FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

# Implementation of a vectorized Quicksort using AVX-512 intrinsics

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
im Studiengang Informatik

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik

eingereicht von Frank Thiemicke
geb. am 28.07.2000 in Weimar

Betreuer: Mark Blacher
Zweitgutachter: Dr. Lars Kühne

Jena, den 2. August 2021

## Zusammenfassung

Jahrzehntelang wurden Verbesserungen der Rechengeschwindigkeit erreicht, indem die Taktfrequenz der CPU erhöht wurde. Im Laufe der letzten Jahre wurde dieser Mechanismus durch physikalische Einflüsse gebremst. Daher müssen moderne Single-Thread-Anwendungen stärker CPU-Funktionen ausnutzen, um von den Fortschritten neuer Prozessorgenerationen zu profitieren.

Eine dieser Funktionen ist die Vektorverarbeitung, um mehrere Datenelemente gleichzeitig zu verarbeiten. Diese Arbeit untersucht die Verwendung von AVX-512-Befehlen zur Sortierung von primitiven Typen der Länge 32 bit. Die Nutzung von Vektorinstruktionen für die Sortierung ist eine Herausforderung, da Sortieralgorithmen erst vektorisierungsfreundlich umgestaltet werden müssen. Glücklicherweise hat Mark Blacher einen effizienten vektorisierten Sortieralgorithmus auf Basis von Sortiernetzwerken und einem nichtquadratischen Quicksort entwickelt.

In dieser Arbeit wird Blachers AVX2-basierte vektorisierte Implementierung auf den moderneren AVX-512-Befehlssatz portiert. Im Geschwindigkeitsvergleich schlägt Blachers AVX2-Version die in dieser Arbeit entwickelte AVX-512-Implementierung. Diese ist jedoch in der Lage, den bisherigen AVX-512-Sortieralgorithmus von Bramas zu übertreffen.

## Abstract

For decades, improvements in computation speed have been achieved by increasing the CPU's clock speed. Over the course of the last years, this mechanism has been curbed by physical influences. Therefore, modern single-threaded applications need to leverage modern CPU features to benefit from the advances of recent processor generations.

One of these features is vector processing – using vector instructions to process multiple data items simultaneously. This thesis explores the use of AVX-512 vector instructions for sorting arrays of 32 bit primitive types. Using vector instructions for this task is a challenging endeavor, because sorting algorithms have to be adapted to vectorization. Fortunately, Mark Blacher has developed an efficient vectorized sorting algorithm based on vectorized sorting networks and a non-quadratic vectorized Quicksort.

This thesis ports Blacher's AVX2-based vectorized implementation to the more modern AVX-512 instructions. In benchmarks, his AVX2 implementation compares favorably to the dedicated AVX-512 implementation described in this thesis. Nevertheless, the AVX-512 port is able to beat Bramas' AVX-512 sorting algorithm.

# Contents

# 1 Introduction

Sorting is one of the most well-known problems in computer science. Sorting algorithms are applied to solve a variety of problems. Ranging from ordering search results in online stores to sorting indices for databases, sorting is inherent to many problems on vastly different levels of development.

A major use case for sorting is the minimization of effort spent on tasks that benefit from strict ordering. Binary search is a prime example for that: searching an item after sorting a large array may be significantly faster than executing a search routine on the unsorted array. Sorting may thus be considered an important subproblem of many more complex problems.

While sorting is commonly used in introductory courses to illustrate different aspects of algorithmics, it is far from being trivial. The field of sorting is still advancing, especially in terms of optimizing implementation speed. Modern implementations nearly ubiquitously feature hybrid algorithms switching from one algorithm for sorting large datasets to even faster algorithms for smaller ranges or arrays.

One goal of optimizing these implementations is the better utilization of hardware resources. For CPU-bound tasks like (in-cache) sorting on modern processors, it is of great interest to stay up to date with recent CPU features and instructions. Vectorization is one approach to reach this aim by exploiting data-level parallelism. Its application to sorting is subject to active research.

For years, consumer hardware has provided vector instructions. Primarily used by multimedia applications at first, vector instructions speed up many algorithms today, from cryptography to linear algebra. In high performance computing environments that are still based on the x86 instruction set architecture, the AVX-512 vector extensions have gained market share. Although they are not the default for implementing compute-intensive processes yet, they may gain more traction by Intel integrating AVX-512 into recent consumer CPUs, for desktop computing as well as laptops.

In 2018, Mark Blacher implemented a vectorized Quicksort using AVX2, five years after AVX-512's introduction. This thesis aims to port his ideas from

AVX2 to AVX-512 using the advanced capabilities of AVX-512 and determine the speedup of this algorithm over the AVX2 implementation on processors with AVX-512 feature set. Algorithmically, this revolves around Blacher's hybrid algorithm of Quicksort and sorting networks.

After a short introduction to vector processing and AVX intrinsics in chapter 2, relevant sorting algorithms and sorting networks will be discussed in chapter 3. Previous approaches to the vectorization of sorting algorithms in general and Quicksort in particular are presented in chapter 4. Chapter 5 highlights the differences of an AVX-512 implementation compared to Blacher's AVX2 implementation. In chapter 6 the resulting implementation's benchmarking results are discussed and its performance is compared to common well performing sorting implementations.

I am grateful to my supervisor Mark Blacher for enlightening discussions and technical insight. Thanks also go to Dr. Lars Kühne for introducing me to the topic of algorithm engineering and offering the topic of this thesis to me. I appreciate the feedback and proofreading by Paulo Roberto Massa Cereda and Christoph Wernike.

## Code examples and accompanying code

Throughout this thesis, code examples will aid the textual explanations. Unless otherwise stated, they are written in the Rust programming language. Some of them may not be executable on their own and most of them require compilation on a machine with AVX-512 support. Please note that all these code examples are prepared for typesetting, which includes the use of coding ligatures.

The repository at `https://git.uni-jena.de/li73jeh/bachelor-thesis` provides the executable code accompanying this thesis. That is, the AVX-512 sorting implementation, unit and fuzz tests, benchmarking code as well as the LaTeX code for this thesis may be found there. Some sections link to relevant source files indicating the relative path within the repository.

# 2 Vectorization on modern hardware

## 2.1 Vector processing and SIMD

This thesis aims to implement a vectorized Quicksort. To define vectorization it is useful to step back and introduce the term “vector”. MacKenzie describes vector processing in the following way:

> “ Vector processing means hardware and software provision for a single instruction to be executed on all the members of an ordered set of data items. [45, p. 103] ”

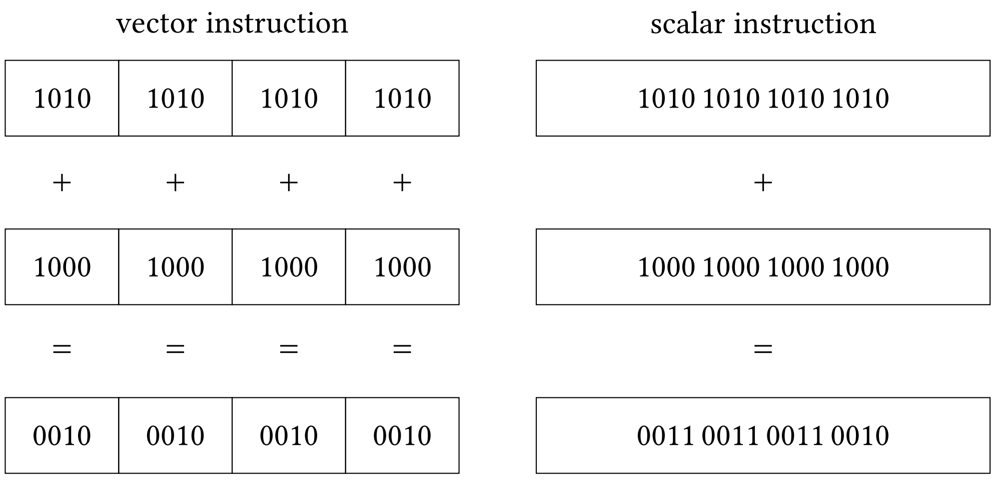
Based on this, the term vector will be used to describe an ordered set of data items. For the sake of simplicity, vectors will be seen as registers which are used as contiguous storage of several elements of the same length. For example, a register of a length of 512 bit is a vector of 8 elements for 64-bit-integers.

The idea of vector processing is more than 50 years old. The first successful vector computer was the Cray I from 1976. Many principles of this kind of processing date back to the 1950s when Daniel Slotnick explored the field of processor arrays to perform the same bit-serial operation on multiple words (although his first attempts used a word size of 1 bit). Back then, computing pioneer John von Neumann dismissed Slotnick’s ideas as requiring “too many tubes” [45, p. 103 ff.].

Flynn devised a classification of computing systems in 1972. Slotnick’s approach is Flynn’s prime example for the SIMD, single instruction multiple data, principle [18]. SIMD processing is one way to achieve data-level parallelism.

On modern systems, SIMD instructions are ubiquitous. The following explanations will concentrate on computers using the x86 instruction set ar-

Figure 2.1: Addition using a vector instruction and a scalar instruction on registers with the same content.



chitecture. There, SIMD instructions are commonly referred to as vector extensions of an instruction set. Common vector extensions include MMX (Multi Media Extension, 64 bit registers), SSE (Streaming SIMD Extensions, 128 bit registers), AVX, AVX2 (Advanced Vector Extensions, 256 bit registers each) and AVX-512 (512 bit registers).

A vector instruction operates on one or more vectors providing a semantic abstraction from the register's bit level to the vector's element level. Vectorized addition, for instance, will add two vectors element-wise. Vector instructions are contrasted to scalar instructions where adding two registers would simply add the numbers in the registers without considering them to be multiple elements.

To exemplify vectorized and scalar operations, fig. 2.1 depicts both a vectorized and a scalar addition. These additions are performed on a machine with a register size of 16 bit and vector instructions that use one register as a vector of 4 elements. Note that the content of the source registers is the same in both the scalar and the vectorized case but there is a difference in the output/destination register.[1]

This leads to the process of turning code of scalar instructions into code maximizing the use of vector instructions, viz. vectorization [36]. There

[1]This difference is achieved by computing an addition with overflow. On this contrived architecture integer overflow is silently ignored.

are different approaches to vectorization. Many developers rely on auto-vectorization as performed by the compiler, affecting simple loops with independent iterations and similar constructs. However, in many cases code has to be rewritten to be vectorized effectively.

As chapters 5 and 6 will concentrate on Rust code, it is important to note that the Rust compiler enables auto-vectorization by default. Rust delegates this task to LLVM, which has two major vectorizers: a loop vectorizer that unrolls and vectorizes loops and a “superword-level parallelism” (SLP) vectorizer combining multiple similar instructions into one vector instruction [43].

## 2.2 Vector instructions and intrinsics

In the previous section, vector instructions have been introduced as instructions operating on vector registers. As this thesis will focus on Intel’s AVX-512 instructions, there are three relevant types of vector registers: SSE registers (`xmm0` to `xmm15`, width: 128 bit), AVX/AVX2 registers (`ymm0` to `ymm15`, width: 256 bit) and AVX-512 registers (`zmm0` to `zmm31`, width: 512 bit).

It is important to note that these registers are not independent of each other. For all AVX registers their lower half constitutes the SSE registers and equally AVX registers constitute the lower half of AVX-512 registers [50]. Due to the latter there are more AVX and SSE registers available on AVX-512 hardware than on AVX2-only hardware [12]. This configuration of the registers is shown in fig. 2.2.

The AVX-512 vector extensions complement the x86_64 instruction set architecture to utilize the respective 512 bit registers. Initially developed by Intel, the x86 instruction set gained 64 bit additions by AMD back in 2003 [9, p. 269]. Albeit introduced in 2013, the AVX-512 vector extensions are one of the latest major additions to the instruction set [50].

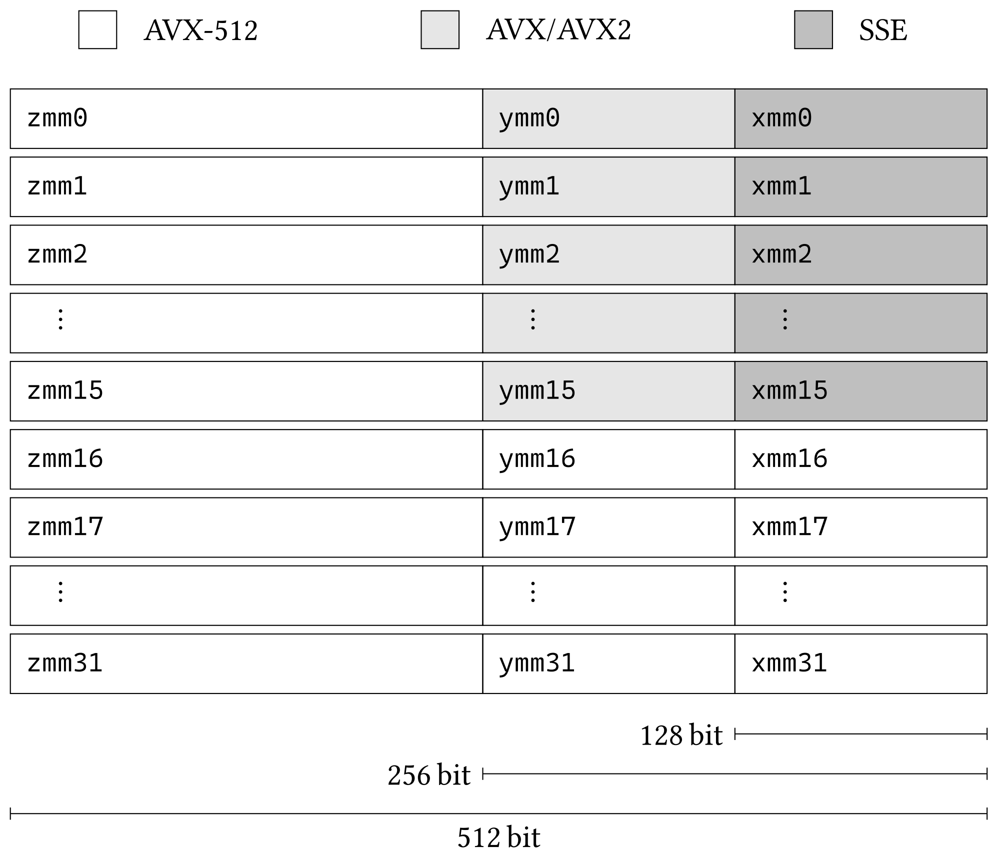
One of these new vector instructions is `vpminsd`, which computes the (element-wise) minimum of signed 32 bit integers in 512 bit vector registers. A call to this instruction could look like

```asm
vpminsd zmm0, zmm1, zmm2
```
Asm

which assigns the element-wise minimum of `zmm1` and `zmm2` to `zmm0`.

Vector instructions like `vpminsd` are called in the same way as scalar instructions. The main difference with respect to their assembly code is the ability

Figure 2.2: AVX-512 registers on the x86_64 architecture.



to use vector registers. As the AVX registers use the lower half of AVX-512 registers many vector instructions like `vpminsd` may be used with `zmm` type as well as `ymm` type registers.

When displaying instructions or assembly code, this thesis makes use of the Intel ASM syntax [44] where calls follow the order instruction, destination, source. Some of the assembly output is generated with the help of Compiler Explorer [21].

In theory, assembly code as shown would be sufficient to make use of vector instructions in higher-level languages. The following Rust example illustrates the use of these assembly vector instructions.

```rust
#![feature(asm)]
#![feature(stdsimd)]
use std::arch::x86_64::*;
```

```
fn min(a: __m512i, b: __m512i) -> __m512i {
    let c: __m512i;
    unsafe {
        asm!(
            "vpminsd {0}, {1}, {2}",
            out(zmm_reg) c,
            in(zmm_reg) a,
            in(zmm_reg) b,
        );
    }
    c
}
```

Writing such a function has multiple disadvantages. One of the most significant is the increase in code complexity and decrease in maintainability. Furthermore, it is hard to reason about the correctness of certain optimizations, even for compilers. Thus, including assembly calls in a way similar to the one above might result in less optimal code generation.

The most important alternative to assembly calls are intrinsic functions (intrinsics). These are functions whose implementation is not provided by a language's libraries but on compiler or runtime level. For instance, Rust's toolchain is based on the LLVM compiler infrastructure and the Rust compiler generates LLVM intermediate representation code. Rust's standard library provides the following function interface corresponding to the interface described by Intel [34, 51]

```rust
pub unsafe fn _mm512_min_epi32(a: __m512i, b: __m512i) -> __m512i {
    transmute(vpminsd(a.as_i32x16(), b.as_i32x16()))
}
```

where `vpminsd` is treated as a regular Rust function. This `vpminsd` function is declared in the same module by using an external C declaration corresponding to the C interface of a LLVM intrinsic [53]:

```rust
#[allow(improper_ctypes)]
extern "C" {
    // …
    #[link_name = "llvm.x86.avx512.mask.pmins.d.512"]
    fn vpminsd(a: i32x16, b: i32x16) -> i32x16;
    // …
}
```

This LLVM intrinsic in turn compiles to the vpminsd instruction described earlier.[2]

The following explanations will concentrate on the intrinsics as defined by the Intel Intrinsics Guide [34], i.e. the level of _mm512_min_epi32. When dealing with these abstractions, it is important to clarify a caveat of that level: these intrinsics do not always translate into exactly one vector instruction.

Essential instructions like _mm512_set_epi32 (setting an AVX-512 register by using the integers that the vector will hold) are of type "sequence", i.e. they will generate more than one instruction. Intel poses a performance warning on these instructions.

> "This intrinsic generates a sequence of instructions, which may perform worse than a native instruction. Consider the performance impact of this intrinsic. [34]"

Indeed, the Rust source code for simply inserting 16 integers into an AVX-512 register results in two instructions being issued.

```rust
#![feature(stdsimd)]
use std::arch::x86_64::*;

pub unsafe fn set_vec() -> __m512i {
    _mm512_set_epi32(
        1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
    )
}
```

The resulting assembly shows two vmovaps instructions responsible for loading the values from LCPI0_0 into the corresponding registers.

```asm
.LCPI0_0:
        .long   16
        .long   15
        .long   14
        .long   13
```

[2]Actually, it is a resolution process starting with LLVM's auto-upgrade mechanism which maps it to the smin intrinsic that is subject to further translations. It is not useful to follow the path here. In the end it resolves to the corresponding instruction.

```
        .long   12
        .long   11
        .long   10
        .long   9
        .long   8
        .long   7
        .long   6
        .long   5
        .long   4
        .long   3
        .long   2
        .long   1
example::set_vec:
        mov     rax, rdi
        vmovaps zmm0, zmmword ptr [rip + .LCPI0_0]
        vmovaps zmmword ptr [rdi], zmm0
        vzeroupper
        ret
```

As the second `vmovaps` relies on the availability of the output of the first `vmovaps`, the two instructions have to be executed sequentially. To assess the impact of a specific order of vector instructions on the performance, there are two important metrics: latency and throughput.

The latency of an instruction describes the number of clock cycles until the result of that instruction is available to other instructions [9, p. 501]. Take the line

```asm
vmovaps zmm0, zmmword ptr [rip + .LCPI0_0]
```
Asm

from above. `vmovaps` has a latency of 1. Hence, the result will be available to the next `vmovaps` instruction after 1 clock cycle has completed. In turn, the result of the second `vmovaps` instruction needs another clock cycle to be made available. Thus, the whole `_mm512_set_epi32` is compiled into instructions that need 2 clock cycles to return the result, or assign it to a variable for that matter.

Another important measure is throughput.[3] It describes the number of clock cycles it takes until the next instruction of the same type may be executed on the same execution unit. As an example, consider the instruc-

[3]Throughput follows Intel's terminology as presented in the Intrinsics Guide. Bryant and O'Hallaron call the same measure "issue time" [9, p. 501].

tion vpcompressd, which has a latency of 11 and a throughput of 2. This instruction is wrapped by _mm512_mask_compressstoreu_epi32.

```C++
_mm512_mask_compressstoreu_epi32(base_addr, 0xA0A0, v);
_mm512_mask_compressstoreu_epi32(base_addr + 16, 0xA0A0, v);
_mm512_mask_compressstoreu_epi32(base_addr + 32, 0xA0A0, v);
```

In the preceding code snippet, the computations are independent of each other. That is, the second compressing store may start after 2 clock cycles as the throughput of the previous instruction is 2, it uses the same execution unit and does not depend on the result of the first instruction. Similarly for the third instruction. For the execution unit, all computations are considered finished after 6 clock cycles. However, due to the latency of 11 the results will only be available after a total of $11 + 2 + 2 = 15$ clock cycles.

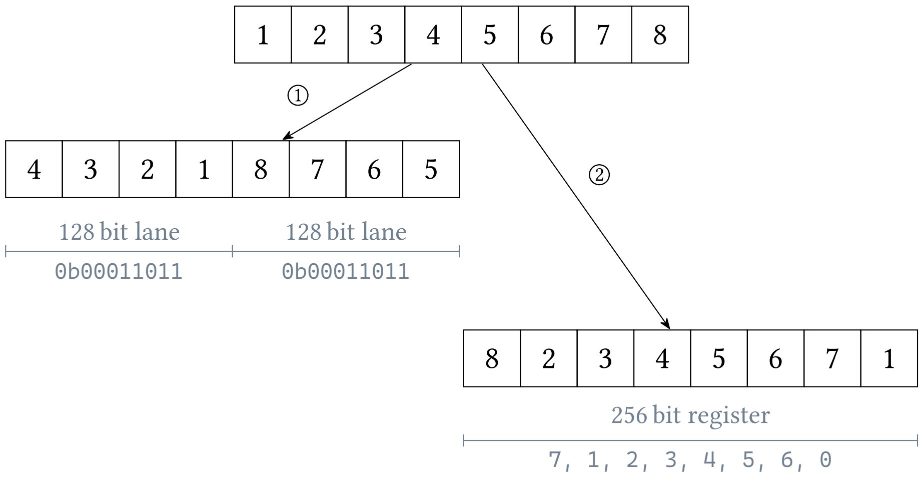
NB: Scalar instructions and vector instructions may have the same latency and throughput. The example from fig. 2.1 on page 10 is a prime example: adding usually needs 1 clock cycle to provide its result regardless of being scalar or vectorized. This is the foundation for the performance improvements gained by vectorization.

To reduce latency, some operations artificially narrow the register width by simultaneously operating on lanes instead of the whole vector register at once. A lane groups multiple contiguous elements. AVX2 features two 128 bit lanes, AVX-512 four. A recurring example in this thesis will be the use of shuffling with latency 1 vs. permuting with latency 3, where both may reorder elements within a vector. Shuffling is constrained to only reorder items within lanes, whereas permuting may cross lane boundaries while reordering.

Figure 2.3 shows an example of shuffling vs. permuting with respect to lanes in AVX2. For ①, it is sufficient to shuffle because all elements are reordered within lane boundaries. The seemingly harmless swap of 8 and 1 in ②, however, crosses lane boundaries and has to be performed on the whole register, i.e. by permuting. Permutations are performed by providing a mapping between target indices and source indices. Shuffles are defined by an immediate value, immediate for short, representing the transformation for one lane.

Immediates are compile-time constants [9, p. 169] which do not require access to registers or memory. The immediate argument for shuffling is a sequence of four numbers in binary representation ranging from 0 (00) to 3 (11). Groups of two binary digits each are read from right to left. The

Figure 2.3: Shuffling vs. permuting on AVX2.



specification `0b00011011` for reversing a lane as used in the figure thus reads as the sequence of indices 3, 2, 1, 0, i.e. "last element of the lane first, third second, second third and first last".

## 2.3 Advances of AVX-512 instructions

In contrast to AVX and AVX2, AVX-512 describes multiple vector extensions. The most important subset is AVX512F (foundation), including 512 bit equivalents of AVX2 extensions as well as additional instructions like the compress or masked instructions. Every AVX-512 compatible CPU must implement AVX512F, which is why this thesis will focus on the AVX512F vector extensions.

Besides AVX512F, most architectures apart from the discontinued Xeon Phi architecture implement AVX512CD, which provides collision detection instructions. The Skylake processors provide three additional vector extensions, AVX512BW (byte and word, i.e. support for additional 8 bit and 16 bit operations), AVX512DQ (double and quad word, i.e. support for additional 32 bit and 64 bit operations) and most importantly AVX512VL (vector length, allowing to use AVX-512 instructions on `ymm` and `xmm` registers) [49, 60].

AVX-512 also introduces improvements in the details. One of these details is that unlike "SSE and AVX that cannot be mixed without performance penalties, the mixing of AVX and Intel AVX-512 instructions is supported without penalty" [50], which in turn allows the re-use of code optimized for

AVX in AVX-512 programs. That is, for larger software projects a migration may happen steadily instead of all at once without losing performance.[4]

To exemplify some of the new instructions: Gueron and Krasnov as well as Blacher developed Quicksort's partitioning based on AVX2 instructions and both used pre-computed permutation masks to implement partitioning [6, 24].[5] For Blacher's Quicksort this results in the following code to partition a single AVX2 vector into memory.

```rust
// which elements are larger than the pivot
let compared = _mm256_cmpgt_epi32(current_vec, pivot_vec);
// extract highest bit from each integer of the vector
let mm = _mm256_movemask_ps(
    *(&compared as *const __m256i as *const __m256)
);
// permute elements greater than the pivot to the right and those
// less than the pivot to the left
current_vec = _mm256_permutevar8x32_epi32(
    current_vec,
    AVX2_PERMUTATION_MASKS[mm as usize],
);
_mm256_storeu_si256(
    left_store as *mut i32 as *mut __m256i,
    current_vec
);
```

In this example, `AVX2_PERMUTATION_MASKS` contains pre-computed permutation masks that help permuting the current vector into a partitioned state given the position of the elements greater than the pivot element.

