DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Optimization of small matrix multiplication kernels on Arm

Jonas Schreier

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Optimization of small matrix multiplication kernels on Arm

Optimierung von Multiplikationskernels für kleine Matritzen auf Arm Architekturen

Author:	Jonas Schreier
Supervisor:	Michael Bader
Advisor:	Lukas Krenz
Submission Date:	15.03.2021

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

Munich, 15.03.2021 Jonas Schreier

Acknowledgments

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
and my advisor Lukas Krenz for giving me guidance.

Abstract

Matrix multiplication is essential for many numerical applications. Thus, efficient multiplication kernels are necessary to reach optimal performance. Most efforts in this direction are targeted towards x86 architectures, but recent developments have shown promise when traditional x86 computing contexts are ported to Arm architectures. This work analyses different approaches of generating highly optimized kernels for small dense GEMM on Arm. Benchmarks of generated code are performed on ThunderX2 (NEON vector extension) and Fujitsu A64FX (Scalable Vector extension). While multiple possible solutions to the DGEMM problem on Arm exist, a general solution that performs best for all use cases could not be found. The largest deciding factor was the vector extension implemented by the respective target CPU.

Contents

1 Introduction

1.1 Motivation

The Arm CPU design is a RISC architecture that is used for various applications from tiny, specialized microcontrollers and small, energy efficient smartphone chips to high performance processors designed for supercomputing contexts. While Arm was traditionally lacking in performance for larger workloads compared to x86 architectures, rising energy demands of supercomputers and new, powerful Arm CPU designs have brought interest towards the Arm architecture and its implementations for the use in desktop CPUs and large scale supercomputers [39]. Modern Arm architectures have shown promise that they can outperform comparable x86 architectures while being more power efficient at the same time. In the context of supercomputing, an Arm CPU made headlines in 2019 when the Fugaku supercomputer was declared the fastest computer in the world [34]. Fugaku is built with the Fujitsu A64FX, an Arm CPU that was designed specifically for use in large scale systems [40]. In general, Arm architectures show a lot of promise for supercomputing workloads where performance and energy efficiency are paramount. As Arm CPUs have just recently found their way into the supercomputing market, many tools and solutions are either not available or not optimised for this new architecture. Furthermore, Arm's inner workings differ greatly compared to the more widespread x86/x64 CPUs, at least in the field of supercomputing. Due to these differences, considerable effort is required when porting or translating existing optimisation solutions. This work presents and evaluates different existing approaches for optimizing small matrix multiplication kernels, which are at the core of many scientific workloads.

1.2 SeisSol

SeisSol is a high resolution simulator for seismic wave propagation developed by a joint research team at Technical University Munich (TUM) and Ludwig-Maximilians University Munich (LMU). Physical seismic processes are modelled using hyperbolic Partial Differential Equation (PDE)s. Solving these PDEs is computationally challenging, whereby higher performance hardware allows for more accurate simulations. High

resolution simulations over large areas allow SeisSol to be scaled to arbitrarily high computing costs. An example for such a large simulation run is presented in [36] by Uphoff et al. At the lowest level, these PDEs are solved by numerous (small) matrix multiplications, either as dense $\times$ dense, dense $\times$ sparse or sparse $\times$ sparse. Because sparsity patterns are known in advance, specifically tuned multiplication kernels can be generated by SeisSol. Sparsity patterns are not relevant for dense multiplications, which make up a large part of SeisSols workload. Generally, sparse and dense calculations both need to be optimized, as focusing on one of each will inevitably create bottlenecks at some point. To evaluate the optimal multiplication kernel, SeisSol performs dry runs with a limited amount of iterations and selects the kernel with the best performance. Even though other highly performant generalized solutions like Intel's MKL or OpenBLAS exist, SeisSol relies on self generating kernels via libxsmm as generic solutions lose a small amount of performance due to their generality [7]. SeisSols simulation runs are performed on a variety of supercomputers with different x86 architectures, where libxsmm can provide adequate performance. Because recent developments made Arm processors more interesting for supercomputing workloads and libxsmm not being available on Arm, another high performance General Matrix Multiply (GEMM) kernel generator is needed.

1.3 Isambard & BEAST

Development and testing of the approaches presented in this work is performed on the Isambard and BEAST (Bavarian Energy-, Architecture- and Software-Testbed) clusters. Isambard was the first production ready Arm supercomputer in 2018. It uses over 20,000 ThunderX2 processors in Cray XC50 racks. Detailed information on this CPU can be found in section 2.4. Isambard is a joint project of the *GW4*, a consortium of four British universities. It was expanded in 2020, doubling its ThunderX2 nodes and making it the largest Arm supercomputer in Europe. As part of that expansion, multiple A64FX and ThunderX3 (when it is released) nodes will be added to the cluster. As one of the only large production ready Arm supercomputers in Europe, Isambard provides the necessary platform to benchmark the approaches shown in this work on a real world, large scale system [18]. BEAST is a smaller test bench by the Leibniz supercomputing centre. It contains multiple A64FX and Cavium ThunderX2 nodes and is used as a primary development and testing platform, while also serving as the testbed for runs on A64FX.

2 Background

In this chapter, we introduce the RISC architecture in general, the Arm implementation of RISC and the concrete Arm processors used for benchmarking. Furthermore, an overview of the General Matrix Multiply (GEMM) algorithm is given.

2.1 General Matrix Multiplication

The basic operations of linear algebra are essential for a wide variety of scientific and engineering fields and are therefore at the core of many numerical problems. Because of this widespread use, a de-facto standard specification for these basic operations was created, called Basic Linear Algebra Subsystem (BLAS) [23]. The operations are grouped into levels according to their operands as displayed in table 2.1.

While the asymptotic complexity of the matrix-matrix product could be lowered below $\mathcal{O}(n^3)$, for example to $\mathcal{O}(n^{2.8074})$ with Strassen's algorithm [33], in practice this slightly lowered asymptotic complexity is only beneficial for very large matrices as for small matrices, an optimized naive approach is more efficient as Strassen's algorithm is hard to optimize. Due to the abundant use of these basic operations, many applications can benefit greatly from efficient implementations of these basic routines. There typically exists an optimized implementation of the BLAS specification for every major processor generation and, while these optimized and tuned implementations already offer great performance, further improvements can be made when targeting specific CPU architectures. Especially micro-optimization that leverages specialized features of a concrete CPU are out of scope for BLAS implementations like OpenBLAS [38] or GotoBLAS [16] which aim to provide good performance on a wide variety of systems

Level	Structures	Typical complexity	Examples
1	Vectors	$\mathcal{O}(n)$	Dot Product, Addition, Norm
2	Matrix-Vector	$\mathcal{O}(n^2)$	Matrix-Vector Product
3	Matrix-Matrix	$\mathcal{O}(n^3)$	Matrix Multiplication

Table 2.1: Overview of BLAS operations [5], n is the number of multiplications required

and use cases. Another approach is used by the ATLAS [1] BLAS implementation, leveraging empirical auto tuning for optimization instead of the more manual approach used in Goto/OpenBLAS. In this work, the only relevant part of BLAS is the level 3 routine General Matrix Multiply (GEMM), that computes the matrix-matrix product of the form $C = \alpha * A * B + \beta * C$ where A, B and C are matrices with dimensions $m * n, n * k$ and $m * k$ and α and β are scalars. Note that in this work the scalars α and β are not included in the implemented GEMM algorithms and can therefore be assumed to be fixed to values of $\alpha = 1$ and $\beta = 0$. GEMM can be optimized for dense or sparse input matrices and can be further subdivided into small or large matrices, where *small* means (in this work) that each of the matrices A,B and C fit inside the processors L1 cache of typically 32 KiB, resulting in maximum matrix dimensions of about 100 rows/columns. Discriminating between these subsets of the GEMM problem is required to reach maximum performance, considering the implementation details differ greatly when comparing, for example, dense GEMM with its sparse counterpart.

2.2 RISC and Arm

The Reduced Instruction Set Computer (RISC) processor architecture is a competing design philosophy to the more common and current de-facto standard Complex Instruction Set Computer (CISC) architecture for desktop and supercomputing use cases. RISC processors offer different advantages compared to CISC architectures, the most impactful ones are presented below. The fundamental difference between Reduced Instruction Set Computer and Complex Instruction Set Computer is already revealed by their respective names. RISC architectures try to reduce the amount and complexity of available instructions to a minimum while CISC architectures generally have many specialized instructions. The RISC design actually involves more than this obvious difference in instruction set size. The discussion here follows [27].

Pipelining

RISC CPUs introduced pipelining to computer processors. The five execution stages Instruction Fetch, Instruction Decode, Execute, Memory Access and Write-back are working in parallel, each executing one instruction per clock cycle. Figure 2.1 shows how the different stages are working next to each other, providing their results to the stage after them and

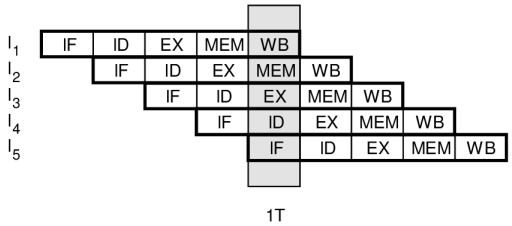

Figure 2.1: Classic RISC Pipeline by Eberle [9]

accepting the result of the stage before them. While pipelining is nowadays also available on CISC, due to varying instruction sizes it is harder to make full use of it. These design strategies are not exclusive to RISC though. Over time, RISC and CISC incorporated beneficial developments of each other into their designs but keeping mostly true to their roots. For example, RISC architectures usually also have vector extensions where some operations do take longer than one cycle to complete. The full instruction set is still much smaller than on a typical CISC implementation though.

