DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Code Generation for Small Matrix Multiplication Kernels Targeting Arm SVE

Alexander Johann Puscas

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Code Generation for Small Matrix Multiplication Kernels Targeting Arm SVE

Codegenerierung von Multiplikationskernels für kleine Matrizen auf Arm SVE

Author:	Alexander Johann Puscas
Supervisor:	Univ.-Prof. Dr. Michael Bader
Advisor:	Lukas Krenz, M.Sc.
Date:	15.11.2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.11.2021 Alexander Johann Puscas

Acknowledgements

I would like to thank:
The Chair of Scientific Computing at TUM and Prof. Dr. Michael Bader for giving me the opportunity to start this project,
my advisor Lukas Krenz for giving me guidance,
and my family and girlfriend for giving me the strength to finish it.

Abstract

The Fujitsu A64FX and the inclusion of Arm's SVE have shown promising results in the field of HPC. Especially for numerical applications, the novel extension of the Aarch64 ISA has been shown to yield significant performance boosts for applications that are correctly ported to Arm architectures. This work extends the matrix multiplication kernel generator PSpaMM to allow generating SVE instructions and analyzes the measured results. We benchmark multiplication kernels generated by PSpaMM containing SVE and NEON instructions, as well as matrix multiplication kernels generated by LIBXSMM. We show that SVE-based kernels can provide a performance boost of a factor of 6.3 for small matrix multiplication kernels when compared to PSpaMM's NEON kernels. Benchmarks including dense-by-sparse multiplication kernels show that the SVE kernels achieve increased performances by a factor of 3.8 compared to their NEON counterparts. Finally, we observe that PSpaMM's SVE generator can compete performance-wise with the more optimized math library LIBXSMM.

Contents

1 Introduction

Historically, the x86 instruction architecture set (ISA) was predominantly used for high performance computing (HPC) systems [44]. This family of ISAs, which has been originally developed by Intel, implements a complex instruction set computer (CISC) architecture. The CISC architecture aims to perform tasks using as few instructions as possible [8]. To achieve this, CISC-based processors are able to perform multiple operations after interpreting a single instruction. The execution of a CISC instruction therefore takes multiple cycles [10]. CISC based processors were able to improve their performance in the last decades by increasing the amount of transistors per CPU while simultaneously decreasing the size of the transistors [44]. However, the rise in processing capabilities is followed by higher heat generation and power consumption [44]. Therefore, more power needs to be used to combat the heat generation, which further increases the amount of power that a processor consumes [44].

As an alternative, reduced instruction set computer (RISC) architectures only include simple instructions that the processor can execute during a single clock cycle. Consequently, programmers need to write more lines of assembly code for RISC processors compared to implementing the same code for CISC processors [10]. Arm is a family of RISC architectures designed for processors that are employed in different environments, for example in low power environments and smartphones. During the last decade, a lot of effort has been made to develop Arm processors that can perform more compute intensive tasks, especially in relation to High Performance Computing (HPC) [44]. In 2019, Fujitsu finished developing a new Arm based CPU, the Fujitsu A64FX [45]. This Arm based processor was then used to build the "Fugaku" supercomputer in collaboration with RIKEN. Since 2020, Fugaku has taken the first place on the Top500 list three times in a row [41]. This list, ranking the 500 fastest supercomputers in the world by measuring their peak performance in FLOP/s, has otherwise been dominated by systems that primarily utilize GPU accelerators.

One HPC application that can potentially make use of the Fujitsu A64FX processor is SeisSol [38], a seismic wave propagation solver that simulates seismic wave phenomena and earthquake dynamics. There are numerous papers that explain in detail how SeisSol works and how it has been optimized [7, 23]. The software utilizes code generators to create matrix multiplication kernels. For example, PSpaMM [34] is a generator specifically created for SeisSol [6]. The code generator is currently run on Intel's x86 many-core processor Intel Xeon Platinum 8174 with AVX-512 and the Marvell ThunderX2 with Arm NEON. However, the latest developments in HPC have made the A64FX, including the Scalable Vector Extension (SVE), an interesting target for developing a new code generator that returns Arm based SVE assembly.

The goal of this thesis is to extend the aforementioned code generator PSpaMM to create inline assembly for small matrix multiplication kernels specifically targeted at the Fujitsu A64FX using Arm's novel SVE instruction set. The generated assembly is tested using matrices that are commonly used in SeisSol. Furthermore, the performance of the generated code will be measured and compared to inline assembly generated by PSpaMM. Chapter 2 discusses general information related to RISC-based architectures, Fujitsu's A64FX, and the fundamental differences between NEON and SVE, whereas chapter 3 discusses general algorithmic ideas that can be used in matrix multiplications. In chapter 4, we present advantages of several loop optimization techniques and how to implement them, as well as discuss the key differences between PSpaMM's NEON generator and the new SVE generator. We analyze the results of benchmarks that test the performance of the generated SVE kernels in chapter 5. Finally, chapter 6 concludes this thesis with a summary and gives an outlook on possible future research.

2 Background

2.1 RISC vs CISC

There are numerous ways in which RISC and CISC processors differ. Some of these key properties are the length of the encoded instructions and the design goals of the respective processor architectures [8]. The instructions used in RISC processors have a fixed encoding length and can be categorized into two types, load/store instructions and operations which exclusively work on registers [21]. On the contrary, instructions used by CISC processors are of variable length depending on their complexity [8]. There are more than two instruction types available as well, since CISC operations allow referencing memory directly inside of an operation instead of having to first load the data into a register [21]. In addition, each processor architecture has different design goals. CISC processors aim to keep a program's length to a minimum by performing as much work as possible per instruction, whereas RISC processors usually generate more lengthy code while being able to execute single instructions faster than CISC-based processors [21].

In order to better understand the key difference between RISC and CISC, i.e. the complexity of the instructions, we will look at an example. Let A, B denote the names of two registers in either architecture type and let `i:j` denote part of the memory where the value of a matrix at the i^{th} row and j^{th} column. Listing 2.1.1 and 2.1.2 contain a side-by-side view of pseudo-code which multiplies two values of a matrix and stores the result back into the matrix.

```
MULT 4:3, 3:5



```

Listing 2.1.1: Pseudo MULT instruction on CISC processor

```
LOAD A, 4:3
LOAD B, 3:5
PROD A, B
STORE A, 4:3
```

Listing 2.1.2: Pseudo MULT instruction on RISC processor

The instruction in listing 2.1.1 is known as a complex instruction. It can operate directly on memory cells without having to load the values into registers or explicitly call a store instruction. In this example, the first source operand also serves as the destination. An advantage of the CISC approach is that the compiler can easily translate high-level code into low-level instructions [8]. If we assign the value of the memory cell 4:3 to x and the value of 3:5 to y, the instruction in listing 2.1.1 is equal to $x = x * y$.

Listing 2.1.2 depicts the same code as as listing 1, but instead uses a RISC approach. In this case, the processor has to explicitly load the values into two registers and multiply these values using a (pseudo-)instruction called `PROD`, before it can store the result back into memory. Although this approach may give the impression that the compiler and the processor have to do more work, this strategy also comes with advantages. Each RISC instruction requires roughly the same amount of time to be executed, i.e. one clock cycle [8]. Therefore, the RISC version of the code takes around as long as the CISC version [8].

Another advantage of the RISC approach is that instruction-pipelining becomes possible because the instructions take a similar amount of time to be executed [8]. An instruction pipeline enables the processor to execute different parts of consecutive instructions at the same time [13]. Figure 2.1.1 depicts a 5-stage pipeline consisting of the following stages: *Instruction Fetch*, *Instruction Decode and Register Fetch*, *Execute*, *Memory Access*, and *Register Write Back*. If the pipeline is at full capacity, the processor can execute all five stages simultaneously in one clock cycle. However, the execution of each instruction must be at a different stage. In an ideal scenario, pipelining keeps all data processing elements busy, thus effectively increasing the rate at which instructions are processed (see 2.1.1). In practice, however, instructions can depend on the results of previous instructions and have to halt their execution while waiting for the previous instruction to finish. This is commonly referred to as a stall and can significantly decrease the speed of a processor [9]. Modern processors can hide stalls by making use of their "out-of-order resources", meaning that they can change the order in which instructions are executed to minimize the amount of pipeline stalls [9].

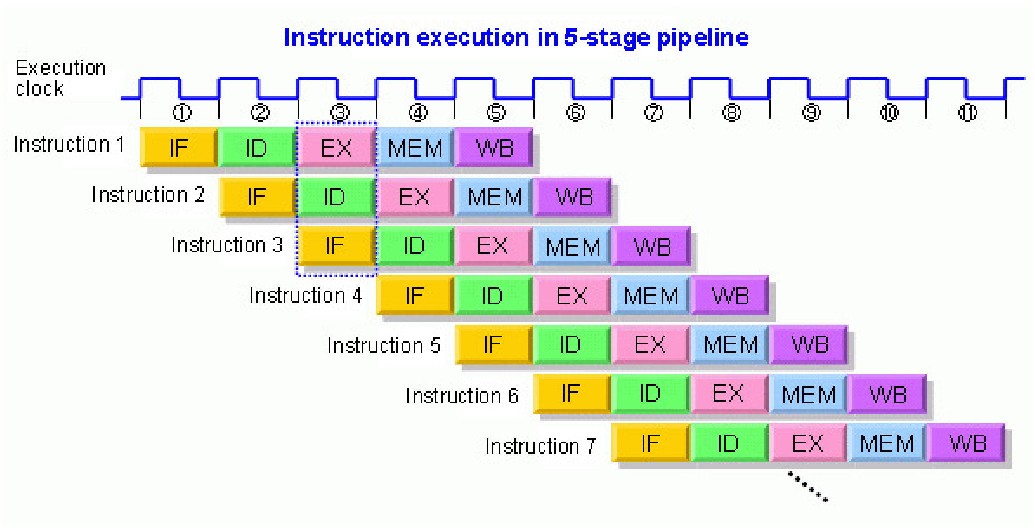

Figure 2.1.1: Ideal scenario of executing multiple instructions in a general 5-stage pipeline, figure taken from [4]

Similar to other modern processors, the processor of the A64FX consists of more stages than the example pipeline in figure 2.1.1. The seven-stage pipeline implemented in the A64FX is shown in figure 2.1.2. A thorough explanation of the pipeline's functionality would be outside of the scope of this thesis. Therefore, we refer to an article published by Fujitsu which contains a more detailed description [31].

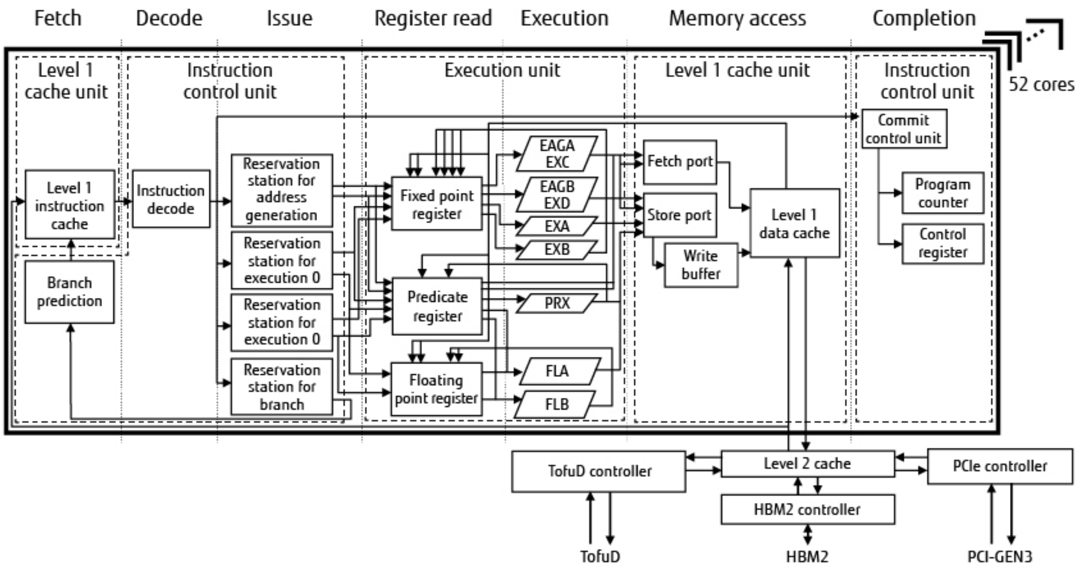

Figure 2.1.2: Diagram of the A64FX seven-stage pipeline, figure taken from [31]

2.2 A64FX

The Fujitsu A64FX is a processor that was jointly developed by Fujitsu and RIKEN Center for Computational Science [28]. Additionally, Fujitsu collaborated with Arm to include the Armv8 instruction set and the Scalable Vector Extension (SVE), which in turn lead to Fujitsu contributing to the design of SVE [36]. The A64FX was used to build the Fugaku supercomputer, which performs roughly 42 times more FLOPS [42] than Japan's previous supercomputer, commonly referred to as the “K computer” [35].