An AVX-512 implementation of the same partitioning of one vector[6] looks like the following snippet.

```rust
// mask of elements are larger than the pivot
let compared = _mm512_cmpgt_epi32_mask(current_vec, pivot_vec);
```

[4]Downs clarifies that there are situations where mixing in light AVX2 instructions may impose a performance penalty [15]. However, this penalty affects both AVX2 and AVX-512 on AVX-512 chips, so it does not change the conclusion.

[5]Partitioning is the problem to transform a vector in a way that all elements with values less than a given pivot are positioned to its left and all elements greater than it to its right.

[6]Actually, the implementations differ as the AVX-512 version will partition 16 elements at once. Still, it is sufficiently similar to introduce the concept.

```
// count ones (every 1 is an element greater than the pivot)
let amount_gt_pivot = compared.count_ones() as usize;
// store elements smaller than the pivot to the left and the ones
// larger than the pivot to the right
_mm512_mask_compressstoreu_epi32(left_store, !compared, current_vec);
_mm512_mask_compressstoreu_epi32(
    right_store.sub(amount_gt_pivot),
    compared,
    current_vec
);
```

There are a few differences. Most notably, the `vpcompressd` instruction introduced in the last section is used instead of permuting and storing separately. This instruction contiguously stores those 32 bit integers in `current_vec` whose respective bit is set in the writemask `compared` to the memory location defined by the provided pointer.

As surfaced by the name of the intrinsic function wrapping `vpcompressd`, `_mm512_mask_compressstoreu_epi32`, this instruction belongs to the family of masked load/store instructions. But even unrelated to memory access AVX-512 introduces a significant amount of masked instructions, many of which without equivalent in the AVX2 instruction set.

Among others, most arithmetic operations gained mask support. For instance, `vpaddd` (adding integer double words) takes one mask argument that allows to control which element's addition results will actually be stored. There are masked compare operations, masked convert operations, masked permute operations and many more.

Returning to masked store instructions and the introductory example, one question remains: is there any benefit from using these instructions compared to the alternative? A first observation in the AVX2 code is the absence of pipelining opportunities due to each statement using the previous statement's result. Disregarding array access to the permutation masks, this results in adding up the latencies of the instructions, in order that is $1 + 2 + 3 + 5 = 11$ clock cycles.

On the other hand, the AVX-512 example can make use of pipelining. The first three statements are sequentially executed and the fourth statement starts after the throughput of the first `vpcompressd` has been exceeded, i.e. two clock cycles after the third statement. This adds up to $3 + 1 + 2 + 11 = 17$ clock cycles, 6 cycles longer than the AVX2 version.

Now, there are two important considerations why the AVX-512 instructions are still of use. First and foremost, they partition 16 elements at once, which would require two partitions of 8 elements each and one additional merge on AVX2. Second, the high-latency AVX-512 instructions benefit from pipelining, which allows for optimizations like manual loop unrolling to result in major speedups.

However, there is one significant caveat to the use of AVX-512. Processors may reduce clock frequency for thermal reasons (throttling) or because of license-based downclocking[7] [55]. They do this for AVX2 instructions as well but the performance drop is about half as severe as for AVX-512. During frequency transitions themselves the processor might be significantly slower than usual [15].

For some applications, throttling is not even measurable [40]. But theoretically AVX-512 code, which triggers a high license and thermal throttling, may still become slower than AVX2 code. Among others, this affects code that primarily consists of multiple "heavy" instructions like those operating on the floating point unit. Whether a program is subject to significant downclocking has to be tested by performing benchmarks on real hardware. Lemire provides some hints on what to take into consideration [41].

It is important to note that more recent generations of processors, Intel's Rocket Lake for example, are not affected by large license differences when performing license-based downclocking. The distinction between light and heavy is nearly inexistent for these CPUs, with the only real source of heavy downclocking on these recent CPUs being throttling [16].

[7]Licenses are categories for the frequencies a processor may run on. The choice of license depends on the instructions, some of which like those from AVX-512 need a higher license and therefore always cause a decrease in clock speed. License-based downclocking does not include thermal throttling. [17].

# 3 Efficient sorting

## 3.1 Properties of sorting algorithms

Sorting is the process of reordering a collection of comparable elements in a way that afterwards the collection is ordered by some (total) ordering. This thesis will focus on sorting 32 bit integers, i.e. the total ordering is defined by the known comparison operations on $\mathbb{Z}$. Integers may be used as sort keys to sort more complex data structures [27, p. 55].

Focusing on 32 bit integers is a very weak restriction. As detailed by Blacher, there are simple transformations between floating point numbers and integers for sorting. Even more complex objects might be represented as integers or at least re-ordered in their bit representation to effectively use a sorting key. If all that is infeasible there are multiple alternatives like using pairs of indices and keys, or repeating every operation on an array of keys on the array of values [6, cf. pp. 67 ff.].

Most general-purpose sorting algorithms like Quicksort or Mergesort are comparison-based sorting algorithms. That is, they are based on an abstract comparison operator applicable to any two items to be sorted which determines the order in which they appear in the sorted list. There is a proven lower bound for the number of comparisons such an algorithm has to perform, $\Omega(n \log(n))$ [27, pp. 87 ff.].

Multiple algorithms achieve this best-case complexity, among them Quicksort and Mergesort. While Mergesort will never perform worse than $\mathcal{O}(n \log(n))$, traditional Quicksort has a worst-case complexity of $\mathcal{O}(n^2)$. Quicksort is still a very good choice because this worst-case can be eliminated or at least be made exceedingly rare, cf. section 3.3.

Apart from complexity analysis, a comparison between multiple algorithms of the same complexity is determined by the implementations and target architectures. A typical Mergesort is implemented as an out-of-place algorithm, i.e. it requires an auxiliary data storage, while Quicksort is usually implemented as an in-place algorithm. Depending on the storage access times this might have significant impact on the constant terms and factors neglected by complexity analysis.

In general, comparison-based sorting algorithms do not take more specific properties of the problem into account. These properties may include a small range of values, a certain distribution, or simply a very small amount of elements. Non-comparison-based sorting algorithms exploit these properties at the cost of being rather inefficient on data of unknown distribution, range etc.

Among the non-comparison-based algorithms are Bucket sort, Radix sort or Counting sort. Counting sort achieves a worst-case time and space complexity of $\mathcal{O}(n+k)$ with the number of possible values $k$ in an array of length $n$. In contrast to Quicksort, which is usually implemented as an unstable sorting algorithm due to space implications of stability, Counting sort is inherently stable, i.e. two equal elements do not change their order in the sorted array compared to their order in the unsorted array.

## 3.2 Sorting networks

Sorting networks are comparison-based sorting algorithms whose only operations are comparison elements.[8] A comparison element receives two inputs and compares them with each other [5, p. 23]. The basic principle is depicted in fig. 3.1. This thesis will stay consistent with the order presented in the figure, the minimum of both inputs will be relayed to the upper output and the maximum to the lower output.

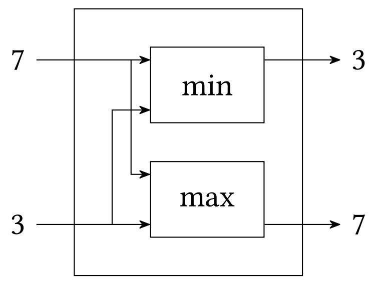
Comparison elements may be implemented in hardware or software. Sorting networks are composed of comparison elements by using the output of previous comparison elements as input of later ones. Comparisons without mutual dependencies may be executed in parallel if the chosen realization supports parallelization. A vectorized implementation of sorting networks requires parallel execution, which is provided by the hardware facilities of the vector processor.

Sorting networks are oblivious. Any input to a sorting network, even a sorted sequence, leads to the execution of all minimum and maximum operations constituting the comparison elements. Sorting networks may be implemented

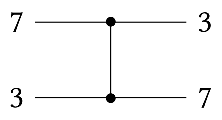
[8]Blacher calls comparison elements “compare-exchange modules”. Especially in code examples, this thesis uses the compare exchange terminology as well.

Figure 3.1: Graphical representation of a comparison element sorting the numbers 3 and 7.

(a) Block diagram.



(b) Simplification.



in a stable way by only comparing adjacent channels [25, p. 26]. The sorting networks presented subsequently will be unstable.

A simple software implementation may look like

```rust
fn comparison_element(a: i32, b: i32) -> (i32, i32) {
    (a.min(b), a.max(b))
}
```

which may be extended to a vectorized version performing (roughly[9]) the equivalent of

```rust
fn comparison_element<'a, I>(a: I, b: I) -> Vec<(i32, i32)>
where
    I: Iterator<Item = &'a i32>,
{
    a.zip(b).map(|(&a, &b)| (a.min(b), a.max(b))).collect()
}
```

by using vector types and instructions.

```rust
fn co_ex(&mut a: __m512i, &mut b: __m512i) {
    let vec_tmp = *a;
    *a = _mm512_min_epi32(*a, *b);
    *b = _mm512_max_epi32(vec_tmp, *b);
}
```

[9]To be semantically identical the iterators must contain 16 elements each and the vector implementation needs to allocate contiguous aligned memory to be castable into a vector type. This code example is kept as simple as possible.

Batcher proposed two types of sorting networks in 1968: Odd-even mergesort and Bitonic sort [5]. Both are algorithmically constructible making them useful for generating larger scale sorting networks. Figure 3.2 shows both networks for 32 elements. Blacher discusses both kinds of sorting networks in depth and evaluates their use in vectorized sorting [6].

Knuth demonstrates how Batcher's strategies, which are designed to generate sorting networks for inputs with a length which is a power of two, may be applied to inputs of arbitrary length [37, pp. 223–225, 230–232]. As illustrated in the figure, both networks follow a repetitive pattern which spans multiple parallel steps. The networks first sort groups of size 2 (step 1), then groups of size 4 (steps 2–3), groups of size 8 (steps 4–6), and so on.

For sorting input of arbitrary length with Odd-even mergesort, one may sort the first $m$ and the second $n$ elements separately and then apply a $(m, n)$-merging network. This merging network is constructed inductively by adding comparison elements for the odd and even sequences and then the last compare exchange steps [37, p. 224].

The asymptotic number of comparison elements in Batcher's networks is $\mathcal{O}(n \log^2 n)$,[10] which is worse than the $\mathcal{O}(n \log n)$ comparisons worst-case complexity of comparison-based sorting algorithms like Mergesort. However, due to the efficiency of implementation and use of parallelization the actual runtime may still outperform other sorting algorithms.

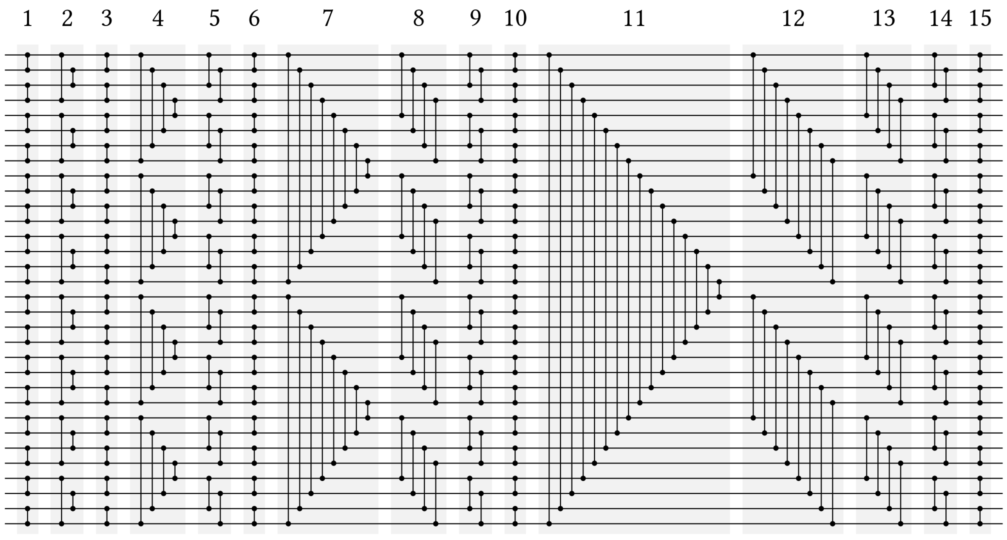
Table 3.1 lists the sizes of the known most optimal sorting networks. If there is only a single number it is proven that the known sorting network is optimal. Otherwise, the first number indicates the known upper bound and the second number the known lower bound of comparison elements or parallel steps respectively. Batcher's Odd-even mergesort networks only achieve optimal characteristics for $n \leq 8$ [37, p. 226].

There are sorting networks of size $\mathcal{O}(n \log n)$ like the ones described by Ajtai, Komlós, and Szemerédi [1]. However, these networks usually have large constant terms making them unsuitable for efficient implementations [63, p. 40]. Chapter 5 will detail how vectorization speeds up sorting networks in a non-asymptotic but practical way.

[10]Odd-even mergesort is more performant with only $(p^2 - p + 4)2^{p-2} - 1$ comparison elements compared to Bitonic mergesort's $(p^2 + p)2^{p-2}$ for $2^p$ elements. Both use $\frac{p(p+1)}{2}$ parallel steps for sorting.

Figure 3.2: Batcher's sorting networks for 32 elements.

(a) Bitonic sorting network with 240 comparison elements and 15 parallel steps [6, Abb. A.2].



(b) Odd-even sorting network with 191 comparison elements and 15 parallel steps [6, Abb. A.3].

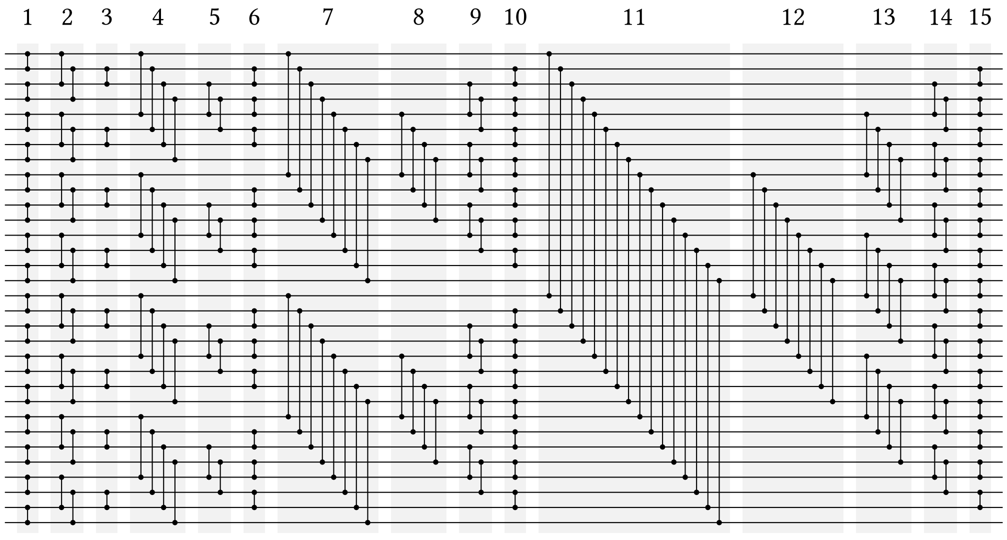


Table 3.1: Characteristics of known most optimal sorting networks for $n \leq 20$ with parallel steps $p(n)$ and number of comparison elements $c(n)$ [11, tab. 1].

| $n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $c(n)$ | 0 | 1 | 3 | 5 | 9 | 12 | 16 | 19 | 25 | 29 | 35 | 39 | 45 | 51 | 56 | 60 | 71 | 78 | 86 | 92 |
| | | | | | | | | | | | 33 | 37 | 41 | 45 | 49 | 53 | 58 | 63 | 68 | 73 |
| $p(n)$ | 0 | 1 | 3 | 3 | 5 | 5 | 6 | 6 | 7 | 7 | 8 | 8 | 9 | 9 | 9 | 9 | 10 | 11 | 11 | 11 |
| | | | | | | | | | | | | | | | | | | 10 | 10 | 10 |

## 3.3 Quicksort and Quickselect

In 1959, Charles Antony Richard Hoare described Quicksort [29, 31], a partitioning-based sorting algorithm. For the sake of completeness the following listing will show an implementation of Quicksort using Hoare's partitioning scheme [28]. The comments will explain Quicksort's steps as a substitute for further introduction to the algorithm.

Rust

```rust
/// Sort the array v using Quicksort.
fn quicksort(v: &mut [i32]) {
    let len = v.len();
    // Arrays of length 1 are considered sorted. Arrays with at least
    // two elements will be split at a pivot within the range of values
    // of the array.
    if len > 1 {
        // Determine the index of the pivot element by partitioning the
        // array so that all elements less than the pivot element are
        // on its left side and elements greater than the pivot element
        // on its right side.
        let pivot_index = partition(v);
        // Sort the part of the array where all elements are less than
        // the pivot value.
        quicksort(&mut v[0..pivot_index]);
        // Sort the part of the array where all elements are greater
        // than the pivot value.
        quicksort(&mut v[pivot_index + 1..len]);
    }
}
```

```rust
/// Partition the array v and return the index of the pivot element
/// determined by Hoare's partition scheme.
fn partition(v: &mut [i32]) -> usize {
    let len = v.len();
    // Use the element in the middle of the array as pivot element.
    // (Hoare used a random number between 0 and len.)
    let pivot = v[len / 2];

    // Initialize running indices for partitioning. These will point
    // to elements that are allowed to be swapped with other elements
    // to ensure consistency with the partitioning condition.
    let mut i = 0;
    let mut j = len - 1;

    loop {
        // Increment i so that it points to the next element that is on
        // the pivot's left side but should be on its right side.
        while v[i] < pivot {
            i += 1;
        }

        // Decrement j so that it points to the next element that is on
        // the pivot's right side but should be on its left side.
        while v[j] > pivot {
            j -= 1;
        }

        // If the next swap would actually destroy the order again,
        // return the index of j as the pivot's index.
        if i >= j {
            break j;
        }

        // If the swap is consistent perform it and start over again.
        v.swap(i, j);
    }
}
```

This basic Quicksort has been subject to a plethora of further research. Among the most significant contributions are Sedgewick's PhD thesis [56] – analyzing many common techniques for speeding up Quicksort – and Samplesort –

a method to split the array into multiple buckets instead of two which eases parallelization – as described by Frazer and McKellar [19].

Sedgewick includes an analysis of Quicksort's worst case based on his implementation. Using the smallest or largest element as the pivot element is the easiest way always to trigger worst-case behavior. Randomizing the pivot element is highlighted as one important resolution strategy. Trying to destroy patterns and bad partitions by randomization or at least shuffling is adapted by hybrid algorithms as well (like pattern-defeating Quicksort introduced in the next section).

A guaranteed mitigation of the quadratic worst case has been implemented by Daoud, Abdel-jaber, and Ababneh. They use two pivot computations, one based on median-of-three and the other based on an adapted Radix sort to guarantee a worst-case complexity of $2 \cdot \mathcal{O}(32 \cdot n)$ for 32 bit integers [14]. One notable aspect of this partitioning is the use of values that do not appear in the array as pivot elements. As a consequence the pivot element does not necessarily appear in the partitioned array. This mitigation technique has been adapted by Blacher [6].

It is important to note that an efficient implementation of Quicksort inherently means that the partitioning is implemented efficiently, which leads to a fast algorithm for the problem of selection.[11] This algorithm based on Quicksort's partitioning has been introduced as Quickselect [30]. An implementation based on the Quicksort code example's partitioning implementation could be the following.

Rust
```rust
/// Get the k th smallest element in v using quickselect.
fn quickselect(v: &mut [i32], k: usize) -> i32 {
    let len = v.len();
    // Arrays of length 1 are considered sorted. Arrays with at least
    // two elements will be split at a pivot within the range of values
    // of the array.
    return if len == 1 {
        // For arrays of length 1 the only selectable element is the
        // first one.
        assert_eq!(0, k);
        v[k]
```

[11] A selection algorithm determines the $k^{\text{th}}$ smallest element in some array of elements. When solved using in-place Quickselect at least the $k^{\text{th}}$ element in the modified array is at the position where it would be in a sorted array. This highlights the relationship to Quicksort.

```rust
    } else {
        // Determine the index of the pivot element by partitioning the
        // array so that all elements less than the pivot element are
        // on its left side and elements greater than the pivot element
        // on its right side.
        let pivot_index = partition(v);

        if k == pivot_index {
            // We have found the k th element at the pivot position.
            v[pivot_index]
        } else if k < pivot_index {
            // k is less than the pivot's index. Then we restart our
            // search to the left of the pivot because at least the k
            // smallest elements will be in that partition.
            quickselect(&mut v[..pivot_index], k)
        } else {
            // k is greater than the pivot's index. Then we restart our
            // search to the right of the pivot because all elements
            // including the pivot are less than the k th smallest element
            // would be. It is important to note that we need to change k
            // to account for the shortening of the array.
            quickselect(&mut v[pivot_index + 1..], k - pivot_index - 1)
        }
    };
}
```

## 3.4 Relevant hybrid sorting algorithms

A hybrid sorting algorithm combines multiple sorting algorithms usually starting with a particularly fast algorithm on large data sets and switching to another algorithm for small problem sizes. In the previous section, the worst case of Quicksort in $\mathcal{O}(n^2)$ has been discussed. This worst case is eliminated in most production-grade implementations of Quicksort by using another fast algorithm for certain distributions or small arrays.

The most notable hybrid sorting algorithm based on Quicksort is Introsort [46] from 1997. In fact, Introsort is used in `std::sort` from the C++ standard template library [20]. Introsort applies Quicksort until a certain recursion depth boundary has been reached (usually logarithmic in the length of the array) and then switches to Heapsort to break the quadratic worst case of Quicksort. Sorting is finished by using insertion sort on very small arrays.

In 2015, Peters published the pattern-defeating Quicksort [47] as an extension of Introsort. It applies multiple techniques to detect patterns in the input data. Among others it applies an insertion sort that is bounded by a maximum number of swaps to seemingly balanced arrays. Additionally, shuffling during the pivot selection reorders some elements deterministically to break up patterns. Pattern-defeating Quicksort is the default algorithm for unstable sorting in the Rust standard library (`std::slice::sort_unstable`) [52].

Apart from the original implementations and those in the standard libraries there are recent efforts to provide formally verified implementations such as the ones provided by Lammich [39]. Due to the availability of the standard libraries the implementations of the C++ standard template library and the Rust standard library will be used for further evaluation of the respective algorithms' speed.

All of the aforementioned implementations are scalar in that they are not vectorized by design. Vectorized hybrid algorithms include those by Gueron and Krasnov [24] (Quicksort and insertion sort), the one by Blacher, Giesen, and Kühne [7] (Quicksort and sorting networks) as well as the one by Bramas [8]. These vectorized approaches will be discussed in the next chapter.

# 4 Related work

As early as in 1977, Siegel examined a SIMD implementation of Batcher's Bitonic sorting networks [58]. Stone implemented a vectorized Quicksort and an adapted version of Batcher's sorting networks on the vector computer CDC STAR only one year later [59]. He did not implement a hybrid algorithm combining both of his vectorized implementations.

A hybrid algorithm using Quicksort and a sorting network on a vector computer has been implemented by Levin in 1990 [42]. His implementation used a vectorized odd-even transposition sort as sorting network, not to be confused with Batcher's more efficient odd-even mergesort. Levin, like Stone, used compression instructions, which have been lacking from the x86 instruction set until the advent of the AVX-512 vector extensions.

On more modern architectures, Inoue et al. implemented a hybrid vectorized merge-based sorting algorithm using IBM's vector media extensions (VMX) in 2007 [32]. One year later, Chhugani et al. used SSE to implement a hybrid vectorized merge-based sorting algorithm as well but introduced a sorting strategy based on vertical sorting and transposition [10].

Satish et al. investigated differences between SIMD-based sorting on CPUs and GPUs in the context of sorting large database keys in 2010 [54]. They concluded from comparing Mergesort with Radix sort that in the future the performance gap between the two will close and a SIMD-friendly bandwidth-oblivious Mergesort will be the sorting algorithm of choice.

Similarly, Polychroniou and Ross devised SIMD sorting algorithms based on Radix sort as well as range partitioning and focused on executing on non-uniform memory access (NUMA) architectures in 2014 [48]. They concluded that their Radix sort implementations are most performant in general but their comparison-based sort is better for balancing loads. This conclusion agrees with the findings of Satish et al.

In 2015, Hayes et al. assessed the potential of multiple vectorized sorting algorithms including Quicksort and, like Levin, combined it with odd-even transposition sort [26]. They concluded that the vectorization of Radix sort is the most promising approach in terms of speedup and devised a cache-friendly sorting algorithm based on Radix sort. For that, they also proposed new vector instructions and their potential implementation in hardware. Their conclusion is contrary to the results of Satish et al. and Polychroniou and Ross.

Gueron and Krasnov vectorized the partitioning function of Quicksort using AVX2 in 2016 [24]. Their stable hybrid implementation of Quicksort needed additional memory and used scalar insertion sort for small partitions. Due to the shortcomings of AVX2 they used a lookup table for shuffle masks to simulate compressing instructions.

Another vectorized hybrid algorithm combining Quicksort with a branchless Bitonic mergesort has been implemented by Bramas in 2017 [8]. He used AVX-512 and its compressing instructions to avoid the inclusion of lookup tables as used by Gueron and Krasnov. By saving the outermost vectors, his partitioning implementation does not need additional memory.

