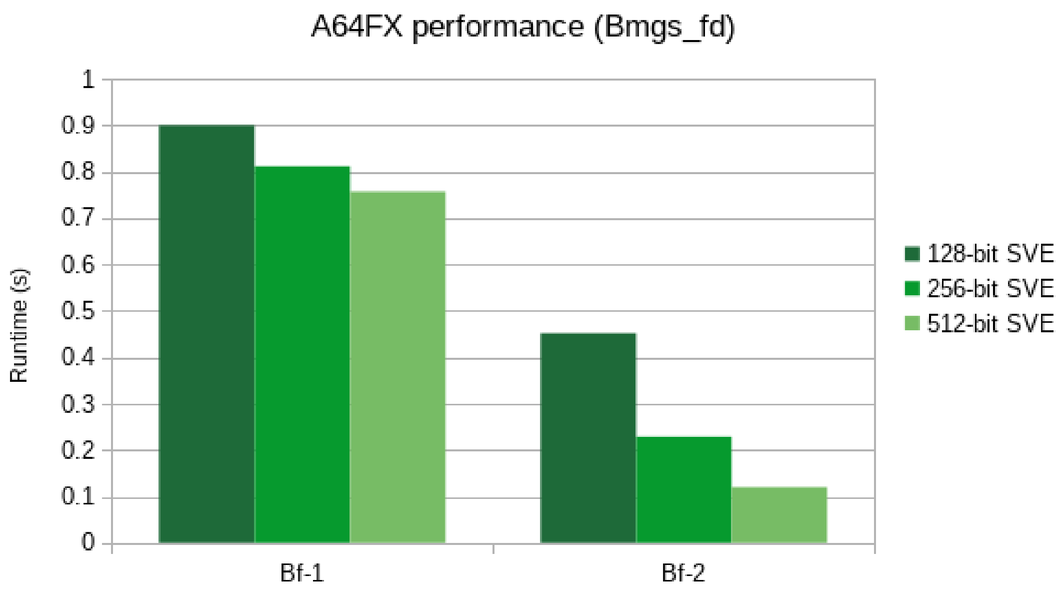


FIGURE 5.16: GPAW A64FX results (Bmgs_fd)

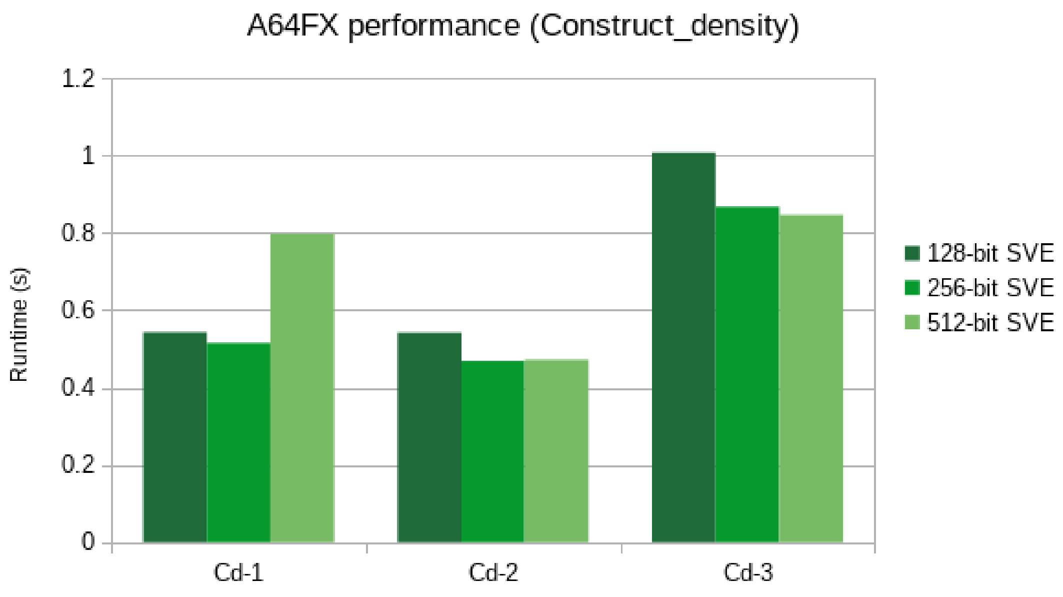


FIGURE 5.17: GPAW A64FX results (Construct_density)

Lastly, we discuss much lower performance than the Gem5 and Graviton 2 results. When the SVE size is 512 bits, the Bf-1 performs roughly eight times slower. Bf-2 performs two times slower. To inspect this further, we collect a set of hardware counters using PAPI. Results are shown in Table 5.35. First, we report the number of L2 cache misses. For all cases, it is approximately four times less than what we observe in the Gem5 model. This shows a similar behavior of both caches, considering the cache line size of the A64FX is four times bigger (256 bits). We also look at the number of backend and frontend stalls. In all cases, the number of backend stalls is very high. For the Bmgs_fd kernel, backend stalls occur in 67% and 59% of all cycles. On top of backend stalls, the Construct_density function also incurs many frontend stalls. Altogether, stalls represent 60%, 64%, and 75% of all cycles for Cd-1, Cd-2, and Cd-3, respectively.

	Bf-1	Bf-2	Cd-1	Cd-2	Cd-3
real_time_nsec	764,594,308	121,721,527	792,873,344	470,573,373	828,450,671
PAPI_TOT_CYC	1,374,678,925	218,277,258	1,425,663,234	846,176,047	1,489,930,293
L2_MISS_COUNT	557,340	568,341	82,601	80,565	77,807
STALL_BACKEND	922,423,594	129,433,773	805,378,239	312,278,975	784,304,334
STALL_FRONTEND	214,464	4,878,853	56,041,044	226,326,689	329,422,066

Table 5.35: GPAW A64FX PAPI counters

5.3.4 MiniFE

5.3.4.1 SpMVM

This section focuses on the SpMVM kernel in the Conjugate Gradient method in MiniFE. First, we evaluate the SELL matrix format for various input (matrix) sizes. Figure 5.18 shows the time spent in matrix-vector multiplication (t_{MatVec}), which we read from the YAML output file. The SpMVM time increases with the bigger input size, which is expected. Looking at a single size, the performance in the Gem5 simulator depends on the SVE size. We observe a 30% performance increase when doubling the SVE length from 128 to 256 bits. Increasing it further to 512 bits improves performance by an additional 20%. Both real-world systems perform better than their Gem5 counterparts. On Graviton 2, the SpMVM time is roughly 40% lower than a 128-bit SVE model in Gem5. The A64FX shows a 17% lower time than the 512-bit Gem5 model.

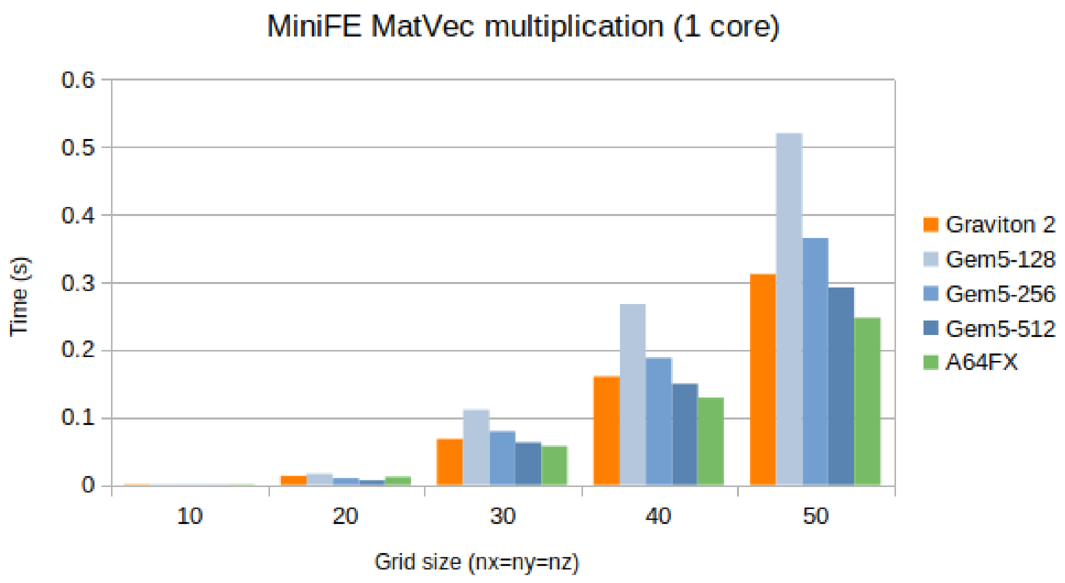


FIGURE 5.18: MiniFE results (1 thread)