Figure 2.2.1 contains a cross-section of the processor. The A64FX has four connected Core Memory Groups (CMG), each containing 12 compute cores and one assistant core. Nevertheless, the A64FX we have access to for this thesis seems to lack assistant cores. Every core has access to a separate L1 cache. However, there is only one L2 cache and memory controller per CMG that is shared between all cores of that CMG. Each CMG can access a separate high-bandwidth memory (HBM2), with each HBM2 offering a memory bandwidth of 256 GB/s [36]. A closer look at the A64FX specifications, as well as the specifications of other processors used for SeisSol, is provided in table 2.2.1.

Extensive tests were executed to benchmark the performance of the A64FX. Researchers at RIKEN executed different benchmarks and applications on the A64FX containing either compute-/memory-bandwidth-intensive kernels or different memory access patterns [28]. They also chose data set sizes that allowed the tests to access the main memory rather than only the caches. The best results are usually seen when performing tests that benefit from a high memory-bandwidth. The researchers accredit this to the fact that performing a high enough number of computational instructions using the 512-bit vectors of the A64FX can benefit from the high memory bandwidth of the processor's HBM2 memory. Additionally,

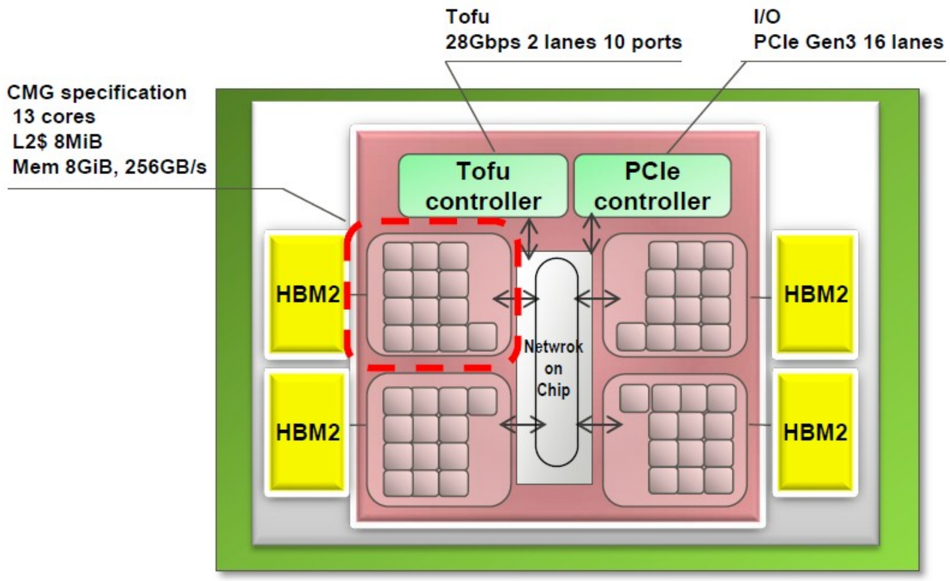

Figure 2.2.1: Block diagram of the Fujitsu A64FX, figure taken from [27]

the processor's computing performance was shown to scale well as the number of threads increases [28].

Jackson et al. show that the A64FX generally outperforms other similar processors when executing benchmarks and applications that are commonly used in HPC [20]. The paper shows that the researchers were able to execute multiple applications and benchmarks on the A64FX without changing the code while also receiving good performances when executing these programs on the processor. Furthermore, when comparing the A64FX's performance to that of the P100 and V100 GPUs when executing the "Nekbone" benchmark, which is a standard poisson equation solver, the processor outperforms the GPUs when using the Fujitsu compiler with the fast maths flag `-Kfast` enabled. However, they note that some of the benchmarks and applications performed worse on the A64FX compared to other Arm or Intel processors. Nonetheless, Jackson et al. point out that the right optimizations for a target architecture could lead to significant performance gains [20]. Moreover, Fujitsu's A64FX specifications show that the processor can reach 90% of the theoretical peak performance of 2.7 TFLOPS when executing the double precision general matrix multiplication (DGEMM) and 80% of the peak memory bandwidth of 1024 GB/s when executing a STREAM triad [14].

Since the A64FX is a processor that was released only recently, the amount of performance-related research is limited [28]. Nevertheless, there are a few more papers that can be referenced for further performance comparisons of the A64FX with other, similar processors. Brank et al. [5] executed a variety of benchmarks using the A64FX, ThunderX2, and the Intel Xeon E5-2660 v3 and compared the obtained performance results. The researchers conclude that the A64FX performs best when executing applications that are memory-bandwidth reliant and can make use of SVE. However, the obtained results from the A64FX were worse when executing benchmarks with a more complex control flow or memory access pattern.

Hardware Specification	**A64FX**	**ThunderX2 CN9980**	**Xeon Platinum 8174**
ISA	Armv8.2-A+SVE	Armv8.1-A + NEON	Intel AVX-512
Number of Cores	48 + 2-4 assistant cores	32 physical - 128 logical	24 physical - 48 logical
Frequency	1.8 - 2.2 GHz	2.0 - 2.5 GHz	3.1 - 3.9 GHz
SIMD Width	512 bits	128 bits	512 bits
L1I Cache	3 MiB (64 KiB/core)	1 MiB (32 KiB/core)	n/a
L1D Cache	3 MiB (64 KiB/core)	1 MiB (32 KiB/core)	n/a
L2 Cache	32 MiB (8 MiB/CMG)	8 MiB (256 KiB/core)	24 MB (1MB/core)
L3 Cache	None	32 MiB	33 MB
Cache-Line Size	256 bytes	* 64 bytes	n/a
Process Technology	7 nm CMOS FinFET	16 nm CMOS FinFET	n/a
Memory Bandwidth	1,024 GB/s	160 GB/s	n/a

Table 2.2.1: Comparison of the hardware specification of the A64FX [15], ThunderX2 CN9980 [43], and Intel Xeon Platinum 8174 [19] processors. Information marked with a * has been taken from processors that we can access on a cluster provided by the Leibniz Rechenzentrum (LRZ)

They further mention that the right optimizations can help to receive better performances [5]. Gupta et al. evaluated the performance of the runtime system "High Performance ParalleX" [22] (HPX), which aims to address common challenges found in HPC like scalability and efficiency, using a variety of benchmarks on different processors, including the A64FX. In their paper, they show that the Fujitsu processor outperforms all other tested processors in benchmarks that test memory-bandwidth, execution time, and GLUPs/s [16].

2.3 Scalable Vector Extension

The Arm Scalable Vector Extension (SVE) does not expand the existing advanced SIMD instruction set architecture, but it is a relatively new extension introducing new A64 instruction encodings [39]. SVE is a vector length agnostic (VLA) instruction set introducing new vector registers named Z0 to Z31 that can contain 8-, 16-, 32-, or 64-bit elements [**?**]. These vector registers are separate from the existing 128-bit advanced SIMD registers named V0 to V31. A vector's length is implementation dependent and can vary between 128 bits and 2048 bits in increments of 128 bits [40]. This means that we can write code once and execute it on different processors that implement SVE for varying vector lengths without recompiling the code or having to know the exact vector length for each processor.

Additionally to the 32 new vector registers, SVE introduces 16 predicate registers P0 to P15. These predicates are used to determine which vector elements are active while executing a predicated instruction [40]. Vector elements are also often referred to as lanes. Predicates can be used to manage loops using SVE instructions. Further insight into SVE is available in [40]. The reference manual containing the exact instructions can be downloaded from [1].

In order to better understand the new registers introduced by SVE, we will look at an example code snippet of inline assembly in C++. This code snippet contains SVE instructions that perform an element-wise multiplication of two arrays and add the resulting elements onto a third array. All three arrays have the same arbitrary length of $n > 0$ elements. To keep the example code as small as possible, the loads of the input variables and the list of ouput, input, and clobbered registers is omitted.

```
1  void array_mul(double* A, double* B, double* C, long length) {{
2      __asm__ __volatile__(
3          /* setup code that loads the pointers to the beginning of
4           * A, B, C into the registers x0, x1, x2 respectively,
5           * while x3 contains the length of the arrays */
6
7          "mov x4, xzr\r\n"
8          "LOOP%=:\r\n"
9          "whilelo p0.d, x4, x3\r\n"
10         "ld1d z0.d, p0/z, [x0, x4, LSL #3]\r\n"
11         "ld1d z1.d, p0/z, [x1, x4, LSL #3]\r\n"
12         "ld1d z2.d, p0/z, [x2, x4, LSL #3]\r\n"
13         "fmla z2.d, p0/m, z0.d, z1.d\r\n"
14         "st1d z2.d, p0, [x2, x4, LSL #3]\r\n"
15         "incd x4\r\n"
16         "b.cc LOOP%=\r\n"
17
18         /* listing of the output, input, and clobbered registers */
19     );
20 }}
```

Listing 2.3.1: SVE example

The example code in listing 2.3.1 contains inline assembly of an element-wise array multiplication. Let `VL` be the vector length at which SVE is implemented. For the A64FX we have `VL = 8`. The registers `x0`, `x1`, and `x2` contain the pointers to the arrays `A`, `B`, and `C`. Register `x3` contains the length of the arrays. Line 7 uses the `xzr` register to move the number 0 into `x4`. The purpose of the special register `xzr` is to have access to a register containing a constant 0 without wasting a general-purpose register. Line 9 takes a predicate and fills it with `true` or `false` values. Setting an element of a predicate to `true` or `false` actually means setting the element to either 1 or 0, respectively. To do this, the `whilelo` instruction repeatedly performs $\mathrm{x4} + i < \mathrm{x3}$ while incrementing `i` from 0 to `VL - 1`. The `i`$^{\texttt{th}}$ element of the predicate register `p0` is set to `true`, if $\mathrm{x4} + i < \mathrm{x3}$ evaluates to `true`. Otherwise, the corresponding element in `p0` is set to `false`. The `whilelo` instruction, however, does not actually alter the value stored in `x4`.

Lines 10 to 12 perform predicated loads. Each instruction loads up to `VL = 8` double elements from an array stored at $\mathrm{xk} + \mathrm{x4} * 8$ and stores the elements in `zk.d`, with $\mathrm{k} \in \{0, 1, 2\}$. The `.d` suffix of the SVE vectors indicates that the vectors contain 64-bit elements. The load instructions all use "zeroing predication", which means that inactive vector elements in `zk.d` are filled with zeros. An element is considered inactive if the corresponding element of the used predicate is set to $false$. Therefore, loads only access memory if an element of the used predicate is set to `true`. This ensures that loading a vector with elements from memory does not access more memory than intended. Using the same predicate register for all loads also guarantees the same number of elements are loaded into each vector register.

Line 13 performs a simple element-wise “fused multiply add” instruction $z2 = z2 + z0 * z1$. Additionally, the used predicate register indicates that the inactive elements of `z2` are filled with zero.

The store instruction in line 14 behaves similarly to the previous load instructions. However, only the active elements of $z2$ are stored back into memory.

Line 15 simply increments `x4` by the number of double words, i.e. 64-bit elements, a vector can hold. Because the A64FX has 512-bit long vectors, the instruction always increments a register by 8. In our example, this instruction is used to update the number of processed array elements contained in `x4`.

Lastly, line 16 performs a branch instruction back to the top of the loop under the condition that the last element of the predicate $p0$ created by `whilelo` is `true`. Another branching condition that we could use is `.any`, which executes a branch if any predicate element was set to `true` by the previous predicate-setting instruction. We can omit an otherwise needed compare instruction right before the branch, because only the `whilelo` instruction actually updates the condition flags. All other instructions executed within the loop do not modify the values of the condition flags.

Although the code performs one unnecessary loop traversal if the length of the arrays is a multiple of 8, we would not see a significant increase in runtime. This is due to the fact that the predicate register `p0` would have all elements set to `false` by the `whilelo` instruction. This in turn entails that the loads do not actually access the memory, because all inactive elements of the vector registers will be set to 0. Analogously, the `fmla` and store instructions would not be performed. Finally, the register `x4` would be incremented one more time before reaching the branch instruction, which is not executed since the last element in the predicate `p0` is set to `false`.

The example shows that the programmer does not have to think about how many elements can fit in a vector or how many elements are left to be processed. The correct usage of predicates eliminates the need to incorporate these values into the code, which is one of the main properties of VLA code. In addition to that, the predicates introduced by SVE make including a “tail loop” at the end of the code above obsolete. When implementing the same function using NEON inline assembly for example, the loop would be able to contain two array elements per vector register. If the length of the arrays was not a multiple of two, we would not be able to fully load a vector register without loading data from outside the arrays. This makes including a tail loop necessary in order to be able to process the last few elements of each array.

Numerous tests have been performed to evaluate the performance of Arm’s SVE. Pohl et al. observed in a series of benchmarks that SVE reaches roughly 90% of the performance that Arm’s NEON reaches [33]. They accredit this performance loss to the predication of the instructions. For these benchmarks, the vector length used by SVE was set to 128 bits, which corresponds to the length of a NEON vector. Furthermore, the paper compares the auto-vectorization rates of the SVE and NEON ISAs with the help of 151 different loop patterns. The researchers show that the GCC compiler was able to vectorize 66 loops using both SVE and NEON. An additional 16 loops were vectorized exclusively with SVE, whereas no loops could be vectorized only using NEON instructions. In other words, the compiler was able to auto-vectorize 24.2% more loops when using SVE compared to NEON. The paper accredits this to the fact that SVE supports predication, as well as gather and scatter instructions, all of which have been missing in NEON [33].