Blacher implemented a hybrid vectorized sorting algorithm consisting of Quicksort, transposition-based sorting as introduced by Chhugani et al., and multiple sorting networks using AVX2 intrinsics. His partitioning is based on Bramas' approach to avoid using additional memory and Gueron and Krasnov's approach to simulate compressing store instructions using AVX2.

In 2019, Yin et al. presented a multi-threaded merge-based sorting algorithm making use of vectorized bitonic sorting and load-balanced merges. They used AVX-512 on the Xeon Phi architecture. While designed for multi-threaded use, it outperforms common sorting algorithms in single-threaded mode [65].

Watkins implemented a vectorized Mergesort on AVX-512 without the help of sorting networks to avoid unnecessary operations on presorted data. He focused on branch avoidance and achieved a speedup of 2 compared to IPP Radix sort [64]. Unfortunately, the code for his implementation is unavailable, as is the code for the implementation of Yin et al.

# 5 Porting AVX2 sort to AVX-512

## 5.1 Choice of language and tooling

The reference implementation of the following AVX-512 sort is written in Rust, a system language designed by Mozilla to achieve maximum performance while providing memory safety by default. A comprehensive summary of Rust's solutions to common problems in system-level development has been published by Jung et al. [35].

While the language itself is young compared to C and C++, the prevalent languages of the field, Rust implementations are highly competitive in terms of performance and the consistent tooling helps increasing development productivity. The extensible build tool `cargo` eases the integration of unit and fuzz tests as well as providing executable examples and benchmarks.

There are some unresolved issues with respect to intrinsics handling though. The most important one is the lack of certain intrinsics in Rust's standard library[12] [13]. Amanieu d'Antras explains ibidem that especially the masked load and store operations cannot be implemented currently due to representation differences between Rust types and the arguments of the respective LLVM intrinsics.

However, this does not restrict the use of Rust. First-class C FFI support is characteristic for that language. A foreign function interface (FFI) allows a language to call functions written in another language, in this case calling C from Rust. Because C and C++ share many characteristics, it is easy to link the missing intrinsics into the Rust program by using code like the following excerpt from a C++ glue code header.

```cpp
void ffi_mm512_mask_compressstoreu_epi32(int32_t* base_addr,
                                          uint16_t k, const int32_t* a) {
  _mm512_mask_compressstoreu_epi32((void*)base_addr, k,
```

[12]Rust's standard library includes `stdarch`, which is developed in a separate repository. `stdarch` provides architecture-dependent intrinsics such as the AVX intrinsics.

```
                                    *((const __m512i*)a));
}
```

The Rust counterpart to complete the FFI bindings is shown in the following snippet. Using these bindings, Rust code may use this missing intrinsic function as if it was native to Rust's `stdarch` library.

Rust
```rust
#[cxx::bridge(namespace = "avx512_bridge")]
mod ffi_internal {
    unsafe extern "C++" {
        include!("avx-sort/include/avx512_bridge.h");

        pub unsafe fn ffi_mm512_mask_compressstoreu_epi32(
            base_addr: *mut i32, k: u16, a: *const i32,
        );
    }
}

pub unsafe fn _mm512_mask_compressstoreu_epi32(
    base_addr: *mut i32, k: __mmask16, a: __m512i,
) {
    ffi_internal::ffi_mm512_mask_compressstoreu_epi32(
        base_addr, k as u16, &a as *const _ as *const i32,
    );
}
```

C FFI may not only be used to include missing intrinsics. The benchmarking application presented in chapter 6 makes extensive use of C FFI to include the other benchmarked algorithms written in C++. For C++ interaction in general David Tolnay's `cxx` crate[13] is used as a build dependency.

The same crate allows to specify a C interface for the Rust side. The AVX-512 sorting crate exposes a C++ interface as header. Building a dynamic library in addition to Rust's library format allows linking it into C++ programs. The following listing shows the functions exposed in the C++ header. They are exported into a common namespace and correspond to Rust functions from several places in the crate's module structure.

C++
```cpp
bool arch_optimal_quickselect_c(int32_t *ptr, size_t n, size_t k);
void arch_optimal_quicksort_c(int32_t *ptr, size_t n);
```

[13] A crate is Rust's equivalent of a library. The inclusion of crates into projects is managed by `cargo`.

```c
bool avx2_quickselect_c(int32_t *ptr, size_t n, size_t k);
bool avx512_quickselect_c(int32_t *ptr, size_t n, size_t k);
void avx2_quicksort_c(int32_t *ptr, size_t n);
void avx512_quicksort_c(int32_t *ptr, size_t n);
```

These functions address the most common operations of the crate by using either the provided methods directly or the runtime-based check to decide in favor of the most optimal Quickselect or Quicksort implementation available. An example of how to use these from the C++ side is included in the project's repository, the most important files of the CMake project may be found in appendix A.1 on page 79.

To get a reasonable baseline for comparisons, Mark Blacher's AVX2-based sorting algorithm has been translated from C++ to Rust. In the course of this translation and writing corresponding unit tests, a bug has been found and corrected.[14] The Rust library provides an interface to use his C++ implementation. The benchmarks in chapter 6 provide evidence that there is only an insignificant performance gain in using the C++ over the Rust implementation, although the Rust implementation deliberately uses more high-level code.

Apart from slightly different levels of abstraction, the Rust implementation differs by providing a test suite with more than 80 % code coverage.[15] Additionally, fuzz tests are provided to help guaranteeing the correctness of the implementation. At least 50 hours of fuzzing have been invested into validating the AVX2 implementation. Due to linker workflow issues, the fuzz tests are not usable to validate the AVX-512 implementation.

## 5.2 Pivot calculation and introduction

Relevant source file:
`avx-sort/src/avx512/pivot/mod.rs`

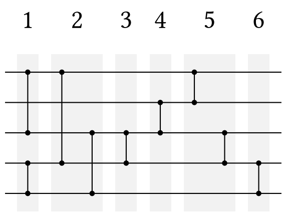
In contrast to other changes, the pivot calculation has not been subject to major changes. The Xoroshiro128+ implementation has been extended to work on `__m512i` vectors, in consequence doubling the number of generated random numbers per iteration.

[14]See `https://github.com/simd-sorting/fast-and-robust/commit/1bc8cc0964f7c0153c39292d6c6ec36b39b0ef7d` for details.

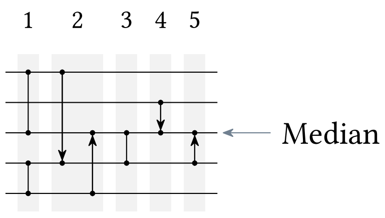
[15]This is a conservative estimate for the coverage when compiled with AVX-512 support. As the code makes extensive use of inlining, the coverage tool is unable to trace all the tested code back to their sources so that it reports less coverage than actually present.

Figure 5.1: An optimal sorting network for 5 elements with 9 comparison elements reduced to an optimal median network with 10 minimum/maximum operations [57, 61].

(a) Sorting network.



(b) Median network.



Apart from that, the pivot calculation uses a sparse array of 5 instead of 9 random vectors of elements gathered from the input. In AVX2, Blacher operates on 72 elements, this approach uses 80 elements in AVX-512. The high-level implementation of the pivot calculation may be found in listing A.3 on page 81.

Because there are only 5 vectors, a smaller median network is used to determine the vector of medians, as depicted in fig. 5.1, along with the sorting network it has been deduced from. Sorting networks may be reduced to median networks by eliminating redundant comparisons, i.e. minimum/maximum operations. Hence, fig. 5.1b contains directed edges where only the respective operation is performed, e.g. only the maximum operation for edges pointing downwards.

In general, compilers are able to perform this optimization on their own. Given the full sorting network in

```rust
#[inline(always)]
fn swap(i: &mut f32, j: &mut f32) {
    if j > i { std::mem::swap(i, j); }
}

pub fn median_of_5 (mut a0: f32, mut a1: f32, mut a2: f32,
                    mut a3: f32, mut a4: f32) -> f32 {
    swap(&mut a0, &mut a2);
    swap(&mut a3, &mut a4);
    swap(&mut a0, &mut a3);
    swap(&mut a2, &mut a4);
    swap(&mut a2, &mut a3);
    swap(&mut a1, &mut a2);
    swap(&mut a0, &mut a1);
    swap(&mut a2, &mut a3);
    swap(&mut a3, &mut a4);
    a2
}
```

the compiler produces the following assembly which is optimal.[16]

```asm
example::median_of_5:
        vminss  xmm5, xmm0, xmm2
        vmaxss  xmm0, xmm2, xmm0
        vminss  xmm2, xmm3, xmm4
        vmaxss  xmm2, xmm2, xmm5
        vmaxss  xmm3, xmm4, xmm3
        vminss  xmm0, xmm0, xmm3
        vminss  xmm3, xmm2, xmm0
        vmaxss  xmm0, xmm0, xmm2
        vminss  xmm0, xmm1, xmm0
        vmaxss  xmm0, xmm3, xmm0
        ret
```

Micro-benchmarking has shown that the AVX2 pivot calculation is still faster with only 16 ns, where the AVX-512 version takes 25 ns.[17] Reducing the number of random vectors has a major influence on the runtime of the pivot selection because it is dominated by `_mm512_i32gather_epi32` with a latency of 30 (compared to approx. 20 on AVX2) with throughput of approx. 10.

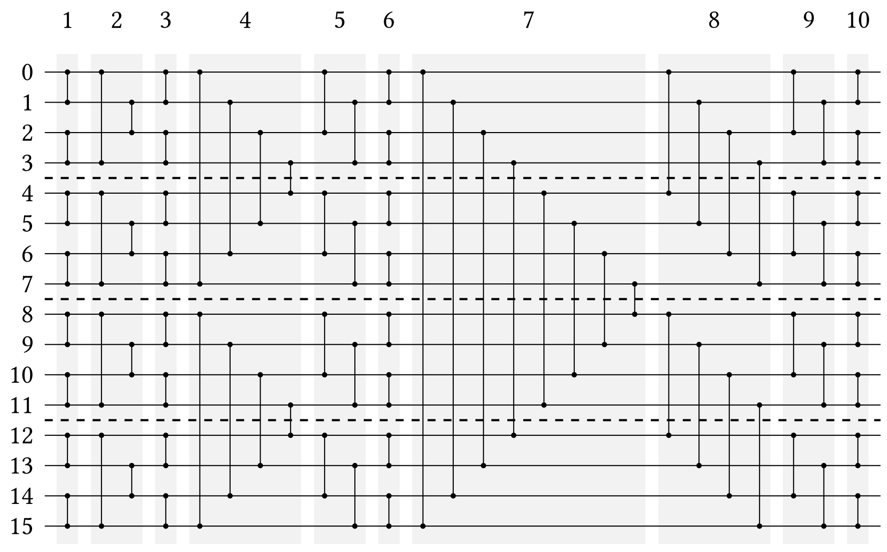
Micro-optimizing the runtime of the pivot selection does not have a major influence on the whole partitioning. Benchmarks showed only marginal impact, which may well be measuring inaccuracy, when using faster vectorized approaches with more locality of memory accesses. One of these very latency-efficient approaches uses loading of whole vectors from random positions instead of gathering elements from random positions in the array.

However, all other tested methods did not guarantee as much randomness and distribution of the selected elements while providing nearly no gain in speed for the selection part of the algorithm, i.e. partitioning and pivot selection. Therefore, Blacher's approach has been kept.

[16]This code uses floats instead of integers to present the simplest assembly output. Similar code for integers is compiled into the optimal 10 minimum/maximum operations as well.

[17]With 9 random vectors as in the AVX2 version, an AVX-512 implementation would take 45 ns. This is nearly twice the runtime compared to using only 5 random vectors.

Figure 5.2: Bitonic sorting network for 16 elements.



After using the median network to determine the vector of medians, the median of elements in this vector is determined by sorting the 16 elements. Therefore, a Bitonic sorting network of that size has been implemented as shown below.[18]

```rust
// fn sort_16(vec: &mut __m512i) {
// groups of 2
coex_shuffle::<1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14>(vec);
// groups of 4
coex_shuffle::<3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8, 15, 14, 13, 12>(vec);
coex_shuffle::<1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14>(vec);
// groups of 8
coex_permute::<7, 6, 5, 4, 3, 2, 1, 0, 15, 14, 13, 12, 11, 10, 9, 8>(vec);
coex_shuffle::<2, 3, 0, 1, 6, 7, 4, 5, 10, 11, 8, 9, 14, 15, 12, 13>(vec);
coex_shuffle::<1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14>(vec);
// group of 16
coex_permute::<15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0>(vec);
coex_permute::<4, 5, 6, 7, 0, 1, 2, 3, 12, 13, 14, 15, 8, 9, 10, 11>(vec);
coex_shuffle::<2, 3, 0, 1, 6, 7, 4, 5, 10, 11, 8, 9, 14, 15, 12, 13>(vec);
coex_shuffle::<1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14>(vec);
// }
```

In fig. 5.2, there are three dashed lines separating the four 128 bit lanes.

[18]This implementation corresponds to the illustration in fig. 5.2.

Wherever no comparison element crosses lane boundaries in a parallel step, shuffle operations are applied. Otherwise, permute operations will perform the exchanges. Shuffles have a latency of 1 compared to the latency of 3 for permutes. This is especially important in a function like this, where the results of each line depend on the result of the previous line so that no pipelining could improve the runtime by itself.

The indices in the figure correspond to the indices as used by the code snippet. Elements are indexed like vectors starting at 0. For example, the index sequence

```text
<15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0>
```

for step 7 represents a comparison element between 0 and 15, 1 and 14 and so on. Because they correspond to comparison elements, the indices will always be symmetric, i.e. if the index at position 0 is 15, the one at position 15 will be 0.

For the sake of completeness, the definition for the permutation will be shown below.

```rust
1  // unsafe fn coex_permute<const a: i32, …>(vec: &mut __m512i) {
2  let permute_mask = _mm512_setr_epi32(
3      a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p
4  );
5  let permuted = _mm512_permutexvar_epi32(permute_mask, *vec);
6  let min = _mm512_min_epi32(permuted, *vec);
7  let max = _mm512_max_epi32(permuted, *vec);
8  *vec = _mm512_mask_blend_epi32(
9      build_mask_16(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p),
10     min,
11     max,
12 );
13 // }
```

Each permute operation starts with setting a permute mask. The important step is permuting in line 5. This results in the vector `permuted` being exactly the same as `vec` with all elements of the comparison elements swapped, i.e. if there is a comparison element between 0 and 1 the vector `permuted` will have the element with index 1 at position 0 and vice versa. Afterwards, it is possible to perform a compare-exchange operation by utilizing element-wise minimum and maximum. Element-wise minima and maxima are blended into the result to provide the output of the parallel execution of multiple comparison elements.

For shuffling, the `permuted` vector is replaced by a `shuffled` vector like the following. Because it only operates on 128 bit lanes the permute mask is replaced by a shuffle mask made up of the first 4 indices.

```rust
let shuffled: __m512i = _mm512_shuffle_epi32(
    *vec,
    _MM_SHUFFLE(d as u32, c as u32, b as u32, a as u32),
);
```

## 5.3 Enlargening sorting networks

Relevant source file:
`avx-sort/src/avx512/networks/mod.rs`

### 5.3.1 Sorting two vectors

As previously shown, the pivot calculation uses a Bitonic sorting network for 16 elements. Based on this, one could define a strategy to sort 32 elements by sorting 16 elements twice and then applying the merge steps of both sorted sequences like below. This is the strategy employed by Bramas [8].

```rust
// unsafe fn sort_32(vecs: &mut CoExI32x16) {
// assert!(vecs.len() == 2);
sort_16(&mut vecs[0]);
sort_16(&mut vecs[1]);

coex_permute::<15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0>(
                                                        &mut vecs[0]);
vecs.co_ex(0, 1);

coex_permute::<8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7>(
                                                        &mut vecs[0]);
coex_permute::<8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7>(
                                                        &mut vecs[1]);
vecs.co_ex(0, 1);

coex_permute::<4, 5, 6, 7, 0, 1, 2, 3, 12, 13, 14, 15, 8, 9, 10, 11>(
                                                        &mut vecs[0]);
coex_permute::<4, 5, 6, 7, 0, 1, 2, 3, 12, 13, 14, 15, 8, 9, 10, 11>(
                                                        &mut vecs[1]);
vecs.co_ex(0, 1);

coex_shuffle::<2, 3, 0, 1, 6, 7, 4, 5, 10, 11, 8, 9, 14, 15, 12, 13>(
                                                        &mut vecs[0]);
coex_shuffle::<2, 3, 0, 1, 6, 7, 4, 5, 10, 11, 8, 9, 14, 15, 12, 13>(
                                                        &mut vecs[1]);
vecs.co_ex(0, 1);

coex_shuffle::<1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14>(
                                                        &mut vecs[0]);
coex_shuffle::<1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14>(
                                                        &mut vecs[1]);
vecs.co_ex(0, 1);
// }
```

While it seems to be efficient – the merge operation may even utilize pipelining – it is a wasteful use of vector registers. The permute and shuffle operations introduced in section 5.2 always allocate four vector registers, one for the mask, one for the shuffled/permuted vector, and one each for minimum and maximum, in addition to the vector register already allocated for the vector itself. And of the minimum and maximum vectors only half the elements are even accessed or needed.

To address this concern, Blacher implemented a strategy for merging two vectors by shuffling and performing compare-exchange operations [6, pp. 32 ff.]. His approach, extended to sort two vectors of 16 elements each, is depicted in fig. 5.3. It follows the Bitonic sorting network introduced in fig. 3.2 on page 25. The circled numbers correspond to the parallel steps of the Bitonic network.

In the figure, the registers contain the indices of the numbers to be sorted, i.e. consider the horizontal lines in fig. 3.2 numbered from 0 to 31 and use these indices as a representation of the actual numbers to be sorted. The starting point is arbitrary; to ease the first parallel step, the even indices are assigned to the values in the first register and the odd indices to the values in the lower register.

The aim of Blacher's transformation is the increase of simultaneous comparisons. Where the previously shown naïve approach performs a multitude of allocations and mutually dependent operations, Blacher uses a vectorized

Figure 5.3: Sort two vectors at once using a Bitonic sorting network. Indices above and below register entries indicate the position after performing the compare-exchange operation.

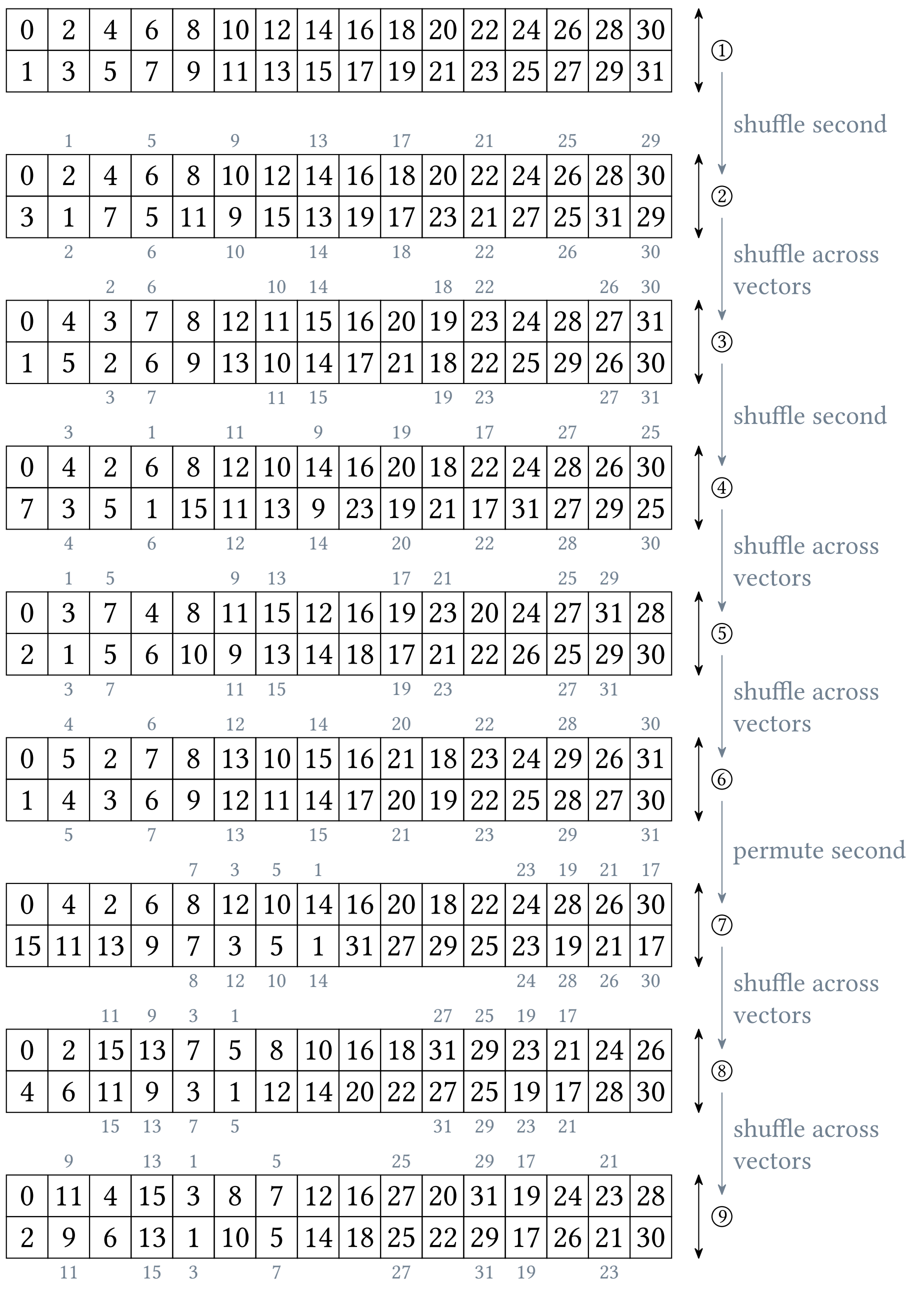
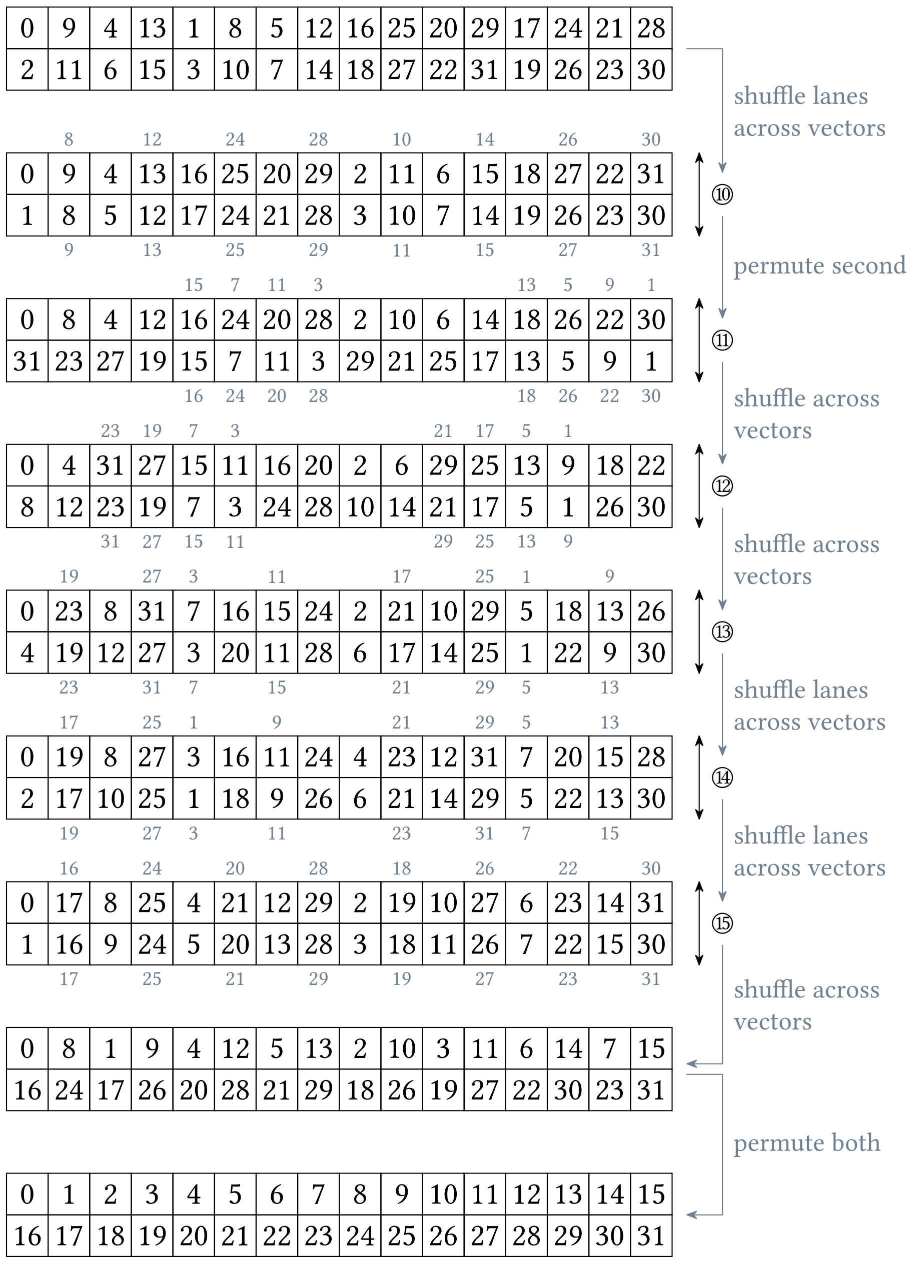


Figure 5.3: Sort two vectors at once using a Bitonic sorting network. Indices above and below register entries indicate the position after performing the compare-exchange operation. (continued)



compare-exchange operation by operating on whole registers. That is, in a parallel step of a Bitonic each element is connected to another element. Each of these compare-exchange operations in the network is realized by having the indices of the compared elements at the same position in both registers.