The CG method (and SpMVM) is a heavily memory-bound kernel. Therefore, to better understand the performance, we first estimate the memory bandwidth during SpMVM. The memory footprint of the matrix A is $m_A = N_{nz} \times (8+4)$ Byte, where the first term in the bracket corresponds to the values of non-zero entries and the second term their column indices (value are stored as `double` and column indices as `int`). We neglect the array of slice offsets. Similarly, we calculate the memory footprint for vector x as $m_x = N_{row} \times 8$. Afterward, we can calculate the attained bandwidth as

$$\frac{n_{iters} \times (m_A + m_x)}{t_{MatVec}}. \tag{5.5}$$

Using this formula, the memory bandwidth of the SpMVM for input $n_x = 50$ is 8.2, 11.8, and 14.8 GByte/s for SVE of 128, 256, and 512 bits, respectively. This is slightly less than what we observe using the STREAM benchmark. Similar numbers are also reported by Gem5. Table 5.36 shows different statistics of data over the memory controller, reported by Gem5. The total reported read bandwidth for 128-bit SVE is 8.5 GByte/s. For 256-bit and 512-bit SVE, the bandwidth rises to 12.1 GByte/s and 15.1 GByte/s, which coincides with our calculation. (Slightly higher bandwidth over memory controller is due to unused hardware prefetches.) The majority of data (95.5%) is read by the hardware prefetcher.

SVE length	128	256	512
mem_ctrls.bytesRead::cpu.data	1,527,936	1,655,360	1,911,168
mem_ctrls.bytesRead::cpu.prefetcher	42,084,160	42,092,992	41,794,752
mem_ctrls.bytesWritten::total	4,222,592	4,222,848	4,222,656
mem_ctrls.bwRead::cpu.data	296,699,253	458,052,663	661,174,408
mem_ctrls.bwRead::cpu.prefetcher	8,172,030,015	11,647,500,893	14,459,022,137
mem_ctrls.bwRead::total	8,468,741,696	12,105,588,975	15,120,218,687
mem_ctrls.bwWrite::total	819,955,740	1,168,499,161	1,460,840,743
mem_ctrls.bwTotal::total	9,288,697,436	13,274,088,137	16,581,059,430

Table 5.36: MiniFE SpMVM memory behavior

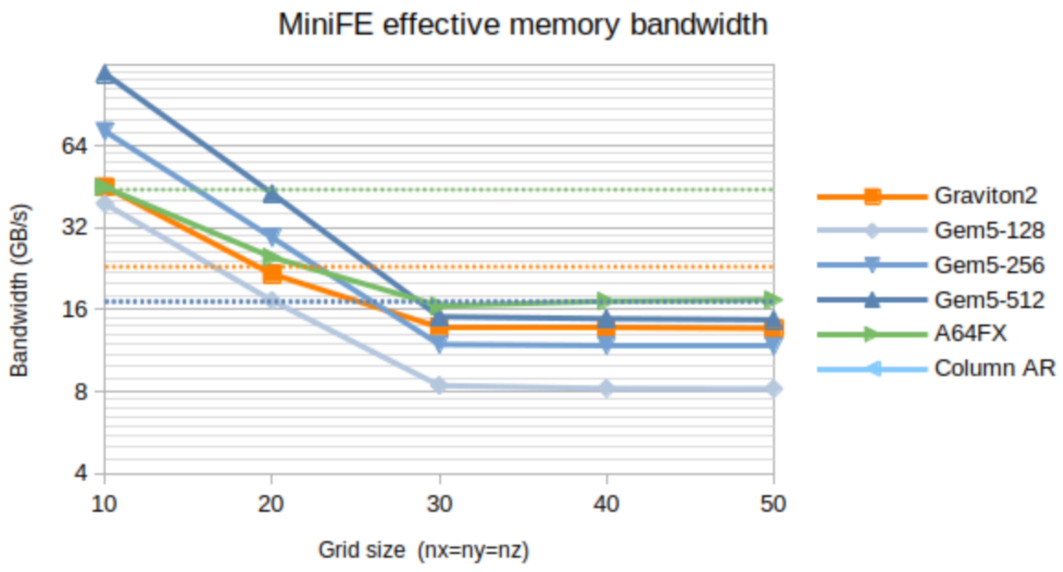


Figure 5.19: MiniFE effective bandwidth

Figure 5.19 shows the MiniFE effective bandwidth for the different grid sizes. (Dashed lines represent the STREAM triad results.) We see that when the size of the grid is 10 or 20, the data fits in the cache which results in higher bandwidth. For larger workloads, the attained bandwidth is 49, 70, and 88% of measured STREAM triad. Table 5.37 shows the main Gem5 counters during the SpMVM kernel. The majority of stalls in the backend are caused by the memory reads. Additionally, we see a different behavior of the ROB and LSQ for different SVE sizes. A big number of *lsqFullEvents* is the main reason for not reaching a better memory bandwidth for a 512-bit case. These occur due to individual memory reads of gather-load instructions (line 9 in Listing 4.11) that require many entries in the LSQ. When reducing the SVE size to 256 bits, the bottleneck moves to the ROB.

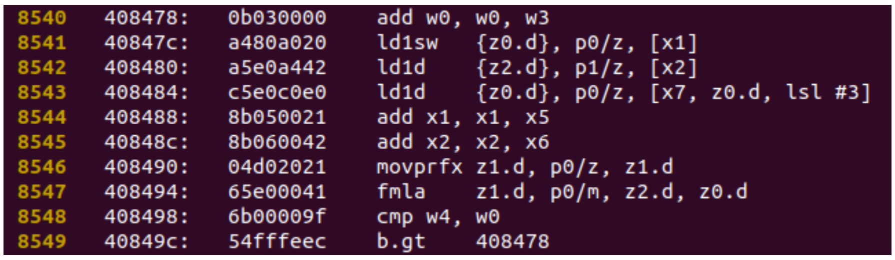
SVE length	128	256	512
numCycles	12,874,451	9,034,769	7,226,414
committedInsts	18,591,352	9,353,515	4,679,536
committedOps	22,057,638	12,842,771	8,170,676
ipc	1.44	1.04	0.65
fuBusy	1,101,967	1,633,498	1,825,322
statFuBusy::MemRead	1,038,119	1,615,610	1,813,869
fuBusyRate	0.05	0.13	0.22
iew.iqFullEvents	1	0	0
iew.lsqFullEvents	0	0	1,007,675
rename.ROBFullEvents	840,676	407,980	112,163
rename.SQFullEvents	1,794	1,794	1,794
rename.fullRegistersEvents	367	631	368

TABLE 5.37: MiniFE SpMVM Gem5 counters

We try to understand LSQ/ROB behavior by applying the analysis described in Section 3.5.2. Figure 5.20 shows the assembly code of the SpMVM kernel for SELL format which we retrieved using the *objdump* command. The inner-most loop of the kernel is composed of ten instructions (lines 8540 - 8549), three of which are memory loads. However, the load in line 8543 is a gather-load which is decoded in l_{SVE} micro-instructions. Therefore, the ratio of load operations μ_{ld} to other operations is

$$\frac{\mu_{ld}}{\mu - \mu_{ld}} = \frac{2 + l_{SVE}}{7}. \tag{5.6}$$

This equals 0.57, 0.86, and 1.43 for 128, 256, and 512-bit SVE, respectively. In the last case, the number becomes larger than the critical factor $f_{LSQ} = 1.29$ (see Eq. 3.8), so the bottleneck shifts from the ROB to the LSQ. Note that depending on how fast load-requests are served, the ROB can still become full. (Gem5 also reports *ROBFullEvents* for 512-bit SVE.)



```
8540  408478:   0b030000    add w0, w0, w3
8541  40847c:   a480a020    ld1sw   {z0.d}, p0/z, [x1]
8542  408480:   a5e0a442    ld1d    {z2.d}, p1/z, [x2]
8543  408484:   c5e0c0e0    ld1d    {z0.d}, p0/z, [x7, z0.d, lsl #3]
8544  408488:   8b050021    add x1, x1, x5
8545  40848c:   8b060042    add x2, x2, x6
8546  408490:   04d02021    movprfx z1.d, p0/z, z1.d
8547  408494:   65e00041    fmla    z1.d, p0/m, z2.d, z0.d
8548  408498:   6b00009f    cmp w4, w0
8549  40849c:   54fffeec    b.gt    408478
```

FIGURE 5.20: SpMVM assembly kernel

Figure 5.21 shows the effective memory bandwidth when running MiniFE across all cores. In this case, MiniFE almost fully saturates the memory bandwidth for all three systems. The result for the Gem5 model at $n_x = 50$ is 21.4 GByte/s for all SVE sizes. For very small input sizes, we observe a significant caching effect. Compared to our Gem5 model, a higher bandwidth (lower SpMVM time) is observed for Graviton 2 (155.5 GByte/s) and the A64FX (593 GByte/s). The STREAM bandwidth for triad kernel reaches 640 GByte/s on the A64FX.[1] Here, we see a benefit of the A64FX's HBM memory which is even more apparent for a higher number of cores.