However, a survey on Arm processors suggests that SVE achieves a speedup of up to 3x and 7x for a vector length of 128 bits and 512 bits respectively, compared to using NEON and its 128-bit registers [44]. It is important to note that the tests in [33] were performed on the gem5 simulator, whereas the tests performed in [44] measured the performance of SVE with a model that was executed on a processor that is not related to an existing one.

Finally, researchers at RIKEN concluded that increasing the vector length used by the processor, as is possible with processors implementing SVE, can lead to performance improvements if the program contains large amounts of arithmetic operations [24]. However, enough physical registers need to be available, otherwise the performance might get worse. If memory bandwidth presents a bottleneck to the performance of a program, increasing the vector length will have a significant impact on the performance. In general, SVE seems to be useful when trying to find the most suitable vector length, because the ISA can be run on processors with different vector lengths without having to adjust the code [24].

3 Algorithms

In this section, we will explore different algorithmic ideas that are used in "PSpaMM" and discuss their strengths and weaknesses. To keep the notations in this thesis consistent, we are going to introduce specific notations as shown in table 3.0.1.

Parameters	*Description*
A, B	Input Matrices
C	Result Matrix
x, y	Vectors of size n
n, k, m	Integers ≥ 1
$n \times k$	Dimensions of A
$k \times m$	Dimensions of B
$n \times m$	Dimensions of C

Table 3.0.1: Common parameters used in this thesis

3.1 Matrix-Matrix multiplication

There are numerous ways to multiply two matrices. Each approach has its own advantages and use cases. These approaches include the naive approach and an outer-product-based formulation. There are other more complex approaches which can perform matrix multiplications in less asymptotic time. However, in the context of this thesis, we will take a closer look only at the first two approaches. We provide code examples for matrix multiplication approaches. To keep it simple, we will assume that in both cases the data is stored sequentially in memory using a one-dimensional layout.

3.1.1 Naive Approach

The most commonly taught matrix multiplication algorithm is the naive approach. In order to calculate $C = AB$, we need to iteratively multiply each row of A with each column of B. To compute an element $c_{i,j}$ of the result matrix C, we need to calculate the dot product of the i^{th} row of A and the j^{th} column of B. Let $a_i = [a_{i,1}, ..., a_{i,k}]$ and $b_j = [b_{1,j}, ..., b_{k,j}]$ denote the i^{th} row and j^{th} column of A and B, respectively. The dot product of two vectors is defined as

$$x \cdot y = xy^T = \begin{bmatrix} x_1, & \cdots, & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} x_i * y_i. \tag{3.1.1}$$

Using the above equation, we can now establish a formula for the naive matrix multiplication with

$$C = AB = \begin{bmatrix} a_{1,1} & \cdots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,k} \end{bmatrix} \begin{bmatrix} b_{1,1} & \cdots & b_{1,m} \\ \vdots & \ddots & \vdots \\ b_{k,1} & \cdots & b_{k,m} \end{bmatrix} \tag{3.1.2}$$

$$= \begin{bmatrix} \sum_{i=1}^{k} a_{1,i} * b_{i,1} & \cdots & \sum_{i=1}^{k} a_{1,i} * b_{i,m} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{k} a_{n,i} * b_{i,1} & \cdots & \sum_{i=1}^{k} a_{n,i} * b_{i,m} \end{bmatrix}.$$

To better understand the naive approach, we introduce a general example of a matrix multiplication using the parameters $n = k = m = 3$. In this case, calculating the result of the multiplication is defined as

$$C = AB = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} \tag{3.1.3}$$

$$= \begin{bmatrix} (a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1}) & (a_{1,1}b_{1,2} + a_{1,2}b_{2,2} + a_{1,3}b_{3,2}) & (a_{1,1}b_{1,3} + a_{1,2}b_{2,3} + a_{1,3}b_{3,3}) \\ (a_{2,1}b_{1,1} + a_{2,2}b_{2,1} + a_{2,3}b_{3,1}) & (a_{2,1}b_{1,2} + a_{2,2}b_{2,2} + a_{2,3}b_{3,2}) & (a_{2,1}b_{1,3} + a_{2,2}b_{2,3} + a_{2,3}b_{3,3}) \\ (a_{3,1}b_{1,1} + a_{3,2}b_{2,1} + a_{3,3}b_{3,1}) & (a_{3,1}b_{1,2} + a_{3,2}b_{2,2} + a_{3,3}b_{3,2}) & (a_{3,1}b_{1,3} + a_{3,2}b_{2,3} + a_{3,3}b_{3,3}) \end{bmatrix}.$$

Listing 3.1.1 shows a typical implementation of the naive approach in C++ consisting of three for-loops. The outer loop iterates over the rows of A, whereas the intermediate loop iterates over the columns of B and the rows of C, respectively. The innermost loop calculates the dot product of the $(ni + 1)^{th}$ row of A and the $(mi + 1)^{th}$ column of B and stores the intermediate results in the accumulator. The calculation is done by executing a series of "fused-multiply-add" instructions, which are counted as two FLOPS each. Finally, the result of the dot product is written into the corresponding cell of the result matrix C.

```
// assume that the matrices A, B, C of type double are defined as in section 3
  for (int ni = 0; ni < n; ++ni) {
      for (int mi = 0; mi < m; ++mi) {
          for (int ki = 0; ki < k; ++ki) {
              C[ni*n + mi] += A[ni*n + ki] * B[ki + mi*m];
          }
      }
  }
```

Listing 3.1.1: C++ example code for a naive matrix multiplication

We can see in listing 3.1.1 that the naive algorithm performs a total of $2 * k$ arithmetic instructions for each execution of the innermost loop. This loop in turn is executed $n * m$ times, resulting in a total of $2 * n * k * m$ FLOPS for the naive approach. Furthermore, the algorithm uses $2 * k$ load instructions every time the innermost loop gets traversed and $n * m * k$ store instructions, causing a total of $2knm + nmk$ memory accesses. In cases where $n = k = m$, i.e. all matrices are square, this approach yields $\mathcal{O}(2 * n^3) = \mathcal{O}(n^3)$ FLOPS, $\mathcal{O}(2 * n^3) = \mathcal{O}(n^3)$ loads, and $\mathcal{O}(n^3)$ stores. One common way to quickly improve the runtime performance of the naive approach is to introduce an accumulator variable, which we will refer to as `sum`. We can assign $sum = C[ni * n + mi]$ before entering the innermost loop, replace the store instruction into the matrix C with the accumulator, and finally store the accumulated result into C with $C[ni * n + mi] = sum$ after exiting the inner loop. This way, we can reduce the amount of store instructions by a factor of `k` to a total of $n * m$.

3.1.2 Outer Product

Contrary to the naive approach presented in section 3.1.1, the following matrix multiplication algorithm makes use of the "outer product". While the multiplication of two vectors using the dot product returns a scalar, the result of the outer product is a matrix. The calculation of the outer product of any two vectors can be defined as

$$x \otimes y = x^T y = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} y_1, & y_2, & \dots, & y_m \end{bmatrix} = \begin{bmatrix} x_1 * y_1 & x_1 * y_2 & \dots & x_1 * y_m \\ x_2 * y_1 & x_2 * y_2 & \dots & x_2 * y_m \\ \vdots & \vdots & \ddots & \vdots \\ x_n * y_1 & x_n * y_2 & \dots & x_n * y_m \end{bmatrix}. \tag{3.1.4}$$

In contrast to the dot product in equation 3.1.1, the sizes of the two vectors x and y in equation 3.1.4 may differ when calculating the outer product. This is possible because instead of a scalar result, a $n \times m$ matrix is created. The i^{th} column of the result of $x \otimes y$ is equivalent to $x * y_i$. Likewise, the i^{th} row of $x \otimes y$ corresponds to $x_i * y$. With the above equation, we can now establish a formula for a general matrix multiplication using the outer product. Let a_i and b_i denote the i^{th} column and row of the matrices A and B, respectively. The matrix C_i is used to store the intermediate results of the outer product. With this, we can write the multiplication of A and B as

$$C = AB = \sum_{i=1}^{k} C_i = \sum_{i=1}^{k} a_i \otimes b_i. \tag{3.1.5}$$

When comparing the example equation 3.1.3 with the same general matrix multiplication using the outer product in equation 3.1.6, we can see that the end result remains the same, only the approach differs.

$$\begin{aligned} C = AB &= \begin{bmatrix} \mathrm{a}_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} \mathrm{b}_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} \\ &= \begin{bmatrix} \mathrm{a}_{1,1}b_{1,1} & a_{1,1}b_{1,2} & a_{1,1}b_{1,3} \\ a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & a_{2,1}b_{1,3} \\ a_{3,1}b_{1,1} & a_{3,1}b_{1,2} & a_{3,1}b_{1,3} \end{bmatrix} + \begin{bmatrix} \mathrm{a}_{1,2}b_{2,1} & a_{1,2}b_{2,2} & a_{1,2}b_{2,3} \\ a_{2,2}b_{2,1} & a_{2,2}b_{2,2} & a_{2,2}b_{2,3} \\ a_{3,2}b_{2,1} & a_{3,2}b_{2,2} & a_{3,2}b_{2,3} \end{bmatrix} + \begin{bmatrix} \mathrm{a}_{1,3}b_{3,1} & a_{1,3}b_{3,2} & a_{1,3}b_{3,3} \\ a_{2,3}b_{3,1} & a_{2,3}b_{3,2} & a_{2,3}b_{3,3} \\ a_{3,3}b_{3,1} & a_{3,3}b_{3,2} & a_{3,3}b_{3,3} \end{bmatrix} \end{aligned} \tag{3.1.6}$$

For the sake of completeness, the general formula for an outer product based matrix multiplication is defined as

$$\begin{aligned} C &= \sum_{i=1}^{k} C_i = \sum_{i=1}^{k} a_i \otimes b_i \\ &= \begin{bmatrix} a_{1,1} * b_{1,1} & \cdots & a_{1,1} * b_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} * b_{1,1} & \cdots & a_{n,1} * b_{1,m} \end{bmatrix} + \cdots + \begin{bmatrix} a_{1,k} * b_{k,1} & \cdots & a_{1,k} * b_{k,m} \\ \vdots & \ddots & \vdots \\ a_{n,k} * b_{k,1} & \cdots & a_{n,k} * b_{k,m} \end{bmatrix}. \end{aligned} \tag{3.1.7}$$

Although basing a matrix multiplication on equation 3.1.5 can seem unintuitive at first, Pal et al. discuss the advantages of this approach and have developed a sparse matrix multiplication accelerator based on the outer product formulation [32]. Some of the advantages of this formulation include maximizing memory reuse as well as avoiding repeating reads to non-zero elements. Additionally, this approach minimizes the number of load and store instructions. After multiplying a column from A and the corresponding row from B using the outer product, we do not use these values again and can remove them from the cache [32].

```cpp
// assume that the matrices A, B, C of type double are defined as
// in section 3
    for (int ki = 0; ki < k; ++ki) {
      // load ki-th column/row of A/B
      double* a = load_column(A, n, k, ki);
      double* b = load_row(B, k, m, ki);
      for (int mi = 0; mi < m; ++mi) {
        for (int ni = 0; ni < n; ++ni) {
            C[ni + mi*m] += a[ni] * b[mi];
        }
      }
  }
```

Listing 3.1.2: C++ example code of an outer product based matrix multiplication

The findings in [32] regarding loads and stores can be reproduced with the code example in listing 3.1.2. The outer loop loads the ki^{th} column and row of matrices A and B, respectively, into separate arrays. The arrays a and b are loaded using two functions which take as input a matrix, the matrix dimensions, and the current state of the outer loop-variable. We will assume that calling these functions does not significantly impact the performance of the matrix multiplication. We can compare these functions with a functionality called "array slicing", which is provided by the python package "numpy". This package would allow us to load the vectors by calling $a = A[:, ki]$ and $b = B[ki, :]$. The outer product approach implies, given a sufficiently large cache, that these vectors stay in local memory while executing the intermediate loop. Additionally, these vectors are not reused in later parts of the calculation, which means that the contents of the vectors a and b can be removed from the local memory after exiting the intermediate loop.

Similar to the naive approach in listing 3.1.1, the innermost loop contains a single `FMA` instruction, accounting for $2 * n$ FLOPS for each time the loop is executed. Since the inner loop is entered $k * m$ times, the multiplication of two matrices takes a total of $2nkm$ FLOPS, thus resulting in the same amount of executed arithmetic instructions as the naive approach. However, since the columns and rows of A and B are only loaded once, we have a total of $nk + km$ load instructions. Finally, there are n stores in the innermost loop, resulting in $n * k * m$ stores to the matrix C. By including a vector c that acts as an accumulator for each column of C, we could move stores to the matrix C outside of the computation loop, thus decreasing the amount of stores to memory by a factor of n, resulting in $\mathcal{O}(k * m)$ stores. Comparing the number of instructions needed for both approaches of the matrix multiplication, we can see that the outer product formulation can provide a performance increase regarding runtime due to the lower number of memory accesses.

