INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Evaluation of C++ SIMD Libraries

Felix Jonathan Rocke

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Evaluation of C++ SIMD Libraries

Felix Jonathan Rocke

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Karl Fürlinger Sergej Breiter
Abgabetermin:	27. April 2023

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 27. April 2023

(Unterschrift des Kandidaten)

Abstract

Single Instruction, Multiple Data (SIMD) units parallelize code through vectorization, thus enabling substantial performance improvements. Over the last two decades, SIMD units have become part of most CPUs. Despite this availability, many applications are not taking full advantage of SIMD units since utilizing the performance potential requires highly hardware-dependent instructions. However, the increase in performance and the substantial energy savings provided by these instructions should no longer be ignored, leading to the need for an efficient SIMD API to allow for an efficient and portable programming model.

This thesis will evaluate six of the most popular SIMD libraries by reviewing their supported extensions, functions, documentation, and ease of use. Furthermore, we will benchmark the performance of the selected libraries using a floating point benchmark and compare their results to dedicated intrinsics implementations using the AVX2, AVX512, SVE, and NEON SIMD extensions. Finally, we will also take a closer look at Google's SIMD library, Highway, which has been rising in popularity recently. We will assess if the library is ready to take on complex real-world algorithms by conducting a case study on the vectorization of an algorithm operating on unsigned integers.

The results of the floating-point benchmark show that multiple libraries can match the performance of compiler intrinsics. Highway excelled with a strong performance across multiple SIMD extensions for the real-world integer algorithm. Thus, Highway may currently be the most suitable SIMD library for many software projects.

Contents

1 Introduction

Single Instruction, Multiple Data (SIMD) units, in Flynn's classification scheme, are an internal hardware feature of most modern processors and enable a form of parallel processing [Mar18]. SIMD functionality was first introduced to vector supercomputers in the 1970s and then found its way into consumer processors in the 1990s. SIMD units can speed up data parallel programs through code vectorization [PCM+16]. SIMD and vector parallelism, in general, employ the concept of Data Level Parallelism (DLP), which they achieve through data-parallel execution [PCM+16, KMSZ15]. In other words, SIMD instructions allow the execution of an operation on multiple data operands simultaneously instead of repeating an operation sequentially on all data points. Most vector instructions typically operate on two SIMD registers and follow the same pattern, which is visualized in Figure 1.1. First, the SIMD unit is supplied with two vectors and the operation that should be performed on them. The operation will then return the resulting vector. SIMD units typically support arithmetic, logical, and bit manipulation operations, e.g., addition, bitwise-AND, and sometimes math functions like an absolute or square root operation [CFCD17, Int22a]. Besides enabling better performance, employing SIMD functionality can also improve energy efficiency because only one instruction needs to be fetched for multiple data points instead of one instruction for each data point [Mar18].

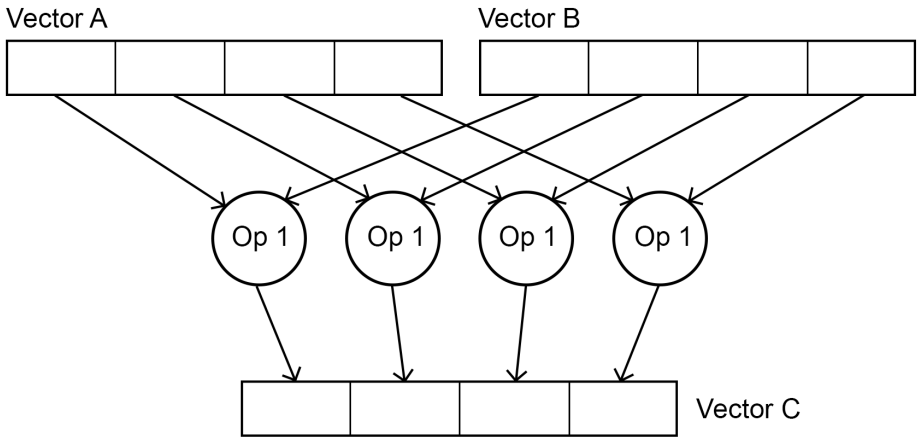

Figure 1.1: Visualization of a typical vector instruction operating on two vector registers with four lanes each.