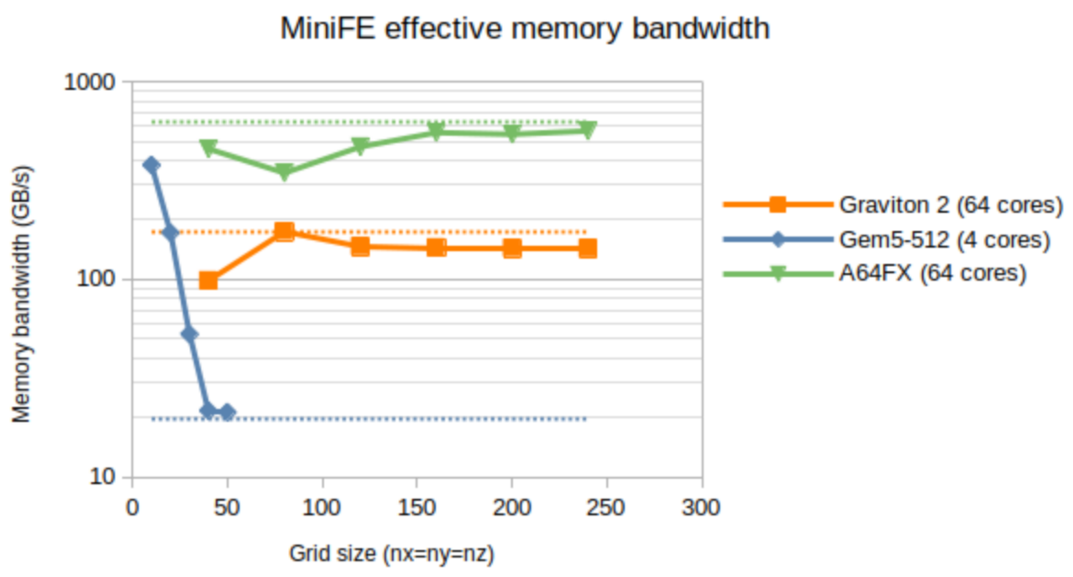


FIGURE 5.21: MiniFE effective bandwidth (full node)

Finally we compare performance for different sparse matrix formats. Here, we separate SELL and two cases of CSR format. In CSR SpMVM (Listing 4.9 in Section 4.4.2), the accumulation of products `val[j] * x[col[j]]` to `y[i]` can be done in two different ways. The first is to use an in-order reduction operation in each SVE iteration. This is the code that compilers generate when using `-O3` optimization. We refer to it as CSR-normal. The second is to use modulo unrolling with only one tree-order reduction at the end of the inner loop. Such vectorization is generated under `-ffast-math`, and we refer to it as CSR-fast. We give a detailed explanation in Appendix A.3. Table 5.38 shows the matrix-vector multiplication time for the case when the grid size $nx = 50$.

[1]Note that the A64FX processor can reach even higher bandwidth with special optimizations, for example the *dc zva* instruction [50].

	1 core		4 cores	
	Gem5 512-bit	A64FX	Gem5 512-bit	A64FX
CSR-normal	0.39	1.17	0.19	0.30
CSR-fast	0.36	0.79	0.19	0.20
SELL	0.29	0.24	0.20	0.06

Table 5.38: MiniFE SpMVM results

The results in the Gem5 simulator show that tree-like reduction with modulo unrolling is approximately 8% faster. However, this would likely be higher for larger N_{nzr} (each inner loop has at most 27 iterations). Additionally, the SELL format performs 34% faster than CSR-fast. All matrix formats perform the same when using four cores due to fully saturated memory bandwidth. The A64FX shows even stronger results in favor of the SELL matrix format. In this case, the SpMVM time for the CSR-normal format takes $1.17s$, while the CSR-fast reduces the time to $0.79s$. However, the SELL format computes SpMVM in only $0.06s$.

Chapter 6

Summary & conclusions

In this thesis, we have analyzed Arm's novel SIMD architecture called Scalable Vector Extension. We looked at selected HPC applications, ported them to SVE, and examined how the performance changes with different SVE sizes. Furthermore, we have studied how the SVE size impacts bottlenecks in different microarchitectural components.

Although SIMD parallelization is well exploited in HPC applications, a vector-length-agnostic concept is not yet fully adopted. The most apparent aspect of SVE is the possibility of running the same binary across systems with different SVE sizes. However, supercomputers rarely use software distributed in binary form due to their unique designs and individually tuned compilations. Therefore, we think it is unlikely that the HPC community would leverage the possibility of a shared VLA binary across different systems. (This might be more applicable in the mobile market.) Furthermore, we observed that developers often work around the VLA concept and use SVE as a traditional fixed-size SIMD extension (for example, GROMACS). In our view, the more considerable benefit of a VLA ISA is that it forces developers not to make assumptions about the SIMD length. This decreases dependence on specific hardware and makes algorithms more portable. Another strong reason in VLA's favor comes from the point of computer architects, who can choose a suitable SIMD size within the scope of the same ISA.

After SVE was introduced in 2018, all major compilers quickly added support for SVE assembly and auto-vectorization. This thesis analyzed GCC, Arm Compiler for Linux, and Fujitsu compiler. When studying the compiler's generated code, we observed no cases where the VLA concept would hinder auto-vectorization. All three compilers can exploit all features of SVE which we encountered in selected HPC applications (gather-load, scatter-store, reduction, vector-compare). However, we also noticed one loop where compilers failed to spot a particular SVE instruction and, therefore, a vectorization opportunity. This should serve as an example that compilers may still miss complex SVE instructions. In the TSVC benchmark, compilers have successfully vectorized between 93 and 97 loops, with 115 loops being vectorized by at least one compiler. This is comparable to the state-of-the-art x86 SIMD extensions. A slight drawback of SVE was observed in loops with a decrementing loop counter. Here, compilers generated inefficient code due to a missing instruction which was added in SVE2. On top of that, we have found three cases where a minor modification of the ISA (adding a new instruction) could open even more vectorization opportunities. We remind SVE designers about the potential prefix-sum SIMD instruction, which appears naturally in many numerical algorithms [51]. Since the *fadda* instruction (ordered sum-reduction) already exists, an extension to a prefix-sum instruction does not seem far-fetched.