3.2 Storage Formats

When processing data, it is common to store it in some kind of array or matrix. This way we can easily access the stored data, divide the data that needs to be processed into subsets, or sort the values in a certain way. Naturally, the more data we have, the more memory we need in order to store it. Matrices that are used in certain applications, for example in PSpaMM which generates a matrix multiplication based on the outer product formulation seen in chapter 3.1.2, sometimes contain a lot of zeros. When using the outer product, a high number zeros stored in one of the input matrices entails that we will perform a multiplication with 0 many times, as well as store the resulting zeros into an intermediate matrix. This redundant calculation can take up a large part of the overall runtime, depending on the ratio of zeros to non-zeros. If the matrix dimensions are large enough and the amount of zeros stored exceeds a certain threshold, we can make use of different storage formats in order to exclusively store non-zero values. Matrices that are used in practical applications tend to have only up to $5-15\%$ of their entries filled with non-zero values, which usually justifies the use of a storage format. The following subsections present a summary of some of the formats that can be used.

3.2.1 Compressed Sparse Row Storage

The first storage format that is commonly used is the "compressed sparse row" (CSR) format. This storage format essentially takes a matrix and compresses it row for row, discarding the zero values while keeping the non-zeros and storing them consecutively into a new array. Because of this, we lose information about the row and column indices of the non-zeros but in return, we need to occupy less memory space to store these values. We have to introduce additional arrays which we can use to infer the row and column indices of the matrix values, so that we can continue to calculate correctly with this compressed version of the matrix.

Let A denote an arbitrary array with dimensions $m \times n$. The first additional array is used to store the column index of the non-zero values. This means that the i^{th} element of the column index array `col_ind` stores the column index of the i^{th} non-zero stored in the value array `non_zeros`. The second array that we need to introduce is called `row_ptr`. This array allows us to infer the amount of non-zero values of each row of the original matrix. To do this, we need to subtract the i^{th} element from the $(i+1)^{th}$ element. The result is the amount of non-zeros that were stored in the i^{th} row. If the i^{th} row of A contains only zeros, we have row_ptr[i] = row_ptr[i+1] and, therefore, row_ptr[i+1] $-$ row_ptr[i] $= 0$. Assuming that the original matrix A has dimensions $n \times m$ with k non-zero values, the memory space that the CSR format takes up can be calculated as follows. The first two arrays contain k elements each, whereas the third array stores $n+1$ elements. Therefore, storing the matrix A in CSR requires storing $2k+n+1$ elements, while storing the entire matrix A would take up $n * m$ memory space.

For better understanding, we will focus on a small example matrix that will be stored in CSR format. The size of the example matrix would not warrant the use of the CSR format. Nonetheless, it suffices to illustrate the storage format. A larger example can be found in [13].

$$A = \begin{bmatrix} 7 & 0 & -2 & 0 & 0 \\ 0 & 1 & 0 & 4 & 0 \\ 9 & 2 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.2.1}$$

Let the matrix A (equation 3.2.1) denote a sparse matrix with $n = m = 5$. The new arrays that are introduced using the CSR storage format are shown in table 3.2.1.

non_zeros	7	-2	1	4	9	2	8
col_ind	1	3	2	4	1	2	3
row_ptr	0	2	4	6	7		

Table 3.2.1: Arrays needed to construct a CSR version of matrix A from 3.2.1

The first array contains the non-zeros of A, whereas the second array is used to store the respective column indices. Finally, the last array helps us to calculate the number of non-zeros in each row of A. If we were to use A as the left-hand side multiplicand in a naive matrix product, we would infer the values of the first row as follows, assuming that the array indices are 1-based: we first calculate row_ptr[2] − row_ptr[1] = 2 to obtain the number of non-zeros of the first row. With this information, we now know that $a_{1,1} = 7$ and $a_{1,3} = -2$, while all other $a_{1,k} = 0$ for $k = \{2, 4, 5\}$.

3.2.2 Compressed Sparse Column Storage

The compressed sparse column (CSC) storage format works analogously to the CSR format. The only difference is that we now transform our original $n \times m$ matrix column by column and rename the third storage-array to `col_ptr`. Although the concepts remain the same for both formats, there are cases where we would prefer one storage format over the other. If A is a “tall” matrix, i.e. $n >> m$, then we could use CSC in order to keep the `col_ptr` array small. If A is a “wide” matrix however, meaning that $n << m$, we could store A using CSR. Finally, if A is square, we are free to choose one of the two formats.

To illustrate how CSC storage works, we will reuse the example from section 3.2.1. Let A denote the same matrix using 1-based indices as in equation 3.2.1. Table 3.2.2 shows the arrays constructed by CSC.

non_zeros	7	9	1	2	-2	8	4
row_ind	1	3	2	3	1	4	2
col_ptr	0	2	4	6	7		

Table 3.2.2: Arrays needed to construct a CSC version of matrix A from 3.2.1

Similar to the example in table 3.2.1, the `non_zeros` array contains the non-zero values in matrix A, the difference being that A is processed column by column. The second array stores the row index of each corresponding non-zero, whereas the third array is used to determine the amount of non-zeros for each column, similar to the CSR format.

Other comparable storage formats include the block sparse row (BSR) and block sparse column (BSC) storage formats. They work similarly to CSR and CSC. Although each sub-matrix must have the same shape, there are implementations where this constraint is relaxed. These additional storage formats are mentioned in the sparse matrix multiplication generator PSpaMM, albeit only by name [34]. Since explaining these more thoroughly would exceed the bounds of this thesis, we refer to [12] for more information.

3.2.3 Coordinate Storage

The storage format that is currently being used for sparse matrices in PSpaMM is the coordinate storage format. Similar to the other presented formats, we introduce three new arrays. The first array stores the non-zero values, whereas the other two contain the corresponding row and column indices. If the original $n \times m$ matrix has k non-zeros, the coordinate storage format now stores $3k$ elements. Therefore, using this format is only helpful in saving memory space if the ratio of total number of elements $n * m$ to the number of non-zeros k is $k \leq \frac{n*m}{3}$. As the ratio between k and nm decreases, more memory space can be saved by using this format. Naturally, if the sparse matrix contains too many non-zero values, we should continue working with the original matrix.

One practical implementation in which the coordinate format can be found is called the “Matrix Market” exchange format [29]. The goal of this format is to provide a simple standardised way of storing, exchanging, and parsing matrices. Additionally, the format can be extended to fit the needs of more complex applications. The information needed to reconstruct a sparse matrix is stored in “.mtx” files. Listing 3.2.1 shows an example mtx file constructed using the following sparse matrix:

$$A = \begin{bmatrix} 0 & 9 & -2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 4 & 0 & 0 & 0 \\ 39 & 2 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & -16 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 2 & 0 \\ 0 & 10 & 8 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.2.2}$$

```
%%MatrixMarket matrix coordinate real general
%
% comments
%
6 7 11  //<- m n nnz
1 2 9
1 3 -2
2 2 1
3 6 8
2 4 4
3 1 39
3 2 2
4 3 -16
5 2 2
6 2 10
6 3 8
5 6 2
```

Listing 3.2.1: Example .mtx file generated for the matrix A in equation 3.2.2

The first line in listing 3.2.1 is the header line. It contains information about the type of object stored in the file, the type of data stored, the storage format, as well as potential symmetry patterns. In this case, the header tells us that the file contains a matrix stored in coordinate format containing real entries. Other possible data types include integer and complex values. The symmetry pattern is denoted as "general", meaning that the matrix is asymmetric.

Lines two to four reserve space for possible comments we may want to include. The comments can consist of zero or more comment lines, meaning that if we do not want to include a comment, the reserved space is reduced to one line containing a single %-sign.

The line following the comment section stores information about the dimensions of the matrix and its number of non-zeros. In our example, the matrix represented by the mtx file has $m = 6$ rows, $n = 7$ columns, as well as $nnz = 11$ non-zero entries.

The rest of the file is nnz lines long. Each line denotes a pair of row and column indices, as well as the value stored in the corresponding matrix cell. Mtx files use 1-based indexing, meaning that the top left element of a matrix A is located in `A[1][1]`.

4 Implementation

In this section, we will explore different loop optimization techniques that are implemented in PSpaMM, as well as discuss the key differences between the two Arm generators and how we modified the NEON generator to be able to return inline assembly containing SVE instructions. Additionally, we will talk about some problems we encountered during the implementation.

4.1 Loop Optimization

PSpaMM makes use of several loop optimization techniques. These optimizations increase the performance of the generated inline assembly, for example by promoting data locality and thus reducing the number of cache misses. We will discuss these advantages and how the optimizations can be applied using an example loop (listing 4.1.1).

```
// Initialization of arrays a, b with some values
for (int i = 0; i < length; ++i) {
    a[i] = a[i] + b[i];
}
```

Listing 4.1.1: Unoptimized example loop performing a simple vector addition

Loop Tiling

Loop tiling, also known as loop blocking, is another way of transforming the iteration space of a loop [3]. This optimization divides a loop into blocks of equal size by splitting it into two separate loops. The outer loop iterates over the number of blocks, whereas the inner loop processes an individual block. In the case of our SVE-based generator, the block size used for loop tiling is chosen such that it divides the iteration space perfectly. This ensures that we do not have to process an otherwise existing overhead in case the last block does not fully fit the end of the iteration space. Furthermore, loop tiling can hint to the compiler that the loop body can be optimized using vector instructions.

```
// make sure that the array lengths are divisible by the block size
int blocksize = 5
assert((length % blocksize) == 0)
for (int ii = 0; ii < length; ii += blocksize) {
  for (int i = ii; i < ii + blocksize; ++i) {
    a[i] = a[i] + b[i];
  }
}
```

Listing 4.1.2: Loop tiling with a block size of 5

Listing 4.1.2 shows an optimized version of our example loop using loop tiling. The outer loop iterates over the whole length of the arrays in steps of size `blocksize`, effectively fixating the block we are going to process. The inner loop then processes an individual block. Generally, we apply loop tiling to increase data locality and cache reuse [3], which consequently reduces the amount of time consuming cache misses. The resulting performance boost can vary drastically depending on the chosen block size [26]. The best performing block size is hard to determine as it highly depends on the processor's caches and the memory accesses performed in the loop [26]. Nevertheless, finding the right block size can be a key factor to ensure high performances, especially when multiplying dense matrices [3].

Loop Unrolling

The first optimization is called "loop unrolling". For this technique, we take a part of the iteration space of a loop and replace it with explicit instructions [3]. This is especially advantageous when the loop body is particularly small and the number of iterations is high. This holds true for our example loop if we set $n = 100,000$. If the loop body contains only a few simple instructions, the majority of the execution time is spent incrementing the iteration variable and performing "end-of-loop" checks. Loop unrolling allows for a higher number of executed instructions before incrementing the loop variable and testing whether we have reached the end of the loop [17].

```
// make sure that the array lengths are divisible by the unroll factor
assert((length % 5) == 0)
for (int i = 0; i < length; i += 5) {
    a[i] = a[i] + b[i];
    a[i+1] = a[i+1] + b[i+1];
    a[i+2] = a[i+2] + b[i+2];
    a[i+3] = a[i+3] + b[i+3];
    a[i+4] = a[i+4] + b[i+4];
}
```

Listing 4.1.3: Loop unrolling with an unroll factor of 5

Listing 4.1.3 contains our example loop after being unrolled by a factor of 5. We now perform 5 arithmetic instructions per loop iteration. Thus, we only need to increment i and perform end of loop checks a total of $\frac{\text{length}}{5}$ times. Compilers can also further optimize the loop by pre-calculating the offsets of every memory access within the loop body. This leads to compilers being able to generate more efficient code [25]. Similar to loop tiling, unrolling a loop can provide a hint to the compiler that the loop body can be optimized using vectorization, i.e. replacing the instructions with "single instruction multiple data" (SIMD) instructions [17].

A larger unroll factor implies that fewer branch instructions need to be performed when executing the loop [25]. Since branch instructions are rather expensive compared to the simple arithmetic instructions in the loop body, we can expect a performance boost. Additionally, compilers are able to fully unroll a loop. This is presumably the most efficient way to traverse a loop, since all potential branch and loop-control-related instructions are removed. However, fully unrolling a loop comes with a disadvantage. The size of the file containing the loop grows linearly with the unroll factor due to the increasing length of the loop body. If we were to completely unroll the example loop while setting $length = 100.000$, the file would grow by 100.000 lines of code. Nevertheless, this is not a problem in the case of files generated by PSpaMM, because the matrices used in the multiplications are rather small (matrix dimensions < 100) and we are not restricted by file size.