This method of sorting involves four basic operations:

- compare-exchange steps, indicated by the vertical double-ended arrow in the figure and performed at the end of each parallel step,
- shuffle and permute within a vector to get the correct opposing positions for the compare-exchange step,
- shuffling across vectors using `_mm512_shuffle_ps` to adapt to differing compare-exchange patterns requiring intra-lane inter-register changes (e.g. interleaving the lanes of both registers), and
- shuffling of lanes across vectors using `_mm512_shuffle_i32x4` to avoid permuting before or after performing an intra-lane inter-register shuffle which would be the alternative.

Reasoning about the respective latencies for both implementations gives a vague idea of the possible speedup of this kind of vectorization. The non-optimized implementation has a latency of

$$2 \cdot \texttt{sort\_16} + 5 \cdot \texttt{coex\_permute} + 4 \cdot \texttt{coex\_shuffle} + 5 \cdot \texttt{co\_ex} = 119\,,$$

whereas the optimized implementation has a latency of only

$$\begin{aligned}15 \cdot \texttt{co\_ex} &+ 2 \cdot \texttt{\_mm512\_shuffle\_epi32} + 2 \cdot \text{single permute}\\ &+ 1 \cdot \text{double permute (with pipelining)} + 8 \cdot \text{double } \texttt{shuffle\_2\_vecs}\\ &+ 3 \cdot \text{double } \texttt{\_mm512\_shuffle\_i32x4} = 72\,.\end{aligned}$$

This latency calculation is not meant to be precise. Register transfers and especially temporary variables have been omitted for the sake of simplicity. Still, the expected speedup is approximately $119/72 \approx 1.65$. Micro-benchmarking real implementations results in runtimes of 30 ns for the non-optimized version and 18 ns for the optimized version, which is $30/18 \approx 1.66$, even more than expected.

Explicit vectorization like this is a useful step for optimizing the overall runtime due to frequent calls to the function. As there is no proven optimal sorting network for 32 elements [11], the implementation is based on a Bitonic sorting network. However, for this kind of vectorization the number of parallel steps is decisive, so that a network with less parallel steps might improve the overall performance. This is left for further evaluation.

Furthermore, this implementation uses shuffling across vectors and shuffling of lanes across vectors. While the former has low latency, the shuffling of lanes across vectors needs to be pipelined to be efficient. Interleaving two vectors by means of unpacking instructions like `_mm512_unpackhi_epi32` might be worth exploring and is left for further evaluation as well.

### 5.3.2 Merging sorted columns

For sorting larger arrays using sorting networks, the construction of a network like the one previously shown is infeasible. Therefore, Blacher devised the following basic algorithm (numbers adapted for AVX-512):

1. Pad the array to a multiple of twice the vector length using the integer maximum for padding.[19] In this case, the array is padded to a multiple of 32 because the vector length is 16 elements. Doubling the vector length is necessary because groups of two vectors are the minimal expected input for sorting.
2. Sort chunks of 32 vectors column-wise. This exploits data-level parallelism and is easy to construct using algorithmic sorting networks with a low number of comparison elements like Batcher's Odd-even mergesort. In contrast to sorting two vectors at once, this needs to be optimized for comparison elements instead of parallel steps because the comparisons are executed sequentially.
3. Merge the sorted columns of 32 elements. Details on this will follow.
4. Perform a Bitonic merge on the whole padded array. This finishes the sorting. Afterwards, the padding will be cut off and the input array is sorted.

Of these steps, the first and last are trivial and the second is explained in the previous section. The remaing, third, step is interesting as it includes sorting $16 \cdot 32 = 512$ elements at once. A sorting network of size 512 is rather large, so a simple approach to merging the sorted columns uses transposition. This is the approach used by Chhugani et al. [10].

In contrast to their implementation, the 16 by 32 array is not a square but rectangular matrix. Hence, a useful transposition consists of transposing both 16 by 16 blocks separately and then swapping rows so that the rows of one column form a contiguous region in memory. Afterwards, the same

[19]The choice of the padded value is arbitrary as long as the elements are easy to remove after sorting. Using the integer maximum allows cutting off the padding after sorting.

Bitonic merge applicable to the fourth step may be used to merge the vectors, starting from the assumption that 32 elements each are already sorted (i.e. perform the Bitonic steps for groups of 64 up to groups of 512 elements).

The following code snippet implements the transposition-based approach. This high-level function makes use of an AVX-512-based transposition [62] and a Bitonic merge method following closely what Blacher implemented for AVX2 in his `bitonic_merge_16` function. A visualization of this code snippet using indices is presented in fig. 5.4.

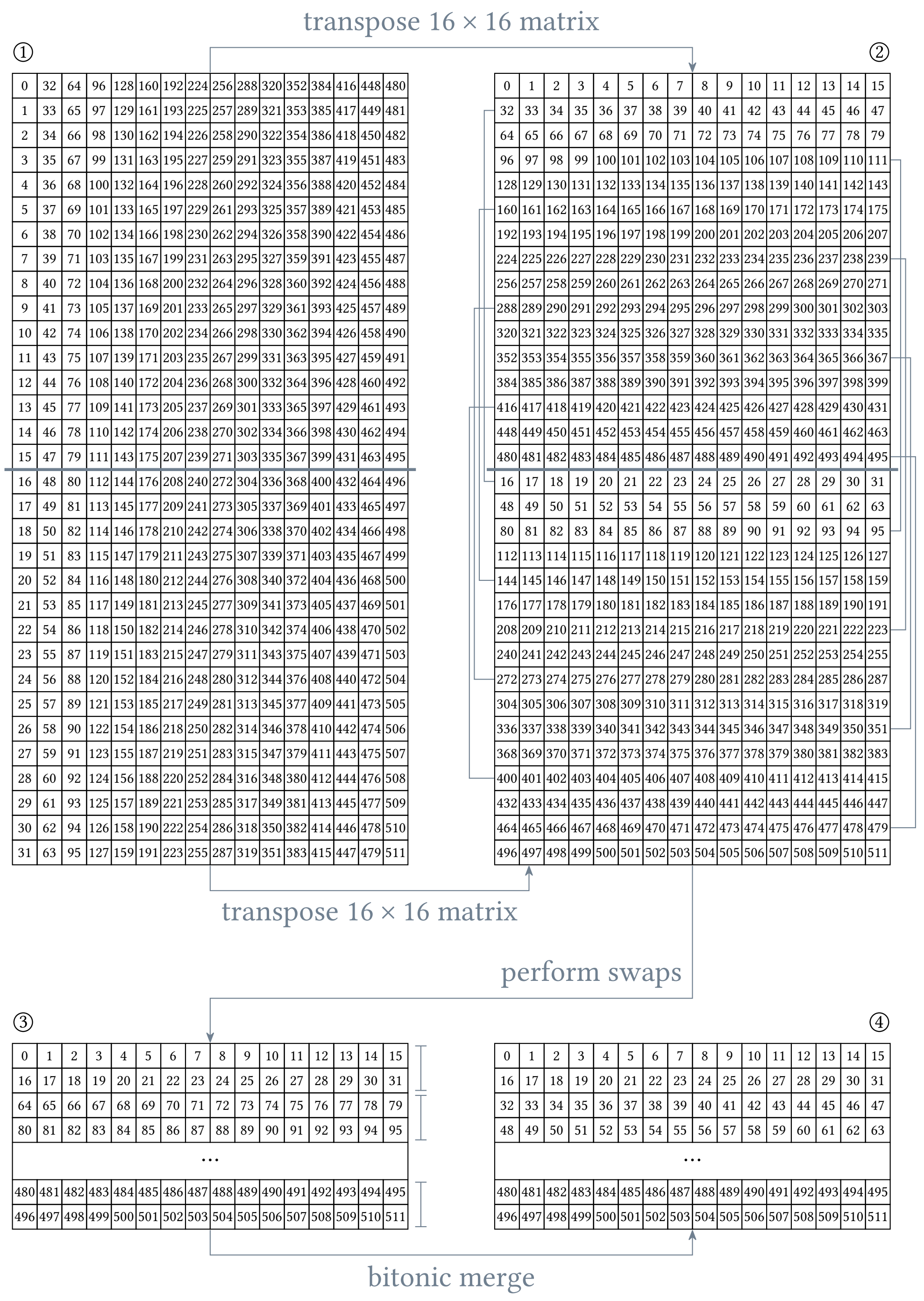
Rust

```rust
#[inline]
unsafe fn merge_16_columns_with_32_elements(vecs: &mut CoExI32x16) {
    // sort blocks
    transpose_16_by_16(&mut vecs[0..16]);
    transpose_16_by_16(&mut vecs[16..32]);

    // perform swaps
    vecs.swap(1, 16);
    vecs.swap(3, 18);
    vecs.swap(5, 20);
    vecs.swap(7, 22);
    vecs.swap(9, 24);
    vecs.swap(11, 26);
    vecs.swap(13, 28);
    vecs.swap(15, 30);

    // two vectors each are sorted, merge them
    bitonic_merge_32(vecs);
}
```

As can be seen from the structure of the code snippet, the transposition-based approach has multiple stages: transposing, swapping, and merging. However, swapping could as well be replaced by element-wise (vectorized) comparison elements and the Bitonic merge may already start from the matrix of sorted columns. Hence, the remaining part of this section will outline an approach without transposition.

Consider the first two columns of step ① in fig. 5.4: both are sorted and should be merged. In a Bitonic sorting network, the first step contains comparison elements between 0 and 63, 1 and 62, 2 and 61, and so on until 31 and 32 are compared. Swapping the first and second column in the lower square matrix, it is obvious that element-wise comparison elements may be applied to the

Figure 5.4: Merging sorted columns by applying transposition. In step ②, vertical connections indicate rows to be swapped in order to create contiguously stored sorted sequences of 32 elements.



vectors, from comparing the outer-most vectors to comparing the inner-most vectors.

After the first group of Bitonic merging, 64 elements are sorted, i.e. two columns, after another merge process 128 elements, i.e. four columns, and so on until all 512 elements have been sorted. The primary operations are shuffle and permute operations and element-wise comparison elements on registers. Only for the last steps of the respective Bitonic merging, in-register comparison elements have to be applied (the functions `coex_permute` and `coex_shuffle` as already introduced).

For a more visual description, consider the smaller example in fig. 5.5. It shows the merge of 4 columns with 4 elements each. Steps ② to ⑤ perform a Bitonic merge operation on two vectors, so that 8 elements each are sorted afterwards. In contrast to the implemented merging strategy for 512 elements, this example only ever swaps within lanes, which allows for the use of shuffle operations in addition to element-wise comparison elements.

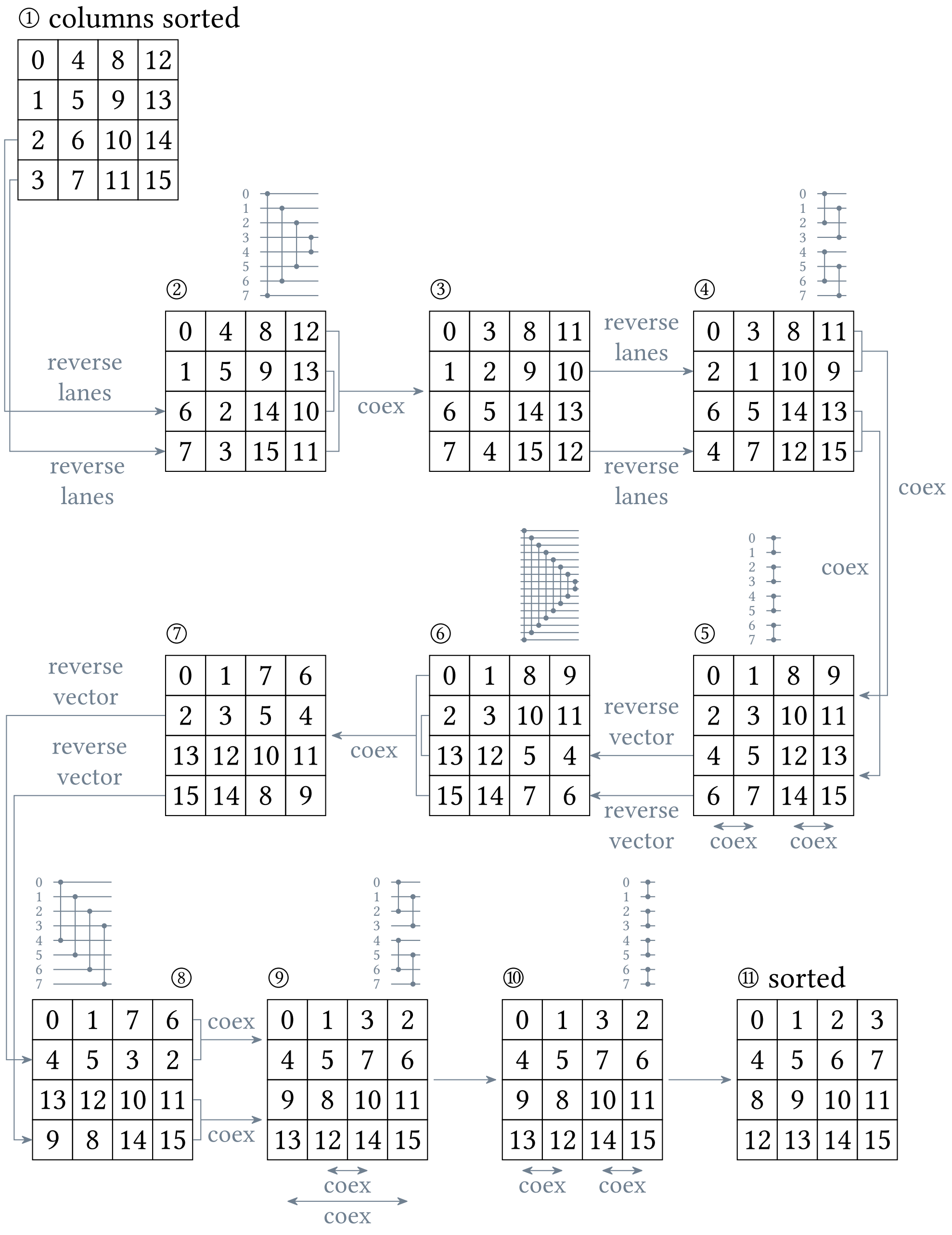
Starting from step ⑥, the final merge of the two sorted halves is performed so that the whole array is sorted in step ⑪. With each time the size is doubled the number of slow in-register comparison elements increases by one (step ⑨ and ⑩ vs. step ⑤).

In the AVX-512 implementation of sorting an array of $16 \cdot 32 = 512$ elements, this results in 4 in-register comparison elements per vector, i.e. 128 in-register comparison elements compared to only 80 permutes and 80 element-wise comparison elements. That is, the merge of two sorted halves of 256 elements is already dominated by in-register comparison elements, which shows that the process of constructing transpositionless merge networks does not scale indefinitely.

Returning to the performance evaluation, there is a measurable gain in efficiency when using the transpositionless approach over the transposition-based one. The latter implementation needs 492 ns, whereas the former one only needs 332 ns to complete the merge of 16 sorted columns. This is a speedup of $^{485}/_{335} \approx 1.48$.

Experiments with manual loop unrolling – as used by Blacher in his C++ implementation of a transpositionless merge – and other simple optimization techniques did not result in further speedup. Using a merging algorithm

Figure 5.5: Merging sorted columns without transposition using vectors of length 4 and lanes of 2 elements each [6, p. 38]. The strategy is based on the Bitonic sorting network from fig. 5.2, parts of which are depicted where relevant.



with less comparison elements like odd-even mergesort[20] instead of Bitonic merging could be beneficial to the overall performance but has not been explored and is left for further research.

## 5.4 Updating partitioning

Relevant source file:
`avx-sort/src/avx512/partition/mod.rs`

The changes to the partitioning function for a single vector have been the introductory example in section 2.3 on page 17. Compared to the major change in instructions for partitioning a single vector, the changes to the partitioning of larger arrays have been insignificant.

Figure 5.6 visualizes the basic principle of partitioning an array using vectors of size 4 on an array of 20 elements as devised by Blacher [6, p. 53]. Elements are stored from `l_store` or `r_store` onwards. `left` and `right` (exclusive) are running indices to determine which elements remain to be processed.

In contrast to Blacher's version of storing the entire vector of 4 elements to the store points due to the restrictions of AVX2, AVX-512 provides compressed store instructions, which are used to avoid writing duplicates. Thus, `r_store` is the inclusive right bound where to store the elements greater than the pivot as opposed to the start of the vector to be stored in the source figure.

Micro-benchmarking has proven this change to be very effective. Without compressing store instructions, partitioning a vector takes 11 ns where it takes only 5 ns using compressing stores. That is, a speedup of $11/5 = 2.2$ could be achieved.

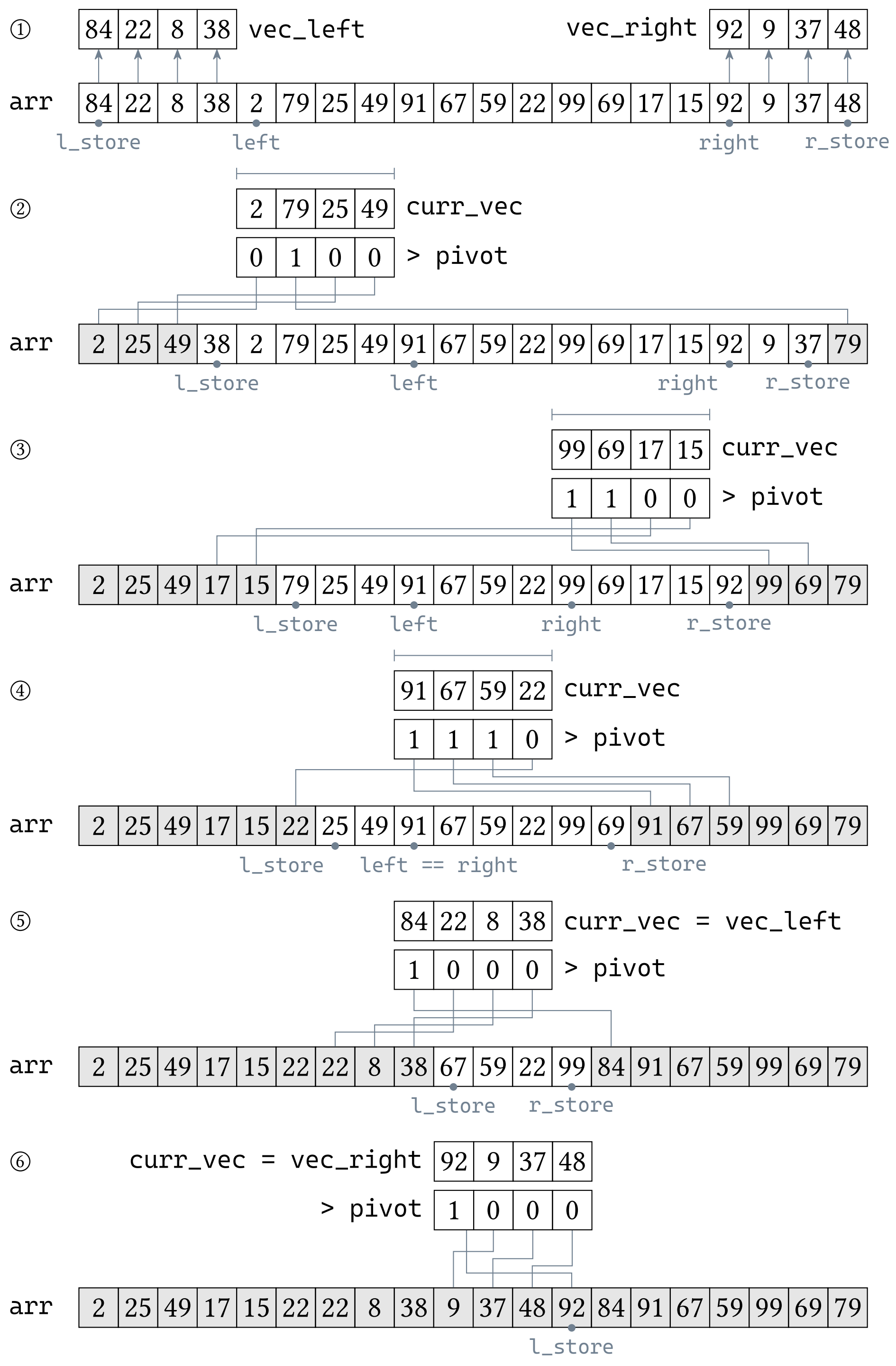
Actually, `partition_vec` does not partition a single vector but directly stores the elements at their partitioned location as described in section 2.3. It uses the bitmasks computed by performing

```rust
_mm512_cmpgt_epi32_mask(current_vec, pivot_vec)
```
Rust

and stores elements with corresponding mask entry 0 to the left and those with a corresponding mask entry 1 to the right.

[20]The problem of merging the 16 sorted columns of 32 elements each without transposition is dominated by many element-wise comparison elements, i.e. the same type of comparison elements used when sorting the columns. Hence, the same reasoning applies here.

Figure 5.6: Partitioning of an array arr using a pivot value of 49 and Blacher's approach [6, Abb. 6.1]. Already partitioned values are highlighted with gray background.



①
84 22 8 38 vec_left
vec_right 92 9 37 48
arr 84 22 8 38 2 79 25 49 91 67 59 22 99 69 17 15 92 9 37 48
l_store
left
right
r_store
②
2 79 25 49 curr_vec
0 1 0 0 > pivot
arr 2 25 49 38 2 79 25 49 91 67 59 22 99 69 17 15 92 9 37 79
l_store
left
right
r_store
③
99 69 17 15 curr_vec
1 1 0 0 > pivot
arr 2 25 49 17 15 79 25 49 91 67 59 22 99 69 17 15 92 99 69 79
l_store
left
right
r_store
④
91 67 59 22 curr_vec
1 1 1 0 > pivot
arr 2 25 49 17 15 22 25 49 91 67 59 22 99 69 91 67 59 99 69 79
l_store
left == right
r_store
⑤
84 22 8 38 curr_vec = vec_left
1 0 0 0 > pivot
arr 2 25 49 17 15 22 22 8 38 67 59 22 99 84 91 67 59 99 69 79
l_store
r_store
⑥
curr_vec = vec_right 92 9 37 48
> pivot 1 0 0 0
arr 2 25 49 17 15 22 22 8 38 9 37 48 92 84 91 67 59 99 69 79
l_store

Blacher's partitioning implementation is based on Bramas' approach to in-place partitioning [8]. It starts off by saving the outer-most elements in vectors ① and then moving towards the middle. While moving, elements are loaded in groups into `curr_vec` ②. `partition_vec` is applied to the current vector, which stores those elements less than the pivot contiguously at the left boundary and those greater than the pivot starting at the right boundary. The vector `curr_vec` is either taken from the left as in ② or from the right as in ③ if more elements have been partitioned on the left than on the right.

As soon as the last elements from the middle are processed in ④ the only elements left for partitioning are those which have been saved in the beginning. Their partitioning is carried out last (⑤ and ⑥). When finished, `l_store` marks the partition boundary pointing to the first elements greater than the pivot. Afterwards, the whole array `arr` is partitioned.

In contrast to Blacher, this whole process uses a simulated register width of 128 instead of 64 elements to iterate through the array. Simulating wide vector registers helps reducing the average latency, especially by pipelining high-latency instructions like load and store. Furthermore, performing multiple contiguous loads at once improves caching behavior.

An important detail of Blacher's partitioning implementation is the bookkeeping of the smallest and largest values. This is used later on to determine if the array has already been sorted and to ensure the deterministic nature of Quicksort. The implementation in listing A.4 on page 83 shows the partitioning of an array without simulated registers.

## 5.5 Assembling Quicksort's recursion

Relevant source file:
`avx-sort/src/avx512/mod.rs`

The previous sections discussed the components of the vectorized Quicksort implementation and how they differ from Blacher's AVX2 version. With these modifications in place, the following implementation for Quicksort's recursion may be assembled.

```rust
/// Recursion for quicksort
#[inline]
unsafe fn quicksort_core(
    arr: &mut [i32], mut choose_avg: bool, avg: i32
) {
    if arr.len() < 2049 {
        // sorting networks for small sub-slices
        let mut buffer = [_mm512_setzero_si512(); 65];
        let buff: &mut [i32] = &mut *(&mut buffer as *mut [__m512i]
                                                  as *mut [i32]);
        sort_int_sorting_network(arr, buff);
        return;
    }

    // avg is average of largest and smallest value in array
    let pivot = if choose_avg { avg } else { get_pivot(arr) };

    let PartitionResult {
        partition_boundary: bound,
        // smallest value after partitioning
        smallest,
        // largest value after partitioning
        biggest,
    } = partition_vectorized_128(arr, pivot);

    // ratio of smaller partition in array
    let ratio = (arr.len() - bound).min(bound) as f64 / arr.len() as f64;
    // if unbalanced sub-slices, change pivot strategy
    choose_avg ^= ratio < 0.2;

    if pivot ≠ smallest {
        // different values in left sub-slice
        quicksort_core(&mut arr[..bound], choose_avg,
                       average(smallest, pivot));
    }
    if pivot + 1 ≠ biggest {
        // different values in right sub-slice
        quicksort_core(&mut arr[bound..], choose_avg,
                       average(biggest, pivot));
    }
}
```

Following the previous discussion, there are two components that are only conditionally executed. The first one consists of the sorting networks discussed in section 5.3, which are used to sort small arrays. Their implementation is hidden behind the high-level function `sort_int_sorting_network` responsible for dividing the elements into pieces that may be sorted by the given sorting networks. This is a translation of the AVX2 version without major adjustments.