SIMD extensions, such as AVX512, NEON, and SVE, are extensions to instruction set architectures (ISAs), e.g., x86, ARM, and RISC-V, which enable the use of vector operations that can lead to a substantial speed-up compared to a scalar execution. For example, the addition of two vectors with eight floating point numbers each could experience a theoretical speed-up of a factor of eight.

Besides the simultaneous operations on a vector, the memory access concept of SIMD

units may give rise to further increases in performance. Vector load and store instructions can load multiple data items from or to the memory in blocks, improving the memory bus bandwidth utilization compared to scalar load and store instructions [CFCD17]. The most significant performance improvements can be found in applications with a significant level of data parallelism. Therefore, high-performance applications, e.g., data mining or multimedia applications, often utilize vectorization. In general, computations heavily reliant on linear algebra benefit the most [Mar18]. Accordingly, for performance-critical applications, using the vector functionality of modern CPUs is highly desirable.

In order to employ the SIMD hardware of a CPU, the code needs to be vectorized first. There are several approaches to achieving this. The most common concept is to rely on the auto-vectorization features of modern compilers. The compiler recognizes data-parallel patterns and replaces the scalar arithmetic instructions with vector instructions. Although auto-vectorization has significantly improved over the last few years, compilers only achieve a fraction of the theoretically possible speed-up since they regularly fail to vectorize code with complex control flows or structured data layouts [PCM+16].

The best performance can often only be achieved by using intrinsic functions that directly inline assembly instructions offered by the SIMD extension while allowing for a higher level of abstraction than programming in assembly language. Unfortunately, this approach to SIMD programming requires a significant amount of effort and knowledge. Although the programmer can still think sequentially and does not need to worry about race conditions and other problems associated with parallel programming, using intrinsics is accompanied by other challenges [Mar18]. The most notable problems are a lack of compatibility and portability, thus requiring a different implementation for each SIMD extension. Accordingly, while being challenging and labor-intensive, intrinsics programming often results in the best performance through its low-level approach [PCM+16].

Library-based programming approaches attempt to bridge the gap between effort and performance. The functions of these libraries map to corresponding compiler intrinsics of various extensions or offer code patterns that can be vectorized by the compiler more efficiently. In this thesis, we will evaluate a variety of libraries that utilize different approaches and compare them with respect to multiple factors. Specifically, the evaluation factors will include the supported extension sets, available functions, documentation quality, ease of use, and performance. A floating point benchmark will be used for the performance evaluation. We will also examine one of the libraries more closely by conducting a case study on its ability to efficiently vectorize a real-world algorithm utilizing bitwise operations on unsigned integers.

The remaining chapters of this thesis are structured as follows. Chapter 2 will provide an in-depth explanation of SIMD's core programming principles and will introduce important terms and concepts referred to in this thesis. In Chapter 3, we will present a brief overview of the different SIMD programming approaches before more closely reviewing six different libraries in Chapter 4. Chapter 5 will present the Mandelbrot benchmark and its different implementations. Chapter 6 will describe a case study on using the Highway library to vectorize an algorithm for epistasis detection. The subsequent chapter will evaluate the results of the Mandelbrot benchmark and the case study. The final chapter will summarize the findings and give an outlook on the future of SIMD programming.

2 Core SIMD Principles

Since the widespread introduction of SIMD units to microprocessors in the 1990s, various extensions have been developed. The first SIMD capabilities were introduced to Intel processors with the MMX extension. Since then, Intel has developed multiple improved and more powerful extension sets [Sie16]. The integer-only MMX extension set was replaced by Streaming SIMD Extensions (SSE), which introduced vector operations on floating point numbers [Int22a]. Nowadays, the focus has transitioned away from integers toward floating point numbers [JR15].

Nevertheless, the concepts of SIMD programming remain similar across the many extensions available today. This chapter will introduce the most common terminology and the principles of SIMD programming.

2.1 SIMD Registers

SIMD extensions are usually implemented using SIMD units, which contain SIMD registers. These registers can store multiple values and typically have a larger size than regular registers, allowing them to hold more data. The size of vector registers has been growing over the last two decades. These registers can be found in most, if not all, modern CPUs across multiple architectures [JR15]. SIMD registers are sometimes also called vector registers. In the following chapters, these terms are used interchangeably.