Register Blocking

Another loop optimization implemented in PSpaMM is called "register blocking". It works similarly to loop tiling. However, the block size is chosen with the number of available registers in mind instead of the cache size [25]. Additionally, we unroll the inner computation loop of a block and ideally increase data reuse within the unrolled loop [25]. Since we would not see the benefits of register blocking when applying the technique to our example loop, we will look at the following excerpt of a file generated by PSpaMM.

```
// Block GEMM microkernel
  // Load A register block @ (d=0,r=0)
  "ld1d z2.d, p7/z, [x0, 0, MUL VL]\r\n"          // A [0,0] [0,0]
  "ld1d z3.d, p7/z, [x0, 1, MUL VL]\r\n"          // A [0,0] [8,0]
"ld1rd z4.d, p7/z, [x1, 0]\r\n"                 // B[0,0][0,0]
"add x11, x1, #544\r\n"                         // move to next element of B
"ld1rd z5.d, p7/z, [x11, 0]\r\n"                // B[0,0][0,1]
"add x11, x1, #1088\r\n"                        // move to next element of B
"ld1rd z6.d, p7/z, [x11, 0]\r\n"                // B[0,0][0,2]
"add x11, x1, #1632\r\n"                        // move to next element of B
"ld1rd z7.d, p7/z, [x11, 0]\r\n"                // B[0,0][0,3]
"add x11, x1, #2176\r\n"                        // move to next element of B
"ld1rd z8.d, p7/z, [x11, 0]\r\n"                // B[0,0][0,4]
"add x11, x1, #2720\r\n"                        // move to next element of B
"ld1rd z9.d, p7/z, [x11, 0]\r\n"                // B[0,0][0,5]
"add x11, x1, #3264\r\n"                        // move to next element of B
"ld1rd z10.d, p7/z, [x11, 0]\r\n"               // B[0,0][0,6]
"add x11, x1, #3808\r\n"                        // move to next element of B
"ld1rd z11.d, p7/z, [x11, 0]\r\n"               // B[0,0][0,7]
"fmla z16.d, p7/m, z2.d, z4.d\r\n"          // C[0:8,0] += A[0:8,0]*B[0,0][0,0]
"fmla z18.d, p7/m, z2.d, z5.d\r\n"          // C[0:8,1] += A[0:8,0]*B[0,0][0,1]
"fmla z20.d, p7/m, z2.d, z6.d\r\n"          // C[0:8,2] += A[0:8,0]*B[0,0][0,2]
"fmla z22.d, p7/m, z2.d, z7.d\r\n"          // C[0:8,3] += A[0:8,0]*B[0,0][0,3]
"fmla z24.d, p7/m, z2.d, z8.d\r\n"          // C[0:8,4] += A[0:8,0]*B[0,0][0,4]
"fmla z26.d, p7/m, z2.d, z9.d\r\n"          // C[0:8,5] += A[0:8,0]*B[0,0][0,5]
"fmla z28.d, p7/m, z2.d, z10.d\r\n"         // C[0:8,6] += A[0:8,0]*B[0,0][0,6]
"fmla z30.d, p7/m, z2.d, z11.d\r\n"         // C[0:8,7] += A[0:8,0]*B[0,0][0,7]
"fmla z17.d, p7/m, z3.d, z4.d\r\n"          // C[8:16,0] += A[8:16,0]*B[0,0][0,0]
"fmla z19.d, p7/m, z3.d, z5.d\r\n"          // C[8:16,1] += A[8:16,0]*B[0,0][0,1]
"fmla z21.d, p7/m, z3.d, z6.d\r\n"          // C[8:16,2] += A[8:16,0]*B[0,0][0,2]
"fmla z23.d, p7/m, z3.d, z7.d\r\n"          // C[8:16,3] += A[8:16,0]*B[0,0][0,3]
"fmla z25.d, p7/m, z3.d, z8.d\r\n"          // C[8:16,4] += A[8:16,0]*B[0,0][0,4]
"fmla z27.d, p7/m, z3.d, z9.d\r\n"          // C[8:16,5] += A[8:16,0]*B[0,0][0,5]
"fmla z29.d, p7/m, z3.d, z10.d\r\n"         // C[8:16,6] += A[8:16,0]*B[0,0][0,6]
"fmla z31.d, p7/m, z3.d, z11.d\r\n"         // C[8:16,7] += A[8:16,0]*B[0,0][0,7]
```

Listing 4.1.4: Excerpt of inline assembly generated by PSpaMM

Listing 4.1.4 contains part of the generated inline assembly for multiplying two dense 32×32 matrices A and B and adding the result to a matrix C. Comments that contain matrix accesses with two sets of brackets use the following naming convention. The first pair of brackets denotes the absolute coordinates of the top left element of a matrix block. This is used as a starting point to process the block. The second set of brackets indicates the coordinates of an element relative to the first element of a block. Registers p0 – p7 represent the SVE predicates. The position of the vector registers used for `FMLA` instructions have the following meaning: the first register acts as the addend as well as the result vector, whereas the other two vector registers are multiplied and added to the result register.

The excerpt shows that register blocking aims to reuse data loaded into registers as much as possible. While registers z16 to z31 were filled with values from C earlier in the file, we can see that the registers used to hold values from A are reused 8 times in this block micro-kernel, while vectors containing elements from B are reused 2 times. Although not shown to keep the example short, vectors storing the results are reused several times in different "block gemm microkernels" before they are stored back into main memory. This technique helps us avoid redundant data accesses, therefore allowing faster processing of matrix elements.

4.2 Key Differences of SVE Generator

Although both the NEON and SVE ISA are part of the Arm family of RISC architectures, rewriting the NEON-based generator such that it returns SVE instructions is not a trivial task. Since we cannot perform a one-to-one mapping from NEON instructions to SVE instructions, we first had to examine in more detail how the NEON generator approaches creating the inline assembly. Then, we were able to select the relevant SVE instructions and rewrite parts of the generator where necessary in order to have it return correct SVE inline assembly. In this subsection, we will discuss the key aspects in which the SVE generator differs from its NEON counterpart, how we implemented necessary changes, and how we solved problems that occurred.

Vector register and block size

One of the major changes that SVE brings is the vector register size. While SVE allows processors to implement the ISA for a vector length between 128 and 2048 bits, the A64FX comes with a maximum vector size of 512 bits. Compared to the 128 bit NEON vector registers, we can store four times more elements in a single vector. Naturally, this means that a SVE register can now hold up to 8 double precision and 16 single precision elements, while NEON registers can only store 2 double and 4 single precision values. The change in vector length also affects the largest possible blocks we can use to process matrices. These blocks are defined by the block size parameters bm and bn, which denote the dimensions of the $bm \times bn$ submatrices of C. These parameters also determine how many elements of a column and row of A and B, respectively, we need to process. The block size, with $bm * bn$ being as large as possible, is determined with respect to the following inequality with v_size being the maximum number of elements a vector can hold: $(bn + bk) * (bm/\text{v\_size}) + bn + 2 <= 32$. Since SVE provides 32 registers, the left hand side of the equation must not exceed 32. Otherwise, some registers would need to be loaded multiple times during one “block gemm microkernel” as seen in section 4.1, defeating the purpose of register blocking.

Predicate registers

As discussed in section 2.3, the predicates introduced by SVE enable us to take a VLA approach to rewriting the generator. We realised that we can avoid using “predicate counted loop” instructions like `whilelo`, which are usually needed to dynamically determine how many vector elements a predicated instruction is supposed to use. Instead, we statically initialized all predicate registers `p0` to `p7` using the following naming convention: the register `pk` would have its first $k + 1$ elements set to `true`. The remaining elements are set to `false`. Let $k \in \{0, ..., 7\}$. The instruction `ptrue pk.d, VL(k+1)` initializes a predicate register according to the introduced naming convention. This way, we can omit using `whilelo` every time we start processing a block.

In the end, we decided to implement a different naming convention after realising that the block size used to generate a file does not change. Therefore, we only need to initialize either one or two predicate registers as follows. The first predicate we need is the “all-true” predicate. This predicate is always needed in our implementation as it is used to load an element of B and broadcasting it into all elements of a vector. If bm is at least 8, we use the all-true predicate to load a whole vector with elements of A or C where necessary, as seen in the code excerpt 4.1.4. The second predicate register we may need is the ”overhead” predicate. Let $k = bm \bmod v\_size$. Then, if $k \neq 0$, the overhead predicate will have its first k elements set to true. The all-true and overhead predicates are always initialized as `p7` and `p0` respectively, using `ptrue p7.d, ALL` and `ptrue p0.d, #k`.

The new naming convention allows us to extend the generator to single precision more easily. The old naming convention would make it necessary to use all 16 predicate registers. However, arithmetic SVE instructions only allow `p0` to `p7` to be used as valid predicates. Therefore, we would have been forced to include “predicate counted loop” instructions in the case of single precision multiplications, which is why we decided to use the new naming convention.

Broadcasting scalar values

Unlike the NEON ISA, the FMLA instruction provided by SVE does not allow for the multiplication of a vector and a scalar value [1]. This means that we need to broadcast the scalar elements loaded from the matrix B into vector registers before we can multiply them with a vector of A using FMLA. Although there are instructions we could use to explicitly broadcast a scalar value into a vector, this would entail executing one broadcasting instruction for every scalar loaded from B. Alternatively, we decided to use the `ld1rd` instructions. This instruction loads an element from memory and simultaneously broadcasts it into a vector register. This means that we need to execute one less instruction every time we process an element of B.

Loads and stores

The logic behind computing the immediate offsets used to load and store values from memory is similar to the NEON generator. One difference is that SVE does not offer "store pair" or "load pair" (`stp/ldp`) instructions. Nevertheless, we do not need an alternative to these instructions, because the A64FX allows us to load/store up to 8 double precision values at once, whereas ldp/stp can only load/store 4 double precision values. In the case of single precision, SVE can load/store 16 elements, whereas NEON can only load/store 8 elements.

Another key difference between SVE and NEON memory accesses is the way immediate offsets are used. The instructions ld1d/st1d load/store consecutive elements according to the true/false values stored in the utilized predicate. Instead of passing the immediate offsets directly to the memory access instructions, we need to divide the offset by the number of bytes that a vector register takes up in the main memory. For the A64FX, a vector is 64 bytes long. If we define $k = \text{offset}/64$, we can execute loads using `ld1d zn.d, pm/z, [xj, k, MUL VL]` with MUL VL indicating that k is multiplied by the vector length in bytes before being added to the adress stored in the general purpose register `xj`. The only restriction for k is $k \in \{-8, ..., 7\}$. If this is not the case, we need to explicitly increment `xj` before we can access the main memory. The same restrictions apply to the store instruction `st1d` [1].

4.3 Compiler Related Errors

While testing the SVE-based generator, we came across a problem that seems to be related to specific versions of the gcc compiler. When using gcc 8.4.1, which represents the default on the BEAST cluster, the test suite we provide is executed without errors. However, using newer versions of gcc breaks a few test cases. In particular, we discovered that using gcc 10.2.1 and gcc 11.0.0 leads to a segmentation fault which did not occur when using gcc 8.4.1. We can use gdb to see where the segmentation fault occurs. Some of the information provided in the tables below is highlighted to increase readability.

```
Program received signal SIGSEGV, Segmentation fault.
gemm_ref (C=0x4dc400, B=0x4c6840, A=0x4c35c0, BETA=0.20000000000000001, ALPHA=0, LDC
    =32, LDB=2576980378, LDA=32, K=50, N=80, M=32) at testsuite.cpp:73
73      C[row + col*LDC] += ALPHA * A[row + s*LDA] * B[s + col*LDB];
```

Listing 4.3.1: Segmentation fault as reported by gdb

As seen in listing 4.3.1, the segmentation fault occurs when we try to calculate $C = C + AB$. Using backtracking, i.e. the "bt" command provided by gdb, we receive more details about which functions were called before the error occurred.

```
#0  gemm_ref (C=0x4dc400, B=0x4c6840, A=0x4c35c0, BETA=0.20000000000000001, ALPHA=0,
    LDC=32, LDB=2576980378, LDA=32, K=50, N=80, M=32) at testsuite.cpp:73
#1  post (M=M@entry=32, N=N@entry=80, K=K@entry=50, LDA=LDA@entry=32, LDB=2576980378,
    LDB@entry=50, LDC=LDC@entry=32, ALPHA=ALPHA@entry=0, BETA=BETA@entry
    =0.20000000000000001, A=A@entry=0x4c35c0, B=B@entry=0x4c6840, C=C@entry=0x4e1440,
     Cref=Cref@entry=0x4dc400, DELTA=DELTA@entry=9.9999999999999995e-08) at testsuite
    .cpp:156
#2  0x0000000000403ad8 in main () at testsuite.cpp:357
```