The first kernel which we analyzed was OpenBLAS's implementation of DGEMM. (Main results are shown in Table 6.1). Porting OpenBLAS functions to a new architecture is a difficult process that requires a lot of debugging efforts. Here, the vector-length-agnostic architecture is advantageous because the same assembly kernel works for all SVE sizes. This does not mean that one kernel will achieve good performance across all SVE devices. However, we see a significant benefit for developers because we can reuse complicated parts of existing kernels and only modify specific aspects (for example, microkernel dimension n_r, prefetching, or unrolling) to tune it for another processor. Seeing that the same kernel can perform well across various SVE sizes is very encouraging in this sense. In DGEMM, software and hardware prefetching are crucial for good performance (>90% of the peak). This is more evident for a bigger SVE size because of the higher floating point performance and the same cache/memory bandwidth.[1]

Although DGEMM is a compute-bound kernel, the execution on a full node of a modern server often uses a non-negligible portion of the available memory bandwidth. Let us analyze this for the Neoverse N1 CPU. The measured STREAM bandwidth (174 GByte/s) results in roughly 2.7 GByte/s per-core memory bandwidth. On the other hand, our Gem5 results show that the traffic over the memory controller for DGEMM was between 0.4 and 1.5 GByte/s. We see that the 512-bit SVE Neoverse N1 architecture would utilize around 55% of the available memory bandwidth when running DGEMM on a full node of such architecture. Considering other factors like higher clock frequency or more cores, it might be worthwhile to include the HBM technology for architectures with $b_{SVE} \geq 512$. Another aspect of DGEMM is the high throughput of executed instructions. In our simulations, we observed an *ipc* between 3.37 and 3.87. For such cases, the core's frontend must be able to feed the core with enough instructions not to introduce stalls in the backend. This is even more important for future architectures

[1]Due to an increased data storage of a wider SVE register, the same number of SVE load instructions loads more cache lines in the L1 cache. Therefore, the load-to-use latency of a core with wider SVE vectors is bigger.

with more than two SVE pipelines. For example, in the case of four SVE pipelines, the SVE DGEMM kernel reaches six committed instructions per cycle at maximum performance. Therefore, the computer architects should increase the decode and fetch width to at least seven instructions per cycle in this specific case.

Porting efforts	We wrote specialized SVE macrokernels for many BLAS3 routines (GEMM, TRMM, and others). Vector-length-agnostic algorithm naturally extends the algorithm for a fixed-width SIMD.
Performance & SIMD scaling	Performance of DGEMM is proportional to the SIMD size and we observed a very good scaling ($\eta_{256} = 0.98$ and $\eta_{512} = 0.86$). The vector-length-agnostic kernel is able to reach good performance for all tested SVE sizes (128, 256, 512 bits).
Prefetching	The majority of the load requests hit the L1 cache. Data is prefetched in software (with prefetch instructions) and by the hardware prefetcher.
Out-of-order buffers	When $l_{SVE} = 512$, the number of *FullRegistersEvents* increased. On average, such an event was resolved in approximately five to six cycles.
Pipeline depth	Latency of the *fmla* instruction impacts the correct microkernel size. In the case of very deep pipelines (for example, on the A64FX) new micro-kernels have to be written to reach a maximum throughput of *fmla* operations. (For architectures with a similar latency, we may, to some extent, reuse existing kernels.)
Number of SIMD units	DGEMM has a high ratio of executed instructions per cycle. We observed a slower performance for $N_{SIMD} = 4$ and $b_{SVE} = 128$. The main bottleneck was insufficient fetch and decode width.

Table 6.1: Summary of DGEMM results

The next selected application was GROMACS. (See Table 6.2 for a summary of results.) GROMACS fixes the SVE size at compile time and relies on fixed-size SVE intrinsic types. This approach minimizes the porting effort since the application does not need to be restructured to allow for *sizeless* types. Our results show that GROMACS is compute-bound and benefits significantly from a bigger SIMD size. The main observed bottleneck in the nonbonded benchmark was in preoccupied SVE units. (Instructions could not be issued immediately when all operands are available.) Additionally, we observed long chains of read-after-write dependencies in the assembly. Here, the latency of SIMD instructions plays a significant role since faster writeback of results can free up resources sooner. We confirmed this with results on the A64FX processor, which has very deep execution pipelines. (Performance was three times slower than in our Gem5 model.) This should remind computer architects that latency and throughput can be equally important when targeting different HPC applications. Ensuring low latency of the most common floating point instructions (*fmul*, *fadd*, *fmla*) can significantly improve performance in this particular case.

Porting efforts	GROMACS relies on an internal SIMD library to map important kernel operations to SIMD instructions. Due to *sizeless* SVE intrinsic types, SVE size is fixed at compile-time and fixed-size SVE types are used.
Performance & SIMD scaling	Algorithm is compute-bound and we observed a good SIMD speed-up ($\eta_{256} = 1.70$ and $\eta_{512} = 2.72$) for computation of nonbonded interactions when increasing the SVE size. For a full protein simulation, the speed-up was lower ($\eta_{256} = 1.38$ and $\eta_{512} = 1.74$).
Out-of-order buffers	For all SVE sizes, we observed many *ROBFullEvents* and *FullRegistersEvents*. This causes the rename stage to block until resources are available (30-40% of cycles in renaming are stalled). The 4xM kernel performs better than the 2xMM kernel when $b_{SVE} = 256$.
Pipeline depth	We observed worse performance for a model with longer pipelines (higher execution latency of SIMD instructions). This is confirmed when looking at the A64FX processor. Many parts of the nonbonded kernel are latency limited and involve many chained operations with read-after-write dependencies.
Number of SIMD units	The performance dropped significantly for a configuration of four 128-bit SVE pipelines. Current nonbonded kernels are not optimized for such a case (four SIMD units).

Table 6.2: Summary of GROMACS results

Busy backend resources broaden the out-of-order window (number of instructions in the ROB) and negatively impact other pipeline stages. For example, we noticed an increased number of events for fully occupied physical registers and ROB. On average, the penalty for such events was between 2.7 and 3.4 cycles for different SVE sizes. This is less compared to the DGEMM kernel, where the source of these events are load instructions. (Load requests often take more cycles to complete.) Here, we suspect that the application would benefit from a larger ROB size and/or more physical register. However, we did not run such simulations and leave this for future work. Finally, we analyzed how the performance changes with different configurations of N_{SVE} and b_{SVE}. We observed that the configuration of four 128-bit SVE units is significantly slower than the 2x256 or 1x512 case. The main reason for this result is that the current implementation of the Verlet algorithm is not optimized for issuing so many instructions per cycle. Therefore, we want to encourage developers to think about how GROMACS could utilize four SIMD pipelines better. New kernels must likely be written to allow for a wider instruction-level parallelism. Seeing that the number of *fuBusy* events is significantly lower for a configuration of four SVE units, we believe that the proper implementation could make a good use of such architecture.

Our third application, GPAW, depends on many external libraries, and the internal code relies on the automatic vectorization of compilers. To simplify the analysis, we extracted two GPAW internal functions for studying. The main results of the Bmgs_fd function (a stencil operator) are shown in Table 6.3. Simulations of the Bmgs_fd kernel exposed the drawbacks of SVE *sequential* instructions. We use this term to describe instructions that are <u>not</u> pipelined or are decoded into multiple micro-instructions. One example of such instruction is the *fadda* which computes the sum of all lanes in an SVE register. Such instruction blocks the SVE unit for the duration of execution. (On A64FX, it is decoded into l_{sve} sequential micro-instructions, producing a similar effect.) We want to point out that such instructions have a negative effect on performance, and the developers should try to restructure the code to avoid them. If this is not possible, a faster *faddv* instruction should be used, which has lower latency (but breaks the floating-point compliance). One way to work around these instructions is to vectorize outer loops, which usually results in gather-load and scatter-store memory operations.

Gather/scatter operations are frequently used to vectorize loops with sparse or irregular data patterns, yet their effect is often poorly understood. Since such instructions are decoded into l_{SVE} micro-instructions, a separate memory flow is issued for each lane.[1] This can exert high pressure on the load/store pipelines and the load/store queue. Indeed, we observed such a case in the outer-loop vectorization of the Bmgs_fd kernel, where the performance was limited by the throughput of load/store pipelines. We want to emphasize that in a vectorized loop with gather-load instructions, only a computational part of the loop is parallelized, whereas the memory part is still sequential. This means that SVE pipelines are often underutilized since a majority of dispatched micro-instructions are issued to load/store units. Such an effect is even more apparent when the SVE size is big. Both developers and hardware architects should well understand such problems. Developers should always strive to organize their data for sequential access and minimize irregular access patterns. Here, tools like Spatter [52] can be used to investigate the behavior of different gather/scatter patterns. On the other hand, computer architects should include a sufficient number of load/store pipelines to handle

[1] There may be some flexibility on how these are implemented in the microarchitecture.

many simultaneous memory requests. (For example, the Arm Neoverse V1 includes five load/store pipelines, three more than the N1.)

