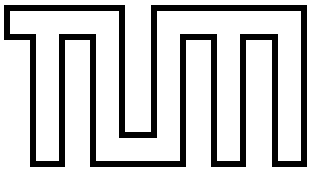


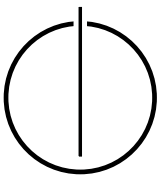
FAKULTÄT FÜR INFORMATIK

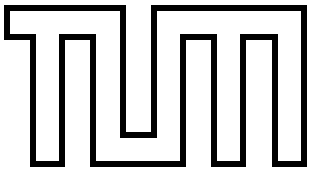
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

AutoPas on A64FX: Evaluation of Arm SVE Vectorization for Optimizing Molecular Dynamics Simulations

Timur Eke





FAKULTÄT FÜR INFORMATIK

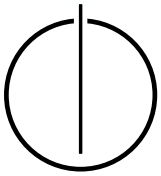
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

AutoPas on A64FX: Evaluation of Arm SVE Vectorization for Optimizing Molecular Dynamics Simulations

AutoPas am A64FX: Evaluierung der Arm SVE Vektorisierung für die Optimierung Molekularer Dynamik-Simulationen

Author:	Timur Eke
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	March 15, 2022



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

Munich, March 15, 2022 Timur Eke

Abstract

Molecular dynamics simulations of high-density, compute-intensive scenarios are well suited for SIMD vectorization. The simulation kernel of AutoPas, a particle simulation library, is already implemented with manual AVX2 vectorization for x86 architectures, as common compilers are unable to auto-vectorize the code.

The Fujitsu A64FX is an Arm CPU developed for the Fugaku supercomputer of the RIKEN Center for Computational Science in Japan, which leads several HPC performance rankings at the time of writing. To achieve peak performance, it supports Arm SVE, a novel SIMD instruction set extension featuring variable-length vectors and per-lane predication.

In this thesis, AutoPas is optimized to run on the A64FX. Specifically, the computation of the pairwise Lennard-Jones force is manually vectorized for the Arm SVE instruction set. Additional optimizations to hide instruction latency and utilize instruction level parallelism of the A64FX are evaluated, and the performance differences quantified and explained. A speedup factor of 9 compared to the unvectorized version is measured in appropriate simulation scenarios, and the performance is found to be comparable to the existing x86 implementation.

Contents

Part I.

Introduction and Background

1. Introduction

As power constraints limited the clock frequency increase for modern processors, a strong trend towards the usage of parallelism to achieve higher peak performance emerged. Vector processing is a vital part of these efforts. Over the years, Intel has released several instruction set extensions for vector computing, notably the SSE and AVX instruction families. All of them operate on different vector lengths and are designed for different tasks, from media processing to computational applications. SVE, a novel vector instruction set for Arm processors takes a different approach: the vector length is not fixed, and is determined by the hardware running the code. The supercomputer with the highest peak performance at the time of writing, Fugaku, as well as latest smartphones support SVE. Needless to say, this instruction set is promising for HPC applications: Fugaku was co-developed with SVE.

Molecular simulations are highly parallelizable and thus suitable for evaluating a vector instruction set. AutoPas, an auto-tuning particle simulation library, is especially suitable for this task, as optimal data layouts and iteration methods are chosen automatically. Thus, the optimization efforts are well targeted. The workload is too complex for auto-vectorization by compilers, so it will be manually vectorized for the CPU used in Fugaku, the A64FX. Performance will be evaluated, and bottlenecks recognized and eliminated. Finally, higher-level optimizations are performed to better utilize the parallel capabilities of the processor.

2. Background

2.1. AutoPas

AutoPas[1] is a short-range particle simulation library, which sustains optimal performance by continuous algorithmic tuning. The library simulates the interaction of particles with a pairwise force acting between them. Different particle containers, parallel traversal methods, and data layouts are evaluated periodically at run-time, so an optimal algorithmic approach can be used continuously throughput the simulation [SGH+21]. To the user, AutoPas is a black box particle container, which only requires an interaction force function and a list of particles to run a simulation. Scaling across multiple threads is supported using OpenMP [DM98]. The Lennard-Jones potential, which will be discussed in Subsection 2.1.1, is implemented as a force function in AutoPas because of its popularity [WRHDF20]. MD-Flexible, an example application for molecular dynamics simulations, is supplied with AutoPas. It can generate particles with various stochastic and deterministic placement methods, such as in a grid or following a normal distribution. MD-Flexible is used throughput this thesis to evaluate the performance of optimizations.

2.1.1. Particle Containers

To simulate one time instant (also called iteration) of a molecular simulation, the interaction force for every pair of particles has to be calculated and applied.

Short-range particle simulations allow to reduce the number of (expensive) force calculations by introducing a cutoff radius: Only the pairs of particles with a distance less than the cutoff radius can interact. This is facilitated by the potential (and force, which is the derivative) rapidly approaching zero for larger distances, as can be seen on the function graph 2.1. The Lennard-Jones potential is given for reference as a function of the distance between two particles. ϵ and σ are parameters which govern the shape of the curve:

$$V(r) = 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right) \tag{2.1}$$

[1] https://github.com/AutoPas/AutoPas

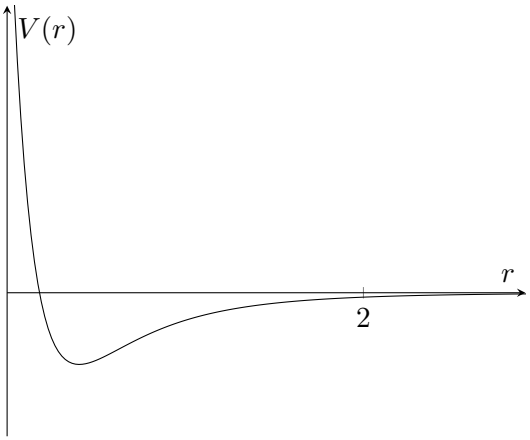


Figure 2.1.: The Lennard-Jones potential as a function of r, the particle distance. The cutoff can be at $r = 2$, where the the potential and thus the force are negligible.

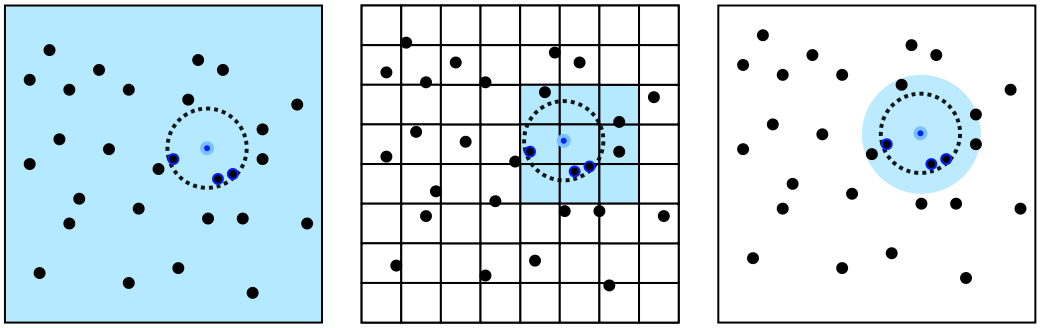


Figure 2.2.: Particle containers in AutoPas. Interaction partners are searched for the particle in the middle of the dashed cutoff radius circle. Only the distances to particles in the blue area are calculated. The force is calculated for highlighted particles.

The direct way to implement the simulation is to organize all n particles in a single list. For every pair of particles, if they are closer than the cutoff radius, the force is calculated and applied. This particle organization and iteration scheme is called **Direct Sum**, and results in $\mathcal{O}(n^2)$ distance checks per iteration. This is illustrated in Figure 2.2: For a particle, every other particle has to be checked as a potential interaction partner, while only those within the cutoff radius can interact.

Linked Cells is a more optimal, spatially aware scheme, which avoids quadratic scaling. The space is divided into a grid of cells, with a separate particle list per cell and particles moving between cells based on their position. A two-dimensional grid is shown in Figure 2.2. Each side of a cell is usually longer [SGH+21, p. 25] than the cutoff radius, while smaller cells are also possible with respective adjustments. Thus, for a particle, only the directly neighboring cells (highlighted) have to be checked for potential interaction targets. This cuts the complexity down to $\mathcal{O}(n)$ in terms of distance checks. Still, many of those checks are unnecessary: in three dimensions, the volume of the nine neighboring cells is significantly larger than of a sphere with the cutoff radius: $27c^3 \geq \frac{4}{3}\pi c^3$.

With **Verlet Lists**, neighbor lists for each particle are used to further reduce the number of distance checks. A maximum particle speed is assumed such that the neighbor lists remain valid for multiple iterations: A neighbor list for a particle contains references to all particles it could interact with until the next time the list must be recalculated [Ver67]. This is shown in Figure 2.2, where particles are inside of a sphere (or a circle in two-dimensions) with a radius slightly larger than the cutoff radius[2] [Ver67]. While the amount of distance checks is lower compared to previous variants, the non-contiguous particle access pattern along with the large number of neighbor lists places a comparatively large strain on the memory. Former particle organization and iteration schemes, which inhibit streaming particle access pattern, tend to be more computationally expensive, as determined in further analysis.

In AutoPas, the algorithms mentioned are implemented as particle containers. **Linked Cells** are used as the underlying storage mechanism for **Verlet Lists**. Various additional variants of **Verlet Lists** are available, but have similar properties and are not relevant for this thesis. As implemented in AutoPas, traversal methods govern the order of cell processing, and their distribution for parallelization. They are out of scope for this thesis.

2.1.2. Data Layouts

There are two common options for arranging the particle data in memory, which are illustrated in Figure 2.3. With **Array of Structures (AoS)**, all properties of a particle are stored contiguously in a single array of particle objects. In contrast, **Structure of Arrays (SoA)** means that values for every particle property are stored in separate arrays. In the context of computer architecture, performance benefits differ based on the access pattern. For sequential access to a subset of particle properties, SoA is beneficial: The respective arrays are able to be streamed, while arrays of unneeded properties are not read. On the contrary, AoS is advantageous for non-sequential access and when most of the properties are read [FSS13]. AutoPas is capable of automatically tuning the data layout during the simulation for optimal performance.