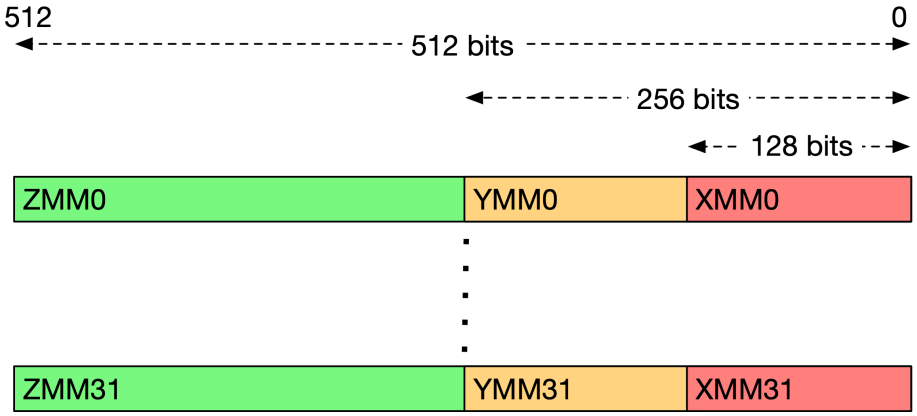

Figure 2.1: x86-64 vector registers on a CPU supporting AVX512. Visualization from Ref. [Sch16].

One of Intel's most recent SIMD extensions, AVX512, has 32 vector registers, each having a size of 512-bit [Int22a, PCM+16]. These registers are addressed as `ZMM0-ZMM31`. Their lower half can be addressed as `YMM0-YMM31` and corresponds to 256-bit vector registers introduced with AVX (compare Figure 2.1). Moreover, the lower quarter of those `ZMM` registers

corresponds to 128-bit registers introduced with SSE [JR15, Int22a]. As a result, AVX extensions allow for backward compatibility but not forward compatibility.

2.2 Lanes

In SIMD registers, data is divided into multiple lanes wherein each lane holds a single scalar value. So a SIMD lane is a unit that operates on a single scalar value. The number of lanes describes how many individual scalar values fit into one vector register, i.e., how many values a vector operation can operate on simultaneously.

The number of lanes that fit in a SIMD register depends on two factors, namely, the register size and the scalar type stored in the register. For example, SSE offers 128-bit vector registers so that each register can store four 32-bit single precision floating point numbers, comparable to Figure 1.1, or two 64-bit double precision floating point numbers [PCM+16, JR15, CFCD17].

Accordingly, larger registers allow for more lanes and, therefore, more operations to be performed simultaneously, which explains the trend towards an increase in vector register sizes.

2.3 Memory Alignment

Operations, capable of loading and storing multiple elements in one instruction, are essential to extract the highest performance out of a SIMD unit. The fastest memory operations are possible when the array we load from or store into allows for contiguous memory accesses, and the memory address is aligned to the beginning of a cache line [CFCD17, Int23]. Thus, allowing the provided data locality to be exploited by loading entire cache lines into SIMD registers [Pac22].

While both aligned and unaligned load/store operations exist, using aligned operations is generally preferred. Operating on aligned memory improves performance because unaligned load operations can make multiple load and shift instructions necessary when a cache line is crossed to piece together the misaligned data inside a SIMD register. However, aligning memory requires additional steps, e.g., using `__attribute__((aligned(64)))`, and may lead to a higher memory consumption since the data must be padded or the memory is fragmented leaving addresses unused, which can no longer be utilized effectively [Int22a, CFCD17, Pac22].

2.4 Vectorizing Loops

SIMD operations are commonly used to vectorize loops. However, for nested loops, only the innermost loop can, in most instances, be vectorized easily. Additionally, there can be no data dependence between the vector lanes because, for example, no lane can wait to read the result of another lane to finish its calculation. I.e., the calculations taking place in each lane must be independent of all the other lanes [NDR+11].