One important question in a hybrid algorithm is the choice of thresholds of when to switch from one algorithm to the other. In this case, the threshold of 2049 elements for the transition to sorting networks instead of Quicksort's recursion has been determined by experiments. Lower values like 1024 resulted in nearly twice the runtime, larger values like 3000 or 4000 elements did not improve the situation and even larger values resulted in a decrease of performance again.

The second component whose call is guarded is the pivot calculation as introduced in section 5.2. If the smaller of the two partitions after partitioning contains less than 20 % of the array's elements, the next step will choose the pivot element by using the average of the current pivot and the smallest or largest value respectively. This follows the pivot selection as described by Daoud, Abdel-jaber, and Ababneh [14].

Ibidem, it is described how this pivot selection influences the termination behavior on arrays with equal values. Due to the fact that the non-quadratic Quicksort uses pivot elements that may not be present in the array and based on that cannot position the pivot element to its sorted position in the array, the checks guarding the recursive calls are required to ensure termination on such arrays.

Based on this implementation of Quicksort, an implementation of Quickselect is achieved by replacing the sorting networks by a selection algorithm. In this case, Rust's standard library will be used.

```rust
/// Recursion for quickselect
#[inline]
unsafe fn quickselect_core(
    arr: &mut [i32], k: usize, mut choose_avg: bool, avg: i32
) {
    if arr.len() < 1025 {
        // use standard library for few elements
        arr.select_nth_unstable(k);
        return;
    }
    // … as in the Quicksort implementation above …
}
```

# 6 Experiments and benchmarks

## 6.1 Benchmarking methodology

All benchmarks have been performed on an Intel Core i9-10980XE processor with 18 cores. This processor clocks with a base frequency of 3 GHz and reaches 4.8 GHz in Intel's Turbo Boost mode [33]. It provides the AVX-512 vector instruction set extensions and two AVX-512 FMA units. 128 GB RAM have been available.

The host operating system has been Ubuntu 20.04 LTS (Linux kernel 5.4) compiling using the target triple `x86_64-unknown-linux-gnu`. As the implementation requires Rust's nightly toolchain, the Rust toolchain version 1.55.0-nightly (2021-07-02) has been used. More recent versions of Rust introduce breaking changes to some of the nightly features, thus using a specific version is important for nightly Rust.[21] The Rust compilation uses the `RUSTFLAGS` for native compilation (`-C target-cpu=native`).

A dedicated benchmarking application has been written to conduct the benchmarks in the following section. It creates arrays of a certain length using given generation functions/distributions. For each array length and distribution each sorting algorithm will be applied to the same array. Such an execution of the sorting algorithm includes multiple warmup runs followed by multiple benchmarking runs.

There is one exception to the aforementioned rule of running on the same array: the sorting algorithm IPS²Ra operates on unsigned instead of signed integers. To guarantee that it actually operates on the same distribution of numbers instead of on the same bits, this algorithm will be applied to a different array.

Getting exact measures is difficult. A very common measure for single-threaded program performance is IPC – instructions per cycle. The IPC count

[21]Using Rust's toolchain manager `rustup`, the required toolchain may be installed by executing `rustup toolchain install nightly-2021-07-02`.

benefits from pipelining, i.e. instructions with low throughput but possibly higher latency. Sometimes its inverse, CPI – cycles per instruction – is used. Alameldeen and Wood elaborate on common pitfalls of IPC measurement, focusing on multi-processor workloads but describing common architectural influences on IPC, like including idle loops, as well [2].

A different approach to measure similar aspects are CPE – cycles per element. In contrast to IPC this measure may be used to extrapolate performance on a certain input size. There is one major difference to IPC: as instructions are irrelevant for this measure, ignoring idle loops and similar situations where no actual work is done is not a flaw per se. CPE can be seen as a normalized wall time.

Wall time itself is a common measure for micro-benchmarking applications. The time elapsed for the user is commonly referred to as wall time. This is usually measured by reading the CPU's clock. Another approach when thinking about runtime is the concept of a task clock only counting the time the processor actually works on the task at hand. The latter has its caveats, most importantly being rather small for I/O-heavy applications doing a lot of waiting, which has a large influence on the evaluation of results.

To reduce outliers in performance measurements, benchmarks are usually executed multiple times, 25 times in all following benchmarks. Before performing the benchmark iterations, 5 warmup runs are executed on the same input as the following benchmarking iterations. Usually, only one warmup run is necessary but in this case, the warmup runs assert the correctness of the result and reproducibility. This saves time during the real benchmark, in which it is only validated that the result is sorted, ignoring possible alterations of the array's content.

The goal of the AVX-512 implementation of a vectorized Quicksort is the maximization of core utilization. Hence, the benchmarking application will export the following measures for each benchmark (100 benchmarking runs):

- median and mean of the execution's wall time,
- the maximum deviation from the median of the execution's wall time,
- the IPC and CPE count,
- the clock frequency during the run in GHz,
- the number of branch misses (BM),
- the number of cache misses (CM), and
- whether the algorithm actually sorted successfully.

The clock frequency is calculated using $^{\text{cycles}}/_{\text{task clock}}$ in this case. Hence, it allows to determine unreasonably high system activity because cycles would outnumber the task-bound clock ticks in that case. AVX-512 requires to consider the clock frequency as most CPUs reduce their clock frequency when executing an AVX-512-heavy program for thermal reasons as explained in section 2.3 on page 17.

Most of the aforementioned measurements are conducted using the Linux kernel's `perf_event_open` API. Hence, the benchmarking application will only run on systems running a Linux kernel.

## 6.2 Benchmarks against relevant sorting algorithms

To follow up on Blacher, Giesen, and Kühne [7], the main objective in this section will be to assess the speedup over the `std::sort` implementation in C++'s standard library. Figure 6.1 depicts this speedup for several algorithms and distributions. Absolute values for the respective benchmarking runs is available in tabular form in appendix A.3 on page 86.

The algorithms under test include:

- `cxxstd` and `rsstd`, the standard libraries of C++ and Rust, i.e. Introsort and pattern-defeating Quicksort as detailed in section 3.4. While the former is the baseline for the speedup calculations, the latter generally performs better and demonstrates a more robust handling of the tested distributions.
- `cxxavx` and `rsavx`, the C++ and Rust implementations of Blacher's AVX2 sort implementation [7]. Due to sharing many implementation details they perform very similar, with the C++ version outperforming the Rust version by a small margin. They will be treated as one sorting algorithm and only the C++ version will be considered in the following analysis.
- `ips4o` and `ips2ra`, the IPS$^4$o [4] and IPS$^2$Ra [3] algorithms as described by Axtmann et al. They implement a scalar Samplesort (cf. section 3.3) and Radix sort respectively, and perform very well on a wide variety of distributions.
- `avx512`, the AVX-512 sorting algorithm by Bramas [8], which defines the baseline for AVX-512-based sorting algorithms in this section because it is the only AVX-512-based sorting algorithm under consideration with public source code that can be used for evaluation.
- `rsavx512`, the vectorized Quicksort implementation as described in this thesis.

They have been tested on the following distributions:

- `asc`, ascending numbers from 0 to $n - 1$ where $n$ represents the array size. This is a special kind of sorted number distribution.
- `equ`, where all elements have been set to 1, another kind of sorted number distribution.
- `alm`, where $\lfloor \frac{1}{2} \cdot 2^{\lg n} \rfloor$ numbers are misplaced. This is generated by creating an array of ascending numbers and then providing local modifications that destroy the order.
- `pip`, a pipe-organ distribution where the first half of the array consists of ascending numbers and the second half consists of descending numbers.
- `gau`, a Gaussian random number distribution ($\mu = 0, \sigma = 100$) created by rounding random floating point numbers to the nearest integer.
- `uni`, a uniform random number distribution over the whole range of `i32`, i.e. $[-2^{31}, 2^{31} - 1]$, or `u32`, i.e. $[0, 2^{32} - 1]$, respectively.

Before turning to the results, it is to be mentioned that benchmarking all algorithms takes exceedingly long when Bramas' sorting algorithm `avx512` is taken into account on more than $10^6$ samples. Not only does it require increasing the stack size, most probably due to the recursion depth, but sometimes it even overflows the stack and is reproducibly at least one order of magnitude slower than the other sorting algorithms on large arrays and certain distributions. As it is the main competitor in the field of sorting on AVX-512,[22] the evaluation will concentrate on results up to $10^6$ elements.

Of the results in fig. 6.1, some confirm the expected outcome. `asc`, for instance, is handled well by most algorithms, including the C++ standard library. IPS²Ra and pattern-defeating Quicksort stand out for being significantly faster because of their distinctive features as Radix sort or pattern-defeating respectively. `avx512` performs bad on distributions with many duplicate keys like `equ` as highlighted by Blacher, Giesen, and Kühne [7].

The most interesting cases, however, are those with random elements, i.e. `gau` and `uni`, because real data usually does not adhere to specific distributions. In summary, the AVX-512 implementation performs very well early on with its peak performance at $10^4$ elements. The performance gap to the AVX2 implementations increases with more elements to the point that the AVX-512 implementation achieves only a stable third place on these distributions. Hence, the cases $10^4$ and $10^6$ elements have been picked for more detailed analysis.

[22]The code for [64, 65] is not freely available for reproduction and could not be included.

Figure 6.1: Speedup of the respective algorithms over cxxstd for the chosen distributions.

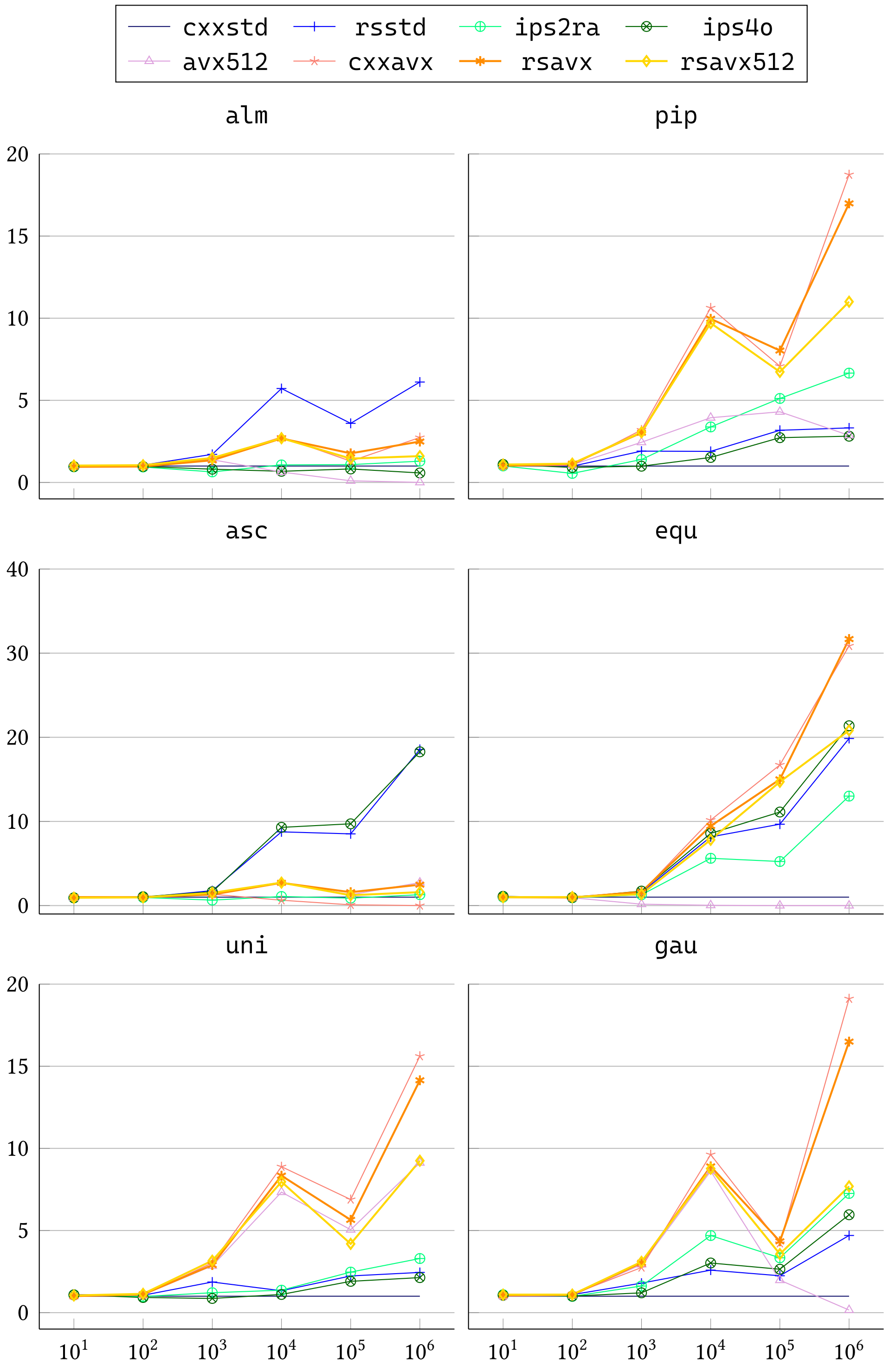


Table 6.1: Performance indicators from a run of all algorithms on uni with $10^4$ elements.

| Algorithm | Median [ns] | CPE | IPC | GHz | BM |
|---|---|---|---|---|---|
| rsavx512 | 117 292 | 12.34 | 2.22 | 1.16 | 97 |
| rsavx | 112 035 | 11.57 | 3.05 | 1.14 | 41 |
| cxxavx | 105 125 | 10.54 | 3.14 | 1.12 | 33 |
| rsstd | 700 135 | 87.74 | 1.78 | 1.27 | 21 120 |
| ips2ra | 683 166 | 85.33 | 1.19 | 1.27 | 12 727 |
| ips4o | 842 842 | 107.67 | 1.88 | 1.29 | 11 733 |
| avx512 | 127 280 | 13.63 | 1.66 | 1.17 | 617 |
| cxxstd | 934 240 | 182.59 | 0.63 | 1.97 | 58 631 |

In table 6.1 detailed performance indicators are presented for sorting arrays of $10^4$ elements. In this case, rsavx512 relies on sorting networks with only few layers of partitioning, as does cxxavx. As a consequence, the oblivious part of the hybrid sort dominates where the distribution is of less importance.

rsavx512 uses the second least cycles per elements, beating avx512 and staying close behind cxxavx. One reason for this is the effective use of whole vector registers in Blacher's approach, e.g. when merging two vectors. This kind of optimization is not exploited by Bramas.

The other measures are competitive as well. While rsavx512 is slightly slower than cxxavx in terms of wall time, it has only few branch misses and better IPC than all except the AVX2 implementations. It is interesting to note that rsstd is not explicitly vectorized but achieves a rather high IPC, comparable to ips4o, merely through compiler optimization and auto-vectorization.

The heavy use of AVX-512 instructions does not seem to cause significant throttling as the clock frequency is in the same range as for the AVX2-based sorting algorithms and slightly less than for those which are not explicitly vectorized. This is probably caused by license-based downclocking. However, due to the very small task of sorting $10^4$ elements, this may as well be statistic noise due to the measurement of the frequency only covering a small time frame.

Table 6.2 covers detailed results for $10^6$ elements. One of the first things to note there is the lower base frequency as well. Both rsavx512 and avx512 are clocking slower than the other algorithms, which is evidently caused by license-based downclocking. In section 6.3, it will be discussed how to compensate the impact of this lower frequency on the other measures.

Table 6.2: Performance indicators from a run of all algorithms on `uni` with $10^6$ elements.

| Algorithm | Median [ns] | CPE | IPC | GHz | BM | CM |
|---|---|---|---|---|---|---|
| `rsavx512` | 6 282 035 | 24.98 | 2.16 | 3.98 | 28 292 | 5 |
| `rsavx` | 4 107 068 | 18.76 | 2.97 | 4.57 | 42 546 | 21 |
| `cxxavx` | 3 721 175 | 17.00 | 3.04 | 4.58 | 50 781 | 8 |
| `rsstd` | 23 731 565 | 108.77 | 2.04 | 4.59 | 2 283 345 | 9 |
| `ips2ra` | 17 632 010 | 80.67 | 1.30 | 4.58 | 1 286 574 | 59 |
| `ips4o` | 27 228 984 | 124.49 | 2.12 | 4.57 | 1 189 386 | 62 |
| `avx512` | 6 367 628 | 25.30 | 1.37 | 3.98 | 241 660 | 32 |
| `cxxstd` | 58 099 660 | 265.69 | 0.61 | 4.57 | 8 625 367 | 12 |

For $10^6$ elements, it still applies that the IPC of `rsavx512` is between the AVX2 implementations and the other sorting algorithms, though `ips4o` is closing up. In terms of CPE and wall-time `rsavx512` is roughly on par with `avx512`. The only measures where `rsavx512` shines is the very low number of branch and cache misses.

The most probable reason for staying behind the AVX2-based sorting implementations is `rsavx512`'s inefficiency in partitioning. The AVX-512-based Quickselect performs far worse on all sizes of arrays suggesting that the partitioning function is particularly slow. Profiling the partitioning did not identify a single bottleneck, the distribution of runtime turns out to be very similar to the one of `rsavx`.

In conclusion, it can be stated that the pure translation of Blacher's approach to AVX-512 did not yield a superior sorting implementation with its bottleneck being the partitioning. On the other hand, that means that improving the partitioning will be of big influence on the overall performance and might turn `rsavx512` into an even more competitive algorithm. This is left for further research.

## 6.3 Assessing the implementation's efficiency

The last section concentrated on typical user-facing benchmarks, making wall time one of the deciding factors. On the other hand, many other performance indicators are important. To follow up Blacher, Giesen, and Kühne and their performance indicators [7], this section will concentrate on finding reasonable heuristics to assess how efficient the respective implementations are for sorting numbers.

Regardless of sorting on high-performance devices or low-end laptops, one important aspect of an optimal implementation is low power consumption. This becomes even more important with the advent of green computing as one of the design goals of high-performance algorithms.

NB: While many papers on green computing describe larger issues like processor power management or data center cooling influencing overall power consumption of whole data centers [22, 38], application-level efficiency contributes to the goal of reducing power consumption, especially when affecting common tasks like sorting. Guermouche and Orgerie discuss why AVX-512's influence on power consumption is beneficial [23].

Considering these aspects of efficiency, a first estimate of "optimal" could be the following:

- few cycles per element to finish quickly
- with high IPC to make sure cycles are used as efficiently as possible.

Let $\mathscr{P}_1$ describe this measure as in $\mathscr{P}_1 = \text{IPC}/\text{CPE}$ (cycles cancel out, instructions per element remain). This is particularly useful for comparisons because it will be in the range $(0, 1]$ for all useful samples. The aim is to maximize $\mathscr{P}_1$ for a given algorithm. Specific values based on the results from table 6.2 for $\mathscr{P}_1$ as well as the measures to follow are shown in table 6.3.

One deficiency of $\mathscr{P}_1$ is the non-consideration of clock ticks and with that the omission of the frequency. On the one hand, this is useful for the assessment of user-facing efficiency. From another perspective, the CPU's frequency is a hardware restriction, i.e. it might improve in future iterations of the architecture and should be corrected in the measures. Hence, let $\mathscr{P}_2 = \mathscr{P}_1/\text{GHz}$ to address this.

These are basic efficiency measures. Given the specific problem at hand, it might be useful to penalize certain properties of the implementation. One, admittedly controversial, example of this are branch mispredictions.[23] To avoid overweighting these when including them into an efficiency measure, only the order of magnitude is considered, i.e. $\mathscr{P}_3 = \mathscr{P}_2/\lg(\text{branch misses})$.

[23]Branch misses do not always indicate bad performance or optimization, but large numbers of branch misses are typically a sign of subpar branch predictability, which may harm performance stability across input distributions and platforms.

Table 6.3: Different calculations of implementations' efficiency applied to the performance indicators from a run of all algorithms on `uni` with $10^6$ elements (cf. table 6.2). Results for all measures except $\mathscr{P}_1$ are scaled to ease comparisons.

| Algorithm | $\mathscr{P}_1$ | $10 \cdot \mathscr{P}_2$ | $100 \cdot \mathscr{P}_3$ | $100 \cdot \mathscr{P}_4$ |
|---|---|---|---|---|
| `rsavx512` | 0.0865 | 0.2173 | 0.4880 | 0.0976 |
| `rsavx` | 0.1583 | 0.3464 | 0.7484 | 0.0356 |
| `cxxavx` | 0.1788 | 0.3904 | 0.8297 | 0.1037 |
| `rsstd` | 0.0188 | 0.0409 | 0.0643 | 0.0071 |
| `ips2ra` | 0.0161 | 0.0352 | 0.0576 | 0.0010 |
| `ips4o` | 0.0170 | 0.0373 | 0.0613 | 0.0010 |
| `avx512` | 0.0564 | 0.1417 | 0.2631 | 0.0082 |
| `cxxstd` | 0.0023 | 0.0050 | 0.0072 | 0.0006 |

In contrast to branch misses, cache misses are actively harming program performance, especially on large arrays that cannot be handled in cache. A large number of cache misses usually describes a poor access pattern to the main memory. Hence, let the last measure $\mathscr{P}_4 = \mathscr{P}_3/\text{cache misses}$.

As illustrated in table 6.2, the most efficient sorting algorithms after considering all these additional influences are the ones based on Blacher's approach. All other approaches are at least one order of magnitude less efficient with the most efficient being `rsstd`, i.e. pattern-defeating Quicksort.

On the other hand, without considering cache misses, Bramas' `avx512` algorithm actually performs similarly well to the ones based on Blacher's strategy. Apart from stack usage and recursion depth, cache misses are `avx512`'s weak points.

The branch misses themselves do not have a large influence on the evaluation. $\mathscr{P}_2$ and $\mathscr{P}_3$ show the same trends. The frequency correction which initiated this discussion shows the most significant impact as the $\mathscr{P}_2$ values for `avx512` and `rsavx512` show, contrasted to the $\mathscr{P}_1$ values.

# 7 Conclusion

Vector processing is ubiquitous in modern instruction sets. Apart from auto-vectorization, few implementations of sorting algorithms explicitly rely on modern AVX-512 instructions. This thesis ported Blacher's Quicksort for 32 bit integers from AVX2 to AVX-512 to provide an implementation to fill the gap.

On AVX-512, Blacher's design decisions for his unstable Quicksort without need for additional memory have proven to be very efficient on their own. Vectorized sorting networks efficiently sort small arrays by treating them as matrices and using a combination of element-wise vectorized comparison elements to sort columns and a transpositionless Bitonic merge of these sorted columns.

For larger arrays, Quicksort requires partitioning. As it turned out, partitioning on AVX-512 is more performance-sensitive. Balancing the cutoff between efficient sorting networks for small and partitioning-based Quicksort for large arrays has been one of the major challenges without satisfactory solution.

While being on par with Blacher's AVX2-based sort for small arrays, the weak partitioning performance lets the AVX-512 implementation stay behind his implementation on large arrays. The performance is still competitive with high-performance sorting algorithms like IPS$^2$Ra though, achieving a speedup of about 10 over `std::sort` and approx. 3 over the Radix sort IPS$^2$Ra and Samplesort IPS$^4$o on random numbers.

A Quickselect based on the same AVX-512 core implementation is not competitive, Blacher's AVX2-based Quickselect remains the better choice. As with Blacher's implementation, the Quicksort may be used to sort other 32 bit types than signed integers by applying pre- and post-processing.

Further work could explore more efficient sorting networks by reducing the number of comparison elements as well as the number of parallel steps in the appropriate places. In addition, a detailed analysis of the shortcomings in partitioning and corresponding improvements could lift the AVX-512 implementation to the top of the benchmarks.

# Bibliography

[1] M. Ajtai, J. Komlós, and E. Szemerédi. "An $\mathcal{O}(n \log n)$ Sorting Network." In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. STOC '83. New York, NY, USA: Association for Computing Machinery, 1983, pp. 1–9. ISBN: 0897910990. DOI: 10.1145/800061.808726.

[2] A. R. Alameldeen and D. A. Wood. "IPC Considered Harmful for Multiprocessor Workloads." In: *IEEE Micro* 26.4 (2006), pp. 8–17. DOI: 10.1109/MM.2006.73.

[3] Michael Axtmann et al. *Engineering In-place (Shared-memory) Sorting Algorithms*. Computing Research Repository (CoRR). Sept. 2020. arXiv: 2009.13569.

[4] Michael Axtmann et al. "In-Place Parallel Super Scalar Samplesort (IPSSSSo)." In: *25th Annual European Symposium on Algorithms (ESA 2017)*. Ed. by Kirk Pruhs and Christian Sohler. Vol. 87. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017, 9:1–9:14. ISBN: 978-3-95977-049-1. DOI: 10.4230/LIPIcs.ESA.2017.9. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7854.

[5] K. E. Batcher. "Sorting Networks and Their Applications." In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1968, pp. 307–314. ISBN: 9781450378970. DOI: 10.1145/1468075.1468121.