Larger amount of registers

RISC architectures incorporate a larger amount of registers compared to CISC. The goal is to avoid memory accesses that only store temporary data and instead leave as much data as possible inside registers. The benefit here is the much faster speeds at which registers can be accessed compared to memory. Also, blocking as little memory traffic as possible with temporary data leaves more bandwidth for new data to be streamed in.

Low Number of Clock Cycles per Instruction

The amount of CPU cycles that are needed to decode an instruction is an important metric when looking into computing speed. The more complex a instruction is, the more cycles pass until the processor knows what that concrete instruction actually means. The decoding of an instruction is done by the CPU's control unit, which is responsible for translating instructions into so called μOPs or microoperations. Because CISC uses variable length instructions, the process of decoding is much more complex and time consuming. Consider the x86 assembly instruction that increments an integer value stored in a register:

```
//Add 10 to the value stored at the memory location in register EAX + 10.
ADD [EAX+16],10
```

Before the CPU can now start decoding this instruction into microcode, it first needs to figure out which opcode corresponds to the `ADD` instruction. `ADD` can work on registers, memory addresses and immediates (constant integer values) which all can be of varying size (BYTE, WORD, DWORD), resulting in different opcodes for, at first glance, the same instruction. After the opcode is known, the decoder starts breaking the instruction down into microcode. In our case, this might look like this:

```
Load value in register 'EAX' and add 16 to it
Do memory lookup at the address that was computed
Load that value from memory into temporary register
```

```
Pass temporary register and immediate to the ALU
Fetch result from ALU
Write result to memory location at 'EAX + 16'
```

Each of these microinstructions requires at least one cycle of the CPU clock to complete, resulting in multiple CPU cycles for one instruction. Comparing this instruction to its respective ARM assembler, it might look something like this:

```
ldr r2, [r0, #10] // Load value at address r0+10 into r2
add r1,r2,10 // r1 = r2 + 10
str r1,[r0, #10] // Store r1 into memory at address r0+10
```

RISC architectures require us to explicitly load and store our values, resulting in more verbose assembly code. In addition, each instruction has a fixed size, typically 32 or 64 bits. A fixed instruction length makes it easier to map assembly instructions to opcodes which in turn reduces the complexity of the CPUs instruction decoder. This decoder can thus be implemented on a smaller physical footprint, leaving more space for transistors and other circuitry. While RISC programs are typically up to 50% longer than an equivalent CISC program [27], most RISC instructions can be completed in a single CPU cycle. Nowadays though, a complex assembly output is less of a problem because it is typically generated by compilers rather than being written by hand.

Load-Store Architecture

Reduced Instruction Set Computers are defined by their Load-Store architecture. This means that there is a clear distinction between instructions that can access memory and those that operate exclusively on registers and perform calculations. In fact, only two instructions are allowed to access memory. Those are `ldr(load)` and `str(store)` with subcommands for different register sizes. Vector or floating point extensions add additional load/store instructions for their respective use cases. All other major instructions (`add,␣sub,␣mul,␣etc.`) can only operate on registers filled by load/store beforehand [2].

The Arm RISC architecture

The Advanced RISC Machines (Arm) Holdings Company designs a popular RISC architecture that sees widespread use in the domain of microcontrollers, smartphone or other integrated SoCs and, recently, in desktops and supercomputers. Arm designs can be licensed from them and then be implemented by a hardware manufacturer, Arm themselves do not produce any chips. The Arm architecture specification includes,

among others, the Instruction Set Architecture (ISA), CPU modes and register layouts. All architectures published by Arm include the following RISC features [31]:

- Load/Store Architecture
- fixed instruction width
- (mostly) single clock cycle execution

The latest Arm standard is v8, released in 2011. Over the years, it was extended continuously and the most recent version is called Arm v8.6-A. The Arm standard also includes the vector (SIMD) extensions NEON (also known as Advanced SIMD) and Scalable Vector extension (SVE). NEON was initially intended for media acceleration but grew larger and more capable over the years and is now used as a general purpose SIMD extension [32]. SVE is the more modern of the two with a focus on scientific and high performance computing. It introduces features that are novel to vector extensions. First off, SVE is vector length agnostic meaning that the vector length is not fixed to a particular value like in NEON or x86 SIMD extensions but can be freely chosen (in steps of 128 bits) by the hardware manufacturer, up to a maximum of 2048 bit. SVE instructions themselves are also not fixed to a specific vector size, making programs instantly compatible with CPUs that implement different SVE vector lengths [2]. No rewriting or recompiliation is necessary when moving from a, for example, CPU with 256 bit SVE to a CPU with 2048 bit. Another novelty are the predicate registers. These registers are used exclusively to control SVE instructions and allow a dynamic selection of vector lanes. One bit of a predicate register is mapped to one byte of a SVE register, allowing granular access in 8 bit chunks. Predicate registers can also be used to control loop iterations, eliminating loop overhead and enabling vector length agnostic loop bodies.

2.3 Fujitsu A64FX

The A64FX is a superscalar 64-bit Arm processor designed for high performance computing unveiled by Fujitsu in 2018. It is used in the currently fastest supercomputer "Fugaku" at the RIKEN Center for Computational Science in Kobe, Japan. [34] It features 48 compute cores and 4 assistant cores, high bandwidth memory with a peak bandwidth of 1024 GB/s and peak performance of over 2.8 TFLOPS while also being comparatively energy efficient, ranking 9th on the *Green 500* in GFLOP/s per watt of power consumed [34]. The full spec sheet can be found in table 2.2. The CPU implements the Arm v8.2 instruction set including NEON and Scalable Vector extension (SVE).

While the A64FX has high theoretical peak performance, special optimisations are necessary to reach optimal performance because of the CPUs relatively large instruction latencies for vector instructions. The FMLA instruction (fused multiply add) has a latency of 9 μOps, even though the instruction itself only takes a single μOP to complete [2]. Therefore, optimizing programs so that results from FPU calculations are not reused immediately is necessary. The A64FX's cores are organized into four Core Memory Group (CMG)s with 12 compute cores and one assistant core each. These CMGs share a layer 2 cache and a high bandwidth memory controller. Inter-CMG communication is handled by a routing controller on the chip. Figure 2.2 shows the block diagram of the A64FX processor and how the internal CPU components are connected.

Layer 1 cache is available on a per core basis and therefore not shown in this figure as it is not shared between cores and can be considered to be part of a compute core. Because the A64FX is designed for high performance computing specifically, it contains a TofuD Controller that enables the usage of a high data rate mesh network (26 Gb/s) between compute nodes. High bandwidth extensions via PCIe are also possible [11]. The A64FX is the first CPU to implement the new Arm SIMD Scalable Vector extension. A64FX's maximum vector length is 512 bit, four times larger than the 128 bits that are available using NEON [12].

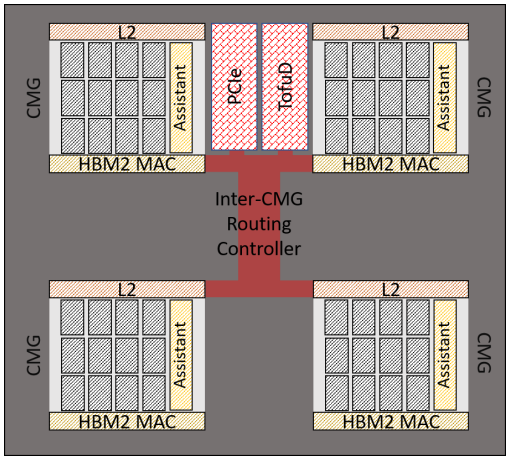

Figure 2.2: A64FX Block diagram, based on [11]

2.4 Marvell ThunderX2

Marvell ThunderX2 is an Arm CPU designed for high performance computing, released in early 2018. It is designed as a System-on-a-Chip (SoC) and produced by Cavium. Naturally for a supercomputer CPU is the high degree of parallelization with 128 threads running on 32 cores. Each core has two floating point units (FPU) with a vector size of 128 bit. Note that the ThunderX2 comes in different configurations, differing mainly in clock speed and core count. The version available to us is the CN9980 clocked at 2.2 GHz [37]. Specific and low level documentation for the ThunderX2 is hard to find as the Marvell does not provide detailed documentation to the public. Especially

	ThunderX2	A64FX
ISA	Arm v8.1 + NEON	Arm v8.2 + SVE
Cores	32	48 + 4
Threads	128	48
Clock Speed	2.0 - 2.5 GHz	1.8 GHz
SIMD width	128 bit	512 bit
L1 Cache	1 MiB / 32 KiB per core	3 MiB / 64 KiB per core
L2 Cache	8 MiB / 256 KiB per core	32 MiB / 8 MiB per CMG
Transistor technology	16 nm CMOS FinFET	7 nm CMOS FinFET
Memory bandwidth	160 GB/s	1024GB/s

Table 2.2: Comparison of A64FX and ThunderX2 [8, 12]

an overview of instruction latencies as provided by Fujitsu for the A64FX would have been of great use for benchmarking and optimising. For the full specifications of the chip, refer to Table 2.2.