The function depicted in Listing 2.1 is a simple example of a loop vectorization, which demonstrates how a vectorized dot product could be implemented using AVX512 compiler intrinsics. To keep the example as simple as possible, we assume that the length of the vectors is a multiple of 16, which is the number of lanes available, so no extra steps are

required to handle remainders. Due to the 512-bit registers available under AVX512 and the use of 32-bit floats, the vector instructions can operate on 16 elements simultaneously. Therefore, the vectorization process can be compared with unrolling a loop, wherein each iteration, we work on 16 elements [NDR+11, PJ15]. Thus, we increment the iterator `i` by the number of lanes (compare line 5 of Listing 2.1).

```
1  float dot_product_avx512(float * a, float * b, size_t length) {
2      assert(length % 16 == 0);   // constraint for simplicity
3      __m512 sum = _mm512_setzero_ps();   // zero initialized vector
4
5      for (size_t i = 0; i < length; i += 16) {
6          __m512 av = _mm512_load_ps(a + i);  // load 16 values of a into av
7          __m512 bv = _mm512_load_ps(b + i);  // load 16 values of b into bv
8          sum = _mm512_fmadd_ps(av, bv, sum); // sum := (av * bv) + sum
9      }
10
11     return _mm512_reduce_add_ps(sum);   // return the sum of all 16 lanes
12 }
```

Listing 2.1: Dot product implementation using AVX512 compiler intrinsics.

Regarding the intrinsics used in Listing 2.1, the first one we utilized is `_mm512_setzero_ps`, which sets every lane inside a vector register to zero (compare line 3). The C type `__m512` represents AVX512's 512-bit vector registers containing single-precision floating point numbers. The precision of the pack of numbers we operate on is also specified in every function through the ending of `_ps` (compare, e.g., line 3), which corresponds to packed single-precision, other endings can be, for example, `_pd` for packed double-precision, or `_ph` for packed half-precision [Int22a]. The subsequent compiler intrinsics function we use is `_mm512_load_ps` which loads 16 single-precision floating point numbers from aligned memory into a vector register (see lines 6 and 7). The following function, `_mm512_fmadd_ps`, is a SIMD function requiring an additional Fused Multiply Add (FMA) unit beside the SIMD unit [Int22a]. This function can perform a multiplication and subsequent addition in a single instruction. The final function necessary for this example is `_mm512_reduce_add_ps`, which computes a sum across all the vector lanes and returns a scalar result (compare line 11). It is important to note that this function consists of a sequence of instructions with narrowing vector lengths. Therefore, it should be avoided in loops to optimize performance [Int22a].

The performance of the function, `dot_product_avx512`, in Listing 2.1 could be further improved by unrolling the loop and adding multiple sum accumulators to increase the throughput. Due to the latency of 4 and throughput of 0.5 CPI of the `_mm512_fmadd_ps` function on Intel Xeon Ice Lake-SP processors [Int22a], multiple independent accumulators can overcome the latency associated with the FMA operation through pipelining. The number of independent accumulators required for optimal performance depends on the number of FMA ports available, the latency, and the throughput associated with the FMA instruction.

3 SIMD Programming Approaches

Multiple programming approaches to vectorize code exist. These approaches differ in the performance they can provide, their portability, and their ease of use. This chapter will dive deeper into those approaches, examine the factors that affect their performance, and discuss their advantages and disadvantages. Furthermore, the need for SIMD libraries in vectorizing code for multiple platforms will be discussed.

3.1 Implicit Vectorization

The programming model of implicit vectorization refers to a technique in which the compiler or interpreter vectorizes scalar code without requiring the programmer's explicit use of vector instructions.

This approach of vectorizing code has several advantages. First, in most cases, the programmer will only be required to add compiler flags (compare Table 3.1) or pragmas to his code to enable the auto-vectorization features of modern compilers [PCM+16, GCC22a]. This results in highly portable code since only a single version is required across all architectures because it can be compiled for all SIMD extensions and platforms, offering a compatible compiler. Therefore, it is only necessary to update the compiler to take advantage of a new SIMD extension set [PCM+16].

3.1.1 Auto Vectorization

This section will focus on the auto-vectorization features of GCC (GNU Compiler Collection) compilers [GCC22b] because we used GCC's vectorizer in Section 7.2. However, most other popular compilers, e.g., LLVM and Intel® Compiler, also have similar auto-vectorization features [LLV23, Int22b].

Work on an auto-vectorization feature for GCC started in 2003, with the primary goal being the creation of a basic vectorizer with the ability to map simple scalar operations to their corresponding vector operation. The basic vectorizer was limited to innermost loops, aligned memory, the absence of function calls, and no if-then-else constructs, besides the need for the code structure and data dependence to allow vectorization [GCC22a]. Since then, the features have been enhanced, and, for example, loops with conditions can now be vectorized in some instances. However, the mentioned restrictions still often prohibit effective vectorization [GCC22a, PJ15].