[6] Mark Blacher. "Entwurf und Implementierung vektorisierter Sortieralgorithmen." Master's thesis. Friedrich-Schiller-Universität Jena, Nov. 1, 2018. URL: https://ci.inf-i2.uni-jena.de/qa74rag/avx2_sort/-/blob/master/Masterarbeit_vektorisierte_Sortieralgorithmen.pdf (visited on 2021-05-04).

[7] Mark Blacher, Joachim Giesen, and Lars Kühne. "Fast and Robust Vectorized In-Place Sorting of Primitive Types." In: *19th International Symposium on Experimental Algorithms (SEA 2021)*. Ed. by David Coudert and Emanuele Natale. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl,

Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, 3:1–3:16. DOI: 10.4230/LIPIcs.SEA.2021.3.

[8] Bérenger Bramas. “A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake.” In: *International Journal of Advanced Computer Science and Applications* 8 (Nov. 2017). DOI: 10.14569/IJACSA.2017.081044.

[9] Randal E. Bryant and David R. O’Hallaron. *Computer Systems: A Programmer’s Perspective.* 2nd. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0136108040.

[10] Jatin Chhugani et al. “Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture.” In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1313–1324. ISSN: 2150-8097. DOI: 10.14778/1454159.1454171.

[11] Michael Codish et al. “Sorting networks: To the end and back again.” In: *Journal of Computer and System Sciences* 104 (2019). Language and Automata Theory and Applications - LATA 2015, pp. 184–201. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2016.04.004. URL: https://www.sciencedirect.com/science/article/pii/S0022000016300162.

[12] Peter Cordes. *Can AVX2-compiled program still use 32 registers of an AVX-512 capable CPU?* Oct. 20, 2020. URL: https://stackoverflow.com/a/48893734 (visited on 2021-05-03).

[13] Alex Crichton, Amanieu d’Antras, et al. *Implement AVX-512 intrinsics.* URL: https://github.com/rust-lang/stdarch/issues/310 (visited on 2021-04-24).

[14] Amjad M. Daoud, Hussein Abdel-jaber, and Jafar Ababneh. “Efficient Non-Quadratic Quick Sort (NQQuickSort).” In: *Digital Enterprise and Information Systems.* Ed. by Ezendu Ariwa and Eyas El-Qawasmeh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 667–675. ISBN: 978-3-642-22603-8.

[15] Travis Downs. *Gathering Intel on Intel AVX-512 Transitions.* Jan. 17, 2020. URL: https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html (visited on 2021-06-07).

[16] Travis Downs. *Ice Lake AVX-512 Downclocking.* Aug. 19, 2020. URL: https://travisdowns.github.io/blog/2020/08/19/icl-avx512-freq.html (visited on 2021-06-07).

[17] Travis Downs. *SIMD instructions lowering CPU frequency.* July 3, 2019. URL: https://stackoverflow.com/a/56861355 (visited on 2021-06-07).

[18] Michael J. Flynn. “Some Computer Organizations and Their Effectiveness.” In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 1557-9956. DOI: 10.1109/TC.1972.5009071.

[19] W. D. Frazer and A. C. McKellar. “Samplesort: A Sampling Approach to Minimal Storage Tree Sorting.” In: *J. ACM* 17.3 (July 1970), pp. 496–507. ISSN: 0004-5411. DOI: `10.1145/321592.321600`.

[20] Free Software Foundation. *__sort*. URL: `https://github.com/gcc-mirror/gcc/blob/d9375e490072d1aae73a93949aa158fcd2a27018/libstdc++-v3/include/bits/stl_algo.h#L1950` (visited on 2021-04-17).

[21] Matt Godbolt et al. *Compiler Explorer*. URL: `https://github.com/compiler-explorer/compiler-explorer` (visited on 2021-05-03).

[22] Gregory Goth. “Chipping Away at Greenhouse Gases.” In: *Commun. ACM* 54.2 (Feb. 2011), pp. 13–15. ISSN: 0001-0782. DOI: `10.1145/1897816.1897823`.

[23] Amina Guermouche and Anne-Cécile Orgerie. *Experimental analysis of vectorized instructions impact on energy and power consumption under thermal design power constraints*. June 2019. URL: `https://hal.archives-ouvertes.fr/hal-02167083v2/file/report.pdf` (visited on 2021-06-28).

[24] S. Gueron and V. Krasnov. “Fast Quicksort Implementation Using AVX Instructions.” In: *The Computer Journal* 59.1 (2016), pp. 83–90. DOI: `10.1093/comjnl/bxv063`.

[25] Tobias Haslop. “Minimal Depth Sorting Networks.” Bachelor’s thesis. University of Bremen, Mar. 10, 2020. URL: `https://www.szi.uni-bremen.de/wp-content/uploads/2020/03/thesis_compressed.pdf` (visited on 2021-05-04).

[26] Timothy Hayes et al. “VSR sort: A novel vectorised sorting algorithm & architecture extensions for future microprocessors.” In: *Proceedings of High-Performance Computer Architecture (HPCA)*. 2015, pp. 26–38. DOI: `10.1109/HPCA.2015.7056019`.

[27] George T. Heineman, Gary Pollice, and Stanley Selkow. *Algorithms in a Nutshell: A Practical Guide*. 2nd. O’Reilly Media, Inc., 2016. ISBN: 1491948922.

[28] C. A. R. Hoare. “Algorithm 63: Partition.” In: *Commun. ACM* 4.7 (July 1961), p. 321. ISSN: 0001-0782. DOI: `10.1145/366622.366642`.

[29] C. A. R. Hoare. “Algorithm 64: Quicksort.” In: *Commun. ACM* 4.7 (July 1961), p. 321. ISSN: 0001-0782. DOI: `10.1145/366622.366644`.

[30] C. A. R. Hoare. “Algorithm 65: Find.” In: *Commun. ACM* 4.7 (July 1961), pp. 321–322. ISSN: 0001-0782. DOI: `10.1145/366622.366647`.

[31] C. A. R. Hoare. “Quicksort.” In: *The Computer Journal* 5.1 (Jan. 1962), pp. 10–16. ISSN: 0010-4620. DOI: `10.1093/comjnl/5.1.10`.

[32] Hiroshi Inoue et al. “AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors.” In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. PACT ’07. USA: IEEE Computer Society, 2007, pp. 189–198. ISBN: 0769529445.

[33] Intel Corporation. *Intel® Core™ i9-10980XE Extreme Edition Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/198017/intel-core-i9-10980xe-extreme-edition-processor-24-75m-cache-3-00-ghz.html (visited on 2021-05-06).

[34] Intel Corporation. *Intel® Intrinsics Guide*. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (visited on 2021-04-28).

[35] Ralf Jung et al. “Safe Systems Programming in Rust.” In: *Commun. ACM* 64.4 (Mar. 2021), pp. 144–152. ISSN: 0001-0782. DOI: 10.1145/3418295.

[36] Evgueny Khartchenko. *Vectorization: A Key Tool To Improve Performance On Modern CPUs*. Jan. 25, 2018. URL: https://software.intel.com/content/www/us/en/develop/articles/vectorization-a-key-tool-to-improve-performance-on-modern-cpus.html (visited on 2021-07-01).

[37] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201896850.

[38] Patrick Kurp. “Green Computing.” In: *Commun. ACM* 51.10 (Oct. 2008), pp. 11–13. ISSN: 0001-0782. DOI: 10.1145/1400181.1400186.

[39] Peter Lammich. “Efficient Verified Implementation of Introsort and Pdqsort.” In: *Automated Reasoning*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Cham: Springer International Publishing, 2020, pp. 307–323. ISBN: 978-3-030-51054-1.

[40] Daniel Lemire. *AVX-512 throttling: heavy instructions are maybe not so dangerous*. URL: https://lemire.me/blog/2018/08/25/avx-512-throttling-heavy-instructions-are-maybe-not-so-dangerous/ (visited on 2021-06-07).

[41] Daniel Lemire. *AVX-512: when and how to use these new instructions*. URL: https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/ (visited on 2021-06-07).

[42] Stewart A. Levin. “A fully vectorized quicksort.” In: *Parallel Computing* 16.2 (1990), pp. 369–373. ISSN: 0167-8191. DOI: 10.1016/0167-8191(90)90074-J. URL: https://www.sciencedirect.com/science/article/pii/016781919090074J.

[43] LLVM Project. *Auto-Vectorization in LLVM*. URL: https://llvm.org/docs/Vectorizers.html (visited on 2021-05-01).

[44] Chris Lomont. *Introduction to x64 Assembly*. Mar. 19, 2012. URL: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-x64-assembly.html (visited on 2021-05-03).

[45] Donald A. MacKenzie. *Knowing Machines. Essays on Technical Change*. MIT Press Cambridge, Mass, 1996. 338 pp. ISBN: 0262133156.

[46] David R. Musser. "Introspective Sorting and Selection Algorithms." In: *Software: Practice and Experience* 27.8 (1997), pp. 983–993. DOI: https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#.

[47] Orson Peters. *pdqsort*. URL: https://github.com/orlp/pdqsort/blob/b1ef26a55cdb60d236a5cb199c4234c704f46726/readme.md (visited on 2021-04-17).

[48] Orestis Polychroniou and Kenneth A. Ross. "A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison- and Radix-Sort." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 755–766. ISBN: 9781450323765. DOI: 10.1145/2588555.2610522.

[49] James Reinders. *Additional Intel® AVX-512 instructions*. July 18, 2014. URL: https://software.intel.com/content/www/us/en/develop/articles/additional-intel-avx-512-instructions.html (visited on 2021-05-04).

[50] James Reinders. *Intel® AVX-512 Instructions*. June 20, 2017. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html (visited on 2021-04-27).

[51] Rust Project. *_mm512_min_epi32*. URL: https://github.com/rust-lang/stdarch/blob/bc5c33cce1d2ff8f39cff2514be1a4ddd90d6d7d/crates/core_arch/src/x86/avx512f.rs#L2239-2247 (visited on 2021-05-01).

[52] Rust Project. *sort_unstable*. URL: https://github.com/rust-lang/rust/blob/57e28ef86fdf528d1e348312f5b2775d9de2cbd0/library/core/src/slice/mod.rs#L2277 (visited on 2021-04-17).

[53] Rust Project. *vpminsd*. URL: https://github.com/rust-lang/stdarch/blob/bc5c33cce1d2ff8f39cff2514be1a4ddd90d6d7d/crates/core_arch/src/x86/avx512f.rs#L35920-35921 (visited on 2021-05-01).

[54] Nadathur Satish et al. “Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort.” In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 351–362. ISBN: 9781450300322. DOI: 10.1145/1807167.1807207.

[55] Robert Schöne et al. “Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance.” In: *2019 International Conference on High Performance Computing Simulation (HPCS)*. July 2019, pp. 399–406. DOI: 10.1109/HPCS48598.2019.9188239.

[56] Robert Sedgewick. *Quicksort*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1975. ISBN: 0-8240-4417-7.

[57] Lukáš Sekanina. “Evolutionary Design Space Exploration for Median Circuits.” In: *Applications of Evolutionary Computing*. Ed. by Günther R. Raidl et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 240–249. ISBN: 978-3-540-24653-4.

[58] Howard Jay Siegel. “The Universality of Various Types of SIMD Machine Interconnection Networks.” In: *SIGARCH Comput. Archit. News* 5.7 (Mar. 1977), pp. 70–79. ISSN: 0163-5964. DOI: 10.1145/633615.810655.

[59] H. S. Stone. “Sorting on STAR.” In: *IEEE Trans. Softw. Eng.* 4.2 (Mar. 1978), pp. 138–146. ISSN: 0098-5589. DOI: 10.1109/TSE.1978.231484.

[60] User: InstLatx64. *The* 19 + 1 *levels of AVX512 in Intel processors according to the 37th Intel ISA-ER*. Aug. 30, 2019. URL: https://github.com/InstLatx64/InstLatx64/blob/5bcd12f2301f3095f2e050fd9e2ada2b7c838786/VennDiagrams/Venn_AVX512_v141.png (visited on 2021-05-04).

[61] User: njuffa. *Standard sorting networks for small values of n*. Aug. 28, 2018. URL: https://stackoverflow.com/a/46854885 (visited on 2021-07-07).

[62] User: Z boson. *How to transpose a 16x16 matrix using SIMD instructions?* May 23, 2017. URL: https://stackoverflow.com/a/41262731 (visited on 2021-06-17).

[63] Berthold Vöcking et al., eds. *Taschenbuch der Algorithmen*. eXamen.press. Springer, 2008. ISBN: 978-3-540-76393-2.

[64] Alex Watkins. “A Fast and Simple Approach to Merge Sorting using AVX-512.” Undergraduate Research Thesis. Georgia Institute of Technology, 2017. URL: https://smartech.gatech.edu/bitstream/handle/1853/61359/WATKINS-UNDERGRADUATERESEARCHOPTIONTHESIS-2018.pdf (visited on 2021-07-02).

[65] Z. Yin et al. "Efficient Parallel Sort on AVX-512-Based Multi-Core and Many-Core Architectures." In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2019, pp. 168–176. DOI: `10.1109/HPCC/SmartCity/DSS.2019.00038`.

# List of Tables

# List of Figures

# Appendix

## A.1 Using the FFI bindings from C++

Listing A.1: `CMakeLists.txt` for using the Rust library from C++

CMake

```cmake
# adapted from the Rust FFI book by Michael F. Bryan
# see https://github.com/Michael-F-Bryan/rust-ffi-guide/blo
↪ b/bb1d0cd108d929d0a551e5a85f4d2198d9a057b8/CMakeLists.txt
# (originally licensed CC0 1.0 Universal;
#  these adaptions follow the crate's license)

cmake_minimum_required(VERSION 3.7)
project(sorting-example)

if (CMAKE_BUILD_TYPE STREQUAL "Debug")
  set(CARGO_RELEASE_FLAG "" CACHE INTERNAL "")
  set(TARGET_DIR "debug" CACHE INTERNAL "")
else ()
  set(CARGO_RELEASE_FLAG "--release" CACHE INTERNAL "")
  set(TARGET_DIR "release" CACHE INTERNAL "")
endif ()

# adjust this variable to change the path to the Rust source
# of the avx-sort crate
set(project_dir ${CMAKE_CURRENT_SOURCE_DIR}/..)

# the following values are derived from the project_dir and
# the structure of the avx-sort crate
set(cargo_library_output
↪ ${CMAKE_CURRENT_BINARY_DIR}/${TARGET_DIR}/libavx_sort.so)
set(cargo_header_output_directory ${CMAKE_CURRENT_BINARY_DIR})
file(GLOB sources ${project_dir}/src/**/*.rs)

```

```
set(compile_message "Compiling ${target_name}")
if(CARGO_RELEASE_FLAG STREQUAL "--release")
  set(compile_message "${compile_message} in release mode")
endif()

add_custom_target(avx_sort ALL
  COMMENT ${compile_message}
  # due to the `CARGO_TARGET_DIR` setting, cargo will create its
  # debug/release folder in the `CMAKE_CURRENT_BINARY_DIR`;
  # due to the setup of avx_sort, the corresponding header with
  # the C bindings will be generated there as well
  COMMAND env CARGO_TARGET_DIR=${CMAKE_CURRENT_BINARY_DIR} cargo build
    ↪ ${CARGO_RELEASE_FLAG}
  WORKING_DIRECTORY ${project_dir})

# although not strictly needed, add a test target for the avx-sort
# crate; this does not test any C++ code from main.cpp
add_test(NAME avx_sort_test
  COMMAND env CARGO_TARGET_DIR=${CMAKE_CURRENT_BINARY_DIR} cargo test
    ↪ ${CARGO_RELEASE_FLAG}
  WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR})

# add the actual executable, linked against the compiled library
# (`avx_sort` target) and including the generated glue code header
# so that the `main.cpp` file is actually able to
# `#include "avx_sort.hpp"`
add_executable(sorting_example main.cpp)
add_dependencies(sorting_example avx_sort)
target_include_directories(sorting_example PRIVATE
  ↪ ${cargo_header_output_directory})
target_link_libraries(sorting_example ${cargo_library_output})
```

Listing A.2: C++ program using the FFI

C++

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

#include "avx_sort.hpp"

int main() {
  std::vector<int32_t> to_sort{8, 7, 6, 5, 4, 3, 2, 1};
  avx_sort::arch_optimal_quicksort_c(to_sort.data(), to_sort.size());

  int return_value = 0;

  std::cout << "The given vector has ";
  if (!std::is_sorted(to_sort.begin(), to_sort.end())) {
    std::cout << "not ";
    return_value = 1;
  }
  std::cout << "been sorted." << std::endl;

  return return_value;
}
```

## A.2 Supplementary code examples

Listing A.3: Calculate new pivot element

Rust

```rust
/// Determine a pivot element for `arr`; not necessarily part
/// of the collection.
#[inline]
unsafe fn get_pivot(arr: &[i32]) -> i32 {
    let bound = _mm512_set1_epi32(arr.len() as i32);

    // seeds for vectorized RNG
    let mut s0 = _mm512_setr_epi64(
        3_443_930_009_025_137_051,
        4_731_979_008_551_603_254,
        8_794_064_498_923_322_450,
        4_874_270_468_522_057_061,
        5_897_544_235_680_999_164,
        8_266_299_549_366_754_966,
        3_541_782_707_195_610_314,
        2_832_473_711_643_591_904,
    );
    let mut s1 = _mm512_setr_epi64(
        3_645_099_076_540_354_483,
        5_458_105_273_946_890_124,
            7_010_914_221_407_653_385,
            2_651_990_308_724_075_580,
            4_232_083_816_489_385_122,
            6_295_913_260_703_919_690,
            7_795_133_987_285_071_557,
            4_008_619_371_956_439_200,
    );
    s1 = _mm512_sub_epi64(s1, _mm512_set1_epi64((arr.len() - 1) as i64));

    // sparse array for median of medians
    let mut v: [__m512i; 5] = collect_into_array_unchecked(
        // fill sparse array
        &mut (0..5).map(|_| {
            // get a vector of 4 random uint64_t
            let mut result = vnext(&mut s0, &mut s1);
            // ZZ between 0 and bound - 1
            result = rnd_epu32(result, &bound);
            // indices for arr
            _mm512_i32gather_epi32(
                result,
                arr.get_unchecked(0) as *const i32 as *const u8,
                size_of::<i32>() as i32,
            )
        }),
    );
    let v = &mut v as &mut CoExI32x16;

    // use a median network for 5 vectors
    v.median_of_five();

    // sort the 16 medians in the middle
    sort_16(&mut v[2]);

    // get the pivot element;
    // AVX-512 does not provide a way to efficiently get an element
    // from a vector so this is achieved by viewing the memory as
    // contiguous i32s
    let int_view = &*(v as *const [__m512i] as *const [i32]);
    // the element v[2] we sorted previously is at 16 * 2 = 32;
    // now we want to get integers 7 and 8 from it (those around
```

```rust
    // the middle to compute the median of medians)
    let elem_7 = *int_view.get_unchecked(39);
    let elem_8 = *int_view.get_unchecked(40);
    average(elem_7, elem_8)
}
```

Listing A.4: Partitioning multiple vectors

Rust

```rust
/// Partition array using AVX512 vectors.
#[inline]
unsafe fn partition_vectorized(arr: &mut [i32], pivot: i32) ->
↪  PartitionResult<i32> {
    // shorten array to multiple of 16
    let (smallest, biggest, mut left, mut right) =
↪  shorten_array_mod16(arr, pivot);

    if left == right {
        // less than 16 elements in array
        return PartitionResult {
            partition_boundary: left,
            smallest,
            biggest,
        };
    }

    // vectors for pivot, smallest and biggest
    let pivot_vec = _mm512_set1_epi32(pivot);
    let mut sv = _mm512_set1_epi32(smallest);
    let mut bv = _mm512_set1_epi32(biggest);

    if right - left == 16 {
        let v = load_vec(arr.get_unchecked(left..left + 16));
        let amount_gt_pivot = partition_vec(
            v,
            &pivot_vec,
            &mut sv,
            &mut bv,
            arr.as_mut_ptr().add(left),
            arr.as_mut_ptr().add(left + 16),
        );
        return PartitionResult {
            partition_boundary: left + 16 - amount_gt_pivot,
            smallest: _mm512_reduce_min_epi32(sv),
            biggest: _mm512_reduce_max_epi32(bv),
        };
    }

    // the first and last 16 values are partitioned last
    let vec_left = load_vec(arr.get_unchecked(left..left + 16));
    let vec_right = load_vec(arr.get_unchecked(right - 16..right));

    // positions where the vectors are stored
    // right position to save to
    let mut r_store = right - 16;
    // left position to save to
    let mut l_store = left;

    // positions to load the vectors to
    // increase and decrease because first and last 8 elements are
    // buffered (see above)
    left += 16;
    right -= 16;

    // parition 16 elements per iteration
    while right - left != 0 {
        // if less elements are stored on the right side then use
        // the next vector from the right, else from left
        let curr_vec = if (r_store + 16) - right < left - l_store {
            right -= 16;
            load_vec(arr.get_unchecked(right..right + 16))
        } else {
            left += 16;
            load_vec(arr.get_unchecked(left - 16..left))
        };

        // partition current vector and store on both sides
        let amount_gt_pivot = partition_vec(
            curr_vec,
            &pivot_vec,
            &mut sv,
            &mut bv,
                arr.as_mut_ptr().add(l_store),
                arr.as_mut_ptr().add(r_store + 16),
            );

            // update positions where vectors are saved to,
            // ensuring that left and right positions are
            // updated by 16 in total
            r_store -= amount_gt_pivot;
            l_store += 16 - amount_gt_pivot;
        }

        // partition left vector and store it
        let amount_gt_pivot = partition_vec(
            vec_left,
            &pivot_vec,
            &mut sv,
            &mut bv,
            arr.as_mut_ptr().add(l_store),
            arr.as_mut_ptr().add(r_store + 16),
        );
        l_store += 16 - amount_gt_pivot;

        // partition right vector and store it
        let amount_gt_pivot = partition_vec(
            vec_right,
            &pivot_vec,
            &mut sv,
            &mut bv,
            arr.as_mut_ptr().add(l_store),
            arr.as_mut_ptr().add(l_store + 16),
        );
        l_store += 16 - amount_gt_pivot;