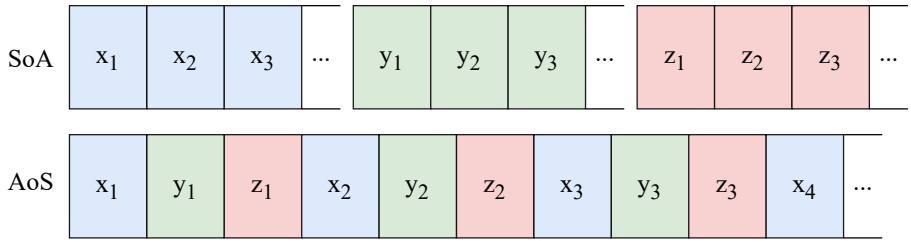


Figure 2.3.: Data layouts supported by AutoPas. Here, sets of three coordinates are stored in a Structure of Arrays and an Array of Structured, respectively. Horizontally neighboring cells are contiguous in memory.

[2]Given d, the maximum distance travelled between list recomputations, the radius of this sphere would be $r_{\text{cutoff}} + d$

2.2. Arm SVE

Arm SVE (Scalable Vector Extension) [SBB+17] is a SIMD extension of the AArch64 architecture, thus available only for 64-bit Arm devices. SIMD (Single Instruction Multiple Data) describes a class of instructions which operate on vectors instead of scalars. A vector consists of multiple scalars, each called a "lane". One SIMD addition instruction results in an element-wise addition of two vectors. SVE is an optional addition to Armv8.2-A and newer instruction sets, and is supported by "standard Armv9-A software platforms" [Lim22, p. A3-126].

In contrast to AVX, the prevalent SIMD instruction set on x86, the vector length for SVE is specified not by the instruction set, but by the underlying hardware and can range from 128 to 2048 bits in increments of 128 bits. This allows for platform-agnostic vectorization, with no need for recompilations. Integer (signed, unsigned, 8-64 bit) as well as floating point (16-64 bit) data types for vectors are supported. Smaller data types can be packed, so a vector consisting of four 64 bit lanes can also be used as eight 32 bit lanes.

A central feature of SVE is per-lane predication. Apart from 32 vector registers, the architecture offers 16 predicate registers. They are independent from vectors, and also variable in length. Most of vector operations require a predicate vector to control, for which lanes the operation shall be performed (which lanes are considered active). Some have modifiers to indicate if inactive lanes of the result are set to zero, merged with the target register, or considered to be arbitrary. A predicated vector addition is illustrated in Figure 2.4, with the zeroing modifier. While immediate predicate creation is limited to repeating 128 bit patterns due to unknown length at compile time, special initializers for loop control[3], and various bitwise operations are available.

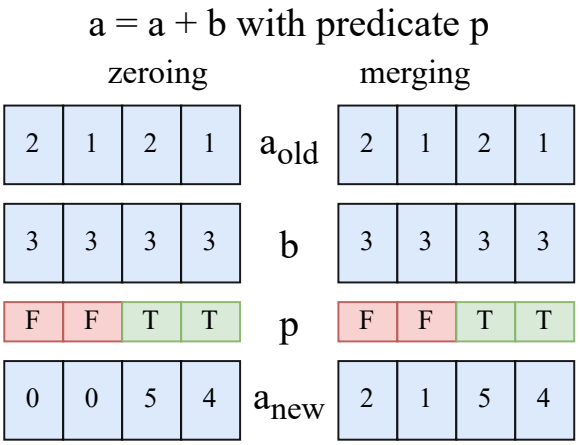


Figure 2.4.: Two vector registers are added with SVE. The predicate `p` controls which lanes the addition is performed for. The merging modifier leaves old values of inactive lanes intact, while the zeroing modifier causes them to be set to zero. Vector length of four was chosen for compactness.

[3] "Set the bits for the first i lanes to active"

The instruction set supports gather-scatter instructions with vector indexing, contiguous and strided memory access instructions. Among other supported features are horizontal reductions, which accumulate elements of a vector to a scalar. The floating-point addition reductions `fadda` (sequential) and `faddv` (tree-based) are specifically used, which require less processor resources compared to a manual accumulation (e.g. on AVX).

Four-operand fused multiply-accumulate operations, unlike the three-operand variants, can have the form $A = B * C + D$, and add more possibilities for fusing multiplication and addition/subtraction instructions to save computational time. Furthermore, vector compaction and slicing instructions for permutation of active elements in vectors are used to fill vectors and maximize useful computation.

2.3. Low-Level Optimization

Concepts to reason about optimization for modern processors, and several optimization patterns are presented in the following.

2.3.1. Processor Features

Instruction Level Parallelism Modern processor architectures, which are the target for optimizations discussed in this thesis, consist of multiple execution units for different types of instructions. As an example, a floating point and an integer addition could be computed in parallel on the respective execution units. To better utilize these resources, a processor may execute instructions out of program order. The reordering subsystem recognizes future instructions with no dependencies on in-progress computations and assigns them to execution units. For this, execution units have the so-called reservation stations, which store pending operations with the corresponding operands. After an execution unit has completed the operation, the resulting state change is applied again in program order, guaranteeing consistency. Given enough independent instructions, the so called superscalar processors can execute multiple instructions in parallel. This is referred to as instruction level parallelism (ILP) [JW89]. The amount or the degree of ILP is the number of instructions that are executed concurrently.

Latency, Pipelining, Dependency Chain The latency of an operation is the number of clock cycles until the results are available. Arithmetic execution units are organized as a pipeline. This means that every instruction, i.e a multiplication, is executed in a number of sequential stages, with every stage needing a clock cycle to complete. While an individual instruction needs multiple cycles to complete, the throughout is still one instruction per cycle for a fully loaded pipeline. Independent instructions are needed to be able to "fill" the pipeline and to hide the latency of individual operations, so a degree of ILP is needed to utilize pipelining in the execution units.

A chain of (usually arithmetic) instructions, where each one is dependent on the previous is called a dependency chain. The latency for executing the whole dependency chain is the sum of individual instruction latencies, as no ILP is available. The critical path dependency chain of a code region is the longest one in terms of latency and determines the execution time of the region.

Operational Intensity, Roofline Model The operational intensity is the ratio between total FLOPs and total bytes transferred to/from the main memory. It is used to quantify the bottlenecks of a program. A high operational intensity means that the program is more likely to be compute-bound, depending on the capabilities of the CPU. The values used include FLOPs from inactive vector lanes and bytes from hardware prefetches.

Roofline models [WWP09] are used to evaluate program performance for a specific processor. They set an upper bound for performance (FLOP/s) of a program depending on its operational intensity. These limits are determined from processor resources.

Performance Events Modern processors record certain microarchitectural events (cache miss, instruction execution) in hardware performance counters during program execution. These can be measured to access low-level statistics, like instruction/cycle ratio or the number of memory transfers.

2.3.2. Optimization Techniques

The following optimization techniques are normally performed by the compiler. For complex workloads, manual intervention is required. The optimizations do not change the workload, only the instruction ordering and/or the loop structure to improve performance. The structural changes to the code are illustrated in Figure 2.5.

Instruction Interleaving

Instruction interleaving is a specific optimization technique for mutually independent, consecutive compute-heavy dependency chains. Normally, the reordering subsystem should recognize the independence of instructions from different dependency chains and hide the latencies with pipelining. Alternating instructions would be issued, improving the performance relative to the serial execution of the chains. If the hardware is unable to do so, for instance if the chains are especially long, instruction interleaving can be applied. For this, the desired reordering behavior is replicated by interleaving multiple dependency chains. This essentially removes the need for an out-of-order processor, and is also effective for in-order systems [CCMH91]. But, as intermediate values for each of the dependency chains must be stored, more registers are used, limiting the attainable level of instruction interleaving.

Loop Unrolling

A conventional method to optimize loops is to unroll them: The loop body is repeated and the header is adjusted such that k iterations of the loop are performed as one iteration of the k-unrolled loop. This is shown in Figure 2.5, with the original loop iterations marked with digits. If the total number of iterations is not divisible by k, the remaining iterations are processed separately from the unrolled loop.

This optimization has two main effects: reduced loop overhead and improved ILP. Loop condition checks have to be performed k times less often, so more "useful" instructions can be executed. Additionally, branches can be avoided due to less frequent loop condition checks. Also, given that the loop is not trivial, more independent instructions are available to the reordering subsystem, allowing for higher ILP.

This can result in "substantial speedup increases" of three times in some cases [MCG+92]. The drawback of higher levels of unrolling is the increased program size and thus a higher load on the instruction cache and decoding subsystem. Also, for complex loop bodies, the processor can be hindered from executing instructions out-of-order due to an insufficient number of intermediate registers [DJ95].

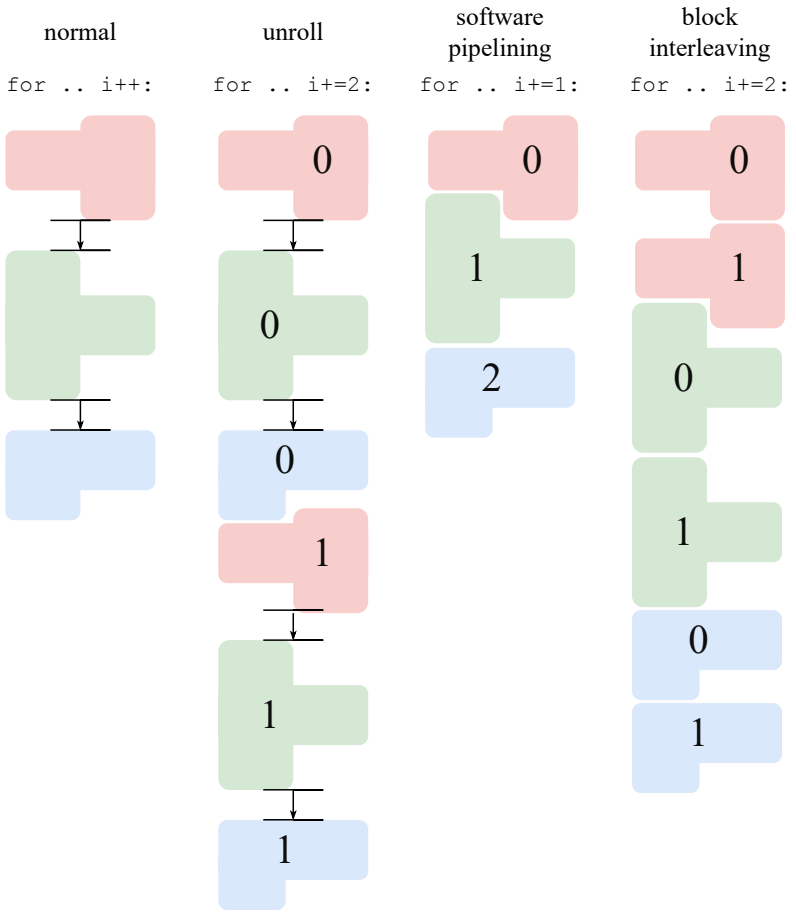


Figure 2.5.: Structural changes for loop optimization techniques are depicted. The loop is divided into three parts (red, green and blue), each with different resource requirements, which is represented by varying heights of the two columns. The second part (green) may execute many memory loads, thus having a larger left column. Subsequent parts from an original iteration are dependent on each other, and their execution cannot be overlapped, depicted by arrows. For all the optimizations, parts from different original loop iterations are performed during one new iterations. These are identified by the offset numbers.