3 Implementation

In this chapter, we explore two existing approaches for optimizing matrix multiplication kernels. These range from attempting to be as general as possible and targeting a wide range of CPUs or even whole architectures, to a highly specialized code generator that targets a specific CPU, making use of a certain feature set. Both approaches optimize GEMM using the outer product approach for general matrix multiplication. To keep the scope of this work bounded, we introduce the following constraints:

- Matrices are dense, meaning all entries are stored explicitly.
- Matrices are small, meaning $m, n, k \in [2; 100]$.
- Matrices are stored as a flattened one-dimensional array in column-major format.
- Algorithms make use of vector instructions but are otherwise not optimized for parallel execution. No optimization is performed to ensure efficient multithreading. We target single core performance exclusively.

Additionally, the variables introduced in table 3.1 always retain their meaning throughout this work.

Variables	
A, B	input matrices of doubles
C	result matrix of doubles
m, n, k	constant integers
$n \times m$	size of A
$m \times k$	size of B
$n \times k$	size of C

Table 3.1: Recurring variables/constants throughout this work

3.1 Outer Product Matrix Multiplication

The outer product approach decomposes the given matrices A and B into their respective columns and rows. Then, the multiplication AB can be formulated as a series of vector-vector multiplications, yielding multiple matrices that can be merged into the result matrix

$$C = \sum_{i=1}^{N} C_i = \sum_{i=1}^{N} a_i b_i, \tag{3.1}$$

with a_i and b_i forming the i^{th} column/row of A and B respectively. Intermediate results are stored inside the matrix denoted by C_i. To be able to leverage performance benefits provided by vector instructions and to optimize memory access, this formula can be rewritten (simplified for square matrices of size $n \times n$) after Pal et al. [25] as

$$C = \sum_{i=1}^{n} C + a_i \otimes b_i, \tag{3.2}$$

with

$$x \otimes y = x \cdot y^T = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \cdot \begin{pmatrix} y_1 & \dots & y_n \end{pmatrix} = \begin{pmatrix} x_1 y_1 & \dots & x_1 y_n \\ \vdots & \ddots & \vdots \\ x_n y_1 & \dots & x_n y_n \end{pmatrix} \tag{3.3}$$

Immediately accumulating the intermediate results into C eliminates the need for caching n matrices consisting of n^2 double values. This is only possible because of the constrained matrix sizes. With bigger matrices, caching all three matrices A, B and C is not feasible due to cache size limitations. With the outer product approach, the only floating point instruction that has to be calculated is of the form $a = a + b \cdot c$, which is implemented in hardware on modern x86 and Arm CPUs. This operation is usually encoded as `fma` or `fmla` (Fused Multiply and Add). The advantage of FMLA is that, due to dedicated and specialized circuitry, it allows the FPU to compute a normally two instruction operation (one multiplication and one addition) in a single one. Also, because there is no need to store the interim result of the multiplication, rounding errors are reduced [29]. Finally, before discussing different optimization options, we examine the pseudocode of the outer product approach in algorithm 1.

Algorithm 1: GEMM using Outer Product

```
// C should be zeroed
Input: A,B,C,n,m,k
Result: C
for i in 0..n do
    a = i-th row of A
    for j in 0..k do
        b = j-th column of B
        for l in 0..m do
            C[i,j] = C[i,j] + a[l] * b[j] // 2 FLOP
        end
    end
end
```

We have a single Fused Multiply-Add Instruction surrounded by setup code that loads the rows and columns into memory. Because all matrices A, B and C are small enough to fit into L1 cache, only $n \cdot m + m \cdot k$ floating point numbers have to be loaded and $n \cdot k$ have to be stored back into memory. Every FMLA instruction consists of 2 FLOPs, resulting in a total of $n \cdot m \cdot k \cdot 2$ FLOPs . Starting with this base algorithm, various loop optimizations can be applied. An overview of loop optimization techniques is given in section 3.2. While some optimizations are already performed by a modern compiler, specific CPUs offer more performance when algorithms are specifically tuned for them [4]. Since this work deals with small matrices exclusively, an approach that aims to keep as much data in the CPUs vector registers as possible is the most promising [24].

3.2 General Loop Optimization Techniques

In the case of the outer product formulation for matrix multiplication, two possible optimization targets exist, which are the nesting of the three loops and the computation inside them. To ensure that loop transformations are valid, i.e do not change a programs semantics, dependency analysis is necessary. Inspecting the outer product formulation in algorithm 1, we can see that the two outer loops indexed by i and j are independent of each other. Rearranging the ordering of these outer loops would only impact how the input matrices A and B are accessed. The inner loop and its contained computation depends on both outer loops. This is illustrated in the dependency graph in figure 3.1.

The inner loop can thus be optimized as long as it is wrapped inside the two outer loops [4]. The two outer loops can be run in any order, no performance benefits can be

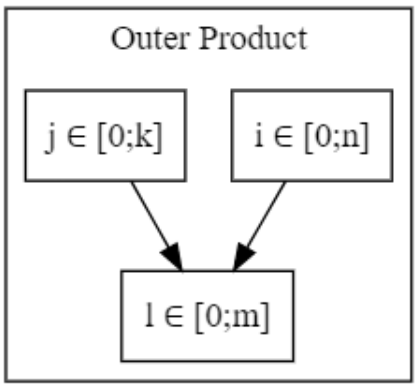

Figure 3.1: Dependency graph illustrating loop dependencies of the outer product

expected from reordering these loops though. To illustrate the different optimization techniques, a simple vector copy loop is used as an example.

```
//Unoptimized
for(int i=0;i<n;i++){
  a[i]=b[i]; //Vector copy
}
```

Loop Tiling

Loop tiling (also called loop blocking) partitions a single loop into smaller chunks of a fixed size. By doing this, one can ensure that data remains inside the CPUs cache, improving data locality and eliminating cache misses. This is especially crucial for dense matrix-matrix multiplications [4]. The tiling size is hard to pin down as it is dependent on the CPUs cache size and the data access layout of the algorithm at hand. In this example, the tile size is chosen arbitrarily. Tiling would provide no benefit in the sample presented here, as a vector copy does not reuse data.

```
//Tiled
assert(n%4 == 0);
for(int i=0;i<n;i+=4){
  for (int j=i;j<tileSize;j++){
    a[j]=b[j]
  }
}
```

Loop Unrolling

Unrolling a loop is done by replacing a part of the iteration space by explicit instructions. Note that unrolling requires the loop to be tiled beforehand, or alternatively loops have to be fully unrolled. Usually, the tile size matches the unroll factor.

```
//Unrolled
assert(n%4 == 0);
for(int i=0;i<n;i+=4){
  a[i]=b[i];
  a[i+1]=b[i+1];
  a[i+2]=b[i+2];
  a[i+3]=b[i+3];
}
```

Loop overhead from tiling the loop is eliminated by unrolling. Instead of n iterations (and therefore n conditional jumps), only a quarter of these conditional jumps is necessary now. Loops can also be fully unrolled, eliminating all jumps from the program. While this results in large assembly files and longer compile times, it will also provide better performance as long as the loop body still fits into the CPUs instruction cache. If this is not the case, a smaller unroll factor might actually provide better performance since it avoids frequent cache misses. An unrolled loop is also easier to vectorize for the compiler, as multiple consecutive calculations can easily be gathered into a single vector instruction. In summary, loop unrolling aims to eliminate loop overhead and enables the usage of vector/SIMD instructions.

Vector/SIMD instructions

Single instruction, multiple data (SIMD) instructions make it possible to load, store and modify a data vector instead of a single data point [26]. In our example, the four data loads of arrays a and b and the subsequent reassignment in each loop iteration could be done in one SIMD instruction. This is an example of instruction level parallelism.

```
assert(n%4 == 0);
for(int i=0;i<n;i+=4){
  //Actual ASM is ISA dependent
  a[i:i+3]=b[i:i+3];
}
```

Four loads/stores are replaced by a single vector load/store. The maximum vector width and the resulting assembly depend on the specific CPU. Writing vectorized

assembly by hand is difficult, time consuming and error prone. Therefore it is beneficial to leave these optimizations to the compiler where possible.

3.3 Loopy

Loopy is a code generator for array based code, mainly for parallel computing languages like CUDA and OpenCL [22]. It can also target generic C-Code that is architecture agnostic and can be compiled for Arm. It aims to provide a high level interface for loop optimizations that would otherwise be hard or tedious to produce by hand.