Kernel characteristics	The stencil operator exhibits a high data locality and most memory requests hit the L1 cache. The measured arithmetic intensity ranged between 2.4 and 2.8 Flop/Byte.
Outer-loop vectorization	Vectorizing the outer loop improves performance from 15% to 36% for different SVE sizes. The main benefit is the removal of the latency-heavy non-pipelined *fadda* instruction (sum reduction).
Performance & SIMD scaling	For both types of vectorizations, we observed higher performance for a bigger SVE size. When vectorizing the inner loop, the speed-up was $\eta_{256} = 1.61$ and $\eta_{512} = 1.93$, and for the outer loop $\eta_{256} = 1.63$ and $\eta_{512} = 2.59$.
Out-of-order execution	When vectorizing the inner loop, the main bottleneck is in insufficient backend resources (SVE pipelines). This is mostly caused by a blocked pipeline caused by the *fadda* instruction. When vectorizing the outer loop, the performance is limited by the throughout of load-store pipelines. The main reason for this are the gather-load instructions which are decoded into l_{SVE} micro-operations.

TABLE 6.3: Summary of Laplace stencil (Bmgs_fd) results

Porting efforts	We implemented a VLA Sliced Ellpack format for storing sparse matrices and the SpMVM kernel with SVE intrinsic functions. The performance is higher than the default CSR format.
Performance & SIMD scaling	Performance is limited by the memory bandwidth. For single-core simulations, longer SVE vectors achieve higher memory bandwidth which results in a better performance. In multi-core simulations, longer vectors bring no benefit.
Out-of-order execution	We observed *ROBFullEvents* for 128 and 256-bit SVE models. When increasing the SVE size to 512 bits, the bottleneck shifts to *lsqFullEvents*. This can be predicted by the buffer sizes and the assembly analysis.

TABLE 6.4: Summary of MiniFE results

The last application, MiniFE, was selected to investigate memory-bound kernels. (A summary of the results is presented in Table 6.4.) We mainly focused on the sparse matrix-vector multiplication, where we implemented the SELL matrix format. In the results, we observed a significant SIMD speed-up for single-core simulations, the same as in the STREAM benchmark. Architecturally speaking, the SIMD vector length and the memory bandwidth are separate entities. However, longer SIMD vectors will increase the performance when a single core cannot fully utilize the available bandwidth. This happens because the same number of in-flight memory requests can transfer more data.

Lastly, we discuss the opportunities for future work. First, we want to point out that this thesis focused only on the first generation of SVE. SVE was superseded by SVE2, which introduced new instructions. Therefore, our work on the TSVC benchmark could be extended to SVE2 and potential improvements to the first version. Also, comparing our findings thoroughly to other SIMD extensions (AVX-512) would be an interesting next step. When analyzing HPC applications, we mainly relied on the Gem5 simulator due to its flexibility and cycle-level details. In our simulations, we only varied the SVE size and the number of SVE units, which is far less than what Gem5 is capable of. Since the SVE size impacts the execution in other microarchitectural components, we could continue the analysis by varying the size of other buffers (ROB, LSQ, physical registers). This would deepen our analysis of potential benefits in the out-of-order execution. Although we tried to configure the model with realistic parameters, the Gem5 model itself has limitations. This raises the question of how well our results translate to real hardware. With more SVE hardware coming to the market (AWS Graviton 3, NVIDIA Grace, mobile implementations based on Cortex X2/X3), it would be worthwhile to confirm the findings of this thesis by evaluating a larger variety of processors. Finally, the study conducted in this thesis mainly focused on the application's performance and how the microarchitecture responds to different SVE sizes. However, configuring the SVE size in real hardware has many consequences we did not address. For example, increasing SVE size leads to more transistors, a larger die area, increased power consumption, and more heat dissipation. In other words, bigger SVE vectors do not come without a cost. Therefore, we should consider all these effects for a more comprehensive analysis of the SVE size. (This would require other specialized simulators.)

Overall, we showed that Gem5 is a powerful simulator that can generate a wide range of results. Our simulations show that a co-design analysis can provide valuable feedback for computer architects and software developers. We hope this thesis will reach both and that some of our observations will support important decisions in future HPC architectures.

Appendix A

SVE examples

A.1 DAXPY

Listing A.1 shows an SVE implementation of the DAXPY kernel. The DAXPY kernel computes $y = ax + y$, where x and y are vectors of size n, and a is a scalar. Before entering the main loop, we initialize a loop counter (stored in register `x4`) to 0 in line 3. Next, we construct a predicate register `p0` with the `whilelt` instruction. The `whilelt` instruction (see Figure 2.3) calculates the number of iterations until the end of the loop and activates the lanes of the predicate register accordingly. (The `x3` register holds the size of the array.) In addition, we load and replicate the value a in the register `z2`.

In the main loop, starting in line 6, we load the values $x[i]$ and $y[i]$ into registers `z0` and `z1`. Then, we use the `fmla` instruction to compute $ax + y$, overwriting the result in the `z1` register. In line 10, we use `st1d` to store values back to array y. Line 11 contains the instruction `incd`, which increments the loop counter in the register `x4` by the number of doublewords in the SVE vector. Afterward, we update the predicate register with the new loop counter value and branch back if the first lane of the predicate register is active (line 13). Note that the predicate-driven loop removes the need for loop tails. The last iteration is the same as the previous ones but with a different predicate register.

```
// x0 = x, x1 = y, x2 = a, x3 = n
.daxpy:
    mov x4, #0                        // initialize loop counter in x4 with 0
    whilelt p0.d, x4, x3              // create a predicate register p0
    ld1rd z2.d, p0/z, [x2]            // replicate value a in all lanes of z3
.loop:
    ld1d z0.d, p0/z, [x0, x4, lsl #3]   // load values of x[i]
    ld1d z1.d, p0/z, [x1, x4, lsl #3]   // load values of y[i]
    fmla z1.d, p0/m, z0.d, z2.d         // compute y = a x + y
    st1d z1.d, p0, [x1, x4, lsl #3]     // store results in y[i]
    incd x4                             // increment loop counter
    whilelt p0.d, x4, x3                // update predicate register
    b.first .loop                       // loop back
ret
```

Listing A.1: SVE DAXPY kernel

A.2 Array permutation

Gather-load and scatter-store instructions are used to load elements from non-contiguous locations in memory. These instructions enable the vectorization of loops with complex memory patterns. The vector that holds the indices of the memory locations is often called an index vector. In Listing A.2 left, we see a simple loop that permutes the elements of the array `b` and stores them in the array `a`. On the right, we see a corresponding SVE code of this loop. In each loop iteration, we first load elements `perm[i]` (indices) in the register `z0` (line 5). Afterward, the `z0` register is used as an input to a second `ld1` instruction which gathers elements `b[perm[i]]`, overriding register `z0`. Gather-load and scatter-store use the same operation code, but take an extra SVE input register for addressing mode.

```
1  for (int i = 0; i < n; i++) {
2      a[i] = b[perm[i]];
3  }
```

```
1      mov      x4, 0
2      cntd     x5
3      whilelo  p0.d, wzr, w3
4  .L3:
5      ld1d     z0.d, p0/z, [x2, x4, lsl 3]
6      ld1d     z0.d, p0/z, [x1, z0.d, lsl 3]
7      st1d     z0.d, p0, [x0, x4, lsl 3]
8      add      x4, x4, x5
9      whilelo  p0.d, w4, w3
10     b.any    .L3
```

Listing A.2: Array permutation in C (left) and SVE assembler (right)

A.3 Sum reduction

A reduction instruction operates between lanes in the same SVE register. As discussed in Section 2.1.2.3, we differentiate between in-order and tree-like reductions. We take a simple example of computing the sum of elements in the array to demonstrate this. Listing A.3 shows the code in C.

```c
// assume sum and a[i] are double
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

Listing A.3: Array sum

The floating-point addition is not commutative. Normally, compilers adhere to the floating-point standard, which preserves the order of additions. We can bypass this with the use of *-ffast-math* flag, which breaks the rule in favor of a better performance. As a result, GCC compilations resuts in two different codes depending on whether the *-ffast-math* flag is used. On the left side of Listing A.4, we see a standard approach that preserves the order of summation. In each loop iteration, the `fadda` instruction is used to compute a sum of l_{sve} elements in the SVE register in order (line 4). Each time, we add the result to the overall sum stored in the register `d0`. On the right side, we show the generated code when we compile with the *-ffast-math* flag. This results in the l_{sve}-way modulo variable expansion. In each iteration, we add the intermediate results to the SVE register `z0` using a normal `fadd` instruction (element-wise addition). After the loop, we use an additional tree-like reduction operation `faddv` to compute the final sum (line 8). On SVE hardware, the second approach is usually faster due to the shorter latency of the `fadd` compared to the `fadda` instruction.