Listing 4.3.2: Further information using gdb backtracking

In listing 4.3.2, three functions are called before the segmentation fault occurs. From the bottom up, the first one is `main()` which is responsible for executing the test cases. The `post()` function is used to compare our generated result with a reference result. Finally, `gemm_ref()` calculates the reference result used in post(). The information provided by gdb tells us that upon entering post(), the parameter LDB was passed with a value of 50 (`LDB@entry=50`). The error occurs after the call to our generated inline assembly function and the call to the calculation of the reference solution for the same test case. Between these two calls, the value of the parameter LDB is altered, leading to faulty memory accesses when calculating the reference solution. Although the issue seems obvious, we can not explain why the value changed. We can see in listing 4.3.3 that between entering post() and calling gemm_ref(), the value of LDB is only adjusted if LDB is 0. However, the value LDB should be changed to is not the same as the value that is passed when calling gemm_ref(), as seen in listing 4.3.2.

```cpp
int post(unsigned M, unsigned N, unsigned K, unsigned LDA, unsigned LDB, unsigned
    LDC, double ALPHA, double BETA, double* A, double* B, double* C, double* Cref,
    double DELTA) {
    if(LDB == 0) {
        LDB = K;
    }

    gemm_ref(M, N, K, LDA, LDB, LDC, ALPHA, BETA, A, B, Cref);

    for(int i = 0; i < M; i++) {
      for(int j = 0; j < N; j++) {
        //test if the difference between the result C and reference      Cref
    exceeds a threshold DELTA
        if(std::abs(C[i + j * LDC] - Cref[i + j * LDC]) > DELTA)
          return 0;
    return 1;
}
```

Listing 4.3.3: post() function used to calculate the reference and compare with our solution

LDB represents the leading dimension of the input matrix B and is necessary to correctly access a matrix that was transformed from a two-dimensional matrix into a one-dimensional matrix. For a $m \times n$ matrix, the leading dimension ldb must be at least as large as n, therefore $ldb \geq n$ must evaluate to true. When converting a two-dimensional matrix `B` into a one-dimensional array `b`, n elements of a row of B are consecutively stored into ldb elements of the new array b. This means that the $m \times n$ matrix is converted into an array of size $m * ldb$. The new array b can contain more values than B if $ldb > n$. However, these cells are not addressed when accessing elements of b, because accessing element `B[i][j]` is equivalent to accessing `b[i*LDB + j]`. Placeholder values stored in the transformed matrix

$$b = \underbrace{[\overbrace{B_{1,1}, \dots, B_{1,n}, *, \dots, *}^{\text{ldb}}, \overbrace{B_{2,1}, \dots, B_{2,n}, *, \dots, *}^{\text{ldb}}, \dots, \overbrace{B_{m,1}, \dots, B_{m,n}, *, \dots, *}^{\text{ldb}}]}_{\text{ldb*m}}$$

are denoted by a '*' when $ldb > n$.

When using gcc versions higher than 8.4.1, including an optimization level higher than `-O1` led to the aforementioned problem of the parameter LDB being altered. Using `-O1` or lower in the makefile resulted in the testsuite being executed without any parameters changing their values. We managed to further narrow down the source of our problem and found that the compiler flags `-fschedule-insns` and `-fschedule-insns2` are likely to be the root cause. Compiling the tests using the optimization level `-O2` while explicitly deactivating the two flags, i.e. including `-fno-schedule-insns` and `-fno-schedule-insns2`, the issue disappeared and the tests were executed without errors. Another way to fix this problem is to include the extended asm qualifier `__inline__` in our generated inline assembly as described in [18]. By adding this qualifier, the error disappears without us having to explicitly disable the aforementioned compiler flags.

Unfortunately, we cannot be certain where the error lies. We ultimately decided to include the `__inline__` qualifier at the start of our generated inline assembly, as we find this solution easier and more compact compared to disabling certain compiler flags.

5 Results

In this chapter, we will discuss our measured results and evaluate whether extending PSpaMM to generate SVE inline assembly was successful. For this, we measured the executed floating point operations per seconds (flop/s) for both SVE and NEON generators. Aditionally, we benchmarked matrix multiplication functions generated by `LIBXSMM` [11], a library that specializes in, among other things, dense matrix operations. The matrices used for the benchmarks are dense and sparse square matrices. During this chapter, we refer to the NEON version and the SVE version of PSpaMM as the "NEON generator" and "SVE generator", respectively. The LIBXSMM release we used for the performance measurements is version `1.16.3`. The benchmarks containing the different kernels are compiled using gcc 11.0.0 and the following optimization flags:

```
// NEON
-std=c++17 -Ofast -march=armv8.2-a -mcpu=a64fx
// SVE and LIBXSMM
-std=c++17 -Ofast -march=armv8.2-a+sve -mcpu=a64fx -msve-vector-bits=512
```

Listing 5.0.1: Compiler flags used for benchmarks

5.1 Simple Benchmarks With PAPI

In order to accurately test the performances of the aforementioned GEMM generators, we decided to use the "Performance Application Programming Interface", commonly referred to as PAPI, for our benchmarks. A guide on how to properly set up the application can be found in [30]. PAPI provides a simple interface that helps us read performance counters for a range of hardware systems. Additionally, PAPI enables us to specify code regions that we want to monitor instead of having to measure the performance of an entire program.

```
Available PAPI preset and user defined events plus hardware information.
--------------------------------------------------------------------------------
PAPI version             : 6.0.0.1
...
output shortened to increase readability
...
--------------------------------------------------------------------------------
================================================================================
  PAPI Preset Events
================================================================================
    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
...
output shortened to increase readability
...
PAPI_FP_OPS  0x80000066  Yes   Yes  Floating point operations
PAPI_SP_OPS  0x80000067  Yes   Yes  Floating point operations; optimized to count
    scaled single precision vector operations
PAPI_DP_OPS  0x80000068  Yes   Yes  Floating point operations; optimized to count
    scaled double precision vector operations
PAPI_VEC_SP  0x80000069  No    No   Single precision vector/SIMD instructions
PAPI_VEC_DP  0x8000006a  No    No   Double precision vector/SIMD instructions
PAPI_REF_CYC 0x8000006b  No    No   Reference clock cycles
--------------------------------------------------------------------------------
Of 108 possible events, 36 are available, of which 16 are derived.
```

Listing 5.1.1: Information received by calling `papi_avail`

PAPI provides numerous preset events, a set of processor events that are commonly used to fine-tune the performance of an application. We receive information about the current hardware and the performance counters that PAPI can process by executing `./papi/src/install/bin/papi_avail`. Although not all counters are available for the A64FX, we can still use PAPI to count the number of executed floating point operations, as seen in listing 5.1.1.

The benchmarks are generated using a python script that returns a c++ wrapper program which is responsible for setting up the matrices, as well as initializing PAPI and executing the functions we want to monitor. Listing 5.1.2 shows how we measure the performance of a single matrix multiplication.

```
#include <papi.h>
/* multiple function and variable declarations declarations */
  // initialize the PAPI library
  int retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval!=PAPI_VER_CURRENT) { // handle error }
  // setup matrix pointers
  /* zero the accumulators */
  acc_real_time = 0.0; acc_proc_time = 0.0; acc_mflops = 0.0; acc_flpops = 0;
  /* execute our generated GEMM function <repetitions> times */
  for (int i = 0; i < repetitions; ++i) {
    // copy pointers for benchmarking purposes
    /* start measuring flops counter here */
    if ( (retval = PAPI_flops_rate(PAPI_FP_OPS, &real_time, &proc_time, &flpops, &
    mflops)) < PAPI_OK ) { // handle error }
    // execute our generated functions
    /* stop measuring flops counter here */
    if ( (retval = PAPI_flops_rate(PAPI_FP_OPS, &real_time, &proc_time, &flpops, &
    mflops)) < PAPI_OK ) {  // handle error }
    /* accumulate measurements */
    acc_real_time += real_time; acc_proc_time += proc_time; acc_mflops += mflops;
    acc_flpops += flpops;
    // free copied pointers
  }
  // compare result of generated function with a reference result
  // free remaining pointers and write measurements averaged for repetitions into a
    .csv file
```

Listing 5.1.2: Benchmarking of a single generated matrix multiplication

First, we need to initialize the PAPI library using `PAPI_library_init()` before calling any other functions provided by PAPI. The initialization also tests if the installed PAPI version is up to date. To monitor the performance of our code, we would normally use low-level events that count for example the accesses to different cache levels. Since we only want to measure the flop/s performed by the A64FX, we can make use of a high-level function that simplifies the process of reading the "floating point operations" counter, namely `PAPI_flops_rate()`. This function can be used to measure the amount of executed double or single precision floating point operations, as well as all types of floating point operations. Before entering the repetition loop, we introduce different accumulator variables to store intermediate results. The repetition loop executes our generated function multiple times. This helps us reduce measurement errors that could occur if we were to execute our generated function only once. We surround our function with two separate calls to `PAPI_flops_rate()` to measure the executed flops within this code region. Afterwards, we add the measurements onto accumulators, because calling `PAPI_flops_rate()` in the next loop iteration resets the measurement variables. After exiting the loop, we clean up the pointers, compare the result of our function to a reference result, and write the measurements done by PAPI into a csv file.

The aforementioned benchmarks were performed on the A64FX provided by the BEAST cluster. The matrices used for dense-by-dense measurements are filled with randomized values between 0.00001 and 1000 [34]. For dense-by-sparse multiplications, we pass a mtx file that contains information about the sparse matrix. If the file does not exist, we generate one which we can use to make sure that redundant executions of the same benchmarks yield the same results. As a result, the sparsity patterns are created at random. We take the average of the measured results by dividing them by the parameter `repetitions`. To measure the highest possible performance, the input matrices A and B are kept in the cache for the duration of the repetition loop.

5.2 Benchmarking Results

While testing the performance of the SVE-based generator, we came across an additional restriction caused by the fixed size of ARM instructions. If the size of the generated files exceeds a certain threshold, we receive the following error message when trying to compile the benchmarks: `/tmp/ccGOHLkk.s: 1260609: Error: conditional branch out of range`. This error occurs when we try to branch to a label that is too far away from the current position of the program counter. According to the Arm documentation [2], the range of a conditional branch instruction is restricted to ±1 MB. Since all Arm instructions have a fixed size of 4 bytes, we may run into this problem when generating matrix multiplications for large matrices. Due to loop unrolling, the generated file size can be rather large. If the loop cannot be fully unrolled, the branch instruction at the end of the function has to be executed. This branch would move the program counter to a label at the top of the function, which then might exceed the maximum allowed offset. Nevertheless, we have encountered this problem only using square matrices with dimensions $m, k, n > 176$. Matrices used in PSpaMM usually do not exceed $m, k, n <= 100$, which is why we do not think that this issue is relevant in practice. Nevertheless, choosing block sizes that are smaller than the ones returned by `max_arm_sve.py` may very well solve this problem, because we noticed during testing that smaller block sizes typically lead to shorter files.

While benchmarking generated functions containing SVE instructions, we noticed that the total number of executed flops often did not match our expectations. To be more precise, if the dimensions of the matrices used during a benchmark were not a multiple of 8, i.e. the SVE vector length for the A64FX, PAPI would report an excess of flops. For example, when we generate a function for matrices with $m = k = n = 2$, we know that the processor executes 16 arithmetic instructions as discussed in section 3.1.1. However, PAPI reports that 64 flops were executed. This "overcounting" is caused by the fact that the performance counter is incremented without taking the number of active elements in the vector registers into account. Instead, the counter is incremented under the assumption that the whole vector register is active. This means that for every FMLA instruction, the flop counter is incremented by 16. To obtain results that are not affected by this overcounting, we decided to increase the sizes of the matrices used in the SVE benchmarks in steps of 8. Matrix sizes in NEON benchmarks, however, are incremented in steps of 2, because the NEON related measurements were not affected by this.

First, we measured the A64FX's performance using PSpaMM's NEON-based generator with matrix dimensions increasing in steps of 2 to set a baseline for later performance comparisons. Figure 5.2.1 shows that for dimensions between 2 and 64, the performance scales well to a peak of about 120 GFLOP/s. Afterwards, the measured performance starts oscillating between 110 and 130 GFLOP/s up until matrix dimensions of 96. From this point on, the performance becomes even more volatile. The measurements then vary in a

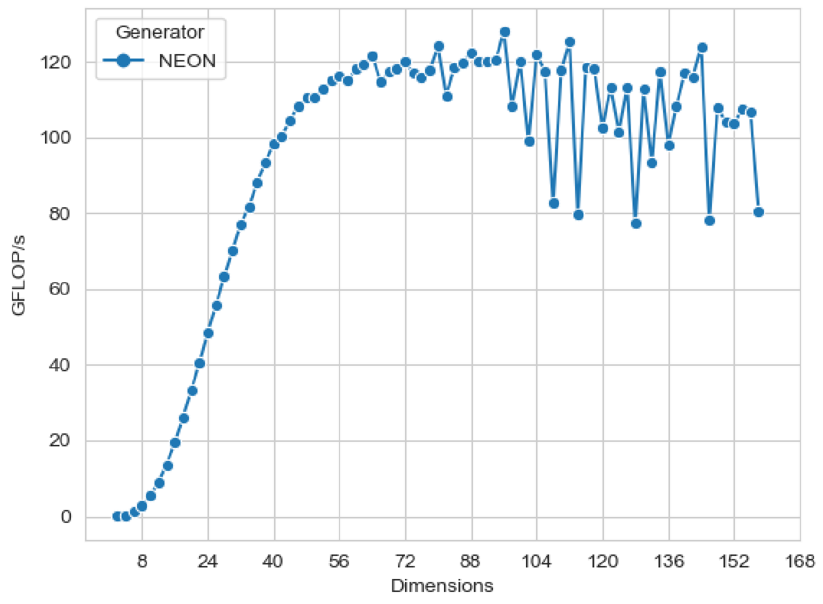

Figure 5.2.1: Scaled performance of PSpaMM's NEON generator on A64FX

range of 80 and 130 GFLOP/s. In the worst case, we observe a performance loss of about 39% between two tests. The peak performance of the NEON generator is approximately 130 GFLOP/s and was measured for a matrix dimension of 96. Similar results were obtained during an earlier thesis about matrix multiplication kernels on Arm architectures [37].