        // store smallest and largest value in vector
        PartitionResult {
            partition_boundary: l_store,
            smallest: _mm512_reduce_min_epi32(sv),
            biggest: _mm512_reduce_max_epi32(bv),
        }
    }
```

## A.3 Raw benchmarking results

Apart from results for the distributions introduced in section 6.2 on page 57, the subsequent table includes the following additional distributions on arrays of size $n$: `dsc`, descending numbers from $n-1$ to 0, `u16`, the respective remainders of the ascending numbers from 0 to $n-1$ modulo 16, shuffled randomly, `pft`, ascending numbers from 1 to $n-1$ followed by 0, and `pmi`, ascending numbers from 0 to $n-1$ without $\lfloor \frac{n}{2} \rfloor$, which is appended as the last element.

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| equ | rsavx512 | $10^1$ | $2.49\times10^4$ | 48.30 | 0.50 | 0.04 |
| | rsavx | $10^1$ | $2.47\times10^4$ | 36.20 | 0.60 | 0.03 |
| | cxxavx | $10^1$ | $2.50\times10^4$ | 28.20 | 1.10 | 0.02 |
| | rsstd | $10^1$ | $2.34\times10^4$ | 33.10 | 0.41 | 0.03 |
| | ips2ra | $10^1$ | $2.48\times10^4$ | 17.50 | 1.31 | 0.01 |
| | ips4o | $10^1$ | $2.24\times10^4$ | 12.90 | 1.26 | 0.01 |
| | bramas | $10^1$ | $2.24\times10^4$ | 26.40 | 0.73 | 0.02 |
| | cxxstd | $10^1$ | $2.47\times10^4$ | 18.30 | 1.24 | 0.01 |
| asc | rsavx512 | $10^1$ | $2.43\times10^4$ | 70.50 | 0.34 | 0.05 |
| | rsavx | $10^1$ | $2.24\times10^4$ | 35.60 | 0.61 | 0.03 |
| | cxxavx | $10^1$ | $2.49\times10^4$ | 28.30 | 1.09 | 0.02 |
| | rsstd | $10^1$ | $2.48\times10^4$ | 12.10 | 1.12 | 0.01 |
| | ips2ra | $10^1$ | $2.47\times10^4$ | 17.50 | 1.31 | 0.01 |
| | ips4o | $10^1$ | $2.45\times10^4$ | 13.00 | 1.25 | 0.01 |
| | bramas | $10^1$ | $2.25\times10^4$ | 26.10 | 0.74 | 0.02 |
| | cxxstd | $10^1$ | $2.26\times10^4$ | 18.90 | 1.20 | 0.02 |
| dsc | rsavx512 | $10^1$ | $2.50\times10^4$ | 51.40 | 0.47 | 0.04 |
| | rsavx | $10^1$ | $2.39\times10^4$ | 56.20 | 0.38 | 0.04 |
| | cxxavx | $10^1$ | $2.27\times10^4$ | 28.20 | 1.10 | 0.02 |
| | rsstd | $10^1$ | $2.33\times10^4$ | 43.00 | 1.17 | 0.03 |
| | ips2ra | $10^1$ | $2.42\times10^4$ | 62.90 | 0.78 | 0.05 |
| | ips4o | $10^1$ | $2.23\times10^4$ | 14.40 | 1.38 | 0.01 |
| | bramas | $10^1$ | $2.36\times10^4$ | 47.90 | 0.41 | 0.04 |
| | cxxstd | $10^1$ | $2.41\times10^4$ | 31.60 | 1.26 | 0.02 |
| uni | rsavx512 | $10^1$ | $2.36\times10^4$ | 65.30 | 0.37 | 0.05 |
| | rsavx | $10^1$ | $2.46\times10^4$ | 28.00 | 0.77 | 0.02 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
|  | cxxavx | $10^1$ | $2.43\times10^4$ | 45.80 | 0.67 | 0.03 |
|  | rsstd | $10^1$ | $2.26\times10^4$ | 16.90 | 1.89 | 0.01 |
|  | ips2ra | $10^1$ | $2.30\times10^4$ | 29.90 | 1.62 | 0.02 |
|  | ips4o | $10^1$ | $2.28\times10^4$ | 31.30 | 1.32 | 0.03 |
|  | bramas | $10^1$ | $2.48\times10^4$ | 23.80 | 0.82 | 0.02 |
|  | cxxstd | $10^1$ | $2.47\times10^4$ | 23.80 | 1.52 | 0.02 |
| u16 | rsavx512 | $10^1$ | $2.35\times10^4$ | 63.20 | 0.38 | 0.05 |
|  | rsavx | $10^1$ | $2.40\times10^4$ | 48.90 | 0.44 | 0.04 |
|  | cxxavx | $10^1$ | $2.27\times10^4$ | 25.30 | 1.22 | 0.02 |
|  | rsstd | $10^1$ | $2.23\times10^4$ | 16.70 | 1.77 | 0.01 |
|  | ips2ra | $10^1$ | $2.49\times10^4$ | 26.80 | 1.57 | 0.02 |
|  | ips4o | $10^1$ | $2.42\times10^4$ | 50.50 | 0.88 | 0.04 |
|  | bramas | $10^1$ | $2.23\times10^4$ | 22.90 | 0.85 | 0.02 |
|  | cxxstd | $10^1$ | $2.40\times10^4$ | 42.50 | 0.73 | 0.03 |
| pip | rsavx512 | $10^1$ | $2.27\times10^4$ | 45.00 | 0.54 | 0.04 |
|  | rsavx | $10^1$ | $2.45\times10^4$ | 28.10 | 0.77 | 0.02 |
|  | cxxavx | $10^1$ | $2.48\times10^4$ | 24.80 | 1.25 | 0.02 |
|  | rsstd | $10^1$ | $2.30\times10^4$ | 15.30 | 1.84 | 0.01 |
|  | ips2ra | $10^1$ | $2.46\times10^4$ | 20.90 | 1.69 | 0.02 |
|  | ips4o | $10^1$ | $2.25\times10^4$ | 24.70 | 1.65 | 0.02 |
|  | bramas | $10^1$ | $2.26\times10^4$ | 22.90 | 0.85 | 0.02 |
|  | cxxstd | $10^1$ | $2.47\times10^4$ | 22.60 | 1.45 | 0.02 |
| gau | rsavx512 | $10^1$ | $2.26\times10^4$ | 44.40 | 0.55 | 0.04 |
|  | rsavx | $10^1$ | $2.34\times10^4$ | 48.60 | 0.44 | 0.04 |
|  | cxxavx | $10^1$ | $2.46\times10^4$ | 24.70 | 1.25 | 0.02 |
|  | rsstd | $10^1$ | $2.43\times10^4$ | 36.70 | 0.72 | 0.03 |
|  | ips2ra | $10^1$ | $2.27\times10^4$ | 25.50 | 1.67 | 0.02 |
|  | ips4o | $10^1$ | $2.34\times10^4$ | 49.00 | 0.90 | 0.04 |
|  | bramas | $10^1$ | $2.48\times10^4$ | 22.90 | 0.85 | 0.02 |
|  | cxxstd | $10^1$ | $2.48\times10^4$ | 22.20 | 1.53 | 0.02 |
| alm | rsavx512 | $10^1$ | $2.27\times10^4$ | 43.20 | 0.56 | 0.04 |
|  | rsavx | $10^1$ | $2.46\times10^4$ | 27.60 | 0.78 | 0.02 |
|  | cxxavx | $10^1$ | $2.26\times10^4$ | 25.00 | 1.24 | 0.02 |
|  | rsstd | $10^1$ | $2.41\times10^4$ | 32.90 | 0.44 | 0.03 |
|  | ips2ra | $10^1$ | $2.46\times10^4$ | 17.70 | 1.36 | 0.01 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| | ips4o | $10^1$ | $2.41\times10^4$ | 40.60 | 0.69 | 0.03 |
| | bramas | $10^1$ | $2.24\times10^4$ | 22.90 | 0.85 | 0.02 |
| | cxxstd | $10^1$ | $2.33\times10^4$ | 39.60 | 0.59 | 0.03 |
| pfr | rsavx512 | $10^1$ | $2.51\times10^4$ | 44.90 | 0.54 | 0.03 |
| | rsavx | $10^1$ | $2.27\times10^4$ | 27.80 | 0.78 | 0.02 |
| | cxxavx | $10^1$ | $2.27\times10^4$ | 24.90 | 1.24 | 0.02 |
| | rsstd | $10^1$ | $2.47\times10^4$ | 14.10 | 1.48 | 0.01 |
| | ips2ra | $10^1$ | $2.46\times10^4$ | 26.30 | 1.06 | 0.02 |
| | ips4o | $10^1$ | $2.23\times10^4$ | 22.10 | 1.34 | 0.02 |
| | bramas | $10^1$ | $2.27\times10^4$ | 23.80 | 0.82 | 0.02 |
| | cxxstd | $10^1$ | $2.47\times10^4$ | 23.40 | 1.05 | 0.02 |
| pmi | rsavx512 | $10^1$ | $2.44\times10^4$ | 64.30 | 0.38 | 0.05 |
| | rsavx | $10^1$ | $2.27\times10^4$ | 28.50 | 0.76 | 0.02 |
| | cxxavx | $10^1$ | $2.34\times10^4$ | 45.70 | 0.68 | 0.04 |
| | rsstd | $10^1$ | $2.46\times10^4$ | 13.00 | 1.33 | 0.01 |
| | ips2ra | $10^1$ | $2.40\times10^4$ | 38.80 | 0.67 | 0.03 |
| | ips4o | $10^1$ | $2.26\times10^4$ | 21.80 | 1.54 | 0.02 |
| | bramas | $10^1$ | $2.32\times10^4$ | 43.70 | 0.44 | 0.03 |
| | cxxstd | $10^1$ | $2.48\times10^4$ | 19.30 | 1.31 | 0.01 |
| equ | rsavx512 | $10^2$ | $2.33\times10^4$ | 9.89 | 1.27 | 0.08 |
| | rsavx | $10^2$ | $2.48\times10^4$ | 9.15 | 2.35 | 0.07 |
| | cxxavx | $10^2$ | $2.49\times10^4$ | 10.23 | 2.02 | 0.07 |
| | rsstd | $10^2$ | $2.26\times10^4$ | 3.76 | 2.38 | 0.03 |
| | ips2ra | $10^2$ | $2.55\times10^4$ | 13.61 | 1.81 | 0.10 |
| | ips4o | $10^2$ | $2.47\times10^4$ | 3.19 | 2.49 | 0.02 |
| | bramas | $10^2$ | $2.54\times10^4$ | 7.85 | 1.26 | 0.06 |
| | cxxstd | $10^2$ | $2.37\times10^4$ | 16.49 | 2.33 | 0.12 |
| asc | rsavx512 | $10^2$ | $2.53\times10^4$ | 9.54 | 1.32 | 0.07 |
| | rsavx | $10^2$ | $2.46\times10^4$ | 10.98 | 1.96 | 0.08 |
| | cxxavx | $10^2$ | $2.38\times10^4$ | 8.32 | 2.48 | 0.06 |
| | rsstd | $10^2$ | $2.49\times10^4$ | 3.71 | 2.41 | 0.03 |
| | ips2ra | $10^2$ | $2.56\times10^4$ | 21.11 | 1.88 | 0.14 |
| | ips4o | $10^2$ | $2.34\times10^4$ | 4.88 | 1.63 | 0.04 |
| | bramas | $10^2$ | $2.44\times10^4$ | 9.89 | 1.00 | 0.07 |
| | cxxstd | $10^2$ | $2.45\times10^4$ | 20.07 | 2.27 | 0.14 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| dsc | rsavx512 | $10^2$ | $2.52\times10^4$ | 9.48 | 1.33 | 0.07 |
| | rsavx | $10^2$ | $2.53\times10^4$ | 9.05 | 2.37 | 0.07 |
| | cxxavx | $10^2$ | $2.30\times10^4$ | 8.20 | 2.52 | 0.06 |
| | rsstd | $10^2$ | $2.29\times10^4$ | 4.61 | 2.64 | 0.04 |
| | ips2ra | $10^2$ | $2.65\times10^4$ | 20.89 | 2.24 | 0.14 |
| | ips4o | $10^2$ | $2.30\times10^4$ | 4.75 | 2.31 | 0.04 |
| | bramas | $10^2$ | $2.52\times10^4$ | 7.80 | 1.27 | 0.06 |
| | cxxstd | $10^2$ | $2.62\times10^4$ | 16.96 | 2.39 | 0.12 |
| uni | rsavx512 | $10^2$ | $2.33\times10^4$ | 9.12 | 1.38 | 0.07 |
| | rsavx | $10^2$ | $2.39\times10^4$ | 11.06 | 1.94 | 0.09 |
| | cxxavx | $10^2$ | $2.52\times10^4$ | 8.25 | 2.50 | 0.06 |
| | rsstd | $10^2$ | $2.52\times10^4$ | 25.55 | 3.47 | 0.16 |
| | ips2ra | $10^2$ | $2.72\times10^4$ | 43.39 | 2.06 | 0.26 |
| | ips4o | $10^2$ | $2.89\times10^4$ | 52.68 | 3.64 | 0.30 |
| | bramas | $10^2$ | $2.47\times10^4$ | 9.89 | 1.00 | 0.07 |
| | cxxstd | $10^2$ | $2.67\times10^4$ | 23.24 | 2.55 | 0.15 |
| u16 | rsavx512 | $10^2$ | $2.38\times10^4$ | 9.87 | 1.27 | 0.07 |
| | rsavx | $10^2$ | $2.46\times10^4$ | 9.08 | 2.37 | 0.07 |
| | cxxavx | $10^2$ | $2.37\times10^4$ | 8.36 | 2.47 | 0.06 |
| | rsstd | $10^2$ | $2.60\times10^4$ | 22.57 | 3.35 | 0.15 |
| | ips2ra | $10^2$ | $2.76\times10^4$ | 36.24 | 2.23 | 0.22 |
| | ips4o | $10^2$ | $2.87\times10^4$ | 57.00 | 3.66 | 0.32 |
| | bramas | $10^2$ | $2.38\times10^4$ | 7.81 | 1.27 | 0.06 |
| | cxxstd | $10^2$ | $2.66\times10^4$ | 22.72 | 2.36 | 0.15 |
| pip | rsavx512 | $10^2$ | $2.30\times10^4$ | 9.66 | 1.30 | 0.08 |
| | rsavx | $10^2$ | $2.31\times10^4$ | 9.14 | 2.35 | 0.07 |
| | cxxavx | $10^2$ | $2.53\times10^4$ | 8.24 | 2.51 | 0.06 |
| | rsstd | $10^2$ | $2.67\times10^4$ | 23.36 | 3.33 | 0.15 |
| | ips2ra | $10^2$ | $4.78\times10^4$ | 338.94 | 1.82 | 0.90 |
| | ips4o | $10^2$ | $2.88\times10^4$ | 52.37 | 3.61 | 0.30 |
| | bramas | $10^2$ | $2.45\times10^4$ | 7.82 | 1.27 | 0.06 |
| | cxxstd | $10^2$ | $2.64\times10^4$ | 26.99 | 2.55 | 0.17 |
| gau | rsavx512 | $10^2$ | $2.40\times10^4$ | 9.59 | 1.31 | 0.07 |
| | rsavx | $10^2$ | $2.45\times10^4$ | 9.16 | 2.35 | 0.07 |
| | cxxavx | $10^2$ | $2.37\times10^4$ | 8.22 | 2.51 | 0.06 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| | rsstd | $10^2$ | $2.41\times10^4$ | 24.96 | 3.40 | 0.18 |
| | ips2ra | $10^2$ | $2.67\times10^4$ | 27.26 | 2.03 | 0.18 |
| | ips4o | $10^2$ | $2.63\times10^4$ | 36.77 | 3.72 | 0.23 |
| | bramas | $10^2$ | $2.54\times10^4$ | 7.83 | 1.27 | 0.06 |
| | cxxstd | $10^2$ | $2.64\times10^4$ | 21.66 | 2.70 | 0.14 |
| alm | rsavx512 | $10^2$ | $2.34\times10^4$ | 9.52 | 1.32 | 0.07 |
| | rsavx | $10^2$ | $2.53\times10^4$ | 9.03 | 2.38 | 0.07 |
| | cxxavx | $10^2$ | $2.40\times10^4$ | 8.29 | 2.49 | 0.06 |
| | rsstd | $10^2$ | $2.32\times10^4$ | 3.38 | 2.87 | 0.03 |
| | ips2ra | $10^2$ | $2.61\times10^4$ | 19.38 | 2.06 | 0.14 |
| | ips4o | $10^2$ | $2.53\times10^4$ | 5.59 | 3.20 | 0.04 |
| | bramas | $10^2$ | $2.51\times10^4$ | 7.87 | 1.26 | 0.06 |
| | cxxstd | $10^2$ | $2.46\times10^4$ | 19.12 | 2.39 | 0.13 |
| pfr | rsavx512 | $10^2$ | $2.22\times10^4$ | 9.69 | 1.30 | 0.08 |
| | rsavx | $10^2$ | $2.51\times10^4$ | 9.10 | 2.36 | 0.07 |
| | cxxavx | $10^2$ | $2.29\times10^4$ | 8.29 | 2.49 | 0.07 |
| | rsstd | $10^2$ | $2.32\times10^4$ | 6.12 | 2.80 | 0.05 |
| | ips2ra | $10^2$ | $2.45\times10^4$ | 22.29 | 2.03 | 0.16 |
| | ips4o | $10^2$ | $2.52\times10^4$ | 7.95 | 2.64 | 0.06 |
| | bramas | $10^2$ | $2.38\times10^4$ | 7.81 | 1.27 | 0.06 |
| | cxxstd | $10^2$ | $3.08\times10^4$ | 74.37 | 2.47 | 0.39 |
| pmi | rsavx512 | $10^2$ | $2.30\times10^4$ | 9.39 | 1.34 | 0.07 |
| | rsavx | $10^2$ | $2.31\times10^4$ | 9.09 | 2.36 | 0.07 |
| | cxxavx | $10^2$ | $2.46\times10^4$ | 8.19 | 2.52 | 0.06 |
| | rsstd | $10^2$ | $2.33\times10^4$ | 4.93 | 2.67 | 0.04 |
| | ips2ra | $10^2$ | $2.50\times10^4$ | 20.60 | 2.06 | 0.15 |
| | ips4o | $10^2$ | $2.52\times10^4$ | 8.02 | 3.28 | 0.06 |
| | bramas | $10^2$ | $2.48\times10^4$ | 7.83 | 1.27 | 0.06 |
| | cxxstd | $10^2$ | $2.55\times10^4$ | 27.45 | 2.33 | 0.19 |
| equ | rsavx512 | $10^3$ | $2.97\times10^4$ | 6.42 | 2.01 | 0.34 |
| | rsavx | $10^3$ | $2.47\times10^4$ | 1.49 | 2.62 | 0.11 |
| | cxxavx | $10^3$ | $2.57\times10^4$ | 1.27 | 2.55 | 0.09 |
| | rsstd | $10^3$ | $2.56\times10^4$ | 1.69 | 4.25 | 0.12 |
| | ips2ra | $10^3$ | $3.14\times10^4$ | 7.08 | 2.24 | 0.37 |
| | ips4o | $10^3$ | $2.34\times10^4$ | 1.45 | 4.88 | 0.11 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| | bramas | $10^3$ | $2.42\times10^5$ | 283.72 | 2.56 | 1.23 |
| | cxxstd | $10^3$ | $4.03\times10^4$ | 21.93 | 2.71 | 0.73 |
| asc | rsavx512 | $10^3$ | $2.82\times10^4$ | 6.42 | 2.01 | 0.37 |
| | rsavx | $10^3$ | $3.25\times10^4$ | 8.93 | 2.76 | 0.43 |
| | cxxavx | $10^3$ | $2.93\times10^4$ | 8.11 | 2.91 | 0.43 |
| | rsstd | $10^3$ | $2.41\times10^4$ | 1.69 | 4.25 | 0.12 |
| | ips2ra | $10^3$ | $6.57\times10^4$ | 54.46 | 1.62 | 0.98 |
| | ips4o | $10^3$ | $2.60\times10^4$ | 1.46 | 4.86 | 0.10 |
| | bramas | $10^3$ | $3.16\times10^4$ | 9.20 | 1.80 | 0.44 |
| | cxxstd | $10^3$ | $4.27\times10^4$ | 25.09 | 2.69 | 0.78 |
| dsc | rsavx512 | $10^3$ | $3.02\times10^4$ | 6.37 | 2.02 | 0.34 |
| | rsavx | $10^3$ | $3.14\times10^4$ | 8.60 | 2.82 | 0.42 |
| | cxxavx | $10^3$ | $2.91\times10^4$ | 8.06 | 2.91 | 0.43 |
| | rsstd | $10^3$ | $2.50\times10^4$ | 2.72 | 3.75 | 0.19 |
| | ips2ra | $10^3$ | $6.51\times10^4$ | 55.37 | 1.60 | 1.02 |
| | ips4o | $10^3$ | $2.68\times10^4$ | 2.58 | 3.91 | 0.17 |
| | bramas | $10^3$ | $3.53\times10^4$ | 14.58 | 2.03 | 0.57 |
| | cxxstd | $10^3$ | $3.87\times10^4$ | 19.62 | 2.82 | 0.70 |
| uni | rsavx512 | $10^3$ | $2.85\times10^4$ | 6.42 | 2.01 | 0.37 |
| | rsavx | $10^3$ | $3.11\times10^4$ | 8.73 | 2.77 | 0.42 |
| | cxxavx | $10^3$ | $2.97\times10^4$ | 8.16 | 2.84 | 0.40 |
| | rsstd | $10^3$ | $4.83\times10^4$ | 33.67 | 3.69 | 0.88 |
| | ips2ra | $10^3$ | $7.43\times10^4$ | 66.47 | 0.89 | 1.04 |
| | ips4o | $10^3$ | $1.04\times10^5$ | 102.35 | 1.75 | 1.09 |
| | bramas | $10^3$ | $3.16\times10^4$ | 8.49 | 1.73 | 0.42 |
| | cxxstd | $10^3$ | $9.01\times10^4$ | 87.24 | 0.94 | 1.11 |
| u16 | rsavx512 | $10^3$ | $3.08\times10^4$ | 6.40 | 2.02 | 0.36 |
| | rsavx | $10^3$ | $2.99\times10^4$ | 8.58 | 2.71 | 0.45 |
| | cxxavx | $10^3$ | $2.98\times10^4$ | 8.06 | 2.80 | 0.43 |
| | rsstd | $10^3$ | $3.53\times10^4$ | 16.34 | 3.48 | 0.66 |
| | ips2ra | $10^3$ | $4.92\times10^4$ | 31.91 | 1.25 | 0.83 |
| | ips4o | $10^3$ | $5.10\times10^4$ | 31.65 | 2.45 | 0.80 |
| | bramas | $10^3$ | $2.99\times10^4$ | 8.25 | 1.77 | 0.41 |
| | cxxstd | $10^3$ | $5.75\times10^4$ | 43.83 | 1.57 | 0.94 |
| pip | rsavx512 | $10^3$ | $2.97\times10^4$ | 6.40 | 2.02 | 0.34 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| | rsavx | $10^3$ | $2.99\times10^4$ | 8.32 | 2.83 | 0.41 |
| | cxxavx | $10^3$ | $2.83\times10^4$ | 7.71 | 2.97 | 0.42 |
| | rsstd | $10^3$ | $4.75\times10^4$ | 30.61 | 3.73 | 0.81 |
| | ips2ra | $10^3$ | $6.49\times10^4$ | 51.54 | 1.42 | 0.95 |
| | ips4o | $10^3$ | $9.15\times10^4$ | 87.57 | 2.12 | 1.09 |
| | bramas | $10^3$ | $3.70\times10^4$ | 16.98 | 2.04 | 0.65 |
| | cxxstd | $10^3$ | $9.06\times10^4$ | 87.54 | 2.12 | 1.10 |
| gau | rsavx512 | $10^3$ | $2.84\times10^4$ | 6.40 | 2.02 | 0.37 |
| | rsavx | $10^3$ | $2.90\times10^4$ | 8.68 | 2.78 | 0.46 |
| | cxxavx | $10^3$ | $3.20\times10^4$ | 7.94 | 2.89 | 0.40 |
| | rsstd | $10^3$ | $4.89\times10^4$ | 31.45 | 3.71 | 0.81 |
| | ips2ra | $10^3$ | $5.42\times10^4$ | 39.41 | 1.54 | 0.89 |
| | ips4o | $10^3$ | $7.29\times10^4$ | 63.13 | 1.98 | 1.02 |
| | bramas | $10^3$ | $3.00\times10^4$ | 9.55 | 1.68 | 0.49 |
| | cxxstd | $10^3$ | $8.79\times10^4$ | 84.73 | 0.98 | 1.10 |
| alm | rsavx512 | $10^3$ | $2.86\times10^4$ | 6.46 | 1.99 | 0.37 |
| | rsavx | $10^3$ | $3.16\times10^4$ | 8.68 | 2.84 | 0.42 |
| | cxxavx | $10^3$ | $2.98\times10^4$ | 8.06 | 2.93 | 0.42 |
| | rsstd | $10^3$ | $2.49\times10^4$ | 1.93 | 3.81 | 0.14 |
| | ips2ra | $10^3$ | $6.63\times10^4$ | 55.54 | 1.58 | 0.98 |
| | ips4o | $10^3$ | $5.27\times10^4$ | 39.03 | 2.31 | 0.93 |
| | bramas | $10^3$ | $3.08\times10^4$ | 9.24 | 1.79 | 0.44 |
| | cxxstd | $10^3$ | $4.27\times10^4$ | 24.90 | 2.71 | 0.77 |
| pfr | rsavx512 | $10^3$ | $2.99\times10^4$ | 6.44 | 2.00 | 0.35 |
| | rsavx | $10^3$ | $2.87\times10^4$ | 9.01 | 2.82 | 0.47 |
| | cxxavx | $10^3$ | $2.87\times10^4$ | 8.23 | 2.96 | 0.44 |
| | rsstd | $10^3$ | $2.44\times10^4$ | 3.98 | 3.82 | 0.27 |
| | ips2ra | $10^3$ | $6.25\times10^4$ | 51.47 | 1.75 | 0.99 |
| | ips4o | $10^3$ | $5.59\times10^4$ | 38.00 | 2.32 | 0.85 |
| | bramas | $10^3$ | $4.41\times10^4$ | 26.47 | 2.38 | 0.79 |
| | cxxstd | $10^3$ | $1.82\times10^5$ | 207.21 | 1.61 | 1.21 |
| pmi | rsavx512 | $10^3$ | $2.86\times10^4$ | 6.40 | 2.01 | 0.36 |
| | rsavx | $10^3$ | $3.13\times10^4$ | 9.90 | 2.57 | 0.48 |
| | cxxavx | $10^3$ | $3.05\times10^4$ | 9.26 | 2.63 | 0.44 |
| | rsstd | $10^3$ | $2.69\times10^4$ | 2.77 | 4.05 | 0.18 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| | ips2ra | $10^3$ | $6.31\times10^4$ | 51.28 | 1.76 | 0.99 |