```
1  .L3:
2      ld1d    z1.d, p0/z, [x0, x1, lsl 3]
3      add     x1, x1, x2
4      fadda   d0, p0, d0, z1.d
5      whilelo p0.d, w1, w3
6      b.any   .L3
```

```
1  .L3:
2      ld1d    z1.d, p0/z, [x0, x1, lsl 3]
3      add     x1, x1, x2
4      fadd    z0.d, p0/m, z0.d, z1.d
5      whilelo p0.d, w1, w3
6      b.any   .L3
7      ptrue   p0.b, all
8      faddv   d0, p0, z0.d
```

Listing A.4: Array sum in SVE

A.4 Complex arithmetics

Let us consider the case of an element-wise multiplication of two vectors (a and b) with complex numbers. Additionally, let us assume that the complex numbers are stored as an *Array of Structures* (real components in the even elements and imaginary components in the odd elements). The code is shown in Listing A.5. Here, we rely on complex numbers from the C standard library.

```c
void complex_multiply(double complex *a, double complex *b)
    for (int i = 0; i < n; i++) {
        a = a * b;
    }
}
```

Listing A.5: Complex multiplication in C

There are two approaches to vectorizing the above loop with SVE. The first case is shown in Listing A.6 (intrinsic code). (For readability, we only show the computation of a single loop iteration.) In this case, we use an `ld2` instruction which separates real and imaginary components into two vectors in an SVE tuple. Afterward, we use a combination of normal multiplications and additions to construct the result. The resulting SVE vector is written back to memory with a scatter-store `st2` instruction.

```c
svfloat64x2_t a_v = svld2(pg, &a[i]);
svfloat64x2_t b_v = svld2(pg, &b[i]);

svfloat64_t res_r, res_i;
res_r = svmul_x(pg, a_v.v0, b_v.v0);              // res.real  = a.real * b.real
res_i = svmul_x(pg, a_v.v1, b_v.v0);              // res.imag  = a.imag * b.real
res_r = svmls_x(pg, res_real_vec, a_v.v0, b_v.v1); // res.real -= a.imag * b.imag
res_i = svmla_x(pg, res_imag_vec, a_v.v0, b_v.v1); // res.imag += a.real * b.imag

svst2(pg, &a[i], {res_real_vec, res_imag_vec});
```

Listing A.6: SVE complex multiplication (1)

Listing A.7 shows an alternative approach using SVE instructions for complex arithmetics. The `fcmla` instruction assumes that the SVE register stores the real components in even lanes and imaginary components in odd lanes. The third input operand is first rotated by 0, 90, 180, or 270 degrees in polar representation.[1] Then, the third operand is multiplied with duplicated real (imaginary) components of the second operand when rotation is 0 or 180 (90 or 270) degrees. The result is destructively added to the first operand without intermediate rounding. Using the `fcmla` instructions, we do not need

[1]Hardware designers can implement rotation by a multiple of 90 degrees only with swapping and negating real and imaginary components.

to separate real and imaginary parts in two different SVE registers. Therefore, we can use normal `ld1` and `st1` instructions to load and store values (see lines 1, 2, and 8).

```
1  svfloat64_t a_v = svld1(pg, &a[i]);
2  svfloat64_t b_v = svld1(pg, &b[i]);
3
4  svfloat64_t r = svdup(0.0f);          // initialize res
5  r = svcmla_x(pg, r, a_v, b_v,  0);    // res +=  a.real * b.real + I * a.real * b.imag
6  r = svcmla_x(pg, r, a_v, b_v, 90);    // res += -a.imag * b.imag + I * a.imag * b.real
7
8  svst1(pg, &a[i], res_vec);
```

Listing A.7: SVE complex multiplication (2)

A.5 *strlen* function

Listing A.8 shows an SVE implementation of the *strlen* function, which returns the length of a string.[1] The main idea is that each iteration first uses the FFR load instruction to suppress possible segmentation faults (line 6). In line 7, we read the FFR register and create a predicate register `p1` that only holds valid memory requests before checking for null characters in line 8. Finally, we create a predicate register `p2` with the `brkbs` instruction which activates all lanes until the null character and count number of active lanes with the `incp` instruction (line 10).

```
1   strlen:
2       mov x1, x0
3       ptrue p0.b
4   mainloop:
5       setffr                        // initialize the ffr register
6       ldff1b z0.b, p0/z, [x1]       // speculative load
7       rdffr  p1.b, p0/z             // read the ffr register
8       cmpeq  p2.b, p1/z, z0.b, #0   // check for null characters
9       brkbs  p2.b, p1/z, p2.b       // activate lanes until first active
10      incp   x1, p2.b               // increment counter by the number of active lanes
11      b.last mainloop               // branch back if all lanes are valid
12      sub x0, x1, x0                // calculate final length
13      ret
```

Listing A.8: SVE *strlen* function

[1]This implementation can be further optimized by separating the last iteration where the null character is found. This removes the counting of individual elements in all but last iteration. (See `https://github.com/ARM-software/optimized-routines/blob/master/string/aarch64/strlen-sve.S`)

Appendix B

Gem5 configuration

B.1 Execution units

Listing B.1 shows the configuration of the execution units for the (Neoverse N1) Gem5 model.

```

# This class refers to FP/ASIMD 0/1
class O3_ARM_Neoverse_N1_FP(FUDesc):
    # copied from Neoverse V1 optimization guide,
    # latency taken for specific instruction is written in brackets
    opList = [
        OpDesc(opClass='SimdAdd', opLat=2), # ASIMD arithmetic basis (add & sub)
        OpDesc(opClass='SimdAddAcc', opLat=4), # ASIMD absolute diff accum (vaba)
        OpDesc(opClass='SimdAlu', opLat=2), # ASIMD logical (and)
        OpDesc(opClass='SimdCmp', opLat=2), # ASIMD compare (cmeq)
        OpDesc(opClass='SimdCvt', opLat=3), # ASIMD FP convert to 64b (scvtf)
        OpDesc(opClass='SimdMisc', opLat=2), # ASIMD move, immed (vmov)
        OpDesc(opClass='SimdMult',opLat=4), # ASIMD integer multiply (mul)
        OpDesc(opClass='SimdMultAcc',opLat=4), # ASIMD multiply accumulate, (mla)
        OpDesc(opClass='SimdShift',opLat=2), # ASIMD shift by immed, (shl)
        OpDesc(opClass='SimdShiftAcc', opLat=4), # ASIMD shift accumulate (vsra)
        OpDesc(opClass='SimdSqrt', opLat=9), # ASIMD reciprocal estimate (vrsqrte)
        OpDesc(opClass='SimdFloatAdd',opLat=2), # ASIMD FP arithmetic (vadd)
        OpDesc(opClass='SimdFloatAlu',opLat=2), # ASIMD FP absolute value (vabs)
        OpDesc(opClass='SimdFloatCmp', opLat=2), # ASIMD FP comapre (fcmgt)
        OpDesc(opClass='SimdFloatCvt', opLat=3), # Aarch64 FP convert (fvctas)
        OpDesc(opClass='SimdFloatDiv', opLat=11, pipelined=False), # ASIMD (fdiv)
        OpDesc(opClass='SimdFloatMisc', opLat=2), # ASIMD (vneg)
        OpDesc(opClass='SimdFloatMult', opLat=4), # ASIMD FP (vmul)
        OpDesc(opClass='SimdFloatMultAcc',opLat=4), # ASIMD FP (vmla)
        OpDesc(opClass='SimdFloatSqrt', opLat=12, pipelined=False), # ASIMD (vsqrt)
        OpDesc(opClass='SimdReduceAdd', opLat=10), # SVE reduction, arithmetic (saddv)
        OpDesc(opClass='SimdReduceAlu', opLat=12), # SVE reduction, logical (andv)
        OpDesc(opClass='SimdReduceCmp', opLat=9), # SVE reduction, arithmetic (smaxv)
        OpDesc(opClass='SimdFloatReduceAdd', opLat=8, pipelined=False), # SVE (fadda)
        OpDesc(opClass='SimdFloatReduceCmp', opLat=9), # SVE (fmaxv)
        OpDesc(opClass='FloatAdd', opLat=2), # Aarch64 FP arithmetic (fadd)
        OpDesc(opClass='FloatCmp', opLat=2), # Aarch64 FP compare (fccmpe)
        OpDesc(opClass='FloatCvt', opLat=3), # Aarch64 Fp convert (vcvt)
        OpDesc(opClass='FloatDiv', opLat=11, pipelined=False), # Aarch64 (vdiv)
        OpDesc(opClass='FloatSqrt', opLat=12, pipelined=False), # Aarch64 (fsqrt)
        OpDesc(opClass='FloatMultAcc', opLat=4), # Aarch64 Fp (vfma)
        OpDesc(opClass='FloatMisc', opLat=3), # Aarch64 miscelleaneaus
        OpDesc(opClass='FloatMult', opLat=3) ] # Aarch64 Fp multiply (fmul)
    count = 2