Software Pipelining

Software pipelining is a technique to execute loop iterations in a pipelined way. The loop body is divided into k subsequent parts (size can range from individual instructions to code blocks), which are used as pipeline stages. In Figure 2.5, the loop body is already subdivided. Similarly to unrolling, original iterations are numbered. The workload of one original iteration is executed in parts during k iterations of the pipelined loop: One stage is completed per iteration. In turn, computations for k original iterations are ongoing during one iteration of the pipelined loop, once the pipeline is full. At the beginning, the pipeline has to be filled for $k-1$ iterations, as no computation has reached the latter stages. Similarly, for the last $k-1$ iterations, the pipeline is also not operating at the full capacity: Remaining computations have to be completed, and the initial stages are idle.

The major difference to general pipelining is limited parallelism: Stages are still executed sequentially within the loop body. Thus, software pipelining does not improve performance for in-order processors.

But, for out-of-order processors, software pipelining can be beneficial. When there are limited dependencies between subsequent iterations, overlapping them makes more independent instructions available for the processor. Compute-heavy dependency chains can be broken up this way to utilize pipelining in the execution unit and to hide instruction latencies. This can lead to an increase in ILP. In the example illustrated in Figure 2.5, the second and the third stage can overlap and utilize the full capacity of the processor, because the dependence between the stages is removed.

The major disadvantage of software pipelining for complex loops is that intermediate values need to be stored between pipeline stages [LVAG98]. So, the number of registers (or the size of the L1 cache) becomes the limiting factor for the number of stages. Unlike unrolling, the register assignment can be influenced directly by the programmer or the compiler.

Software Prefetching

For memory-bound code regions, the memory bandwidth must be utilized as much as possible to achieve optimal performance. In modern processors, the hardware is able to recognize some basic access patterns (streaming, strided) and to preemptively load predicted future memory addresses in a process known as *hardware prefetching*. The data is then available in the cache when the address is accessed by the program. For more complex access patterns, a hint may be given to the hardware, that an memory address will be accessed in the future with a special instruction. This is called software prefetching. Its usefulness highly depends on the context and on the application as it often interferes with hardware prefetching [LKV12].

2.4. Fujitsu A64FX

The A64FX (introduced in 2019) is a superscalar processor implementing the Armv8.2-A instruction set architecture including SVE with a vector width of 512 bit. It is used in the Fugaku supercomputer of the RIKEN Center for Computational Science in Japan, which is leads major performance ratings at the time of writing. [Lim19]

Cores / Threads	48
Frequency	1.8GHz
SIMD Width	512 bit
Peak Flops (64 bit)	2.7648 TFLOP/s
L1D Cache Size	3MiB (64KiB /core)
L2 Cache Size	32MiB (8MiB x 4)
Cache-Line Size	256 bytes
Memory Bandwidth	1,024 GB/s
Memory Capacity	32 GiB (8GiB x 4)

Table 2.1.: A64FX specifications. The given frequency of 1.8 GHz was used throughout this thesis; Higher frequencies are configurable

To expand on the processor specifications in Table 2.1, the A64FX is subdivided into four Core Memory Groups (CMG), each with an independent L2 cache, and an independent memory controller. Each CMG has a separate physical memory space in a non-uniform memory configuration. Cache coherency between CMGs is guaranteed by the hardware [Lim].

Each core is equipped with two floating point execution units with reservation stations of 20 instructions each. The first execution unit supports more instructions. For instance, floating point division cannot be executed on the second one. As listed in Table 2.2, the latency of floating point operations is relatively high: Multiplication takes 9 cycles to complete. This means that utilizing ILP and pipelining is crucial for achieving near-peak performance. Also, the A64FX supports out-of-order execution: Four instructions can be issued in one cycle, and up to 128 instructions can be considered for reordering.

SVE add	9
SVE subtract	9
SVE multiply	9
SVE FMA	9
SVE divide	154
SVE adda	114
SVE addv	49

Table 2.2.: A64FX floating-point instruction latencies

AVX tests are performed on a node with two Intel Xeon CPU E5-2697 v3 processors.

Part II.

Implementation and Results

3. Methodology

In this thesis, AutoPas will be optimized to utilize the vector processing capabilities of the A64FX. For this, a consistent compute-bound workload must be found for valid performance measurements, and examined for vectorization candidates.

3.1. A Compute-Bound Scenario

First, a consistent compute-bound scenario is constructed by choosing an appropriate particle generator and a set of parameters for MD-Flexible. It will be used for all further benchmarks.

3.1.1. Simulation Parameters

To ensure consistent performance, a deterministic particle generator, `closestPacking`, is chosen. It generates the particles in a hexagonal grid such that they are packed as densely as possible, but maintain a certain separation distance to other particles. This distance can be adjusted using the `particle spacing` parameter, consequently affecting particle density. The total number of particles is set by the size of the bounding box, in which the particles are generated.

Given that other parameters are fixed, this particle generator is also beneficial for maximizing the number of interactions for the same number of particles, which is limited by the relatively small 32GiB system memory of the A64FX.

Additionally, the time difference between subsequent iterations, `deltaT`, is set to zero to effectively stop particles from moving. This is done to preserve consistency between iterations, while not affecting the computational workload.

Other parameters, including ϵ and σ for the Lennard-Jones potential, are left at their default values[1]:

`Generator`	`closestPacking`
δt	0
Cutoff radius	2 units
ϵ	1
σ	1

Table 3.1.: Parameter values for all scenarios

[1] `Newton3` calculation is enabled, and `globals` calculation is disabled for all results in the thesis, unless specified otherwise

3.1.2. Bounding Box Size

Using a particle spacing set to the default value of 1.1225 units, for which the forces on particles are balanced, the following parameters are benchmarked on the non-vectorized implementation (48 threads) to determine the optimal box size:

Size	Side length of the box for particle generation
Iterations	Number of iterations simulated

Table 3.2.: Parameters to be tuned for further benchmarks

For a small number of iterations, fixed costs like domain initialization reduce the achievable performance. Similarly, a small box size may prevent the workload from being fully parallelized, for instance when threads process an insufficiently large number of particles. On the other hand, if the iteration number or the box size are set too high, the runtime may be suboptimal for rapid data collection. Also, as 128 bytes are stored per particle, the relatively small 32GiB memory size of the A64FX has to be considered for avoiding large particle numbers.

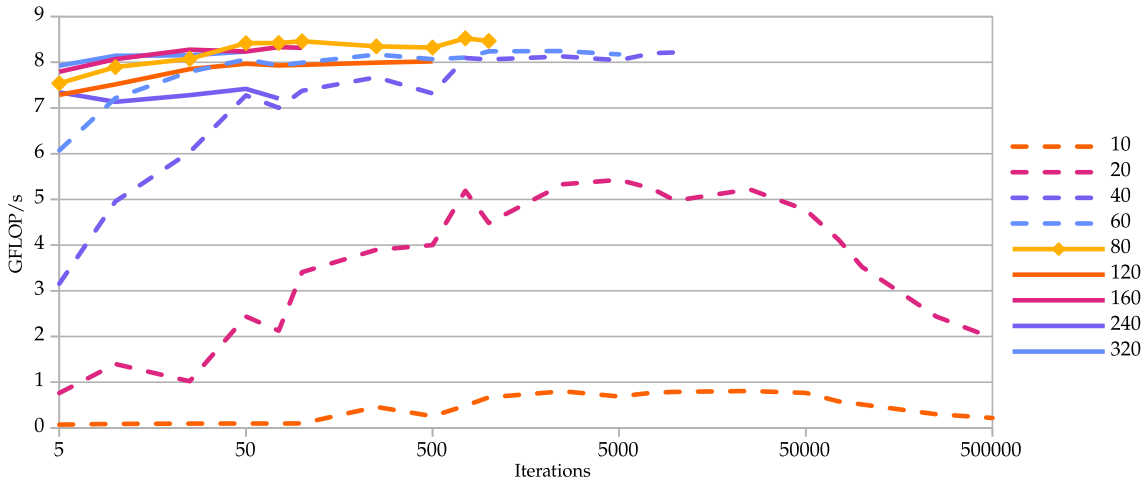


Figure 3.1.: The performance is measured for various box sizes and iteration counts with a capped execution time. The configuration AutoPas is using is the optimal one it determined for higher box sizes and iterations: SoA, Linked Cells, lc_08 iterator.

As can be seen from Figure 3.1, a box with the side length of 80 units is the smallest one to not lose performance for low iteration counts due to initialization and other overheads. Also, it is well parallelizable, unlike smaller box sizes, which hit a performance bound. The largest scenario, which will be determined in the following, avoids this memory limitation with 26.9 million particles and 3.2 GiB of memory usage. For further benchmarks, the iteration count will be set individually such that the performance is not impacted by initialization time.

3.1.3. Spacing and Operational Intensity

With a fixed box size, particle spacing can be adjusted to steer the operational intensity.