When using the vectorization feature of GCC, a few flags are of particular importance (see Table 3.1). The first and most crucial flag is `-ftree-vectorize`, which turns on GCC's tree vectorizer [GCC22a]. Another essential flag is `-ftree-vectorizer-verbose`, which outputs information on which loops were vectorized and the reason for any failed loop vectorizations. The level of detail this flag outputs can be adjusted; more information can be found in Ref. [GCC05].

Flag	Description
`-ftree-vectorize`	Enables vectorization, by default turned on with `-O2`.
`-ffast-math`	Required for the vectorization of most floating point operations since instruction reordering can lead to different rounding errors.
`-ftree-vectorizer-verbose`	Outputs information on which loops were and were not vectorized, as well as the cause. There are multiple debug levels to this flag. More information can be found in Ref. [GCC05].

Table 3.1: Important GCC auto-vectorization flags. Details from Refs. [GCC23e, GCC22a, GCC05].

Although vectorization features of modern compilers have improved since their inception, they still need to catch up with explicit vectorization using intrinsics [PCM+16]. Compilers fail to vectorize code with more complex control flows because they are often limited by the information not coded into the algorithm and, therefore, cannot generate highly efficient vectorized code [KL12]. Studies have shown that auto-vectorized code, on average, only takes advantage of two vector lanes [PJ15].

3.1.2 OpenMP Pragmas

More than 20 years ago, OpenMP (Open Multi-Processing) was created by compiler and hardware manufacturers to provide a more straightforward way to use threads by defining a compiler-directed threading model rather than using libraries such as Pthreads. Most popular C/C++ compilers support a recent version of OpenMP [Bre20, Ope18, PCM+16].

In Version 4.0 of OpenMP, support for SIMD was added to the API [Ope18]. Since then, the pragma `#pragma omp simd` can advise the compiler that a loop is data parallel and to vectorize it. Additional clauses can help the compiler choose safe vector lengths, inform the compiler about memory alignment and possible reductions [Ope18, Bre20, PCM+16].

3.2 Explicit Vectorization

Explicit vectorization is commonly performed using compiler intrinsics or by writing assembly routines. Unfortunately, both strategies are labor-intensive due to their poor cross-platform portability [KL12]. Multiple projects attempted to solve this issue by creating highly portable and performant libraries such as the libraries in Refs. [Goo22c, Vc22a, Lib22a].

Unlike implicit vectorization, explicit vectorization requires in-depth knowledge of the principles of vectorization and mindful incorporation into the code. However, there are distinct differences in programming with intrinsics, assembly and libraries, which will be highlighted in the following two subsections.

3.2.1 Compiler Intrinsics

Compiler intrinsics functions directly map to assembly instructions that are inlined by the compiler. As a result, compiler intrinsics provide one of the most performant ways of vectorizing code. In addition, intrinsics offer an abstraction layer above assembly code, facilitating use by a broader audience since only few developers are comfortable writing code in assembly language [KL12]. The GCC compiler, for example, will also further optimize code utilizing compiler intrinsics or inline assembly [GCC23b, Lir09]. Nevertheless, intrinsics have multiple drawbacks.

The first and primary reason is that intrinsics are highly hardware-dependent, making portability a significant issue. For example, AVX intrinsics lack forward compatibility, i.e., code using, for example, AVX-512 intrinsics can not be executed on older machines which, for example, support AVX2 [Int22a]. Thus, maintaining multiple versions of the same code is often necessary to ensure highly efficient execution on all machines [KL12].

Another factor is that SIMD extensions for x86 architectures do not provide complete type safety, making bugs difficult to identify [KL12]. For example, the only AVX512 type for integers is `__m512i`. As a result, it is possible to, e.g., perform an addition for 64-bit integers on a vector packed with 32-bit integers without causing an error.

3.2.2 Libraries

SIMD libraries address some issues known from compiler intrinsics. For example, many libraries map their functions to the underlying intrinsic functions but give them more readable names. For example, most libraries we examine offer overloaded functions so that functions have the same name across different data types, vector lengths, and SIMD extension sets. Turning functions such as the AVX function, `_mm256_add_ps`, for the addition of single precision floating point numbers, into, for example, `Add`, makes the code easier to read and platform independent [Int22a, Goo22e]. This further abstraction opens up vectorization to a broader audience than intrinsics and assembly.