| | ips4o | $10^3$ | $5.45\times10^4$ | 39.69 | 2.43 | 0.90 |
| | bramas | $10^3$ | $3.31\times10^4$ | 12.02 | 2.10 | 0.55 |
| | cxxstd | $10^3$ | $9.06\times10^4$ | 88.21 | 2.12 | 1.11 |
| equ | rsavx512 | $10^4$ | $3.69\times10^4$ | 1.85 | 2.19 | 0.70 |
| | rsavx | $10^4$ | $3.05\times10^4$ | 0.94 | 3.36 | 0.47 |
| | cxxavx | $10^4$ | $2.84\times10^4$ | 0.97 | 2.90 | 0.49 |
| | rsstd | $10^4$ | $3.53\times10^4$ | 1.54 | 4.55 | 0.62 |
| | ips2ra | $10^4$ | $5.13\times10^4$ | 3.56 | 3.51 | 0.89 |
| | ips4o | $10^4$ | $3.36\times10^4$ | 1.27 | 5.50 | 0.57 |
| | bramas | $10^4$ | $8.07\times10^6$ | 3183.44 | 2.36 | 3.95 |
| | cxxstd | $10^4$ | $2.88\times10^5$ | 34.31 | 2.67 | 1.24 |
| asc | rsavx512 | $10^4$ | $1.14\times10^5$ | 12.08 | 2.25 | 1.16 |
| | rsavx | $10^4$ | $1.15\times10^5$ | 11.95 | 2.90 | 1.14 |
| | cxxavx | $10^4$ | $1.12\times10^5$ | 11.47 | 2.92 | 1.14 |
| | rsstd | $10^4$ | $3.55\times10^4$ | 1.52 | 4.61 | 0.63 |
| | ips2ra | $10^4$ | $2.88\times10^5$ | 34.35 | 2.55 | 1.24 |
| | ips4o | $10^4$ | $3.34\times10^4$ | 1.27 | 5.50 | 0.57 |
| | bramas | $10^4$ | $4.89\times10^5$ | 60.48 | 2.46 | 1.26 |
| | cxxstd | $10^4$ | $3.11\times10^5$ | 37.25 | 2.76 | 1.24 |
| dsc | rsavx512 | $10^4$ | $1.15\times10^5$ | 12.24 | 2.22 | 1.16 |
| | rsavx | $10^4$ | $1.10\times10^5$ | 11.25 | 3.10 | 1.14 |
| | cxxavx | $10^4$ | $1.03\times10^5$ | 10.23 | 3.19 | 1.12 |
| | rsstd | $10^4$ | $4.31\times10^4$ | 2.59 | 3.86 | 0.80 |
| | ips2ra | $10^4$ | $2.92\times10^5$ | 35.08 | 2.49 | 1.24 |
| | ips4o | $10^4$ | $4.33\times10^4$ | 2.58 | 3.88 | 0.80 |
| | bramas | $10^4$ | $2.35\times10^5$ | 27.53 | 2.15 | 1.23 |
| | cxxstd | $10^4$ | $2.46\times10^5$ | 28.82 | 2.76 | 1.23 |
| uni | rsavx512 | $10^4$ | $1.17\times10^5$ | 12.34 | 2.22 | 1.16 |
| | rsavx | $10^4$ | $1.12\times10^5$ | 11.57 | 3.05 | 1.14 |
| | cxxavx | $10^4$ | $1.05\times10^5$ | 10.54 | 3.14 | 1.12 |
| | rsstd | $10^4$ | $7.00\times10^5$ | 87.74 | 1.78 | 1.27 |
| | ips2ra | $10^4$ | $6.83\times10^5$ | 85.33 | 1.19 | 1.27 |
| | ips4o | $10^4$ | $8.43\times10^5$ | 107.67 | 1.88 | 1.29 |
| | bramas | $10^4$ | $1.27\times10^5$ | 13.63 | 1.66 | 1.17 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| | cxxstd | $10^4$ | $9.34\times10^5$ | 182.59 | 0.63 | 1.97 |
| u16 | rsavx512 | $10^4$ | $1.16\times10^5$ | 12.20 | 2.21 | 1.16 |
| | rsavx | $10^4$ | $5.79\times10^4$ | 4.51 | 3.26 | 0.96 |
| | cxxavx | $10^4$ | $5.35\times10^4$ | 3.94 | 3.33 | 0.92 |
| | rsstd | $10^4$ | $1.25\times10^5$ | 13.19 | 3.53 | 1.16 |
| | ips2ra | $10^4$ | $1.35\times10^5$ | 14.46 | 3.30 | 1.17 |
| | ips4o | $10^4$ | $1.87\times10^5$ | 20.60 | 3.35 | 1.16 |
| | bramas | $10^4$ | $8.62\times10^5$ | 164.63 | 2.54 | 1.93 |
| | cxxstd | $10^4$ | $6.62\times10^5$ | 82.77 | 1.14 | 1.27 |
| pip | rsavx512 | $10^4$ | $1.14\times10^5$ | 12.09 | 2.21 | 1.16 |
| | rsavx | $10^4$ | $1.11\times10^5$ | 11.43 | 3.08 | 1.14 |
| | cxxavx | $10^4$ | $1.04\times10^5$ | 10.52 | 3.16 | 1.12 |
| | rsstd | $10^4$ | $5.85\times10^5$ | 72.54 | 1.99 | 1.27 |
| | ips2ra | $10^4$ | $3.28\times10^5$ | 39.49 | 1.85 | 1.25 |
| | ips4o | $10^4$ | $7.28\times10^5$ | 92.18 | 2.24 | 1.29 |
| | bramas | $10^4$ | $2.81\times10^5$ | 33.49 | 1.96 | 1.24 |
| | cxxstd | $10^4$ | $1.11\times10^6$ | 217.86 | 1.52 | 1.98 |
| gau | rsavx512 | $10^4$ | $1.17\times10^5$ | 12.37 | 2.20 | 1.16 |
| | rsavx | $10^4$ | $1.15\times10^5$ | 11.89 | 3.02 | 1.14 |
| | cxxavx | $10^4$ | $1.07\times10^5$ | 10.74 | 3.12 | 1.12 |
| | rsstd | $10^4$ | $3.97\times10^5$ | 48.31 | 2.21 | 1.25 |
| | ips2ra | $10^4$ | $2.19\times10^5$ | 25.53 | 2.52 | 1.23 |
| | ips4o | $10^4$ | $3.40\times10^5$ | 40.89 | 2.55 | 1.25 |
| | bramas | $10^4$ | $1.19\times10^5$ | 12.46 | 1.70 | 1.15 |
| | cxxstd | $10^4$ | $1.03\times10^6$ | 150.46 | 0.64 | 1.48 |
| alm | rsavx512 | $10^4$ | $1.15\times10^5$ | 12.09 | 2.25 | 1.16 |
| | rsavx | $10^4$ | $1.16\times10^5$ | 11.94 | 2.90 | 1.14 |
| | cxxavx | $10^4$ | $1.12\times10^5$ | 11.48 | 2.91 | 1.14 |
| | rsstd | $10^4$ | $5.43\times10^4$ | 3.91 | 4.03 | 0.91 |
| | ips2ra | $10^4$ | $2.91\times10^5$ | 34.67 | 2.53 | 1.24 |
| | ips4o | $10^4$ | $4.57\times10^5$ | 57.34 | 2.50 | 1.29 |
| | bramas | $10^4$ | $4.89\times10^5$ | 60.47 | 2.46 | 1.26 |
| | cxxstd | $10^4$ | $3.10\times10^5$ | 37.15 | 2.77 | 1.24 |
| pfr | rsavx512 | $10^4$ | $1.16\times10^5$ | 12.18 | 2.25 | 1.16 |
| | rsavx | $10^4$ | $1.15\times10^5$ | 11.93 | 3.00 | 1.15 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| | cxxavx | $10^4$ | $1.03\times10^5$ | 10.23 | 3.20 | 1.12 |
| | rsstd | $10^4$ | $5.21\times10^4$ | 3.55 | 4.23 | 0.86 |
| | ips2ra | $10^4$ | $2.88\times10^5$ | 34.48 | 2.54 | 1.24 |
| | ips4o | $10^4$ | $4.75\times10^5$ | 59.10 | 2.42 | 1.27 |
| | bramas | $10^4$ | $1.11\times10^6$ | 218.40 | 2.49 | 1.98 |
| | cxxstd | $10^4$ | $1.27\times10^6$ | 249.31 | 1.79 | 1.98 |
| pmi | rsavx512 | $10^4$ | $1.17\times10^5$ | 12.28 | 2.26 | 1.16 |
| | rsavx | $10^4$ | $1.17\times10^5$ | 12.05 | 2.90 | 1.14 |
| | cxxavx | $10^4$ | $1.13\times10^5$ | 11.52 | 2.91 | 1.14 |
| | rsstd | $10^4$ | $4.34\times10^4$ | 2.53 | 4.35 | 0.79 |
| | ips2ra | $10^4$ | $2.90\times10^5$ | 34.60 | 2.53 | 1.24 |
| | ips4o | $10^4$ | $4.80\times10^5$ | 59.19 | 2.54 | 1.26 |
| | bramas | $10^4$ | $6.73\times10^5$ | 84.67 | 2.53 | 1.28 |
| | cxxstd | $10^4$ | $1.10\times10^6$ | 142.45 | 1.87 | 1.31 |
| equ | rsavx512 | $10^5$ | $9.25\times10^4$ | 1.74 | 2.31 | 2.02 |
| | rsavx | $10^5$ | $9.07\times10^4$ | 0.86 | 3.56 | 1.09 |
| | cxxavx | $10^5$ | $8.13\times10^4$ | 0.75 | 3.70 | 1.07 |
| | rsstd | $10^5$ | $1.41\times10^5$ | 1.50 | 4.65 | 1.17 |
| | ips2ra | $10^5$ | $2.60\times10^5$ | 3.08 | 3.91 | 1.24 |
| | ips4o | $10^5$ | $1.22\times10^5$ | 1.26 | 5.54 | 1.15 |
| | bramas | $10^5$ | $1.03\times10^9$ | 40 998.75 | 1.83 | 3.97 |
| | cxxstd | $10^5$ | $1.36\times10^6$ | 40.31 | 2.80 | 2.98 |
| asc | rsavx512 | $10^5$ | $9.75\times10^5$ | 18.20 | 2.24 | 1.88 |
| | rsavx | $10^5$ | $7.53\times10^5$ | 14.71 | 3.10 | 1.98 |
| | cxxavx | $10^5$ | $9.28\times10^5$ | 13.20 | 3.22 | 1.44 |
| | rsstd | $10^5$ | $1.40\times10^5$ | 1.50 | 4.66 | 1.17 |
| | ips2ra | $10^5$ | $1.35\times10^6$ | 36.20 | 2.93 | 2.69 |
| | ips4o | $10^5$ | $1.23\times10^5$ | 1.29 | 5.41 | 1.15 |
| | bramas | $10^5$ | $1.33\times10^7$ | 527.13 | 2.23 | 3.98 |
| | cxxstd | $10^5$ | $1.20\times10^6$ | 41.30 | 3.03 | 3.47 |
| dsc | rsavx512 | $10^5$ | $1.15\times10^6$ | 18.10 | 2.24 | 1.59 |
| | rsavx | $10^5$ | $1.08\times10^6$ | 14.75 | 3.08 | 1.38 |
| | cxxavx | $10^5$ | $6.99\times10^5$ | 13.60 | 3.12 | 1.97 |
| | rsstd | $10^5$ | $2.22\times10^5$ | 2.57 | 3.89 | 1.22 |
| | ips2ra | $10^5$ | $1.33\times10^6$ | 39.43 | 2.88 | 2.98 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| | ips4o | $10^5$ | $2.21\times10^5$ | 2.56 | 3.91 | 1.22 |
| | bramas | $10^5$ | $1.31\times10^6$ | 43.85 | 2.15 | 3.37 |
| | cxxstd | $10^5$ | $1.09\times10^6$ | 32.34 | 2.93 | 2.97 |
| uni | rsavx512 | $10^5$ | $1.17\times10^6$ | 18.42 | 2.19 | 1.59 |
| | rsavx | $10^5$ | $8.67\times10^5$ | 15.24 | 2.98 | 1.78 |
| | cxxavx | $10^5$ | $7.11\times10^5$ | 13.84 | 3.06 | 1.97 |
| | rsstd | $10^5$ | $2.19\times10^6$ | 99.40 | 1.93 | 4.55 |
| | ips2ra | $10^5$ | $1.98\times10^6$ | 88.30 | 0.94 | 4.46 |
| | ips4o | $10^5$ | $2.57\times10^6$ | 113.03 | 1.97 | 4.41 |
| | bramas | $10^5$ | $9.69\times10^5$ | 19.81 | 1.39 | 2.06 |
| | cxxstd | $10^5$ | $4.89\times10^6$ | 223.97 | 0.61 | 4.58 |
| u16 | rsavx512 | $10^5$ | $5.14\times10^5$ | 7.85 | 2.22 | 1.55 |
| | rsavx | $10^5$ | $3.44\times10^5$ | 4.15 | 3.26 | 1.25 |
| | cxxavx | $10^5$ | $3.00\times10^5$ | 3.58 | 3.49 | 1.24 |
| | rsstd | $10^5$ | $1.02\times10^6$ | 20.16 | 2.24 | 1.99 |
| | ips2ra | $10^5$ | $7.95\times10^5$ | 13.16 | 3.39 | 1.67 |
| | ips4o | $10^5$ | $9.61\times10^5$ | 18.85 | 3.57 | 1.98 |
| | bramas | $10^5$ | $4.67\times10^7$ | 1851.87 | 2.55 | 3.96 |
| | cxxstd | $10^5$ | $2.07\times10^6$ | 94.44 | 1.21 | 4.57 |
| pip | rsavx512 | $10^5$ | $9.22\times10^5$ | 18.11 | 2.22 | 1.98 |
| | rsavx | $10^5$ | $7.73\times10^5$ | 15.02 | 3.03 | 1.96 |
| | cxxavx | $10^5$ | $8.77\times10^5$ | 13.67 | 3.10 | 1.58 |
| | rsstd | $10^5$ | $1.95\times10^6$ | 85.40 | 2.04 | 4.38 |
| | ips2ra | $10^5$ | $1.21\times10^6$ | 35.77 | 1.93 | 2.96 |
| | ips4o | $10^5$ | $2.28\times10^6$ | 101.48 | 2.20 | 4.47 |
| | bramas | $10^5$ | $1.44\times10^6$ | 55.04 | 2.00 | 3.83 |
| | cxxstd | $10^5$ | $6.21\times10^6$ | 283.93 | 1.66 | 4.58 |
| gau | rsavx512 | $10^5$ | $9.38\times10^5$ | 18.43 | 2.20 | 1.98 |
| | rsavx | $10^5$ | $7.70\times10^5$ | 11.41 | 3.00 | 1.50 |
| | cxxavx | $10^5$ | $7.92\times10^5$ | 9.97 | 3.08 | 1.28 |
| | rsstd | $10^5$ | $1.49\times10^6$ | 35.32 | 2.63 | 2.38 |
| | ips2ra | $10^5$ | $1.00\times10^6$ | 19.58 | 2.86 | 1.97 |
| | ips4o | $10^5$ | $1.26\times10^6$ | 31.91 | 3.08 | 2.54 |
| | bramas | $10^5$ | $1.69\times10^6$ | 50.34 | 2.16 | 2.99 |
| | cxxstd | $10^5$ | $3.34\times10^6$ | 149.60 | 0.79 | 4.48 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| alm | rsavx512 | $10^5$ | $9.26\times10^5$ | 18.21 | 2.23 | 1.98 |
| | rsavx | $10^5$ | $7.59\times10^5$ | 14.71 | 3.10 | 1.96 |
| | cxxavx | $10^5$ | $1.04\times10^6$ | 13.21 | 3.21 | 1.28 |
| | rsstd | $10^5$ | $3.74\times10^5$ | 4.57 | 4.01 | 1.26 |
| | ips2ra | $10^5$ | $1.24\times10^6$ | 36.27 | 2.93 | 2.96 |
| | ips4o | $10^5$ | $1.64\times10^6$ | 72.58 | 2.42 | 4.45 |
| | bramas | $10^5$ | $1.33\times10^7$ | 528.12 | 2.23 | 3.96 |
| | cxxstd | $10^5$ | $1.35\times10^6$ | 41.31 | 3.03 | 3.08 |
| pfr | rsavx512 | $10^5$ | $9.17\times10^5$ | 17.90 | 2.26 | 1.97 |
| | rsavx | $10^5$ | $7.62\times10^5$ | 14.78 | 3.07 | 1.96 |
| | cxxavx | $10^5$ | $9.34\times10^5$ | 13.68 | 3.10 | 1.48 |
| | rsstd | $10^5$ | $2.92\times10^5$ | 3.50 | 4.28 | 1.24 |
| | ips2ra | $10^5$ | $1.41\times10^6$ | 36.19 | 2.94 | 2.58 |
| | ips4o | $10^5$ | $1.67\times10^6$ | 72.72 | 2.43 | 4.36 |
| | bramas | $10^5$ | $6.35\times10^7$ | 2521.26 | 2.11 | 3.97 |
| | cxxstd | $10^5$ | $5.80\times10^6$ | 264.50 | 2.04 | 4.56 |
| pmi | rsavx512 | $10^5$ | $9.17\times10^5$ | 17.97 | 2.25 | 1.98 |
| | rsavx | $10^5$ | $7.57\times10^5$ | 14.78 | 3.08 | 1.97 |
| | cxxavx | $10^5$ | $9.82\times10^5$ | 13.39 | 3.17 | 1.38 |
| | rsstd | $10^5$ | $2.15\times10^5$ | 2.50 | 4.39 | 1.22 |
| | ips2ra | $10^5$ | $1.53\times10^6$ | 36.22 | 2.93 | 2.38 |
| | ips4o | $10^5$ | $1.65\times10^6$ | 73.47 | 2.48 | 4.46 |
| | bramas | $10^5$ | $2.35\times10^7$ | 934.12 | 2.15 | 3.97 |
| | cxxstd | $10^5$ | $3.30\times10^6$ | 150.64 | 2.15 | 4.58 |
| equ | rsavx512 | $10^6$ | $4.92\times10^5$ | 1.74 | 2.31 | 3.57 |
| | rsavx | $10^6$ | $3.23\times10^5$ | 0.90 | 3.41 | 2.84 |
| | cxxavx | $10^6$ | $3.32\times10^5$ | 0.74 | 3.74 | 2.28 |
| | rsstd | $10^6$ | $5.16\times10^5$ | 1.51 | 4.65 | 2.96 |
| | ips2ra | $10^6$ | $7.88\times10^5$ | 3.02 | 3.97 | 3.87 |
| | ips4o | $10^6$ | $4.79\times10^5$ | 1.38 | 5.08 | 2.91 |
| | bramas | $10^6$ | $1.06\times10^{11}$ | 421 401.51 | 1.78 | 3.97 |
| | cxxstd | $10^6$ | $1.02\times10^7$ | 46.94 | 2.86 | 4.58 |
| asc | rsavx512 | $10^6$ | $6.15\times10^6$ | 24.40 | 2.22 | 3.97 |
| | rsavx | $10^6$ | $3.95\times10^6$ | 18.06 | 3.08 | 4.57 |
| | cxxavx | $10^6$ | $3.59\times10^6$ | 16.37 | 3.16 | 4.56 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| | rsstd | $10^6$ | $5.29{\times}10^5$ | 1.51 | 4.65 | 2.91 |
| | ips2ra | $10^6$ | $7.58{\times}10^6$ | 34.74 | 3.06 | 4.58 |
| | ips4o | $10^6$ | $5.37{\times}10^5$ | 1.29 | 5.45 | 2.42 |
| | bramas | $10^6$ | $1.64{\times}10^9$ | 6515.99 | 2.04 | 3.98 |
| | cxxstd | $10^6$ | $9.82{\times}10^6$ | 44.92 | 3.28 | 4.58 |
| dsc | rsavx512 | $10^6$ | $6.13{\times}10^6$ | 24.37 | 2.22 | 3.98 |
| | rsavx | $10^6$ | $3.95{\times}10^6$ | 18.06 | 3.09 | 4.58 |
| | cxxavx | $10^6$ | $3.60{\times}10^6$ | 16.44 | 3.14 | 4.58 |
| | rsstd | $10^6$ | $8.95{\times}10^5$ | 2.65 | 3.77 | 2.98 |
| | ips2ra | $10^6$ | $8.44{\times}10^6$ | 38.65 | 3.00 | 4.58 |
| | ips4o | $10^6$ | $8.47{\times}10^5$ | 2.52 | 3.97 | 2.99 |
| | bramas | $10^6$ | $1.64{\times}10^7$ | 64.98 | 2.13 | 3.97 |
| | cxxstd | $10^6$ | $7.60{\times}10^6$ | 34.50 | 3.20 | 4.54 |
| uni | rsavx512 | $10^6$ | $6.28{\times}10^6$ | 24.98 | 2.16 | 3.98 |
| | rsavx | $10^6$ | $4.11{\times}10^6$ | 18.76 | 2.97 | 4.57 |
| | cxxavx | $10^6$ | $3.72{\times}10^6$ | 17.00 | 3.04 | 4.58 |
| | rsstd | $10^6$ | $2.37{\times}10^7$ | 108.77 | 2.04 | 4.59 |
| | ips2ra | $10^6$ | $1.76{\times}10^7$ | 80.67 | 1.30 | 4.58 |
| | ips4o | $10^6$ | $2.72{\times}10^7$ | 124.49 | 2.12 | 4.57 |
| | bramas | $10^6$ | $6.37{\times}10^6$ | 25.30 | 1.37 | 3.98 |
| | cxxstd | $10^6$ | $5.81{\times}10^7$ | 265.69 | 0.61 | 4.57 |
| u16 | rsavx512 | $10^6$ | $2.02{\times}10^6$ | 7.99 | 2.14 | 3.97 |
| | rsavx | $10^6$ | $1.00{\times}10^6$ | 4.36 | 3.16 | 4.37 |
| | cxxavx | $10^6$ | $8.28{\times}10^5$ | 3.63 | 3.23 | 4.41 |
| | rsstd | $10^6$ | $4.15{\times}10^6$ | 18.88 | 2.51 | 4.55 |
| | ips2ra | $10^6$ | $2.85{\times}10^6$ | 13.02 | 3.40 | 4.58 |
| | ips4o | $10^6$ | $4.12{\times}10^6$ | 18.87 | 3.55 | 4.58 |
| | bramas | $10^6$ | $6.04{\times}10^9$ | 23 993.92 | 1.96 | 3.97 |
| | cxxstd | $10^6$ | $2.23{\times}10^7$ | 102.16 | 1.36 | 4.58 |
| pip | rsavx512 | $10^6$ | $6.17{\times}10^6$ | 24.56 | 2.21 | 3.98 |
| | rsavx | $10^6$ | $4.00{\times}10^6$ | 18.28 | 3.05 | 4.58 |
| | cxxavx | $10^6$ | $3.63{\times}10^6$ | 16.56 | 3.11 | 4.57 |
| | rsstd | $10^6$ | $2.05{\times}10^7$ | 93.72 | 2.17 | 4.58 |
| | ips2ra | $10^6$ | $1.02{\times}10^7$ | 46.70 | 2.16 | 4.58 |
| | ips4o | $10^6$ | $2.41{\times}10^7$ | 110.19 | 2.39 | 4.57 |

continued on next page

| $\mathscr{D}$ | Sort | Samples | $r$ [ns] | CPE | IPC | GHz |
|---|---|---|---|---|---|---|
| | bramas | $10^6$ | $2.37\times10^7$ | 93.98 | 1.91 | 3.97 |
| | cxxstd | $10^6$ | $6.79\times10^7$ | 310.40 | 1.79 | 4.57 |
| gau | rsavx512 | $10^6$ | $4.37\times10^6$ | 17.37 | 2.17 | 3.98 |
| | rsavx | $10^6$ | $2.04\times10^6$ | 9.19 | 3.12 | 4.53 |
| | cxxavx | $10^6$ | $1.76\times10^6$ | 7.98 | 3.21 | 4.55 |
| | rsstd | $10^6$ | $7.15\times10^6$ | 32.71 | 2.78 | 4.58 |
| | ips2ra | $10^6$ | $4.63\times10^6$ | 21.17 | 2.98 | 4.58 |
| | ips4o | $10^6$ | $5.64\times10^6$ | 25.79 | 3.31 | 4.58 |
| | bramas | $10^6$ | $2.13\times10^8$ | 846.56 | 2.52 | 3.98 |
| | cxxstd | $10^6$ | $3.36\times10^7$ | 153.50 | 0.92 | 4.57 |
| alm | rsavx512 | $10^6$ | $6.13\times10^6$ | 24.38 | 2.22 | 3.98 |
| | rsavx | $10^6$ | $3.94\times10^6$ | 18.03 | 3.09 | 4.58 |
| | cxxavx | $10^6$ | $3.59\times10^6$ | 16.35 | 3.16 | 4.56 |
| | rsstd | $10^6$ | $1.61\times10^6$ | 7.32 | 3.97 | 4.57 |
| | ips2ra | $10^6$ | $7.59\times10^6$ | 34.74 | 3.06 | 4.58 |
| | ips4o | $10^6$ | $1.69\times10^7$ | 77.46 | 2.69 | 4.57 |
| | bramas | $10^6$ | $1.64\times10^9$ | 6509.98 | 2.04 | 3.98 |
| | cxxstd | $10^6$ | $9.82\times10^6$ | 44.98 | 3.28 | 4.58 |
| pfr | rsavx512 | $10^6$ | $6.11\times10^6$ | 24.30 | 2.22 | 3.98 |
| | rsavx | $10^6$ | $3.94\times10^6$ | 18.01 | 3.09 | 4.58 |
| | cxxavx | $10^6$ | $3.65\times10^6$ | 16.67 | 3.10 | 4.57 |
| | rsstd | $10^6$ | $9.91\times10^5$ | 3.54 | 4.24 | 3.59 |
| | ips2ra | $10^6$ | $7.59\times10^6$ | 34.75 | 3.06 | 4.58 |
| | ips4o | $10^6$ | $1.70\times10^7$ | 77.60 | 2.69 | 4.57 |
| | bramas | $10^6$ | $6.98\times10^9$ | 27 717.47 | 1.91 | 3.97 |
| | cxxstd | $10^6$ | $6.30\times10^7$ | 288.18 | 2.18 | 4.57 |
| pmi | rsavx512 | $10^6$ | $6.12\times10^6$ | 24.36 | 2.22 | 3.98 |
| | rsavx | $10^6$ | $3.99\times10^6$ | 18.26 | 3.05 | 4.58 |
| | cxxavx | $10^6$ | $3.62\times10^6$ | 16.50 | 3.13 | 4.57 |
| | rsstd | $10^6$ | $6.85\times10^5$ | 2.51 | 4.38 | 3.70 |
| | ips2ra | $10^6$ | $7.59\times10^6$ | 34.77 | 3.06 | 4.58 |
| | ips4o | $10^6$ | $1.71\times10^7$ | 78.17 | 2.76 | 4.58 |
| | bramas | $10^6$ | $2.54\times10^9$ | 10 110.17 | 1.97 | 3.98 |
| | cxxstd | $10^6$ | $3.56\times10^7$ | 163.15 | 2.33 | 4.58 |

# Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Seitens des Verfassers bestehen Einwände, die vorliegende Bachelorarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Jena, den 2. August 2021

Frank Thiemicke