The lower the spacing, the more particles have to be simulated. As they are packed more densely in the same space and for the same cutoff radius, a particle interacts with more neighbors. Let k be the number of interaction partners for a particle, which increases with decreasing spacing. As the particle distribution is homogeneous, k is equal for the vast majority of the particles. k approximates the volume of a sphere with the cutoff as its radius, so it is directly proportional to the number of particles per cell of the Linked Cells container. This metric can be easily calculated from the total number of particles, as the box size and the number of cells are static. *Particle/cell*, or, more generally, the density, is indicative of the computational cost and is used for the rest of this thesis instead of spacings, unless specified otherwise. The cubic relationship with spacing is demonstrated in Figure 3.2.

As k increases, the total number of pairwise interactions increases quadratically, so the computational complexity is $\mathcal{O}(k^2)$. On the other hand, the number of memory transfers can be reduced because adjacent particles have largely overlapping sets of neighboring particles. Thus, the L2 cache on each core can be used for neighboring particles, if the total memory requirement is less than the L2 cache size:

$$k < \frac{\text{size}_{\text{L2}}}{\text{cores} * 128B} = 5460 \text{ particles}$$

The memory complexity is bounded by $\mathcal{O}(k)$, given enough spatial locality for particle iteration. Combined, this leads to a higher operational intensity for higher densities, as illustrated in Figure 3.2.

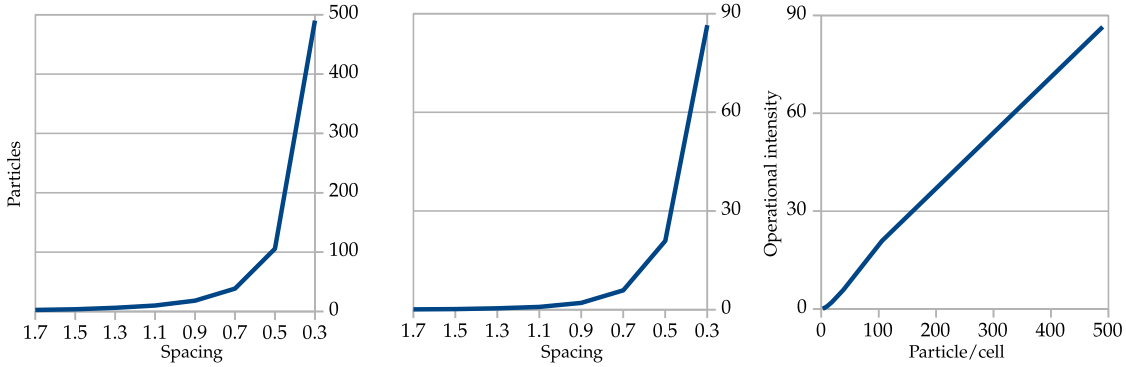


Figure 3.2.: The relationship between spacing and the number of neighbors, or also particle/cell, is cubic, as the simulations are three-dimensional. The operational intensities increase linearly with increasing particle/cell.

Equivalence with Other Scenarios It has to be noted, that the scenarios with the lowest spacings are unrealistically dense for most real-world applications. Nonetheless, equivalent computational behaviour can be achieved for similar particle/cell values. These can occur in scenarios with a lower density but a bigger cutoff radius, or in high-density regions in otherwise more sparse simulations.

The optimizations performed in this thesis will be focused on scenarios with the lowest spacings, which are computationally bound.

3.2. Vectorization Candidates

Once a compute-bound workload is chosen, it is examined to find candidates for vectorization. In the case of AutoPas and the selected scenario, such a candidate is the pairwise force update procedure. As is determined from an AutoPas timing report, it accounts for over 95 percent of the total computation time. Such a report is presented for a similar scenario:

```
Measurements:
Total accumulated               : 335685377639 ns (335.685s)
  Initialization                :  10405573266 ns ( 10.406s) =  3.100%
  Simulate                      : 319738966023 ns (319.739s) = 95.250%
    ForceUpdateTotal            : 319738936143 ns (319.739s) =100.000%
      ForceUpdatePairwise       : 319738931683 ns (319.739s) =100.000%
      ForceUdpateGlobalForces   :         3620 ns (  0.000s) =  0.000%
      ForceUpdateNonTuninng     : 319738931683 ns (319.739s) =100.000%
One iteration                   :  21315931068 ns ( 21.316s) =  6.350%
Total wall-clock time           : 335685377639 ns (335.685s) =100.000%
```

The pairwise force update procedure is implemented for different particle container and data layout combinations in the `LJFunctor.h` file in AutoPas. A method for AoS, and methods for Linked Cells/SoA and Verlet Lists/SoA are available respectively, to be chosen at runtime. The general algorithm for the pairwise force update is shared:

Algorithm 1: Pairwise force update algorithm

```
1 for p_i of all particles do
2     for p_j of neighbors of p_i do
3         F_i = 0
4         if distance d_ij within the cutoff then
5             Calculate the Lennard-Jones force F_ij
6             F_i += F_ij
7             Apply F_ij to p_j
8     Apply F_i to p_i
```

It is structured in two nested loops, while the inner one iterates over the neighbor list for Verlet Lists and over particles in several cells for Linked Cells. The exact number and relative location of the cells is dependent on the traversal method. So, for one execution of the inner loop, **the iteration count is proportional to particle/cell** for Linked Cells. Also, the particles in the outer loop are iterated sequentially within one cell, so the neighbors for all those particles identical, and that data can be cached. The body of the inner loop is the vectorization target, and will be referred to as kernel.

3.3. Implementation Basis

As AutoPas automatically selects SoA and Linked Cells as the best-performing configuration for the selected scenario due to streaming access to particles, optimization efforts are focused accordingly. The SoA Verlet Lists implementation is also optimized, as the only difference to Linked Cells is the memory access pattern, so the kernel can be extracted into a generic method. As discussed previously, Verlet Lists are unsuited for high-density uniform scenarios due to large memory overhead and non-streaming access. Thus, vectorization is evaluated only for Linked Cells.

The workload is too complex for SVE auto-vectorization by common compilers (GCC, Clang, Fujitsu). A manually vectorized AVX2 implementation is present for the SoA data layout, which is used as the basis for SVE vectorization. Arm C Language Extensions are used for SVE intrinsics, which offer C/C++ bindings to SVE instructions and according data types. The existing AVX2 vectorization also utilizes intrinsics.

3.4. Experimental Setup

To analyze performance in detail, performance event counters are used. The used kernel version offers A64FX-specific events only as raw events, which have to be accessed by their hexadecimal addresses. The calculations needed to infer the number of FLOPS, the memory transfers, cache misses, etc. can be found in the microarchitecture manual of the A64FX [Lim] and a full list of events is also available. `perf record` is used to identify hotspots for certain events. Table 3.3 summarizes the experimental setup[2]:

GCC	`11.0.0 20201028`
GCC Flags	`-O3 -DNDEBUG -fno-math-errno -fopenmp-simd -march=native -fopenmp`
OpenMP	`4.5`
CMake	`3.18.3`
Kernel	`4.18.0-193.19.1.el8_2.aarch64`
AutoPas	`22556fb4`

Table 3.3.: Experimental setup

[2]GCC flags `-mtune=a64fx -msve-vector-bits=512` have been considered, but didn't affect the performance

4. Vectorization

4.1. Kernel Vectorization

For manual vectorization, the AVX2 version is used as the starting point for implementation. This reuses the general code structure and maintains the operation ordering and fused-multiply-accumulate instructions to allow for a one-to-one comparison.

One change to the inner loop is needed: As the AVX2 implementation has a known vector width of 4, and a performance penalty for predicated (masked) operations is assumed, the inner loop operates only on the first $n - n \ mod \ 4$ particles instead of the whole n. This avoids masked instructions for the majority of the computations, and the remaining $n \ mod \ 4$ particles can be processed immediately after the inner loop exits.

As the vector length for Arm SVE is not known at compile-time, and predication does not affect performance[1], it is used to simplify the code. The special `whilelt` instruction for loop control is used to construct a predicate which is fully active for the first $n - n \ mod \ k$ particles, and where only the first $n \ mod \ k$ lanes are active for the last iteration. Thus, the code is portable across vector lengths and idiomatic for SVE. The resulting code structure is sketched in Algorithm 2: The inner loop operates in k particles at once.

Algorithm 2: Pairwise force update algorithm, vectorized

```
1 for p_i of all particles do
2     [F_i]_k = [0]_k // vector accumulator with k elements
3     for [p_j]_k of neighbors of p_i in batches of k do
4         pred_dist = lane x active if d_i(j+x) within the cutoff
5         if pred_dist has active lanes then
6             Calculate the Lennard-Jones force [F_ij]_k for active lanes of pred_dist
7             [F_i]_k += [F_ij]_k for active lanes of pred_dist // element-wise
8             Apply [F_ij]_k to p_j for active lanes of pred_dist
9     Apply sum([F_i]_k) to p_i // sum of all elements
```

[1] Verified with a micro-benchmark

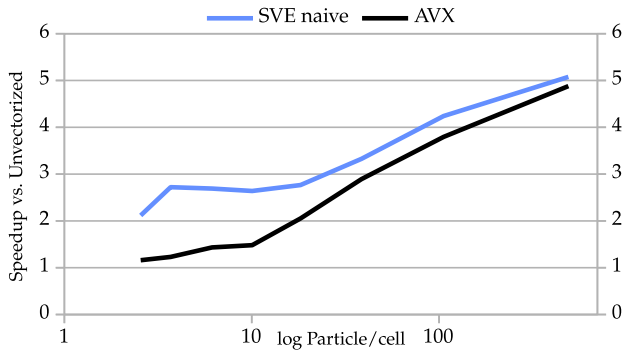


Figure 4.1.: The speedups compared to the unvectorized versions for AVX and SVE are depicted, with full multi-threading supported by the processors and the SoA Linked Cells lc_08 configuration.

Initial Comparison with AVX

As can be seen in Figure 4.1, the speedup for SVE is two times higher than for AVX for small numbers of particles per cell. This coincides with the twofold difference in vector length: 4 for AVX2 and 8 for SVE on the A64FX. As the workload is memory-bound, these bottlenecks outweigh possible computational limitations. Once the data is loaded, the A64FX is quicker to process it due to the longer vectors.

For larger particle/cell (and more than one or two iterations of the inner loop), the speedups are similar. This hints at a compute bottleneck in the kernel for the SVE version. It will be identified in Subsection 4.3.2.