Another benefit of this higher level of abstraction is that the library's functions are typically mapped to various SIMD extensions. Thus, compiling the same code for different architectures and extension sets is possible, resulting in significantly more portable code since only one version of the code needs to be maintained instead of distinct versions for every extension set. However, when a new SIMD extension is released, it will be necessary to update the library.

The maintenance process is where the biggest drawback of libraries arises. In the past, research institutions or open-source projects have typically developed SIMD libraries. However, those projects have often not gained enough traction, so adding support for new SIMD extensions to the library either takes a long time or does not happen. As a result, some libraries no longer receive updates or only consider bug fixes. The problem of long-term support plagues all SIMD libraries and must be considered a significant drawback.

Nonetheless, when only considering the currently available SIMD extensions, those libraries can cut down development time considerably. Even when a new SIMD extension is released, it can take years for the corresponding hardware to become widely available. Accordingly, it is not vital for many applications to immediately support the new extension set. In the worst case, additional code versions must be maintained, while most SIMD extensions are covered by the library, which is still an improvement over not using a library.

Most SIMD libraries that map their functions to compiler intrinsics claim to be zero overhead by utilizing the inline optimization pass of the compiler. While this may result in a zero-cost abstraction layer, there are instances in which a library implementation is slower than a compiler intrinsics implementation. Multiple factors can lead to this issue. The two most prevalent are that there still is overhead associated with the library and the other is that library functions are more limited than intrinsic ones. Therefore, combining various functions may be necessary to substitute missing functions in a library. As a result, it is sometimes possible to write a more efficient implementation utilizing intrinsics rather than a library.

4 C++ SIMD Libraries

This chapter will review six SIMD libraries capable of generating highly portable code, by offering an abstraction layer above intrinsics programming, as discussed in Section 3.2.2. However, we will first go over how we selected the libraries and explain the non-performance-based review criteria before examining the libraries more closely. In the final section of this chapter, we will summarize the key aspects and review the results. The performance evaluation of the libraries introduced in this chapter can be found in Chapter 7.

4.1 Library Selection Methodology

The libraries were selected based on three main factors. The first is popularity, the second is how advanced a library is, and the third and final factor is the design principle.

The first factor in the selection process is how popular a library is. Popular libraries frequently have more thorough documentation, tests with higher code coverage, and a larger developer community, enabling better support and assistance. We determined a library's popularity by comparing how often it is mentioned in literature and other sources. Additionally, we compared how many so-called 'stars' their GitHub pages had to gauge the following they have. Finally, we selected some of the most popular SIMD libraries currently available through this process.

Secondly, we aimed to select the most advanced libraries since they typically support more SIMD extensions, provide more features, and overall have fewer errors. Thus, they allow the programmer to develop more effective and higher-quality code. We determined how advanced a library is by reviewing the supported extensions and functions. The libraries' popularity and advancement often coincided, leading to similar choices in the selection process.

Lastly, the design principle was also essential to the selection process. For example, some libraries put simplicity first, whereas others emphasize flexibility or modularity. Furthermore, not all follow the same vectorization approach. Therefore, we included libraries with different approaches and design philosophies, even if they are less popular or advanced, to ensure we have a diverse selection to highlight the differences.

4.2 Review Criteria

The selected libraries' capabilities will be assessed in four main areas: supported SIMD extension sets, available functions, documentation quality, and ease of use. Performance related factors will not be considered here.

The supported SIMD extension sets are critical determinants of a library's suitability for a project since different CPU architectures support different extensions. Thus, a library supporting a wider variety will be more versatile and suitable for a more extensive range of projects. For example, a library supporting AVX and NEON can generate instructions

for most modern x86 and ARM CPUs. Accordingly, the supported SIMD extensions are a critical determinant in choosing the optimal library for a project.

Besides the extensions, the available functions are equally essential in selecting a library since they must be capable of tackling the project's problems or, even better, solving any task compiler intrinsics could handle. As a result, we will review the functions of the six libraries to determine their capabilities.