Figure 5.2.2 shows the results of PSpaMM's performance using SVE-based multiplication kernels. This time, we tested the performance of our new SVE generator using matrices with increasing dimensions in steps of 8. The results scale well for matrix dimensions between 8 and 104, with an outlier at 80. For larger matrix sizes, the performance starts to vary heavily between 2 tests, ranging between 600 and 800 GFLOP/s. We observed the most significant performance loss to be about 25% moving from `dimensions=136` to `dimensions=144`. We measured the peak performance of our SVE generator to be approximately 800 GFLOP/s for test cases where the matrix dimensions were set to 104, 120, or 136.

In order to determine how well our SVE extension for PSpaMM holds up compared to other more specialized math libraries, we additionally benchmarked DGEMM functions generated by LIBXSMM. Although the library primarily targets Intel architectures [11], we confirmed that the library was also able to generate multiplication kernels using Arm SVE. This was done by executing `objdump` on a binary-dump of the generated kernels, as described in [11]. We included an excerpt of the assembly provided in the object dump in listing 5.2.1. Additionally, we tested whether the computations performed in the generated kernels were correct. We found that for the test cases used in our benchmarks, the results

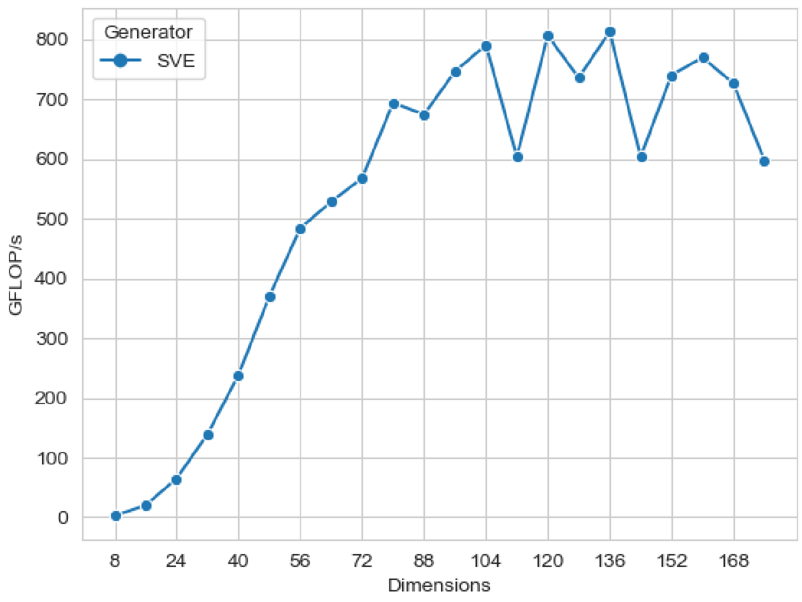

Figure 5.2.2: Scaled performance of PSpaMM's SVE generator on A64FX

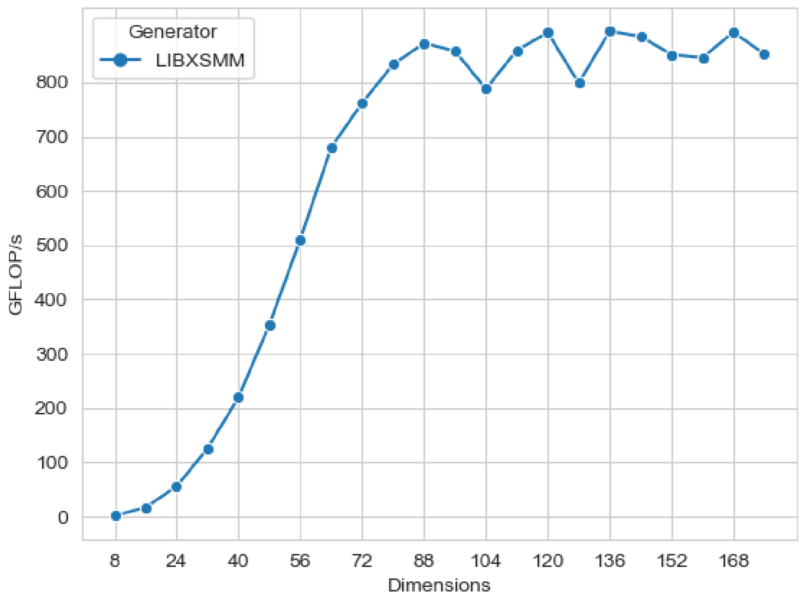

Figure 5.2.3: Scaled performance of LIBXSMM on A64FX

calculated by LIBXSMM kernels were correct for all dimensions, except for the very last test where `dimension=176`. Nevertheless, we decided to keep the results of measuring LIBXSMM kernels to provide an additional application to compare our SVE generator to.

```
command: objdump -D -b binary -m aarch64 -M reg-names-atpcs <filename>.mxm
  ...
  cc:   85804001        ldr     z1, [x0]
  d0:   91010000        add     x0, x0, #0x40
  d4:   85c0e020        ld1rd   {z0.d}, p0/z, [x1]
  d8:   91010021        add     x1, x1, #0x40
  dc:   65e00038        fmla    z24.d, p0/m, z1.d, z0.d
  e0:   85c0e020        ld1rd   {z0.d}, p0/z, [x1]
  e4:   91010021        add     x1, x1, #0x40
  e8:   65e00039        fmla    z25.d, p0/m, z1.d, z0.d
  ec:   85c0e020        ld1rd   {z0.d}, p0/z, [x1]
  f0:   91010021        add     x1, x1, #0x40
  f4:   65e0003a        fmla    z26.d, p0/m, z1.d, z0.d
  f8:   85c0e020        ld1rd   {z0.d}, p0/z, [x1]
  fc:   91010021        add     x1, x1, #0x40
 100:   65e0003b        fmla    z27.d, p0/m, z1.d, z0.d
```

Listing 5.2.1: Excerpt of SVE assembly generated by LIBXSMM

Figure 5.2.3 shows that the performance of the LIBXSMM kernels develop similarly to our SVE generator. The performance scales well with the matrix size increasing from 8 to 88. For larger matrix dimension, the measured performance starts to vary between 800 and 900 GFLOP/s, representing performance differences of up to 11% between matrix sizes. Finally, the variation in performance stabilizes by a small amount starting at matrix sizes of 136. For matrices of this size, the performances vary between 840 and 900 GFLOP/s.

5.3 Interpretation

To compare the results more easily, we have combined the measurements of each generator in figure 5.3.1. The first thing we notice is that the NEON based functions can hold up performance-wise for matrix sizes up to 16×16. For matrices with dimensions of 24 or larger, both the SVE generator and LIBXSMM outperform the NEON version by a large margin. When comparing NEON's peak performance, LIBXSMM and our SVE generator both outperform it by a factor of roughly 7 and 6.3, respectively. We expected both generators to be faster than the NEON version, especially for larger matrix sizes, simply because the A64FX implements SVE for a vector length of 512 bits, i.e. 4 times the vector length of NEON registers. The results are still surprising since a performance boost this large implies that the vector size is not the only reason. We can assume that the ability to process matrices in larger blocks also boosts the performance, as discussed in section 4.1.

For matrices larger than 64×64, we observe that the NEON generator reaches a performance ceiling. The main reason for the plateauing performance is that the matrices used for the benchmarks do not completely fit into the L1 cache for dimensions larger than 64. Since the parameter β is set to 0 for all benchmarks, the DGEMM equation is simplified to $C_{out} = \alpha * AB + \beta * C_{in} = \alpha AB$. We never load values of C from memory into registers, meaning that only A and B have to fit into the L1 cache. The L1d cache has a size of 64

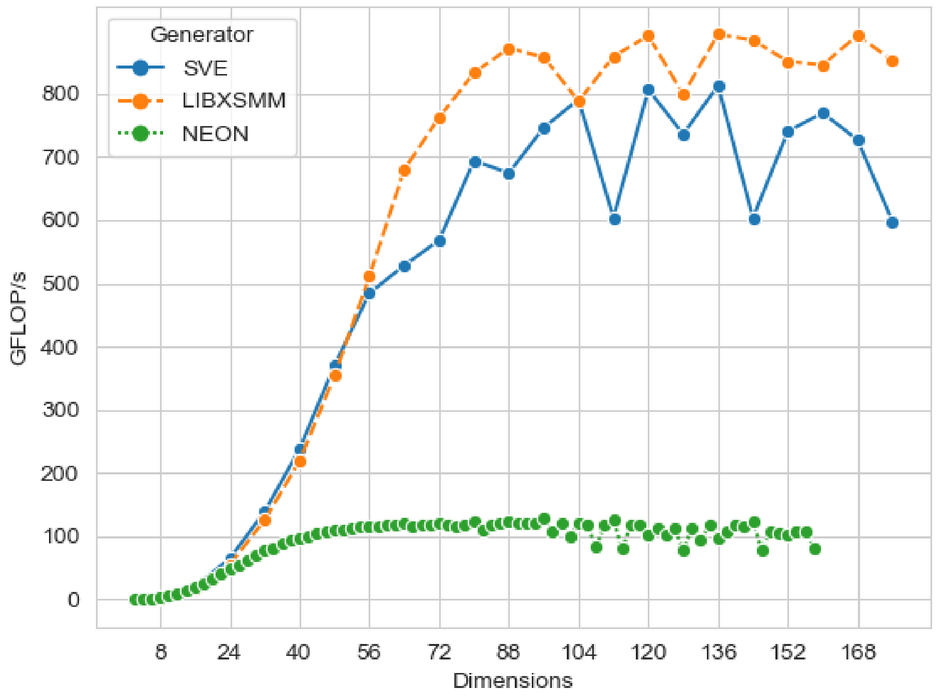

Figure 5.3.1: Combined results of NEON, SVE, and LIBXSMM

KiB/core and is able to hold a total of 8192 double precision values. Therefore, we can fill the L1 cache with two 64×64 matrices. Larger matrices have to be partially stored in the L2 cache first. Another factor that might lead to the observed performance ceiling may be the size of the L1 instruction cache. We assume that the generated instructions do not fit into the L1i cache anymore and have to be partially stored in the L2 cache. The L1i cache of the A64FX is 64 KiB large per core, whereas Arm instructions are 4 byte long. Therefore, we can fit 2^{14} instructions into the instruction cache. However, the Arm documentation specifies that the last 2 bits of all Arm instructions have to be 0, which could entail that the L1i cache can fit 2^{16} instructions. The specification also notes that sometimes the least significant bit (LSB) is used to distinguish Arm and Thumb instructions, which would mean that we can only fit 2^{15} instructions into the L1i cache. In the end, we cannot say with certainty whether the size of the instruction cache impacts the performance of the NEON kernels.

The performances for the SVE generator and LIBXSMM show a similar pattern for small matrix sizes. For dimensions up to 64×64, we observe a steady increase in performance. For larger matrices, the data does not fit into the L1 cache anymore, resulting in a lower performance increase for larger problem sizes. The measurements for the SVE generator seem to be an exception to the steady increase in performance for matrix sizes up to 64×64. For this particular problem size, the relative performance gain compared to the preceding problem size is lower than expected, although the data should still fit into the L1 cache. We assume that this is caused by a sub-optimal choice in block size for our SVE generator.

Both the LIBXSMM kernels and our SVE generator hit a performance ceiling for large matrices starting at 104×104. Interestingly, the SVE generator achieves the same performance as the LIBXSMM kernel for 104×104 matrices. The SVE generator and LIBXSMM kernels achieve a peak performance of 813 GFLOP/s and 894 GFLOP/s, respectively, for problem sizes of 120×120 and 136×136. For large matrices in general, we can observe that both sets of benchmarks exhibit a similar performance pattern starting at matrices of size 120×120. In the case of our PSpaMM kernels, we noticed during testing that for some matrix dimensions, the parameter `bm` of the block size chosen by PSpaMM (see section 4.2) was significantly smaller than for the previous test case. Therefore, we can assume that the drastic performance losses for larger matrices are caused by a sub-optimal choice of block size. The same is presumably true for the drops in performance seen in the LIBXSMM kernels. Unfortunately, we cannot see which block sizes are chosen for LIBXSMM kernels, meaning that we can only make assumptions as to why these drops occur. One possible explanation for the different performances may be that LIBXSMM chooses more appropriate block sizes for its computation kernels than PSpaMM.