4.2. SVE-Specific Kernel Optimization

Some parts of the kernel can be improved with SVE features not available on AVX2, with a breakdown of all performance improvements in 4.2.

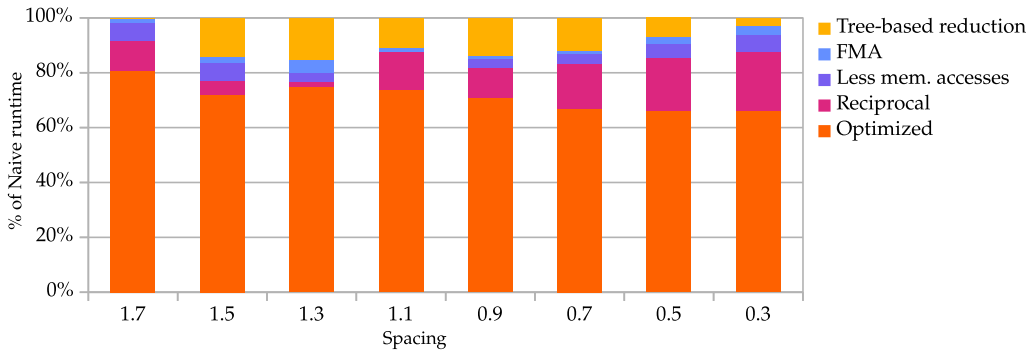


Figure 4.2.: The consecutive runtime improvements resulting from SVE-specific optimizations are depicted for different particle spacings. The higher spacings are memory bound, so analysis is focused on lower particle spacings.

Reciprocal

One such feature is the floating point reciprocal calculation using Newton-Rapson [BPR21] approximation. This replaces the single division operation for the inverse of the squared distance of the Lennard-Jones potential, k:

$$V(r) = 4\epsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right) = 4\epsilon\left(k_2^6 - k_2^3\right); k_2 = \sigma^2\frac{1}{r^2} \tag{4.1}$$

Floating point division has a high latency both on SVE and AVX2 compared to multiplication, of 9 vs. 154 and 4 vs. 35 cycles, and is pipeline blocking. This means that when a division is being calculated, pipelining in the execution unit cannot be used to improve throughput. This is why replacing the division with a reciprocal calculation is beneficial.

While the Newton-Rapson approximation algorithm is not relevant for the thesis, the instruction mix required to execute it is presented. First, an initial approximation is calculated using the `recpre` instruction, then that value is repeatedly multiplied with a step factor, determined at each step using the `recprs` instruction. All in all, four such steps (and multiplications) are sufficient to achieve bit-perfect accuracy in the tested cases.

With this optimization, the runtime is reduced by 20% for the highest particle/cell measured. This is explained by two factors: Firstly, cycles are directly saved: $154_{\texttt{div}} > 4_{\texttt{recpe}} + 4 * (9_{\texttt{mul}} + 9_{\texttt{recps}}) = 76$. Secondly, the multiplications no longer block the pipelining of the execution unit. The number of floating-point operations is increased by seven[2], which leads to a total FLOPS increase by 10% percent for the highest-density scenario. As this optimization happens inside the kernel, its effect grows with increasing iteration counts, and thus with particle/cell.

Tree-Based Reduction

The total force for a particle is the sum of all individual forces from interacting with neighbors. The summation of all elements in a single SIMD vector is a costly operation, so its usage is delayed until after the loop. Consequently, force components (x, y, z) are each accumulated using separate vectors. After the loop is completed, these vectors are summed up individually to three scalars (line 9 of Algorithm 2), thus performing the costly reduction operation only three times. To ensure consistency with the unvectorized implementation, the ordinary vector reduction is used, which sums elements sequentially.

The tree-based reduction instruction `faddv`, performs some of the additions in parallel, and is therefore faster (49 cycles vs 114 cycles, see Table 2.2). It makes use of associativity, and can cause numerical errors, but these are minor and negligible for molecular simulations. In the cases tested for this thesis, results were even identical. The optimization reduces the runtime by 10% for lower densities, but the effect is reduced for higher densities, as the time saved is constant regarding the inner loop.

[2] `recpe` instruction doesn't count as a FLOP

Fused-Multiply-Accumulate

Using the four-operand fused-multiply-accumulate instructions in SVE, additions and subtractions can be fused with preceding multiplication instructions, similar to other platforms. The resulting fused instruction has the latency of the multiplication, so one instruction is saved. For instance, the Lennard-Jones potential can be calculated using a negated fused-multiply-subtract instruction, highlighted in square brackets:

$$V(r) = 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right) = 4\epsilon \left([-(k_6 - k_6 * k_6)] \right); \; k_6 = \frac{\sigma^6}{r^6} \tag{4.2}$$

In this example, one subtraction operation is saved. But, as Lennard-Jones potential, and generally the whole kernel requires significantly more multiplications than additions or subtractions, as will be discussed in the following, the effect of fused-multiply-accumulate instructions is limited to a runtime improvement of around five percent in the best-case.

4.3. Performance Analysis

The performance of the *optimized* implementation, containing the optimizations from Section 4.2, is compared to both the *naive* and the *AVX* versions, and is examined for bottlenecks.

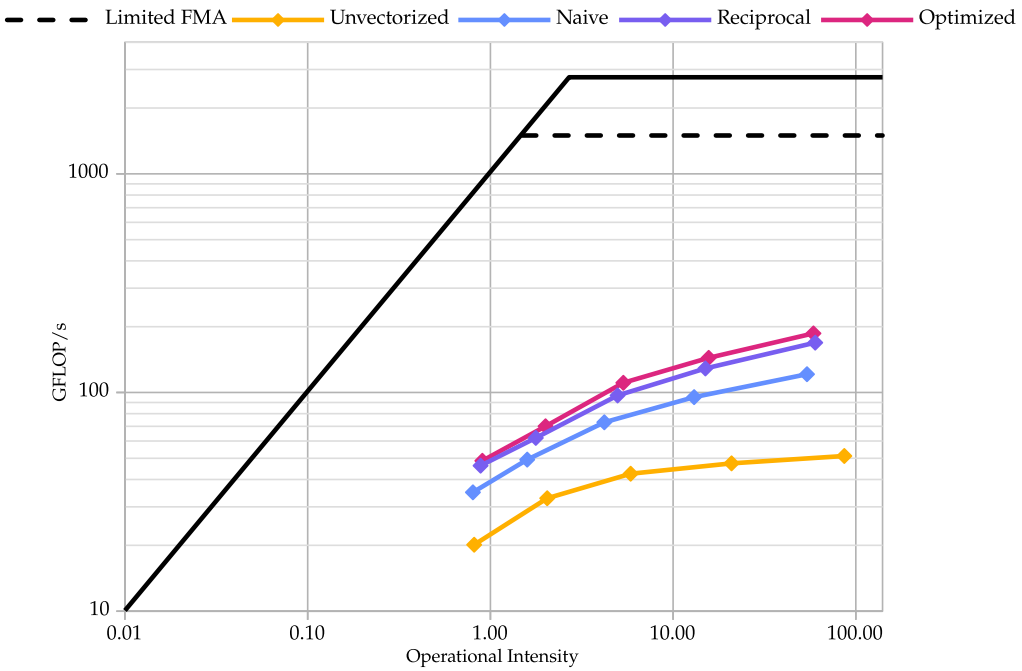


Figure 4.3.: The performance for different operational intensities of the variants is illustrated, along with a bound for peak performance of the A64FX. An additional bound for performance is shown, accounting for the limited fused-multiply-accumulate instructions possible in the kernel.

4.3.1. Performance Impact of Optimizations

By using SVE-specific features, the *optimized* version significantly outperforms the *naive* variant for all but the lowest densities. As the workloads of the latter scenarios are memory-bound, and the optimizations focus on the computational performance, this is insofar expected. For the compute-bound scenarios, the total number of floating-point operations, and thus FLOPS/s and the operational intensity all increased due to the replacement of a division with four Newton-Rapson steps. This difference is illustrated in the Roofline model in Figure 4.3: The *naive* version is to the left (lower operational intensity) and lower (less FLOP/s) than the optimized ones.

The CPU utilization during a simulation run can be illustrated and quantified using cycle analysis. Performance counters reveal the number of cycles the CPU spent stalled, waiting for either a floating-point operation or a memory access to complete, or for other reasons[3]. Also, the number of cycles where one, two, three or four instructions finished execution can be gathered. But, as the analyzed application is too complex for any meaningful conclusions, these will be summarized into one category. The cycle distribution is visualized with stacked bar charts in Figure 4.4.

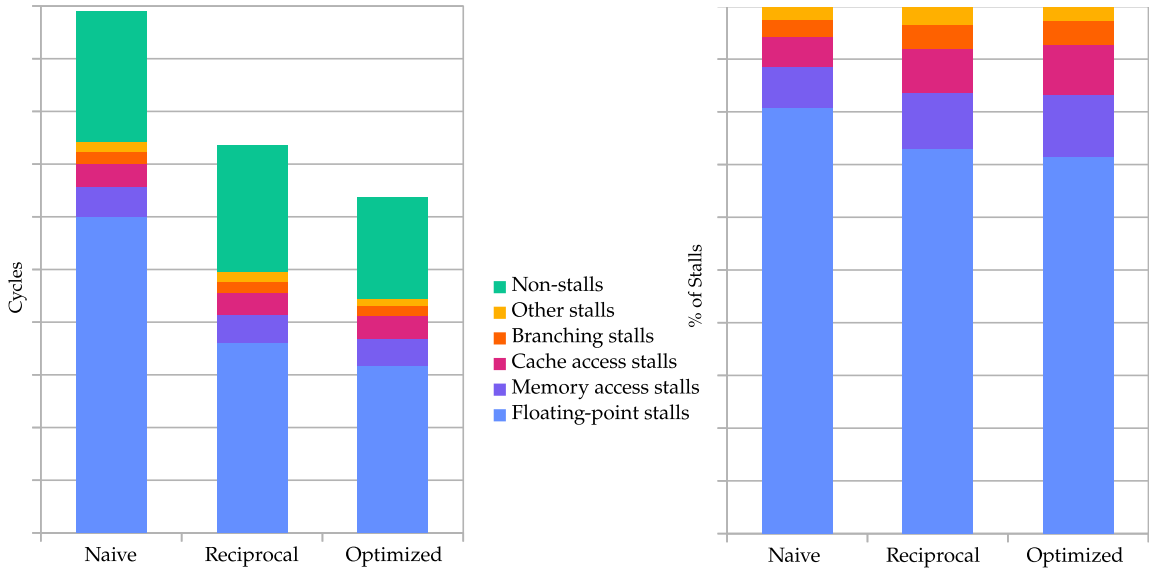


Figure 4.4.: Relative numbers of stall cycles and absolute cycle distributions for cycle categories are compared for optimized variants.