Another crucial factor we consider when reviewing libraries is the quality of their documentation. They should be comprehensive, clear, fast to navigate, and easily understandable since the documentation is a tool for the developer to learn how to use the library correctly. The provided information should, for example, include the library's function signatures, usage examples, and any required dependencies.

In addition to the documentation, the ease of use is also a critical factor for consideration since a library that is difficult to use or has a steep learning curve will slow down the development process and increase the likelihood of errors. This factor is significant because the libraries should simplify SIMD programming and prevent typical errors made with compiler intrinsics. The goal of a library should be an intuitive and straightforward API that can be quickly grasped and used without a long learning period and considerable effort.

Ultimately, the four factors mentioned above can significantly impact a project's development process and overall success. Thus, we examine the six libraries based on these factors. In addition, for some libraries, we will provide supplementary relevant information.

4.3 Selected Libraries

In total, we carefully selected six libraries. The first four libraries for evaluation: Highway, Vc, Libsimdpp, and NSIMD, were chosen based on their popularity and advanced features. These libraries, which we consider among the most popular SIMD libraries on the market, follow a design philosophy that aims to condense the extension sets' available functions down to a subset of functions.

However, we also included SIMD Everywhere in the evaluation to ensure we examine all possible options. This library differs significantly from the previously mentioned libraries regarding its design principle since it aims to map all intrinsic functions between the SIMD extension sets. Nonetheless, the library has a substantial following and an active community working on advanced features. We also included Pure SIMD, which stood out due to its reliance on auto-vectorization, providing a unique approach different from the other libraries' principles.

4.4 Library Reviews

This section presents a comprehensive review of each selected library by evaluating the criteria mentioned in Section 4.2. Each library's introduction will include an analysis of the supported SIMD extensions, whereas the available functions, documentation, and ease of use are split into separate subsections. To facilitate easy comparison between the supported SIMD extensions, Table 4.1 presents an overview for each library at the time of writing.

Library	Supported SIMD Extensions
Highway	SSSE3, SSE4, AVX2, AVX-512*, NEON, SVE, SVE2, WASM SIMD; RISC-V V
Vc	SSE2, SSE3, SSE4.1, SSE4.2, AVX, AVX2
Libsimdpp	SSE2, SSE3, SSSE3, SSE4.1, AVX, AVX2, FMA3, FMA4, AVX512*, NEON, NEONv2, Altivec, VSX v2.06, VSX v2.07, MSA
NSIMD	SSE2, SSE4.2, AVX, AVX2, AVX512*, NEON, SVE, VMX, VSX
SIMD Everywhere	Full Support: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, AVX, AVX2, FMA, GFNI, CLMUL, XOP, SVML, SIMD128 Partial Support: SSE4.2, AVX512*, NEON, SVE, AMX, AES
Pure SIMD	Dependent on compilers auto-vectorization support

Table 4.1: Supported SIMD extension sets by each library. *Multiple AVX512 versions are supported.

4.4.1 Highway

The first library we will examine is Highway [Goo22c], a SIMD library for C++11 and above, developed by Google and distributed under an Apache 2.0 license. Development first started in 2017 and has been ongoing since with frequent updates. This library has become more popular than others before it due to the high code quality and advanced feature set. Additional information will be presented on Highway's type system, as the library will be used in a case study in Chapter 6.

The motivation behind Highway's creation is intrinsic programming's portability and compatibility struggles, which we discussed in Section 3.2.1. Highway aims to address these issues by allowing the same code to be run on various platforms and architectures, eliminating the need for multiple error-prone intrinsic implementations. As presented in Table 4.1, Highway supports the most popular SIMD extensions for x86 and ARM architectures. Therefore, offering a solid development basis. Another aim of the developers was for the library's performance not to differ more than 10% to 20% from that of a handwritten intrinsics implementation on the corresponding platform [Goo22d, Goo22a].

While Highway is a promising solution to the struggles of SIMD programming, it remains to be seen if this library will be a long-term one because Google has a track record of abruptly abandoning projects [Wik23].

Type System

The type system of compiler intrinsics for extensions, such as AVX2, is missing type safety, making bugs hard to find. However, Highway provides type safety with its numerous vector types. These numerous vector types are nonetheless easy to navigate since the library provides constructs that choose the optimal vector type for the selected datatype and current target [Goo22e].