Another possible reason might be that SVE instructions are ordered differently when using our generator compared to LIBXSMM kernels. We noticed that the instructions in the case of LIBXSMM are ordered such that a load instruction is immediately followed by an `add`/`fmla` instruction (see listing 5.2.1). In the case of PSpaMM, we generate a block of mutliple load/store instructions before generating a block of arithmetic instructions (listing 4.1.4). Due to a lower amount of out-of-order resources provided by the A64FX [28], the ordering of instructions may impact the performance of our kernels.

We want to note that the benchmarks include matrices with dimensions that exceed the sizes of matrices for which PSpaMM is used in practice. For real applications, matrices usually do not exceed sizes of 100×100. Although Fujitsu claims that the peak performance of the A64FX is 2.76 TFLOP/s for DGEMM [15], prior work related to PSpaMM measured the peak performance of the processor that we access on the BEAST cluster to be at 1.7 TFLOP/s [37]. We will compare our results to both the theoretical and the measured peak, because theoretical peaks are only, if ever, achieved under ideal conditions. The NEON generator achieves 4.7% of the theoretical and 7.6% of the measured peak performance. The LIBXSMM kernels reach 32.6% and 52.9%, whereas our SVE generator achieves 29.4% and 47.7% of the theoretical and measured peaks, respectively.

Finally, we executed a set of benchmarks using sparse matrices that are only filled by 5%. These sparse matrices are created using their corresponding mtx files. If the file does not exist, we generate a new one with randomized indices and values. These mtx files are used during benchmarking, thus we can expect the same performance each time we perform benchmarks of our sparse PSpaMM kernels. Using sparse matrices will always result in lower performance compared to benchmarking with dense matrices. Nevertheless, performing dense-by-sparse multiplications is the main use case of PSpaMM, making the results shown in figure 5.3.2 interesting as well. Similar to the dense case, the NEON-based kernels can keep up with the performance of the SVE version for matrices with dimensions up to 40×40.

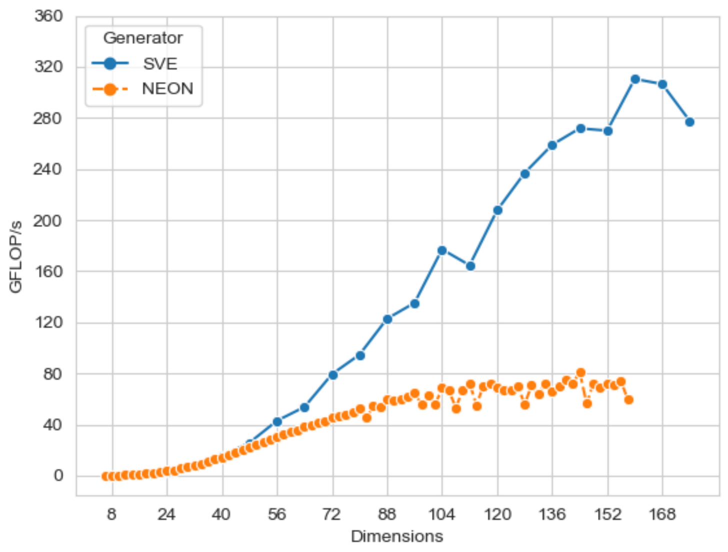

Figure 5.3.2: Performance measurements of sparse kernels for NEON and SVE

When using larger matrices, the SVE kernels outperform their NEON counterparts. Similar to the dense case with 80×80 matrices, the sparse multiplication kernels show an unexpected rise in performance compared to the previous test case, but this time for 104×104 matrices. Examining the peak performances, we can observe that the NEON kernels peak at around 80 GFLOP/s, whereas the SVE kernels peak at 310 GFLOP/s, which is 3.8 times larger.

6 Conclusion

In this section, we provide inspiration for future related work as to how PSpaMM may be improved further and summarize our results. We discussed hardware-related specifications of the A64FX and highlighted key differences between Arm's NEON and SVE. Additionally, we established fundamental matrix multiplication approaches and different storage formats. Afterwards, we discussed different loop optimization techniques and how to implement them, as well as layed out the key differences between PSpaMM's NEON generator and the newly added SVE generator.

Our results show that extending PSpaMM to generate SVE inline assembly leads to dense multiplication kernels that perform at least as well as its NEON counterpart for small matrix dimensions. For larger problem sizes, we showed that the SVE kernels are executed several times faster than NEON kernels. In addition to this, we demonstrated that PSpaMM's performance can compete with matrix multiplication kernels generated by the Intel library LIBXSMM for matrix dimensions up to 56×56. Although LIBXSMM kernels performed better for larger input sizes, PSpaMM's peak performance was only 9.1% lower than the peak achieved by LIBXSMM. Both kernel generators exhibited similar patterns of performance drops, presumably caused by a sub-optimal choice in block size as well as a lack of "out-of-order" resources. Finally, we measured PSpaMM's SVE kernel performance in the case of dense-by-sparse multiplications to be faster by up to 3.8 times compared to sparse NEON kernels generated by PSpaMM.

Promising ideas for future research include improving the algorithm used to choose a block size suitable for a given set of matrices. Currently, block sizes are chosen simply by searching for the largest block size that can be processed using the maximum number of available vector registers. Future work can try to implement a block size algorithm akin to more sophisticated ones used by GEMM generators like LIBXSMM. This may reduce the drops in performance we can observe for SVE-based kernels using larger matrices. Other approaches include reordering the way instructions are generated when processing a block. We did not test whether reordering the SVE instructions significantly affects a kernel's performance, therefore future research may try this approach to potentially stabilize PSpaMM's performance for larger problem sizes.

We can expect that future processors implement SVE with a vector size of more than 512 bits. In this case, researchers will need to investigate the performance gains when utilizing larger vectors, as well as potentially determining suitable vector lengths for different sets of problem sizes. Although larger vectors generally entail that we can process more elements at once, simply choosing the largest vector length available might not lead to better performances for every problem size. PSpaMM can be used for this, because the SVE generator is written in a way that it allows changing the internally used vector length to generate inline assembly for larger vectors. Finally, the SVE generator can be extended to allow the processing of single precision values. Although we have extended PSpaMM in a way that should allow switching between generating inline assembly for double and single precision values, we did not manage to confirm that the single precision version is working as intended.

Bibliography

[1] ARM. Arm Architecture Reference Manual Supplement, The Scalable Vector Extension. `https://developer.arm.com/documentation/ddi0584/ba/` [accessed on: 31/08/2021].

[2] ARM. ARM Cortex-A Series Programmer's Guide for ARMv8-A – Flow control. `https://developer.arm.com/documentation/den0024/a/The-A64-instruction-set/Flow-control?lang=en` [accessed on: 11/11/2021].

[3] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[4] Charles Bauer. Five-Stage Pipeline. `http://www.cs.iit.edu/~cs561/cs350/CPU/5stage.html` [accessed on: 14/09/2021].

[5] 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. IEEE, 2020.

[6] Nathan W. Brei. Generating Small Sparse Matrix Multiplication Kernels for Knights Landing. 2018. (Unpublished masters thesis at the Chair for Scientific Computing, Technical University Munich).

[7] Alexander Breuer, Alexander Heinecke, Sebastian Rettenberger, Michael Bader, Alice-Agnes Gabriel, and Christian Pelties. Sustained petascale performance of seismic simulations with SeisSol on SuperMUC. In *International Supercomputing Conference*, pages 1–18. Springer, 2014.

[8] Crystal Chen, Greg Novick, and Kirk Shimano. RISC Architecture. `https://cs.stanford.edu/people/eroberts/courses/soco/projects/2000-01/risc/risccisc/index.html` [accessed on: 14/09/2021].

[9] Crystal Chen, Greg Novick, and Kirk Shimano. RISC Architecture. `https://cs.stanford.edu/people/eroberts/courses/soco/projects/2000-01/risc/pipelining/index.html` [accessed on: 11/11/2021].

[10] Eric Cheng. Binary Analysis and Symbolic Execution with angr. PhD thesis, 2016.

[11] LIBXSMM Contributors. LIBXSMM. `https://libxsmm.readthedocs.io/en/latest/` [accessed on: 05/11/2021].

[12] Matt Eding. Data Structures - Sparse Matrices. `https://matteding.github.io/2019/04/25/sparse-matrices/#block-sparse-row` [accessed on: 28/09/2021].

[13] Victor Eijkhout, Edmond Chow, and Robert van de Geijn. *Introduction to high performance scientific computing.* lulu.com, 2011.

[14] Fujitsu. Fujitsu Presents Post-K CPU Specifications. `https://www.fujitsu.com/global/about/resources/news/press-releases/2018/0822-02.html` [accessed on: 09/09/2021].

[15] Fujitsu. FUJITSU Processor A64FX. `https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet.pdf` [accessed on: 02/09/2021].

[16] Nikunj Gupta, Rohit Ashiwal, Bine Brank, Sateesh K. Peddoju, and Dirk Pleiter. Performance Evaluation of ParalleX Execution model on Arm-based Platforms. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 567–575. IEEE, 2020.

[17] Jung-Chang Huang and Tau Leng. Generalized loop-unrolling: a method for program speedup. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122)*, pages 244–248. IEEE, 1999.

[18] Free Software Foundation Inc. Extended Asm - Assembler Instructions with C Expression Operands. `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html` [accessed on: 27/10/2021].

[19] Intel. Intel® Xeon® Platinum 8174 Processor. `https://ark.intel.com/content/www/us/en/ark/products/136874/intel-xeon-platinum-8174-processor-33m-cache-3-10-ghz.html` [accessed on: 09/09/2021].

[20] Adrian Jackson, Michele 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. IEEE, 2020.

[21] T. Jamil. RISC versus CISC. *IEEE Potentials*, 14(3):13–16, 1995.

[22] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *2009 International Conference on Parallel Processing Workshops*, pages 394–401. IEEE, 2009.

[23] Martin Käser, Cristobal Castro, Verena Hermann, and Christian Pelties. SeisSol–a software for seismic wave propagation simulations. In *High Performance Computing in Science and Engineering, Garching/Munich 2009*, pages 281–292. Springer, 2010.

[24] 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. IEEE, 2017.

[25] Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, Chu-cheow Lim, John Ng, and David Sehr. An advanced optimizer for the ia-64 architecture. *IEEE Micro*, 20(06):60–68, 2000.

[26] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

[27] Timothy Prickett Morgan. Fujitsu's A64FX Arm Chip Waves The HPC Banner High. `https://www.nextplatform.com/2018/08/24/fujitsus-a64fx-arm-chip-waves-the-hpc-banner-high/` [accessed on: 02/09/2021].

[28] Tetsuya Odajima, Yuetsu Kodama, Miwako Tsuji, Motohiko Matsuda, Yutaka Maruyama, and Mitsuhisa Sato. Preliminary Performance Evaluation of the Fujitsu A64FX Using HPC Applications. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 523–530. IEEE, 2020.

[29] National Institute of Standards and Technology. Matrix Market Exchange Formats. `https://math.nist.gov/MatrixMarket/formats.html` [accessed on 30/09/2021].

[30] University of Tennessee. PAPI. `https://bitbucket.org/icl/papi/wiki/Home` [accessed on: 05/11/2021].

[31] 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. *Fujitsu Technical Review*, pages 2020–03, 2020.

[32] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[33] 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, 2019.

[34] PSpaMM - Code Generator for Sparse Matrix Multiplication. `https://github.com/SeisSol/PSpaMM/` [accessed on: 22/07/2021].

[35] RIKEN and Fujitsu Limited. "K computer" Achieves Goal of 10 Petaflops. `https://www.fujitsu.com/global/about/resources/news/press-releases/2011/1102-02.html` [accessed on: 31/08/2021].

[36] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, et al. Co-Design for A64FX Manycore Processor and "Fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[37] Jonas Schreier. Optimization of small matrix multiplication kernels on Arm. 2021.

[38] SeisSol. `http://www.seissol.org/` [accessed on: 22/07/2021].

[39] Nigel Stephens. ARMv8-A next-generation vector architecture for HPC. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–31, 2016.

[40] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2):26–39, 2017.

[41] Erich Strohmeier, Jack Dongarra, Horst Simon, and Martin Neuer. Top500 Supercomputer rankings. `https://www.top500.org/lists/top500/` [accessed on: 21/07/2021].

[42] Erich Strohmeier, Jack Dongarra, Horst Simon, and Martin Neuer. Top500 Supercomputer rankings. `https://www.top500.org/lists/top500/2020/11/` [accessed on: 31/08/2021].

[43] WikiChip. ThunderX2 CN9980 - Cavium. `https://en.wikichip.org/wiki/cavium/thunderx2/cn9980` [accessed on: 03/09/2021].

[44] Daniel Yokoyama, Bruno Schulze, Fábio Borges, and Giacomo Mc Evoy. The survey on ARM processors for HPC. volume 75, pages 7003–7036. Springer, 2019.

[45] Toshio Yoshida. Fujitsu high performance cpu for the post-k computer. In *Hot Chips*, volume 30, 2018.