The version with all the mentioned optimizations speeds up the computational part of the program. For high densities, this results in a combined ten percent reduction in compute stalls, but an increase of 5% in memory stalls, as memory transfers are executed more often. Overall, 5% less stalls are observed. The reduction in compute stalls is explained by the removal of the floating-point division. Removing this long-latency, pipeline-blocking operation allows other instructions to execute concurrently in the pipeline, reducing stalls.

[3] As multiple instructions can be pending concurrently, the stall category for cycle analysis is determined by the oldest instruction, if no other other instructions have finished execution

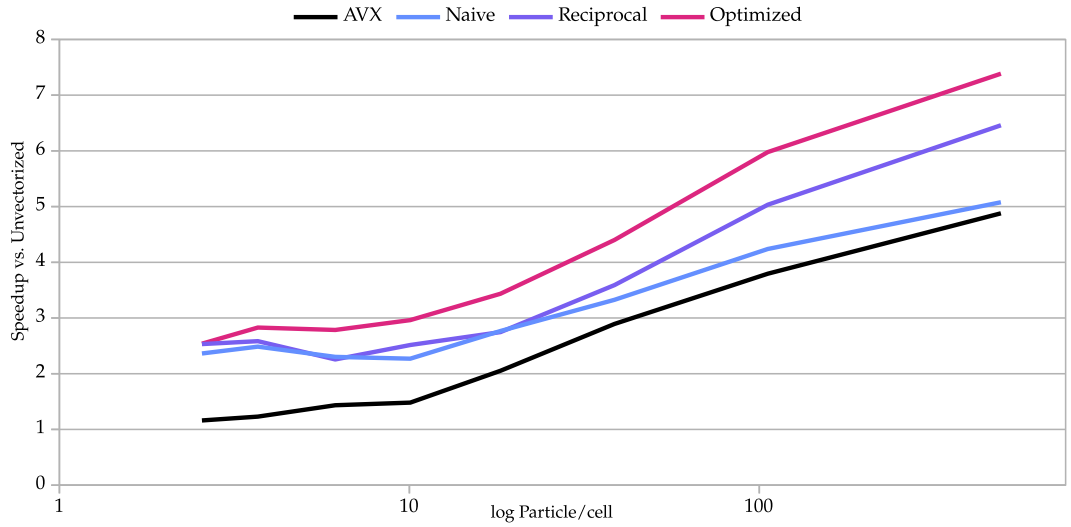


Figure 4.5.: Speedups compared to the unvectorized versions for AVX and SVE variants.

4.3.2. Vectorization Speedup

The speedup graph in Figure 4.5 illustrates the runtime improvement versus the unvectorized version. The major overall improvement in the speedup is achieved through the removal of the division instruction. It must be noted that due to a much lower latency of 43 cycles, the scalar division is faster than the scalar reciprocal calculation with four steps. This underlines the validity of the comparison. The speedup approaches the factor of 8, which would show perfect scaling without any bottlenecks considering the vector length of eight.

Also, the overall shape of the speedup for the version with the division removed is already comparable to the AVX version, which hints at the division being the bottleneck for SVE.

Memory bottlenecks hinder more memory-intensive scenarios, like the ones with low particle/cell values, from fully benefitting from vectorization. In that case, memory bottlenecks have to be mitigated with structural changes like different particle containers or traversal methods, which is out-of-scope for this thesis.

Problematic Number of Stalls

With the improvements in performance compared to the unvectorized version, the number of processor stalls shows a considerable increase. As particles are now processed eight times faster, 15 percent of the execution time is spent waiting for memory and cache accesses, three times more than with the unvectorized version. Furthermore, whereas originally only 30% of cycles are computational stalls, the vectorized variant spends half of the execution time just waiting for floating-point operations to complete. A similar effect is observed between *naive* and *optimized* variants: a 16% reduction of floating-point stalls coincides with a drop in runtime by 33%.

All in all, the reduction of these stall cycles could improve the performance significantly. As already mentioned, reducing the memory load is not the target of this thesis, thus it is expected to increase relatively to the decrease in runtime and computational load.

5. Optimization

5.1. Compute Stalls in the Kernel

With no more in-kernel optimizations apparent, instruction level parallelism has to be leveraged to increase performance by overlapping computations of subsequent inner loop iterations. First, the level of instruction parallelism for the vectorized version shall be determined.

Critical Dependency Chain

To be able to determine the existing degree of ILP for multiple kernel executions, the critical dependency chain must be examined. It is divided into the distance check and the Lennard-Jones force calculation. The third step, force accumulation, is omitted, as it has a large number of independent floating-point instructions and memory accesses. Also, branching is ignored for simplicity.

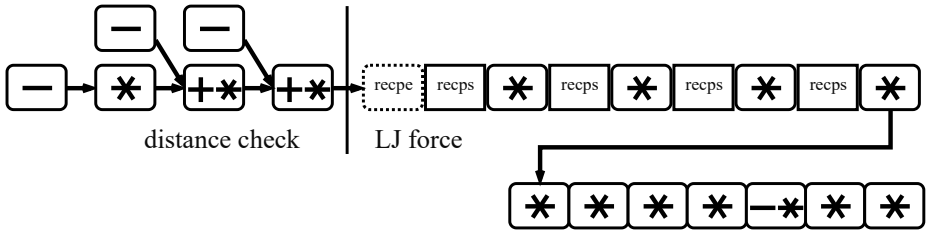


Figure 5.1.: The critical dependency chain, divided into the computational steps. Boxes represent instructions, and arrows the dependencies. For distance calculation, independent instructions can be executed on two execution units. All instructions have a latency of 9 cycles except `recpe`, which does not count as a FLOP.

5.1.1. Upper Bound for ILP

A formula for peak performance depending on the number of overlapping kernel computations is derived based on the critical dependency chain. It can be safely assumed from the stall distribution, that it represents the most computationally intensive, and thus the performance-defining part of the kernel.

A single execution of the dependency chain results in

$$24 * 8_{\text{vector}} = 192\text{FLOPs}$$

on the A64FX, $8 * 8_{\text{vector}}$ FLOPs for distance calculation and $16 * 8_{\text{vector}}$ FLOPs for the force. In the best case, both floating-point execution units can be utilized as depicted in

Figure 5.1, so the total latency, calculated by referencing Table 2.2, is:

$$27_{\text{dist}} + 139_{\text{LJ}} \;=\; 166 \text{ cycles}$$

Otherwise, if only one execution unit is to be used, distance calculation has to be done sequentially, resulting in 193 cycles.

With these values, the maximum achievable performance for a singular execution on the A64FX is determined, which is less than 4% of the single-core peak performance of 57 GFLOP/s.

$$1.8 \text{ GHz} * \frac{192 \text{ FLOP}}{166 \text{ cycle}} = 1.8 \frac{\text{cycle}}{\text{s}} * \frac{192 \text{ FLOP}}{166 \text{ cycle}} = 2.08 \text{ GFLOP/s} \tag{5.1}$$

Up to nine independent computations can be overlapped with pipelining in the execution units, as nine is the prevalent instruction latency in the dependency chain. For n overlaps, and thus an ILP degree of n[1], the performance bound is multiplied by n. For 48 threads and 9 overlapping executions, this bound is already at $48 * 2.08 * 9 \approx 100 * 9 = 900$ GFLOPs, which is much higher than the performance achieved by the vectorized kernel. Thus, there is no need to model higher ILP degrees with concurrency by using both execution units (for which `recprs` instructions, only supported by the first execution unit, would need to be interleaved). A coincidentally simple relationship between maximum performance and the degree of ILP is derived:

$$\text{GFLOP/s} \approx 100 * \text{ILP} \tag{5.2}$$

5.1.2. Experimental Verification

A microbenchmark for verifying the relationship and for gaining further insight is written. The dependency chain from Figure 5.1 is implemented directly, while using as few registers as possible, but still maintaining dependencies between instructions. The code is placed in a loop, with the Lennard-Jones output register is used as an input for the distance calculation, such that subsequent loop iterations are dependent. This way, unwanted overlapping execution of iterations is avoided: The ILP degree is one.

To increase ILP to n, the loop is unrolled by a factor of n. Each copy of the dependency chain receives separate registers, such that they are independent from each other. It is expected that the processor is able to reorder the independent instructions from different dependency chains to utilize pipelining during the execution. The performance for different unroll factors, along with the theoretical computational bound for according ILPs is plotted in Figure 5.2. Additionally, the ratio of cycles, for which floating-point pipelines were active, is plotted, which will be used later.

As can be seen in Figure 5.2, the measured performance matches the theoretical bound exactly. Towards higher unroll factors, the processor is unable to fully exploit ILP, which is observed in minimally lower performance. This is likely due to overheads of unrolling, discussed in Section 2.3.2.

[1] Technically, it is higher due to concurrent instructions in distance calculation. This difference is omitted for simplicity in further analysis.

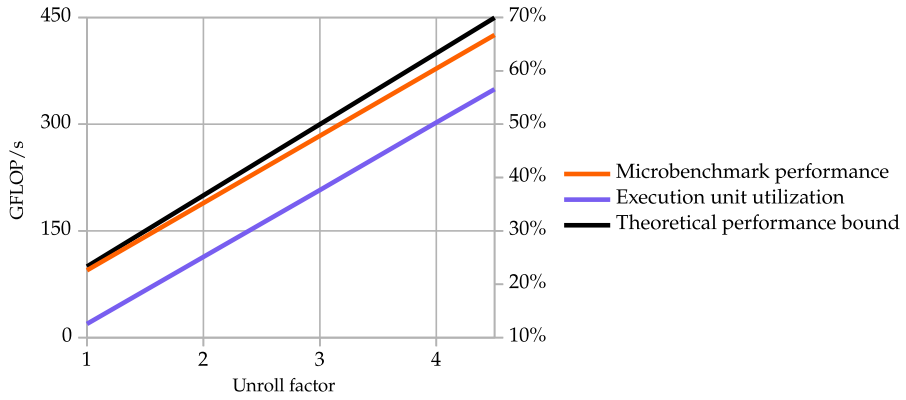


Figure 5.2.: Measured performance of the microbenchmark and the theoretical computational bound. The ratio of active cycles of the floating-point execution units is plotted in purple. Note that the axis is shifted, otherwise it would overlap completely with the measured performance.