3.3.1 Loopy in general

Loopy takes the input, two strings called the loop domain and the loop instructions and produces an intermediate loop represented as a Python object. This object can be modified further and optimized by the user using the convenience of Pythons high level language features. Note though that loopy generates plain C code, requiring a compiler that can make use of processor specifics to achieve the best possible performance. A simple loopy kernel that takes an array and performs element wise addition might look like this:

```
#The loopy Library is imported as lp.
knl = lp.make_kernel(
"{ [i] : 0 <= i < n}", # Domain
"out[i] = out[i] + a[i]", # Instructions
target=lp.CTarget(),assumptions="n>0 and n mod 4 = 0")
```

Now, loopy allows the user to perform optimizations on that kernel:

```
knl = lp.split_iname(knl,"i",4) # Tile with tilesize = 4
knl = lp.tag_inames(knl,dict(i_inner="unr")) # Unroll inner loop
```

To avoid checks on the size of n, loopy can be told to assume that n is a multiple of 4. This means that our loop can always be fully unrolled and giving the wrapping program the responsibility to check that only valid inputs are fed into loopys generated code. When specifying the C language as loopys target, the generated code is first split into two loops with the specified tile size and afterwards the inner loop is unrolled. Loopy also inserts the `__restrict__` keyword for pointers passed to the function. This keyword tells the compiler that these pointers will have no memory overlap and allows it to produce better optimized code.

```
#include <stdint.h>

void loopy_kernel(double const *__restrict__ a, int const n, double *
    __restrict__ out)
{
  for (int i_outer = 0; i_outer <= ((-4 + n) / 4); ++i_outer)
  {
    out[4 * i_outer] = out[4 * i_outer] + a[4 * i_outer];
    out[4 * i_outer + 1] = out[4 * i_outer + 1] + a[4 * i_outer + 1];
    out[4 * i_outer + 2] = out[4 * i_outer + 2] + a[4 * i_outer + 2];
    out[4 * i_outer + 3] = out[4 * i_outer + 3] + a[4 * i_outer + 3];
  }
}
```

3.3.2 Outer Product Loopy Kernel

To generate the outer product kernel using loopy, we again need to specify the loop domain and instructions. First, we declare our loop domains and the statements that should be performed in the loop body. Then, we tell loopy what the data types of our parameters are and what loopy can assume to be always true. In our case, we know that our loop variables m,n and k will never be negative. This allows loopy to eliminate checks on these variables. Another option is fixing the variables to constant values as we will do later.

```
outer_knl = lp.make_kernel(
"{ [i,j,l] : 0 <= i < m and 0<= j < k and 0<= l < n }",
"C[i,j] = C[i,j] + A[i,l]*B[l,j]",
target=lp.CTarget(),assumptions="m>=0 and n>=0 and k>=0")

outer_knl = lp.add_and_infer_dtypes(outer_knl,{"A,B,C":np.float64,"m,n,k":
    knl.index_dtype}) #Data types of the variables
```

Now we need to instruct loopy to perform prefetching. Prefetching in this context means that our kernel streams in rows and columns of our input matrices before performing calculations with them, as shown in algorithm 1. To get loopy to prefetch rows and columns of the input A and B into arrays before they are accessed and to redirect accesses to the now prefetched arrays, loopy can be told to `add_prefetch` on a per array basis.

```
outer_knl = lp.add_prefetch(outer_knl,"A[:,l]")
```

```
outer_knl = lp.add_prefetch(outer_knl,"B[l,:]")
```

This is all that is needed to generate an unoptimized outer product kernel with our matrix dimensions m, n, k as loop variables and the matrices A, B, C as input data. Tiling and unrolling this kernel is performed by modifying it with:

```
outer_knl = lp.split_iname(outer_knl,"i",4) # tilesize 4
outer_knl = lp.tag_inames(outer_knl,dict(i_inner="unr"))
```

The first instruction tiles our loop indexed by `i` into two loops `i_inner` and `i_outer` with the given tile size. The second instruction takes this tiled loop and unrolls the inner one. The chosen tile size of 4 is for brevity of the generated output. In practice, a tile size equal to the matrix dimensions should give the best performance as it allows the loops to be fully unrolled. Keep in mind that while fully unrolled code is faster in theory, in practice the CPU will eventually run out of resources like registers or instruction cache space. Therefore, fine tuning is necessary to ensure that unrolling is only performed to a degree that is actually beneficial for the hardware at hand. Note that right now, our kernel is not fixed to static values of m, n and k. To achieve this, we use:

```
outer_knl = lp.fix_parameters(outer_knl,m=8,n=8,k=8)
```

Fixing parameters to static values allows loopy to eliminate checks on the matrix size and makes the generated kernel much more concise. After parameters are fixed, we can continue unrolling the code. When the middle loop (indexed by j) and the outer loop (indexed by l) are tiled and fully unrolled, all loops have been eliminated and what remains are plain data accesses and calculations that a modern and optimized compiler should be able to vectorize.

```
void loopy_kernel(double const *__restrict__ A, double const *__restrict__
    B, double *__restrict__ C)
{
  double A_fetch[8];
  double B_fetch[8];

  B_fetch[0] = B[0];
  B_fetch[1] = B[1];
  //...

  A_fetch[0] = A[0];
  A_fetch[1] = A[8];
  //...
```

```
C[0] = C[0] + A_fetch[0] * B_fetch[0];
C[8] = C[8] + A_fetch[1] * B_fetch[0];
//...

C[1] = C[1] + A_fetch[0] * B_fetch[1];
C[9] = C[9] + A_fetch[1] * B_fetch[1];
//...
```

Listing 3.1: Shortened fully unrolled outer product generated by loopy

Keep in mind that fully unrolling all of the involved loops for a outer product with $m = n = k$ generates n^4 statements. While this is unproblematic for small matrix sizes, code generation for $n > 30$ becomes very slow. Furthermore, compiling these large source files with GCC is very time intensive and probably not beneficial to performance. Extremely large source files will result in frequent instruction cache misses as for example, loop bodies get too large to fit into instruction cache. To alleviate this problem, we do not unroll the outermost loop fully for $m = n = k > 50$ and instead leave the outer loop intact, reducing the total amount of statements from n^4 to n^3. For even larger input sizes of $50 < n < 100$ we stop unrolling the middle loop fully and instead tile it to a fixed size and unroll with the chosen tile size. For those larger input sizes, the occasional jump/loop is unlikely to impact performance by an noticeable amount. To find a good balance between optimization and practicality, we ran different benchmarks with varying degrees of optimizations, fine tuning which loops got unrolled and to which degree. The generated code is now ready to be linked and compiled into a wrapper executable like SeisSol or other programs. Note that this code is still fully portable and the final assembly depends on the used compiler and preferred target architecture. After compilation, we can inspect the resulting assembly generated by GCC to see which optimizations the compiler applies to our already optimized programs. First off, GCC is able to optimize for NEON and SVE alike, meaning our compiled programs for A64FX and ThunderX2 differ in the usage of the respective vector instructions. We can see that GCC is able to fully utilize SVE features on A64FX. This requires the usage of GCC's newest version, 11 [14], which has not yet been released officially but is available as a feature complete beta (bugs may still occur as of the writing of this thesis, none were encountered during experiments though) [15]. Previous versions of GCC already support SVE but lack auto vectorization features and are generally not able to make full use of the new vector extension. Naturally, GCC is able to utilize NEON features on ThunderX2. A comparison of a FMLA calculation is shown in Table 3.2. Note the lower amount of floating point operations performed by SVE-optimized code. The `insr` instruction is used in SVE for moving general purpose

```
1 str d8, [sp, 136]
2 ldr d8, [sp, 128]
3 fmadd d8, d18, d4, d8
4 str d8, [sp, 128]
5 ldr d8, [sp, 120]
6 fmadd d8, d18, d3, d8
```

ThunderX2 with NEON

```
insr z0.d, d6
insr z0.d, d7
insr z0.d, d16
insr z0.d, d14
insr z0.d, d20
fmla z10.d, p0/m, z0.d, z1.d
```

A64FX with SVE

Table 3.2: Comparison of resulting NEON and SVE assembly after compilation

registers into SVE vector registers.

3.4 PSpaMM

PSpaMM is an assembly code generator for sparse matrix multiplication kernels. It was specifically created for SeisSol. It generates a C-function with inline assembly optimized for Knights Landing (KNL), an x86 architecture designed by Intel for high performance computing [6] and was later expanded to support the newer Skylake architecture. Even though PSpaMM is designed for sparse matrix multiplication, it can also be used for dense $\times$ sparse or even dense $\times$ dense matrix multiplication. While PSpaMM produces highly specialized assembly, the PSpaMM executable itself is very portable as it is written in Python. PSpaMM offers a simple command line interface where matrix dimensions m, n, k, leading dimensions lda, ldb, ldc as well as constant factors α and β have to be provided as arguments. The leading dimension of a matrix is the amount of either rows or columns of the matrix, depending on weather the matrix is stored in column or row major format. This leading dimension is necessary to calculate indices of matrix entries when the 2D matrix is stored inside a flattened 1D array. The index of the element at (i,j) of a column-major matrix with leading dimension `lda` would then be calculated with $i + j * lda$. As we store matrices in column-major format, lda, ldb and ldc are equal to the number of columns of the respective matrix.

```
//pspamm.py m n k lda ldb ldc alpha beta
> ./pspamm.py 8 8 8 8 8 8 1 0 --arch arm
```