# This class refers to pipelines Branch0, Integer single Cycles 0,
# Integer single Cycle 1
class O3_ARM_Neoverse_N1_Simple_Int(FUDesc):
    opList = [ OpDesc(opClass='IntAlu', opLat=1) ]
    count = 3 # Aarch64 ALU (Unfortunately branches are put together with IntALU

# This class refers to pipelines integer single/multicycle 1
class O3_ARM_Neoverse_N1_Complex_Int(FUDesc):
    opList = [ OpDesc(opClass='IntAlu', opLat=1), # Aarch64 Int ALU
               OpDesc(opClass='IntMult', opLat=2), # Aarch64 Int mult
               OpDesc(opClass='IntDiv', opLat=9, pipelined=False), # Aarch64 divide
               OpDesc(opClass='IprAccess', opLat=1) ] # Aarch64 Prefetch
    count = 1 # 1 units

# This class refers to Load/Store0/1
class O3_ARM_Neoverse_N1_LoadStore(FUDesc):
    opList = [ OpDesc(opClass='MemRead'),
               OpDesc(opClass='FloatMemRead'),
               OpDesc(opClass='MemWrite'),
               OpDesc(opClass='FloatMemWrite') ]
    count = 2 #

# Extra class for predicate operations
class O3_ARM_Neoverse_N1_PredAlu(FUDesc):
    opList = [ OpDesc(opClass='SimdPredAlu')  ]
    count = 1


class O3_ARM_Neoverse_N1_FUP(FUPool):
    FUList = [O3_ARM_Neoverse_N1_Simple_Int(),
              O3_ARM_Neoverse_N1_Complex_Int(),
              O3_ARM_Neoverse_N1_LoadStore(),
              O3_ARM_Neoverse_N1_PredAlu(),
              O3_ARM_Neoverse_N1_FP()]
```

Listing B.1: Configuration of execution units and instruction latencies

B.2 Caches

Listing B.1 shows the configuration of the cache levels for the (Neoverse N1) Gem5 model.

```
class O3_ARM_Neoverse_N1_ICache(Cache):
    tag_latency = 1
    data_latency = 1
    response_latency = 1
    mshrs = 8
    tgts_per_mshr = 16
    size = '64kB' # (1)
    assoc = 4 # (1)
    writeback_clean = False
    prefetcher = StridePrefetcher(degree=1)

class O3_ARM_Neoverse_N1_DCache(Cache):
    tag_latency = 3
    data_latency = 3
    response_latency = 1
    tgts_per_mshr = 16
    writeback_clean = False
    size = '64kB' # (1)
    mshrs = 20 # (1)
    assoc = 4 # (1)
    #prefetcher = StridePrefetcher(degree=16, latency = 1)

class O3_ARM_Neoverse_N1_L2(Cache):
    tag_latency = 5
    data_latency = 5
    response_latency = 2
    mshrs = 46  # (1)
    tgts_per_mshr = 16
    clusivity = 'mostly_incl' # (1)
    assoc = 8 # (1)
    size = '1MB' # Graviton2
    writeback_clean= True