5.1.3. Measured Kernel ILP

To roughly estimate the ILP, which is able to be exploited for the vectorized kernel variant in compute-bound scenarios, the attained performance and the execution unit utilization metric mentioned earlier can be used. Assuming that ILP is used through execution unit pipelining in the kernel, as is the case for the microbenchmark, these values can be directly compared. The execution unit utilization is the sum of cycles, for which any execution unit completes an operation. For a completely compute-bound workload, like the microbenchmark, the remainder of cycles are compute stalls. As illustrated in Figure 5.2, the utilization is proportional to the peak performance.

For the vectorized kernel, the execution unit utilization ratio is around 40%, while the performance is at 252 GFLOP/s. The performance value points to an ILP degree of over 2.5, while a value of under 3.5 can be inferred from the utilization ratio. The real degree of instruction level parallelism, which is utilized in the kernel is probably closer to 3.5, as the attained performance is limited by memory-stalls and other factors not present in the microbenchmark.

As a sanity check, a version with two interleaved (at the instruction level) distance calculation and Lennard-Jones calculations is implemented, which removes the need for a reordering subsystem and would therefore also be effective for in-order processors, as discussed in Section 2.3.2. If the ILP factor were lower than two, and given no other bottlenecks, a performance improvement is expected. No significant difference is measured, though. This verifies that ILP of over 2 is already used in the vectorized kernel.

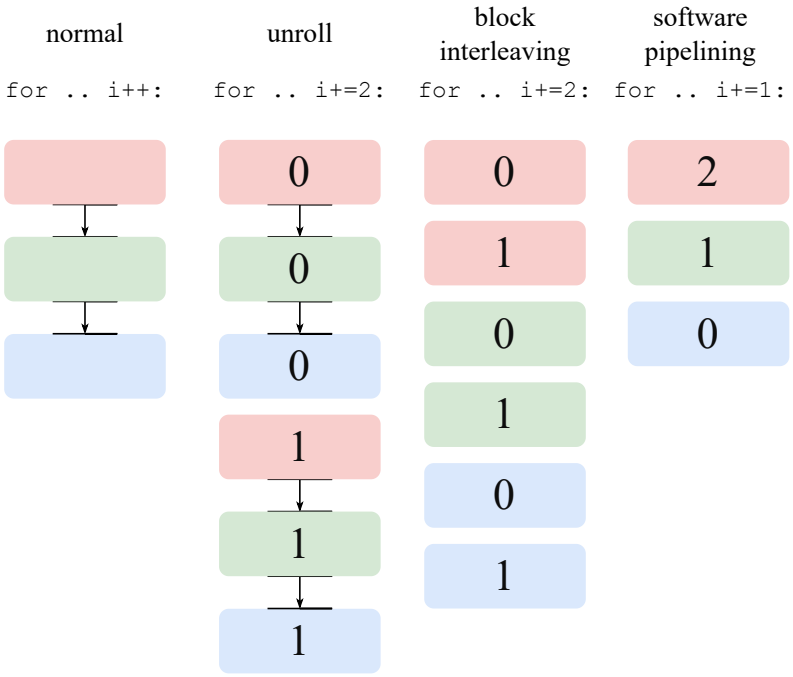


Figure 5.3.: Loop structure after optimization. Red, green and blue represent distance calculation, Lennard-Jones and force accumulation, respectively. This figure is conceptually similar to Figure 2.5.

5.2. Structural Loop Optimization

With an estimated ILP degree of 3.5, the loops around the kernel are restructured to achieve for more instruction level parallelism use and to improve performance. The structural changes are illustrated in Figure 5.3.

5.2.1. Unrolling

The simplest way to improve ILP utilization for a loop is to unroll it, as was already done in the microbenchmark. With the estimated ILP degree, a significant speedup is expected for unroll factors of four and above, if no other overheads are present. However, as performance measurements per unroll factor in Figure 5.4 show, no significantly large improvement is observed. The performance increase is almost 12% at the best unroll factor of four. Execution units show 2% percent more utilization, with the same decrease in compute stalls. A possible limiting factor is the high distance between instructions from subsequent unrolled iterations. The kernel is around 50 instructions in length, and most of those (around 35) are floating-point. So, the reordering subsystem of the A64FX has to be able to execute instructions with a distance of more than 105 out-of-order for the unroll factor of four. The reorder buffer, which holds instructions considered for out-of-order execution, is 128 instructions in length for the A64FX. This explains why there is no performance difference for factors larger than four.

5.2.2. Block Interleaving

To reduce the distance between independent instructions, the multiple kernels of the unrolled version are interleaved on the method-level. For this, the kernel has to be divided into steps of roughly the same length in terms of instructions. An important consideration for such a subdivision is register count. If multiple registers have to be passed to subsequent steps of an iteration, the compiler may hit the architectural limit of 32 floating-point vector registers, and decide to store some of these values in memory. This can obviously result in performance loss, especially if caches are heavily used.

Fortunately, this limit is not exceeded with the logical subdivision from Algorithm 1. Including the six out-of-loop vector registers, the total number of registers to be stored between method executions is 24. It has to be noted that some of them are not needed for the third or the first step, so the actual number of required registers is lower. With the kernels split and interleaved, the distance between independent instructions is reduced to ten between distance checks and to 16 between Lennard-Jones force calculations.

The performance values in Figure 5.4 show a significant speedup of 25% for the interleave factor of four and an estimated ILP factor of four.

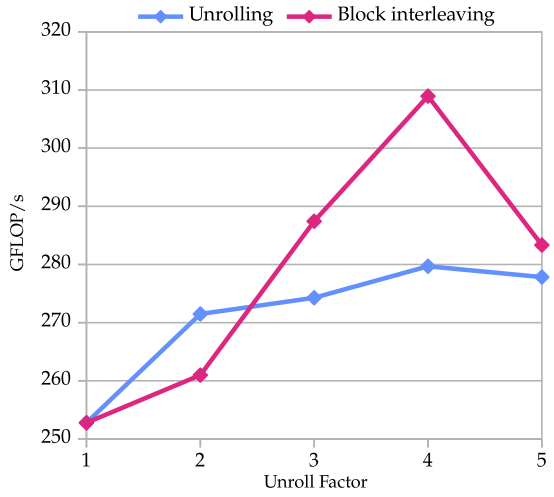


Figure 5.4.: Performance for various unroll or interleaving factors for the highest particle/cell scenario.

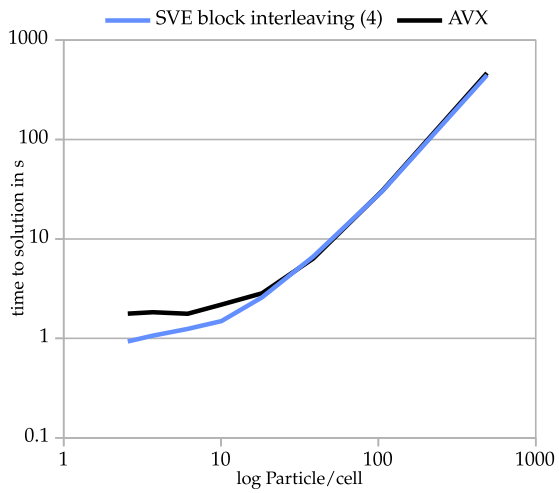


Figure 5.5.: Best-case runtime comparison between AVX and SVE for the highest particle/cell scenario.

Interestingly, for the factor of two, block interleaving is slower than simple unrolling. The reason is the 25% increase in branch mispredictions. To adhere to the distance check, the condition (see line 5 of Algorithm 2) has to be tested for the two latter parts of the kernel, twice as often as for the unrolled version. This offsets any performance gains for minimally improved ILP usage, due to the low factor.

Nevertheless, a speedup factor of nine compared to the unvectorized version is achieved with a performance of 309 GFLOP/s. The absolute runtime, as seen in Figure 5.5, coincidentally matches the runtime for the AVX system, despite having less threads (48 vs. 56). Also, as the instruction latencies on the A64FX are significantly higher, there is room for additional ILP optimization, whereas the AVX implementation is closer to the performance bound.

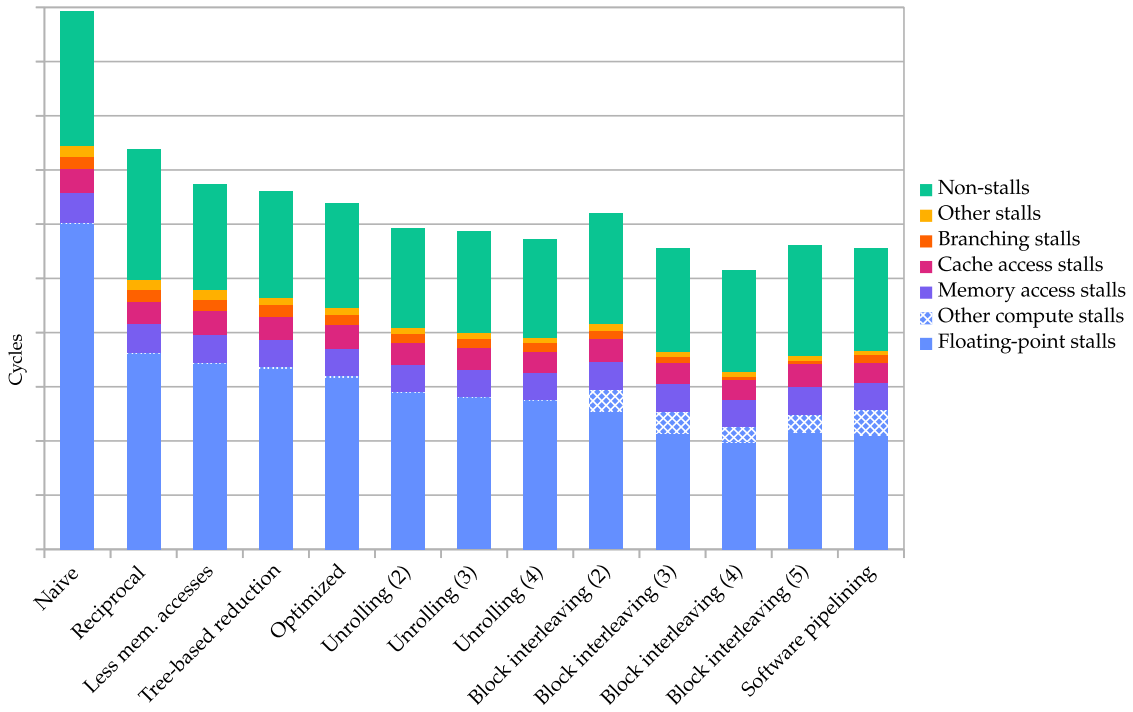


Figure 5.6.: Cycle analysis for vectorized A64FX variants. An extended version of Fig. 4.4.