Listing 3.2: Example PSpaMM call

While PSpaMM mainly targets KNL and therefore x86, it can also output Arm NEON assembly. Listing 3.3 is a shortened example output of Arm inline assembly generated by PSpaMM. The complete output can get very large as PSpaMM tries to fully tile and unroll loops wherever possible. It also makes use of Arm NEON instructions like fused multiply add on vector registers or vector load/store. While PSpaMM produces specialized code that should yield adequate performance, it is not very flexible. Its intended target architecture is KNL and there is no easy way to adjust parameters for a new architecture like Arm. For example, PSpaMM is tuned for instruction and data cache sizes as they would be available on KNL CPUs. Fortunately, KNL and ThunderX2 have the same cache sizes of 32 KiB for instructions and data each. More performance could be gained by optimizing PSpaMM for the A64FX, as its 64 KiB cache and 512 bit wide SIMD (128 bit on the ThunderX2) should lead to higher performance when utilized correctly. Nevertheless, PSpaMM manages to perform most of the standard optimizations (tiling, unrolling) which provide a performance boost no matter the specific architecture. When using PSpaMM for dense matrices (as it is the case for this work), its output is identical to the outer product formulation as presented in Algorithm 1. PSpaMM emits its output directly to standard out or alternatively to the file specified as an command line argument. This file can then be linked into a C/C++ program and compiled as usual.

```
//Called with: > python3 pspamm.py 4 4 4 4 4 4 1 0
void dgemm (const double* A, const double* B, double* C, double alpha,
    double beta, const double* prefetch) {{
    __asm__ __volatile__(
    //Loading args into registers
    //prefetch is unused
    "ldr x0, %0\n\t" //*A
    "ldr x1, %1\n\t" //*B
    //...

    "mov x12, #0\r\n"
    "LOOP_TOP_0_%=:\r\n"
    // zero registers
    "fmov d24, xzr\r\n"
    "fmov d25, xzr\r\n"
    //...

    // Block GEMM microkernel
    // Load A register block @ (d=0,r=0)
    "ldp q2, q3, [x0, 0]\r\n" // A [0,0] [0,0]
    "ldr q4, [x1, 0]\r\n" // B[0,0][0,0]
    "ldr q5, [x1, 32]\r\n" // B[0,0][0,1]
    //...

    "fmla v24.2d, v2.2d, v4.2d[0]\r\n" // C[0:2,0] += A[0:2,0]*B[0,0][0,0]
    "fmla v26.2d, v2.2d, v5.2d[0]\r\n" // C[0:2,1] += A[0:2,0]*B[0,0][0,1]
    //...

    // Store C register block @ (d=0,r=0)
    "stp q24, q25, [x2, 0]\r\n" // C [0,0] [0,0]
    //....

    "add x0, x0, #32\r\n" // Move A to (d=1,r=0)
    //...
    "cmp x12, #1\r\n"
    "b.lo LOOP_TOP_0_%=\r\n"
    : : "m"(A), "m"(B), "m"(C), "m"(alpha), "m"(beta) : "r0","r1" ...
}};
```

Listing 3.3: PSpaMM shortened example output of a 4x4x4 outer product

PSpaMM had a minor bug when generating code for Ar architectures with larger matrix sizes. Arm instructions are fixed to a 32 bit size resulting in issues when trying to fit large immediates into these 32 bits. This can become problematic if one tries to load immediates that are larger than 12 bits for the standard `mov` instruction. Assuming a load of a 16 bit immediate into a register using the `mov` instruction, the operation will fail because the 16 bit immediate cannot fit inside the 12 bits that `mov` has available for immediates. In 64 bit Arm, large immediates have to be moved by splitting up the immediate into 16 bit chunks, moving one chunk at a time into the destination register. PSpaMM did originally not account for this resulting in the generator failing to generate valid assembly for matrices with one dimension larger than 62. The splitting up of the `mov` instruction is shown in Listing 3.4. `movz` and `movk` are 64 bit Arm instructions that can move up to 16 bit immediates. Their 32 bit counterparts are `movw` and `movt`. `movz` zeroes the remaining bits of the destination register while `movk` keeps the unaffected bits unchanged. Additionally, a shift by 16, 32 or 48 bits can be specified to move immediates into the upper regions of the destination register [3].

```
//Moving the 20 bit value 0xBEEEF
mov x0, #0xBEEEF //This will fail

//Replaced by
movz x0, #0xEEEF //Lower 16 bits
movk x0, #0xB, lsl #16 //Upper 16 bits, shifted to the left by 16
```

Listing 3.4: Loading large immediates on 64 bit Arm

3.5 Comparing loopy and PSpaMM

When comparing the loopy approach to PSpaMM, it is noticeable that both approach the problem from a very different angle. Our loopy program aims to be easily modifiable, portable and relies on compiler optimizations to leverage SIMD features. In contrast, PSpaMM's output is fully optimized after code generation and is therefore neither portable nor easy to modify. It is also worth mentioning that loopy's output can be made to be agnostic towards matrix sizes whereby this requires more checks to make sure no unsafe or out of bounds data accesses are made. Code generation is much faster on PSpaMM, especially for larger matrices. In fact, loopy's code generation is too slow to generate fully unrolled code for large matrices in a reasonable amount of time. Inline assembly is not optimized by the compiler used for this work (GCC) [13], therefore PSpaMM will not benefit from further compiler optimizations. This should result in PSpaMM being greatly outperformed on A64FX as SVE should perform better than

NEON. For different or future use cases or even use on a new architecture, PSpaMM would need to be modified considerably before it can be used, while Loopy only requires compiler support for the respective architecture.

4 Results

In this chapter, we evaluate and interpret our results and assess which approach fits which use case. Performance is measured via a wrapper program written in C++. It initializes 50 matrices with random floating point coefficients and increasing dimensions from 2x2 to 50x50. These matrices are then fed into our algorithms with execution time being measured by likwid's marker API and the likwid module `likwid-bench`. The marker API can be used to declare certain code regions that likwid should measure separately [30]. This enables us to run all four algorithms in one wrapper program while also getting performance statistics for each of these regions. Furthermore, declaring only the actual benchmarking loop to be measured eliminates set up overhead and makes results reflect the actual performance Ṫo make results more consistent and to eliminate random noise, this process is repeated multiple times. Listing 4.1 shows one of four loops that make up the benchmarking suite. Note that in the case of PSpaMM and loopy, we store function pointers to the tuned GEMM kernels inside a vector before the benchmark begins. This is necessary as every input dimension requires an explicit function implementation, resulting in 50 different functions that need to be called individually.

```
current_mat_dim = 2;
LIKWID_MARKER_START("PSPAMM");
for (int j = 0; j < matrix_count; j++)
{
  double c[current_mat_dim*current_mat_dim];
  double *mat = pspamm_matrices[j];
  pspamm_pointers[j](mat, mat, c, 1, 0, 0); //pspamm function pointers
  current_mat_dim += 2;
}
LIKWID_MARKER_STOP("PSPAMM");
```

Listing 4.1: Benchmarking loop surrounded by calls to the likwid marker API

To get to our final result, we take the maximal FLOP/s that likwid-bench observed over 10 runs and multiply it by the amount of cores the benchmarked CPU has. This is done to obtain a realistic value for a distributed workload where the single core calculations are distributed onto all available CPU cores. PSpaMM only supports dense

matrices with dimensions divisible by 2 (matrices that are not divisible by 2 need to be padded beforehand), matrices used are always square and dimensions increase in steps of 2. Matrices larger than 100x100 appear very rarely in SeisSols computations and are therefore not relevant for the case presented here. To evaluate our optimization attempts from chapter 3, we also measure two other algorithms and use these results as reference. These two algorithms are a naive GEMM implementation and the Eigen library.

Inner product matrix multiplication (Naive GEMM)

The naive GEMM or inner product algorithm provides an absolute minimal baseline performance. It is implemented completely unoptimized, relying on the compiler to optimize where possible. We expect this algorithm to perform significantly worse compared to both of our implementations, with more noticeable differences with increasing matrix sizes. It follows the "by-hand" matrix multiplication where elements c_{ik} of the result matrix C are calculated like

$$c_{ik} = \sum_{j=1}^{m} a_{ij} b_{ik}. \tag{4.1}$$

Implemented the standard way with three nested for loops and a single calculation in the body, this algorithm is not automatically vectorized by compilers. After loading in three values, ghcalculating the product and writing the result back into memory, a new load is performed. This constant change of operations looses time while waiting for memory accesses to finish, resulting in a slow algorithm.

Eigen Library

Eigen is a C++ Library for a wide range of linear algebra operations. Its implementation of matrix multiplication aims to be efficient and at the same time being easy to use and is dynamically optimized for the matrices at hand. While Eigen is auto vectorized for most vector extensions and does support Arm NEON, it does not yet provide support for Arm SVE [17]. Eigen compares quite well to other high performance matrix multiplication libraries like Intel MKL (x86 only), GOTOBLAS, OpenBLAS and ATLAS. It can generally be considered one of the fastest general matrix multiplication libraries [10].

The full benchmarking suite is then run on both Arm architectures available to us. As constrained at the beginning of chapter 3, all algorithms are single threaded and can only be parallelized by instruction level parallelism if the respective algorithm makes it possible to do so. Furthermore, all algorithms are compiled with:

```
//A64FX
g++ -Ofast -mcpu=a64fx -std=c++17 -msve-vector-bits=512
//ThunderX2
g++ -Ofast -mcpu=thunderx2t99 -std=c++17
```

Using `-mcpu` instead of the x86 standard `-mtune` is recommended for Arm GCC, as `-mcpu` allows the compiler to use standard Arm optimizations as well as CPU specific microarchitecture optimizations. Arm GCC interprets flags differently from x86 GCC where `-mtune` is almost always the preferred flag for highest performance [21]. The `-msve-vector-bits` flag fixes SVE operations to the vector length specified. Without it, the code would be compiled to be vector length agnostic. This would improve portability but could potentially have a small impact on performance, therefore we provide a fixed value for the vector length.