class O3_ARM_Neoverse_N1_L3(L3Cache):
    tag_latency = 48
    data_latency = 48
    response_latency = 16
    assoc = 16 # (1)
    size = '8MB'
    clusivity = 'mostly_excl'
    mshrs = 128
```

Listing B.2: Cache configuration

Bibliography

[1] Jack Dongarra and Piotr Luszczek. TOP500. In *Encyclopedia of Parallel Computing*, pages 2055–2057, Boston, MA, 2011. Springer US. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_157.

[2] Chris A. Mack. Fifty Years of Moore's Law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, Jan 2011. doi: 10.1109/TSM.2010.2096437.

[3] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martínez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37:26–39, May 2017. doi: 10.1109/MM.2017.35.

[4] Bo Zhao, Wei Gao, Rongcai Zhao, Lin Han, Huihui Sun, and Yingying Li. Performance evaluation of NPB and SPEC CPU2006 on various SIMD extensions. In *Big Data Computing and Communications*, volume 9196, pages 257–272, Aug 2015. ISBN 978-3-319-22046-8. doi: 10.1007/978-3-319-22047-5_21.

[5] Mathias Gottschlag and Frank Bellosa. Mechanism to Mitigate AVX-Induced Frequency Reduction. *CoRR*, abs/1901.04982, Sep 2019. doi: https://doi.org/10.48550/arXiv.1901.04982.

[6] G Quintin, S Jubertie, Florent De Martin, and Kenny Péou. Application of the vectorization library NSIMD to the EFISPEC3D kernel. In *Fifth EAGE Workshop on High Performance Computing for Upstream*, volume 2021, pages 1–5, Sep 2021. doi: 10.3997/2214-4609.2021612021.

[7] Matthias Kretz and Volker Lindenstruth. Vc: A C++ library for explicit vectorization. *Software: Practice and Experience*, 42(11):1409–1430, Dec 2011. doi: https://doi.org/10.1002/spe.1149.

[8] Adrien Cassagne, Olivier Aumage, Denis Barthou, Camille Leroux, and Christophe Jégo. MIPP: A Portable C++ SIMD Wrapper and Its Use for Error Correction Coding in 5G Standard. In *Proceedings of the 2018 4th Workshop on Programming*

Models for SIMD/Vector Processing, Feb 2018. ISBN 9781450356466. doi: 10.1145/3178433.3178435.

[9] Arm Limited. *Arm C Language Extensions for SVE.* Arm Limited. Company 02557590 registered in England., 110 Fulbourn Road, Cambridge, England CB1 9NJ, 2022.

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

[11] Siarhei Siamashka. Tinymembench. URL https://github.com/ssvb/tinymembench, 2019.

[12] David H Bailey. The NAS Parallel Benchmarks. Technical Report LBNL-3494E, Lawrence Berkeley National Lab., Berkeley, CA, Nov 2009.

[13] David Callahan, Jack Dongarra, and David Levine. Vectorizing compilers: a test suite and results. In *Supercomputing '88:Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Vol. I*, pages 98–105, Dec 1988. ISBN 0-8186-0882-X. doi: 10.1109/SUPERC.1988.44642.

[14] Zhang Xianyi, Wang Qian, and Yunquan Zhang. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 684–691, Dec 2012. ISBN 978-1-4673-4565-1. doi: 10.1109/ICPADS.2012.97.

[15] Henk Bekker, Herman Berendsen, E.J. Dijkstra, S. Achterop, Rudi Drunen, David van der Spoel, A. Sijbers, H. Keegstra, B. Reitsma, and M.K.R. Renardus. Gromacs: A parallel computer for molecular dynamics simulations. In *Physics Computing*, volume 92, pages 252–256, Jan 1993. ISBN 981-02-1245-3.

[16] Mark Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1-2: 19–25, Jul 2015. doi: 10.1016/j.softx.2015.06.001.

[17] J. Mortensen, L. Hansen, and K. Jacobsen. A real-space grid implementation of the Projector Augmented Wave method. *Physical Review B*, 71:035109, Jan 2005. doi: 10.1103/PhysRevB.71.035109.

[18] Paul Stewart Crozier, Heidi K Thornquist, Robert W Numrich, Alan B Williams, Harold Carter Edwards, Eric Richard Keiter, Mahesh Rajan, James M Willenbring, Douglas W Doerfler, and Michael Allen Heroux. Improving performance via

mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories (SNL), Sep 2009.

[19] Andrea Pellegrini, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, Anitha Kona, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, and Tushar Ringe. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro*, 40:53–62, Mar 2020. doi: 10.1109/MM.2020.2972222.

[20] Amazon Web Services Sebastien Stormacq. New – EC2 M6g Instances, powered by AWS Graviton2. URL https://aws.amazon.com/blogs/aws/new-m6g-ec2-instances-powered-by-arm-based-aws-graviton2/, May 2020. Accessed: 13-11-2022.

[21] Ryohei Okazaki, Takekazu Tabata, Sota Sakashita, Kenichi Kitamura, Noriko Takagi, Hideki Sakata, Takeshi Ishibashi, Takeo Nakamura, and Yuichiro Ajima. Supercomputer Fugaku Cpu A64fx realizing high performance, high-density packaging, and low power consumption. Technical review, 2020.

[22] Miguel Tairum Cruz. Performing SVE Studies using the Arm Instruction Emulator. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 638–638, Sep 2018. doi: 10.1109/CLUSTER.2018.00082.

[23] N Tallent, J Mellor-Crummey, L Adhianto, M Fagan, and M Krentel. HPCToolkit: performance tools for scientific computing. *Journal of Physics: Conference Series*, 125(1):012088, Jul 2008. doi: 10.1088/1742-6596/125/1/012088.

[24] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark Hill, and David Wood. The Gem5 simulator. *SIGARCH Computer Architecture News*, 39:1–7, Aug 2011. doi: 10.1145/2024716.2024718.

[25] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. Accuracy evaluation of gem5 simulator system. In *7th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC)*, pages 1–7, Jul 2012. doi: https://doi.org/10.1109/ReCoSoC.2012.6322869.

[26] Yuetsu Kodama, Tetsuya Odajima, Motohiko Matsuda, Miwako Tsuji, Jinpil Lee, and Mitsuhisa Sato. Preliminary Performance Evaluation of Application Kernels Using ARM SVE with Multiple Vector Lengths. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 677–684, Sep 2017. ISBN 978-1-5386-2326-8. doi: 10.1109/CLUSTER.2017.93.

[27] Andrei Poenaru and Simon McIntosh-Smith. Evaluating the Effectiveness of a Vector-Length-Agnostic Instruction Set. In *Euro-Par 2020: Parallel Processing*, pages 98–114, Aug 2020. ISBN 978-3-030-57674-5. doi: 10.1007/978-3-030-57675-2_7.

[28] Angela Pohl, Mirko Greese, Biagio Cosenza, and Ben Juurlink. A Performance Analysis of Vector Length Agnostic Code. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 159–164, Sep 2019. ISBN 978-1-7281-4485-6. doi: 10.1109/HPCS48598.2019.9188238.

[29] Adrià Armejach, Helena Caminal, Juan Cebrian, Rubén Langarita, Rekai González-Alberquilla, Chris Adeniyi-Jones, Mateo Valero, Marc Casas, and Miquel Moretó. Using Arm's scalable vector extension on stencil codes. *The Journal of Supercomputing*, 76:2039–2062, Mar 2020. doi: 10.1007/s11227-019-02842-5.

[30] 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)*, pages 623–628, Sep 2018. ISBN 978-1-5386-8319-4. doi: 10.1109/CLUSTER.2018.00079.

[31] Christie Alappat, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, Nils Meyer, and Tilo Wettig. Performance Modeling of Streaming Kernels and Sparse Matrix-Vector Multiplication on A64FX. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 1–7, Sep 2020. ISBN 978-1-6654-2265-9. doi: 10.1109/PMBS51919.2020.00006.

[32] Adrian Jackson, Michèle Weiland, Nick Brown, Andrew Turner, and Mark Parsons. Investigating Applications on the A64FX. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 549–558, Sep 2020. ISBN 978-1-7281-6677-3. doi: 10.1109/CLUSTER49012.2020.00078.

[33] 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*, pages 372–382, Oct 2011. ISBN 978-1-4577-1794-9. doi: 10.1109/PACT.2011.68.

[34] Phil Colella. Defining Software Requirements for Scientific Computing, 2004. DARPA HPCS presentation.

[35] Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The Landscape of Parallel Computing Research: A View from

Berkeley. Technical Report EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[36] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communinucations of the ACM*, 52:65–76, Apr 2009. doi: 10.1145/1498765.1498785.

[37] Johannes Hofmann, Christie L. Alappat, Georg Hager, Dietmar Fey, and Gerhard Wellein. Bridging the Architecture Gap: Abstracting Performance-Relevant Properties of Modern Server Processors. *Supercomputing Frontiers and Innovations*, 7 (2):54–78, Jul 2020. doi: 10.14529/jsfi200204.

[38] Romain Dolbeau. FFTW3: Leveraging the Scalable Vector Extension. URL https://www.european-processor-initiative.eu/wp-content/uploads/2019/10/EPI-FFTW3-SVE-Romain.pdf, Sep 2019. Accessed: 12.06.2022.

[39] Walter Lioen, Miguel Avillez, Valeriu Codreanu, Dimitris Dellis, Sagar Dolas, Andrew Emerson, Jacob Finkenrath, Cédric Jourdain, Martti Louhivuori, Cristian Morales, Charles Moulinec, Arno Proeme, and Andrew Sunderland. Evaluation of Accelerated and Non-accelerated Benchmarks. Deliverable D7.5 Grant Agreement Number: EINFRA-730913, PRACE Fifth Implementation Phase Project, Apr 2019.

[40] Tze Low, Francisco Igual, Tyler Smith, and Enrique S Quintana-Orti. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Transactions on Mathematical Software*, 43:1–18, Aug 2016. doi: 10.1145/2925987.

[41] Szilárd Páll and Berk Hess. A flexible algorithm for calculating pair interactions on SIMD architectures. *Computer Physics Communications*, 184:2641–2650, Dec 2013. doi: 10.1016/j.cpc.2013.06.003.

[42] Tom Darden, Darrin York, and Lee Pedersen. Particle Mesh Ewald: An Nlog(N) Method for Ewald Sums in Large Systems. *The Journal of Chemical Physics*, 98: 10089–10092, Jun 1993. doi: 10.1063/1.464397.

[43] Martti Louhivuori. GPAW performance optimisation and energy consumption on KNLs. Presentation, IXPUG spring 2018, Bologna, May 2018.

[44] E. Briggs, D. Sullivan, and J. Bernholc. Real-space multigrid-based approach to large-scale electronic structure calculations. *Physical review. B*, 54:14362–14375, Dec 1996. doi: 10.1103/PhysRevB.54.14362.

[45] Ask Larsen, Marco Vanin, Jens Mortensen, Kristian Thygesen, and Karsten Jacobsen. Localized atomic basis set in the projector augmented wave method. *Physical Review. B*, 80:195112, Sep 2009. doi: 10.1103/PhysRevB.80.195112.

[46] Olga Moldovanova and Mikhail Kurnosov. Auto-Vectorization of Loops on Intel 64 and Intel Xeon Phi: Analysis and Evaluation. In *PaCT 2017: Parallel Computing Technologies*, pages 143–150, Jul 2017. ISBN 978-3-319-62932-2. doi: 10.1007/978-3-319-62932-2.

[47] Arm Limited. Write Streaming mode. Manual Arm Neoverse N1 Core Technical Reference Manual r4p0, 2020. Accessed: 17.08.2022.

[48] Arm Limited. Arm Neoverse N1 PMU Guide Issue 2.0. Manual PJDOC-466751330-547673, 2022. Accessed: 17.08.2022.

[49] Jan Laukemann, Julian Hammer, Georg Hager, and Gerhard Wellein. Automatic Throughput and Critical Path Analysis of x86 and ARM Assembly Kernels. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 1–6, Oct 2019. doi: 10.1109/PMBS49563.2019.00006.

[50] Bine Brank, Stepan Nassyr, Fatemeh Pouyan, and Dirk Pleiter. Porting Applications to Arm-based Processors. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 559–566, Sep 2020. doi: 10.1109/CLUSTER49012.2020.00079.

[51] Guy E. Blelloch. Prefix Sums and Their Applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov 1990.

[52] Patrick Lavin, E. Jason Riedy, Rich Vuduc, and Jeffrey S. Young. Spatter: A benchmark suite for evaluating sparse access patterns. *CoRR*, abs/1811.03743, Nov 2018. doi: 10.48550/arXiv.1811.03743.