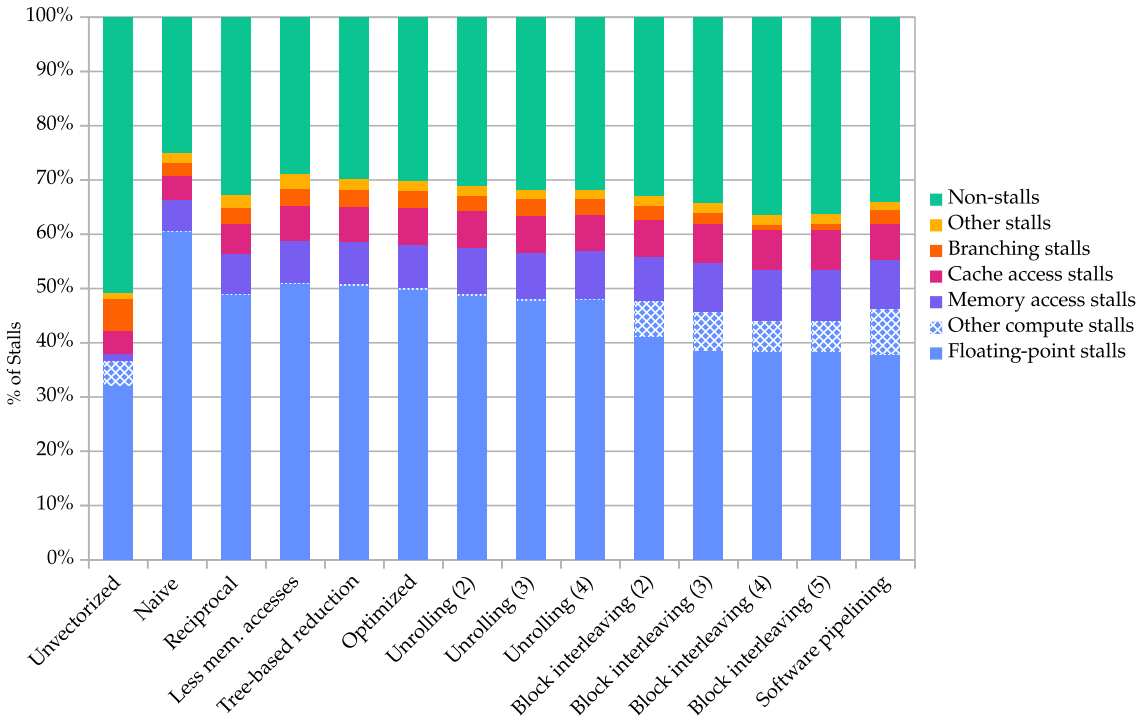


Figure 5.7.: Stall analysis for all A64FX variants. An extended version of Figure 4.4.

5.2.3. Software Pipelining

The most complex loop transformation from the ones attempted is software pipelining, again based on the logical subdivision. The previous optimizations were aimed at using pipelining only in the floating-point execution units. The other possibility is to apply software pipelining to the loop itself to improve the utilization of all resources of the processor. For this, the existing subdivision of the kernel is used. It is unrolled by a factor of three, and software pipelining is applied to achieve three overlapping iteration computations. This is done to simplify the logic, as one whole kernel computation can be done in one unrolled iteration. The architectural register number is exceeded, so the compiler offloads some of the vectors to the memory.

This optimization results in performance close to the best performing versions, with the loop being unrolled only by a factor of three, observed in Figure 5.6. It also has the lowest number of floating-point stalls of all the vectorized versions tested for this thesis, as can be seen in Figure 5.7.

Software Prefetching

Software prefetching is applied to attempt to improve the cache utilization and to hide memory latency. For calculating the distance and the force, relatively many property arrays have to be accessed in a streaming manner due to the SoA data layout. Explicit software prefetching may help the memory subsystem in case hardware prefetch resources are insufficient to keep track of these separate streaming array accesses. However, prefetching with various distances in terms of future array elements does not result in better memory latency hiding and thus less memory stalls for the workload. In contrary, for most of configurations tried, software prefetching is harmful to the performance and increases the count of memory stalls. While the examined workload is not memory-bound and such optimizations are not needed, prefetching could be useful for less dense scenarios and other particle containers.

Compaction

A further, fundamentally different approach for optimization is presented. Using hardware counters, it can be measured that about 15% of FLOPs are performed for inactive vector lanes. This means that the force computation, which is the most compute-intensive part of the kernel, is being calculated more often than needed. Thus, the idea of vector compaction is that particles, which pass the distance check, are "appended" to a vector until it is full and gets passed to the force calculation procedure. For some applications, this optimization can result in 25% improved performance [BCM+20]. Compaction is implemented for the force calculation using the SVE `compact` and `splice` instructions, and avoids all unneeded FLOPs. Nevertheless, the overhead for vector permutation results in overall worse performance. This optimization could work for more compute-intensive force functions, for instance those with multiple divisions, and for less dense scenarios, if Linked Cells is used.

Part III.

Conclusion

6. Conclusion

Throughout the course of this thesis, AutoPas, a particle simulation library, was vectorized for the Arm SVE instruction set and optimized for the Fujitsu A64FX. The pairwise force computation was identified as a hotspot, and optimized for the SoA data layout and specifically for the Linked Cells particle container. A synthetic, high-density scenario was used to ensure a consistent compute-bound workload. A speedup of more than seven over the unvectorized version was achieved using Arm SVE-specific features. The degree of instruction level parallelism for the critical dependency chain of the workload was modeled, and the findings were verified with a microbenchmark. Accordingly, the implementation was further optimized to improve instruction level parallelism utilization on the A64FX. Exceeding the raw computational factor of eight, a final speedup factor of nine with the final performance of 309 GFLOP/s was achieved, which is 11% of the peak and significantly higher than the 3.3% for the industry-standard HPCG benchmark.

Nevertheless, for higher block interleaving factors, a limit in performance was found, with plausible explanations being memory and register count limitations. Further work is needed to quantify and mitigate the bottlenecks, with a focus on the memory and cache architecture of the A64FX. Additionally, more detailed comparisons with x86 architectures and especially with a future AVX512 implementation, which is more comparable to SVE, are subject to further work.

Part IV.

Appendix

List of Figures

List of Tables

List of Algorithms

Bibliography

[BCM+20] Adrián Barredo, Juan M. Cebrian, Miquel Moretó, Marc Casas, and Mateo Valero. Improving predication efficiency through compaction/restoration of simd instructions. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 717–728, 2020.

[BPR21] K J N S Bhargav, Sairam Palisetti, and Madhav Rao. A newton raphson method based approximate divider design for color quantization application. In *2021 18th International SoC Design Conference (ISOCC)*, pages 115–116, 2021.

[CCMH91] Pohua P. Chang, William Y. Chen, Scott A. Mahlke, and Wen-mei W. Hwu. Comparing static and dynamic code scheduling for multiple-instruction-issue processors. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, MICRO 24, page 25–33, New York, NY, USA, 1991. Association for Computing Machinery.

[DJ95] J.W. Davidson and S. Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 125–132, 1995.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[FSS13] Nuno Faria, Rui Silva, and João L. Sobral. Impact of data structure layout on performance. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 116–120, 2013.

[JW89] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, page 272–282, New York, NY, USA, 1989. Association for Computing Machinery.

[Lim] Fujitsu Limited. A64fx microarchitecture manual v1.6. `https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.6.pdf`.

[Lim19] Fujitsu Limited. Fujitsu launches new primehpc supercomputers using fugaku technology. `https://www.fujitsu.com/global/about/resources/news/press-releases/2019/1113-02.html`, 2019.

[Lim22] Arm Limited. *Arm Architecture Reference Manual for A-profile architecture*. Arm Limited (or its affiliates), 110 Fulbourn Road, Cambridge, England CB1 9NJ, ARM DDI 0487H.a edition, Feb 2022.

[LKV12] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1), mar 2012.

[LVAG98] J. Llosa, M. Valero, E. Agyuade, and A. Gonzalez. Modulo scheduling with reduced register pressure. *IEEE Transactions on Computers*, 47(6):625–638, 1998.

[MCG+92] S.A. Mahlke, W.Y. Chen, J.C. Gyllenhaal, W.W. Hwu, P.P. Chang, and T. Kiyohara. Compiler code transformations for superscalar-based high-performance systems. In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 808–817, 1992.

[SBB+17] 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, Mar 2017.

[SGH+21] Steffen Seckler, Fabio Gratl, Matthias Heinen, Jadran Vrabec, Hans-Joachim Bungartz, and Philipp Neumann. Autopas in ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning. *Journal of Computational Science*, 50:101296, 2021.

[Ver67] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159:98–103, Jul 1967.

[WRHDF20] Xipeng Wang, Simón Ramírez-Hinestrosa, Jure Dobnikar, and Daan Frenkel. The lennard-jones potential: when (not) to use it. *Physical Chemistry Chemical Physics*, 22(19):10624–10633, 2020.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.