4.1 Upper performance limits

For theoretical peak performance of A64FX, we can refer to the data sheet provided by the manufacturer. Fujitsu claims 2.8 TFLOP/s at 1.8 GHz for double precision workloads [12]. In the case of ThunderX2 we have to calculate the theoretical peak performance ourselves as the manufacturer does not provide performance statistics. ThunderX2 has two FPUs that can perform one FMLA per clock cycle. The vector length of 128 bit, clock frequency of 2.2 GHz and 32 cores result in a theoretical peak

$$2.2\text{GHz} \cdot 32 \cdot 8\text{FLOP/cycle} \approx 540\text{GFLOP/s} \tag{4.2}$$

of double precision performance or $\approx$ 1080 GFLOP/s of single precision performance respectively. To verify performance figures for the A64FX, we use a benchmarking tool provided by FAU University in Erlangen called *likwid* [35]. *likwid* is a command line interface benchmarking tool with a focus on high performance computing. It is available for x86 and Arm architectures. For ThunderX2, we do not provide own benchmarks as this CPU has been benchmarked multiple times before, for example by Calore et al [8]. Their tests confirm the theoretical calculations of 560 GFLOP/s for double precision workloads with a peak bandwidth of about 120 GByte/s. For A64FX, it was not possible to reproduce the claimed 2.8 TFLOP/s by Fujitsu. Running a benchmark that limits memory access to the absolute minimum and putting the FPUs under full load gave us, at best, 1.7 TFLOP/s. The benchmark loads a single value into a SVE register and performs a large amount of `fmla` operations with this value. The registers z0 and z2 to z15 are filled before the benchmark begins to keep memory access to the absolute minimum possible. Listing 4.2 shows a short excerpt from this benchmark. *Likwid* can then be used to run this benchmark multiple times. Results are

gathered by reading the CPUs performance counters. These performance counters are keeping track of a wide range of metrics on a modern CPU, from cache misses to cycle exact execution time or memory read/write traffic. As they are completely separated from the rest of the processor, gathering of these metrics does not impact performance and yields accurate values [11].

```
//Setting up some data beforehand
.align 32
LOOP 8 //Benchmark begins here
ld1d z1.d, p0/z, [STR0,GPR1, LSL 3]//GPR1: General Purpose Register 1
fmla z0.d, p1/m, z0.d, z1.d
fmla z2.d, p1/m, z2.d, z1.d
fmla z3.d, p1/m, z3.d, z1.d
fmla z4.d, p1/m, z4.d, z1.d
fmla z5.d, p1/m, z5.d, z1.d
//10 more fmla instructions
//...
```

Listing 4.2: Shortened excerpt of the benchmark used to calculate peakflops on A64FX

The performance numbers are shown as a roofline plot in figure 4.1. We do not expect our approaches to reach the full potential performance. The measurements were performed with minimal memory access involved, resulting in very high operational intensities, allowing the FPU to reach its absolute peak throughput. In reality, even with the usage of the outer product, there will be delays where the FPU has to wait for new data to be loaded in.

4.2 Benchmark results

It was not possible to find a general solution that performed the best in both environments and over all matrix sizes. Not surprisingly, the naive algorithm falls short on both CPUs and for all matrix sizes except for the 2×2 matrix product. This is expected as such a small input size leaves little to no room for optimizations. If one is looking for a "good enough" performance solution, the Eigen Library is not only the easiest to use, but also only slightly slower for most scenarios and faster than both optimized approaches in some cases.

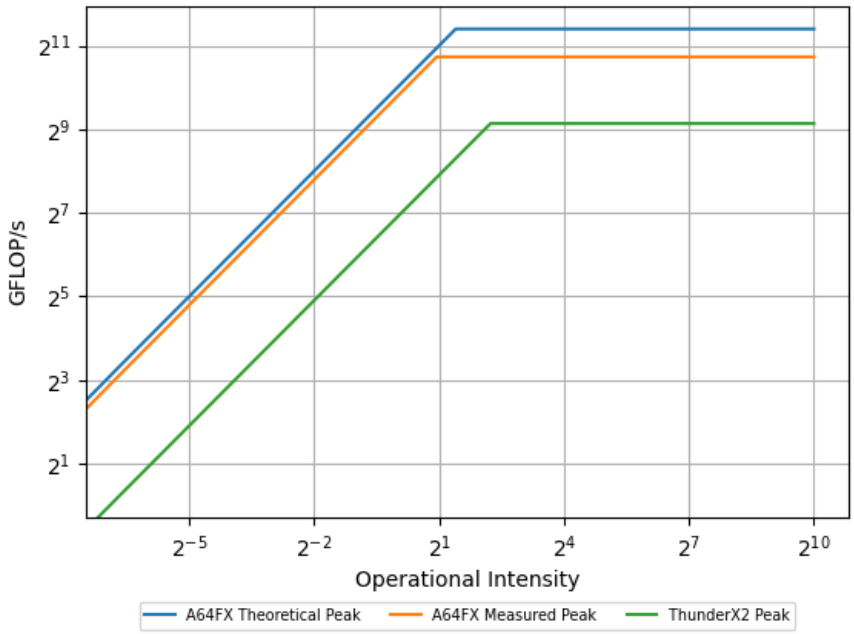

Figure 4.1: Roofline chart of A64FX and ThunderX2

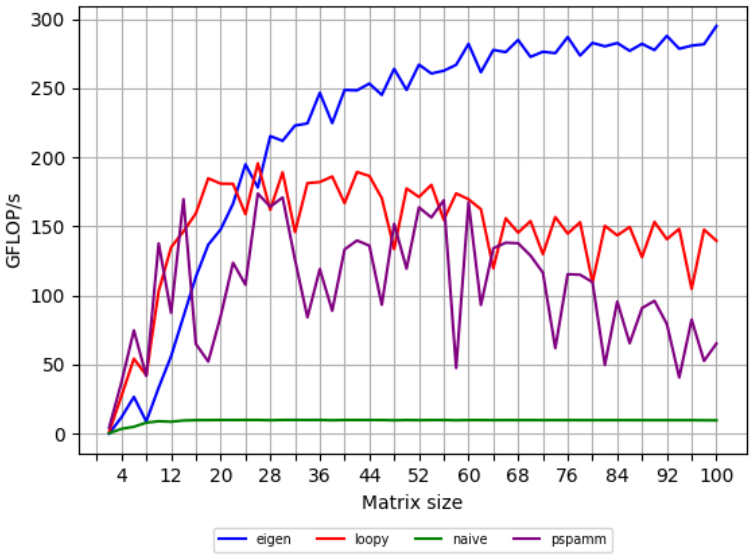

Figure 4.2: Benchmark Results on ThunderX2

4.2.1 ThunderX2

The performance graph in Figure 4.2 for ThunderX2 shows multiple interesting results. First off and unsurprisingly, the naive implementation can not keep up with the other implementations. Its performance gets further behind the larger the input matrices get, never achieving more than 15 GFLOP/s. The only size where the naive algorithm is comparable, is for matrices of size 2×2. For such small matrices, optimizations barely matter and therefore, there is little to no performance to be gained. For small matrices up to dimensions of 18×18, PSpaMM and loopy are close together in performance. PSpaMM experiences a large performance drop for dimensions from 20 to 26 while loopy does not. PSpaMM's performance becomes highly inconsistent from this point onwards. This is probably caused by cache misses in the instruction cache as PSpaMM functions get extremely large with 30,000 statements and more, causing the programs to not fit into the instruction cache any longer. Also, for larger matrix sizes we can no longer fit the whole problem set into vector registers, requiring values to be loaded from memory into the cache. Eigen shows an almost constant performance gain with increasing matrix sizes. Eigen's overhead for smaller matrices is quite large in comparison to the other approaches, but that overhead becomes negligible with larger inputs. From dimensions 30 onwards, Eigen is clearly the fastest solution, increasing

in performance while loopy and PSpaMM both get slower with increasing matrix size. All approaches except the naive one achieve about 50% of theoretical peak double performance of 540 GFLOP/s when extrapolated for all 32 cores. Only Eigen manages to achieve more than 50%, topping out at about 300 GFLOP/s or 55% of possible peak for larger matrices.

4.2.2 A64FX

On A64FX, the picture looks quite different. In Figure 4.3 we can see that, once again, the naive approach can not compete, providing extremely slow performance throughout the benchmark. Eigen and PSpaMM are considerably better than the naive approach and show a similar behaviour as on ThunderX2 with both providing comparable performance. Up until 30×30 matrices, PSpaMM is faster than Eigen while from there on, performance is almost identical with Eigen showing the same occasional performance drops as on ThunderX2. What is interesting though, is that performance for Eigen and PSpaMM is only about half as good as on ThunderX2, reaching ≈ 100 GFLOP/s on A64FX and over 200 GFLOP/s on ThunderX2. Loopy, on the other hand, is much faster on A64FX than on ThunderX2, achieving twice the amount of GFLOP/s on the newer CPU. Compared to ThunderX2, even the fastest measurement of loopy's outer product at 500 GFLOP/s is well short of the 2 TFLOP/s the A64FX is theoretically capable of. Initially, loopy showed a large spike in performance at matrix size 32, jumping ≈ 100 GFLOP/s to over 400 GFLOP/s. Reducing the degree of optimization on the kernel and simply unrolling the innermost loop while leaving the rest of the code completely plain resulted in much better performance on smaller matrices as well. Having less code and trusting the compiler to vectorize the inner compute loop is the driving factor behind execution speed on A64FX as eliminating loop overhead by unrolling over all loops did not improve performance but rather hinder it considerably. While the same phenomenon could be observed on ThunderX2, it was much more pronounced on A64FX. For reference, we included both benchmarks in Figure 4.4 before we stopped unrolling loopy fully, clearly showing the performance jump at matrix size 32.

4.3 Interpretation

The relative results of our benchmarks are, for the most part, as expected. A64FX shows better performance than ThunderX2 as long as modern features of A64FX, in particular vectorization with SVE, are used. What is unexpected though, is that the implementations that are tuned for NEON (Eigen, PSpaMM) are faster on the (theoretically) less capable ThunderX2. While a relative speed-up would have been less

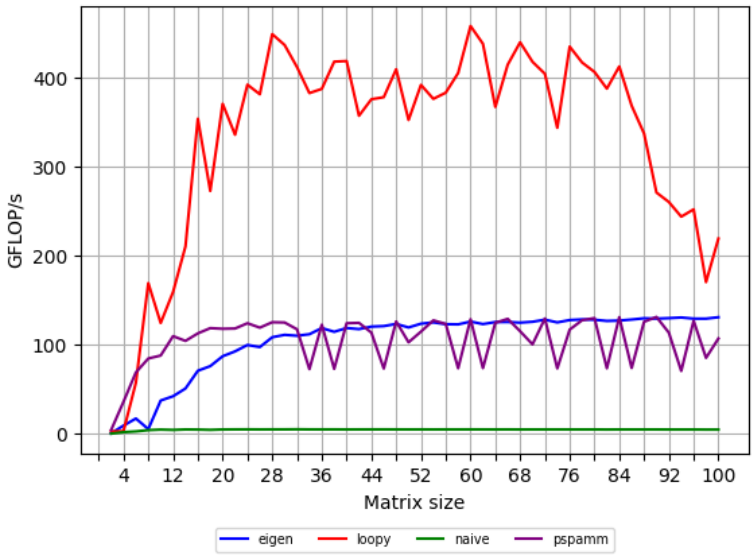

Figure 4.3: Benchmark results on A64FX

surprising, an absolute speed-up of over 100 GFLOP/s is intriguing. Inspecting relative speed-ups as shown in Figure 4.5 shows a very clear picture. We were able to achieve almost 50% of ThunderX2's potential peak with all three optimized implementations while we only reached $\approx$ 32% of A64FX measured peak using loopy, the only one with access to SVE. Eigen and PSpaMM are much further behind and never achieve more than 8% of possible peak. If we go by the claimed peak of A64FX (that we were not able to verify in the first place), we only reached 25% of the maximum with loopy. Issues in utilizing the full potential can be explained by the relative immatureness of the vector extension SVE and the therefore lacking compiler support for it. On the other hand, NEON has been around for some time now, is generally well understood and well supported by compilers. What this also shows, is that there is little performance to be gained by extensive targeted assembly optimizations on ThunderX2/NEON, as the very generic Eigen library is only behind in performance for very small matrices and much faster for larger ones, probably for the same reasons loopy became faster when we optimized less instead of more. Utilization of SVE instructions is paramount for exploiting the potential of the A64FX, as the large performance gap between loopy and Eigen/PSpaMM shows. While we do not have information on instruction latency figures for ThunderX2, we can assume that latencies are probably much lower than on

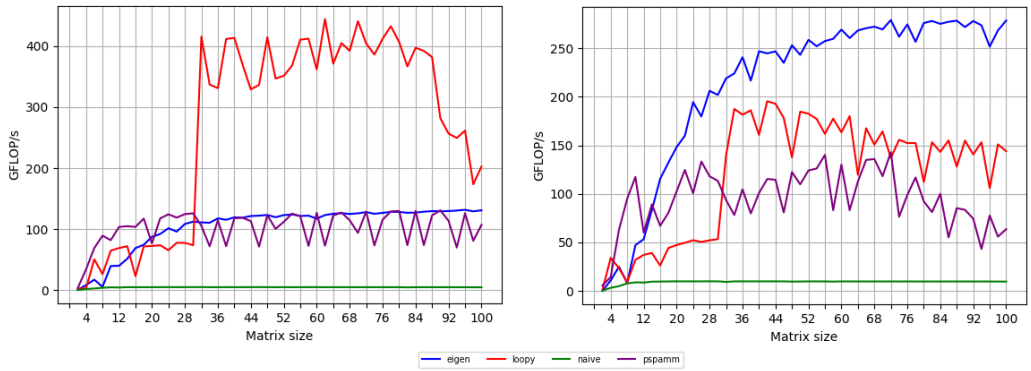

Figure 4.4: Benchmark results with a fully unrolled loopy for small matrix sizes. A64FX left - ThunderX2 right

A64FX. Otherwise, a performance drop when going from a (theoretically) slower CPU to a faster one is hard to explain. When Eigen will eventually be updated to enable vectorization via SVE as it does using NEON already, a significant performance boost is expected on A64FX. Also, compiler support for SVE in general will continue to be improved and therefore result in better performance from more high level programs like the ones generated by loopy. PSpaMM's performance on ThunderX2 for the smallest matrices with dimensions from $2 < n < 30$ should be replicable on A64FX when PSpaMM's code generation is correctly ported to SVE, which might be a good intermediate solution to get adequate performance out of A64FX until compiler support catches up.

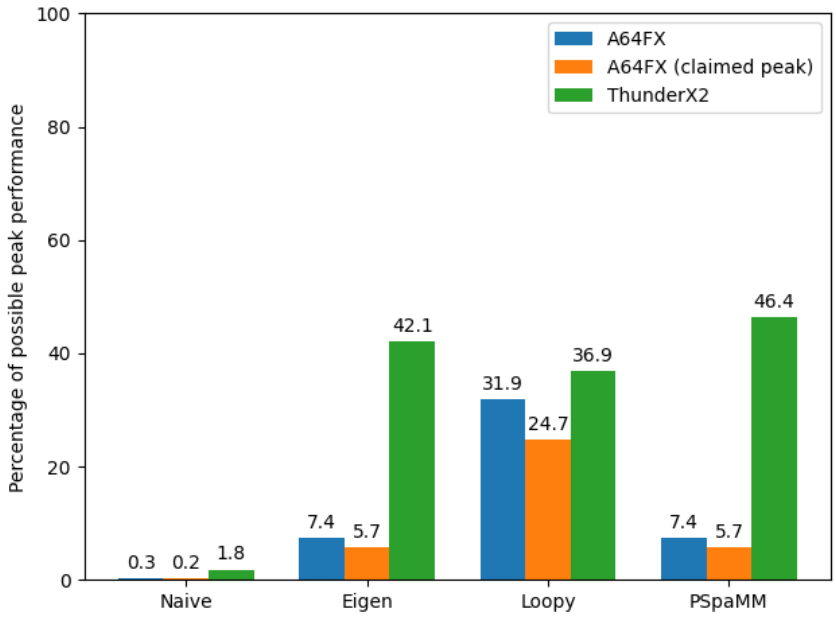

Figure 4.5: Performance measured as a percentage of possible peak

5 Summary

In this chapter, we summarise our findings and provide inspiration for further work to be done in this field. We show where research could continue or what other approaches are available.

5.1 Future work

Multiple other approaches exist that have not been explored in this work. Other Code generators or auto optimizers that work on Arm exist, for example Polly. Polly uses so called *polyhedral optimization* on llvm-generated loop code to improve performance while being completely hardware/architecture agnostic [28]. A combination of loopy and Polly is also possible and might provide better performance than using one of them exclusively. Libxsmm is another code generator to try out. It is currently already used in the SeisSol project in conjunction with PSpaMM when generating code for x86 architectures. Libxsmm provides Arm support via generic code generation on non-Intel architectures [20]. While this generic code will obviously never be as fast as code generated for Intel (as that is libxsmm's primary target platform), a speed up for certain matrix sizes, e.g for sizes where PSpaMM fails to provide good performance, could still be possible [19]. Also, small performance gains could probably be gained by improving when and how loopy is used. In this work we applied a rather general approach with loopy, unrolling until performance showed an obvious drop and reducing the tile size and unroll factor when such a drop was observed. More delicate and targeted modifications could provide a small but noticeable performance boost. Another target would be PSpaMM and its non existent support for SVE. Porting PSpaMM to support SVE instructions should yield, at least in the short term, the best results on A64FX. While that approach is promising, it would also require major changes in the way PSpaMM works right now. Simply exchanging instructions from NEON to SVE would not work in the first place, as there is no one to one mapping from one instruction set to the other. Furthermore, fully utilizing SVE probably requires a change in the structure of the algorithm itself, as A64FX's high instruction latencies, utilization of predicate registers and usage of A64FX four operand FMLA require a more in depth look into how PSpaMM generates code. Apart from porting PSpaMM to the new vector extension, general improvements to its code generation routine could also be performed,

especially for matrices with dimensions larger than 30. PSpaMM is intended for very small matrix sizes and our benchmarks showed that performance becomes unstable and spikes up and down when we venture past the intended use case of PSpaMM.

5.2 Closing remarks

Our results show that generating optimized code can be beneficial for performance, but is not a guarantee. Especially when working with newly released hardware, exploiting special aspects of it via manually optimized or generated code seem to be necessary to get adequate performance, at least as long as it takes for auto tuning solutions to be implemented and adjusted to new hardware. The comparison of the ThunderX2 and its mature architecture and vector extension to the newly released A64FX showed that general solutions like the Eigen library need some time to be adapted for new features (in this case, the Scalable Vector extension). Once this happens though, one can usually only expect slightly better performance by hand optimizing a code generator for a small subset of use cases. When someone is interested in maximum performance on newer hardware, loopy provided very good results. This is attributed to the fact that GCC already supported auto vectorization via SVE, allowing us to generate optimized but still readable C code and leaving the assembly optimizations to the compiler. Would GCC or a comparable compiler not have supported SVE already, code generated by loopy would probably have performed considerably worse on A64FX. If this is the case but one still desires high performance, there seems to be no way around manual optimizations in assembly. This could be achieved by a generator like PSpaMM, that showed good results on NEON and should therefore also perform well when extending it to SVE. While optimizations can and usually are beneficial to performance, they do not have to be. Over-optimizing, which resulted in a large amount of statements in source files, performed inconsistent and/or worse than less optimized code. Fine tuning and regular benchmarking of small optimization steps are necessary to avoid this.

List of Figures

Bibliography

[1] C. W. Antoine, A. Petitet and J. J. Dongarra. "Automated empirical optimization of software and the atlas project." In: *Parallel Computing* 27 (2001), p. 2000.

[2] Arm Limited. *Arm Architecture Reference Manual Supplement The Scalable Vector Extension (SVE), for Armv8-A*. URL: https://developer.arm.com/documentation/ddi0584/latest/ (visited on 19/12/2020).

[3] Arm Limited. *ARM Compiler armasm Reference Guide*. URL: https://developer.arm.com/documentation/dui0802/a/A64-General-Instructions/MOV--wide-immediate- (visited on 13/02/2021).

[4] D. F. Bacon, S. L. Graham and O. J. Sharp. "Compiler transformations for high-performance computing." In: *ACM Computing Surveys* 26.4 (1994), pp. 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406.

[5] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington and R. C. Whaley. "An Updated Set of Basic Linear Algebra Subprograms (BLAS)." In: *ACM Trans. Math. Softw.* 28.2 (June 2002), pp. 135–151. ISSN: 0098-3500. DOI: 10.1145/567806.567807.

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

[7] A. Breuer, A. Heinecke, S. Rettenberger, M. Bader, A.-A. Gabriel and C. Pelties. "Sustained Petascale Performance of Seismic Simulations with SeisSol on SuperMUC." In: *Supercomputing*. Ed. by J. M. Kunkel, T. Ludwig and H. W. Meuer. Cham: Springer International Publishing, 2014, pp. 1–18. ISBN: 978-3-319-07518-1.

[8] E. Calore, A. Gabbana, S. F. Schifano and R. Tripiccione. "ThunderX2 Performance and Energy-Efficiency for HPC Workloads." In: *Computation* 8.1 (2020). ISSN: 2079-3197. DOI: 10.3390/computation8010020. URL: https://www.mdpi.com/2079-3197/8/1/20.

[9] H. Eberle. "Aktuelle Techniken zur Leistungssteigerung von Mikroprozessoren." In: (Feb. 1995), p. 10.

[10] Eigen Project. *Eigen v3 benchmarks against popular competitors*. 2011. URL: http://download.tuxfamily.org/eigen/btl-results-110323/index-110323.html (visited on 13/02/2021).

[11] Fujitsu Limited. *A64FX Microarchitecture Manual*. 2020. URL: https://github.com/fujitsu/A64FX/ (visited on 10/11/2020).

[12] Fujitsu Limited. *A64FX_Datasheet*. 2020. URL: https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet.pdf (visited on 20/11/2020).

[13] GNU Project. *Extended Asm - Assembler Instructions with C Expression Operands*. URL: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html (visited on 23/01/2021).

[14] GNU Project. *GCC Release Series - Changes, New Features and Fixes*. 2020. URL: https://gcc.gnu.org/gcc-11/changes.html (visited on 03/02/2021).

[15] GNU Project. *GCC release timeline*. 2020. URL: https://gcc.gnu.org/develop.html#timeline (visited on 03/02/2021).

[16] K. Goto and R. Van De Geijn. "High-performance implementation of the level-3 BLAS." In: *ACM Transactions on Mathematical Software (TOMS)* 35.1 (2008), pp. 1–14.

[17] G. Guennebaud, B. Jacob et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org (visited on 13/02/2021).

[18] GW4. *GW4 Supercomputer Isambard*. 2020. URL: https://gw4.ac.uk/case_study/gw4-supercomputer-isambard/ (visited on 02/11/2020).

[19] A. Heinecke, G. Henry, M. Hutchinson and H. Pabst. "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation." In: Nov. 2016, pp. 981–991. DOI: 10.1109/SC.2016.83.

[20] Intel Corporation. *libxsmm docs*. 2021. URL: https://libxsmm.readthedocs.io/en/latest/libxsmm_valid/#cross-compilation-for-arm (visited on 20/02/2021).

[21] John Linford. *Compiler flags across architectures: -march, -mtune, and -mcpu*. arm community, 2019. URL: https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/compiler-flags-across-architectures-march-mtune-and-mcpu (visited on 23/01/2021).

[22] A. Klöckner. "Loo.py: transformation-based code generation for GPUs and CPUs." In: *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming*. Edinburgh, Scotland.: Association for Computing Machinery, 2014. DOI: {10.1145/2627373.2627387}.

[23] C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh. "Basic linear algebra subprograms for Fortran usage." In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

[24] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou and J. Dongarra. "High-Performance Matrix-Matrix Multiplications of Very Small Matrices." In: *Euro-Par 2016*. Ed. by P.-F. Dutot and D. Trystram. LNCS Sublibrary: SL1 - Theoretical Computer Science and General Issues. Springer, 2016, pp. 659–671. ISBN: 978-3-319-43659-3.

[25] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge and R. Dreslinski. "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator." In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*. Piscataway, NJ: IEEE, 2018, pp. 724–736. ISBN: 978-1-5386-3659-6. DOI: 10.1109/HPCA.2018.00067. URL: http://blaauw.engin.umich.edu/wp-content/uploads/sites/342/2019/12/OuterSPACE-An-Outer-Product-based-Sparse-Matrix-Multiplication-Accelerator.pdf (visited on 21/10/2020).

[26] D. A. Patterson and J. L. Hennessy. *Computer organization and design: The hardwaresoftware interface*. Fifth edition. The Morgan Kaufmann series in computer architecture and design. Amsterdam and Boston: Elsevier/Morgan Kaufmann Morgan Kaufmann is an imprint of Elsevier, 2014. ISBN: 9780124077263.

[27] D. A. Patterson and C. H. Sequin. "RISC I: A Reduced Instruction Set VLSI Computer." In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. ISCA '81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, pp. 443–457.

[28] L.-N. Pouchet, A. Größlinger, A. Simbürger, H. Zheng and T. Grosser. "Polly-polyhedral optimization in LLVM." In: vol. 2011. Jan. 2011.

[29] E. Quinnell, E. E. Swartzlander and C. Lemonds. "Floating-Point Fused Multiply-Add Architectures." In: *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*. 2007, pp. 331–337. DOI: 10.1109/ACSSC.2007.4487224.

[30] RRZE-HPC/likwid. *likwid-bench*. 2020. URL: https://github.com/RRZE-HPC/likwid/wiki/Likwid-Bench (visited on 05/12/2020).

[31] D. Seal. *ARM architecture reference manual*. Pearson Education, 2001.

[32] N. Stephens. *Technology Update: Scalable Vector Extension (SVE) for Armv8-A*. URL: https://community.arm.com/developer/tools-software/hpc/b/hpc-blog/posts/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture (visited on 20/01/2021).

[33] V. Strassen. "Gaussian elimination is not optimal." In: *Numerische mathematik* 13.4 (1969), pp. 354–356.

[34] E. Strohmaier, J. Dongarra, H. Simon and M. Meuer. *Top500 Supercomputer rankings*. URL: https://top500.org/lists/top500/list/2020/11/ (visited on 19/11/2020).

[35] J. Treibig, G. Hager and G. Wellein. "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments." In: *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. San Diego CA, 2010.

[36] C. Uphoff, S. Rettenberger, M. Bader, E. H. Madden, T. Ulrich, S. Wollherr and A.-A. Gabriel. "Extreme Scale Multi-Physics Simulations of the Tsunamigenic 2004 Sumatra Megathrust Earthquake." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 1–16. ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126948.

[37] *XCI-Marvell Thunder X2*. 2018. URL: https://gw4-isambard.github.io/docs/user-guide/XCI.html (visited on 23/01/2020).

[38] Z. Xian-Yi, W. Qian, Z. Yun-Quan et al. "openblas: a high performance blas library on loongson 3a cpu." In: (2011).

[39] D. Yokoyama, B. Schulze, F. Borges and G. Mc Evoy. "The survey on ARM processors for HPC." In: *The Journal of Supercomputing* 75.10 (2019), pp. 7003–7036.

[40] T. Yoshida. "Fujitsu high performance CPU for the Post-K Computer." In: *Hot Chips*. Vol. 30. 2018.