The code in Listing 4.1 presents the most vital functions of Highway's type system, which

are usually found in the initial portion of code utilizing Highway.

```
1 const ScalableTag<T> d;
2 const size_t N = Lanes(d);
3 using V = decltype(Zero(d));
```

Listing 4.1: Essential functions in Highway's type system.

The first line of code (see line 1) in Listing 4.1 does the following. `ScalableTag` refers to a zero-sized tag type of the form `Simd<T, N, kPow2>` where `T` is the type of the vector's lanes, e.g., `uint32_t` or `float`, and `N` is the number of lanes the vector should have. However, we do not refer directly to the zero-sized tag type since we want our code to be portable. I.e., we should not assume the number of lanes the vector has because we do not know the SIMD register size before the SIMD extension set is selected at compile time. Therefore, we cannot set a value for `N` while writing the code. Thus, the `ScalableTag` type is supplied with a type `T`, for example, `float`, which will then be used at compile time to choose the correct number of vector lanes. The lvalue `d` can then be referred to when creating new vectors to tell the overloaded functions, e.g., `Set` or `Zero`, which type of vector should be created.

The function, `Lanes`, in line 2 makes it possible to retrieve the number of lanes the vector has from the lvalue `d` and stores the scalar value in `N` [Goo22e]. This function is, for example, necessary to increment loops according to the number of vector lanes the current platform provides.

The third and final line (see line 3) extracts the vector type associated with the lvalue `d`. It makes it possible to refer to it as, for example, `V` [Goo22e]. This feature is helpful because, as mentioned above, we should not assume the exact vector type when writing the code.

Functions

The Highway library consists of various overloaded functions, meaning that a function has the same name for every possible type of vector it operates on [Goo22e]. This stands in contrast to compiler intrinsics, where several functions are used for the same operation but differ in the data type on which they operate (compare Section 2.4) [Int22a].

The functions available in the Highway library make it possible to write everything that could be written with intrinsics. They span from functions for initialization, loading, storing, arithmetic, and bitwise operations, to functions for logical operations and masks. If platform-specific code is still necessary, this can be achieved by writing them inside a directive the preprocessor resolves. For example, the if-directive, `#if HWY_TARGET == HWY_NEON`, can be used to declare a NEON-specific implementation [Goo22e, Goo22a].

The library also offers convenient functions for initializing vectors and other use cases, such as the `Iota` function. It returns a vector in which each lane differs from the one next to it by a specified value [Goo22e]. This feature is handy in various applications, and we will use it in the Highway implementation of the Mandelbrot benchmark (see Section 5.4.5). Achieving the same functionality using intrinsics requires more thought because the programmer needs to find the right combination of intrinsic functions to create an `Iota` function [Int22a]. Having this feature built into the library improves usability significantly. Another useful function, missing in most compiler intrinsics libraries, is the `PopulationCount` function which counts the number of bits with the value 1 of an integer in each lane. In Section 6.3, we will closely examine Highway's population count implementation.

Documentation

The documentation of Highway can be found in a folder [Goo22b] on the library's GitHub page. This folder offers documentation on various aspects of the library.

It includes details on the design philosophy behind Highway and the library's aims. Another beneficial file is the FAQ (Frequently Asked Questions) file, which is especially helpful when just starting with Highway. More experienced Highway users may also be interested in the internal implementation documentation of Highway in case they want to add functionalities themselves or verify implementation details.

For most users, the most important document will be the quick reference guide. This guide includes all function definitions, instructions on how to set namespaces, and more. However, this guide could be improved. One problem is that the guide is a simple Markdown file which makes navigating from one function to the next tedious and tiring [Goo22e]. Libraries like Libsimdpp and Vc solved this problem better by providing a dedicated web page displaying all the available functions in a more structured manner [Lib22b, Vc22b].

Ease of Use

In this section on ease of use, we will evaluate Highway in comparison to other SIMD libraries and a compiler intrinsics approach for code vectorization.

Getting started with compiler intrinsics is relatively easy. First, one must include the appropriate header file for the desired intrinsics, like `immintrin.h` for most x86 SIMD extensions [Int22a]. The next step requires setting a compiler flag like `-march=native` to allow platform-specific code generation. Afterward, one can use intrinsic functions and types. However, finding and choosing the correct functions can be a significant hurdle.

Getting to the same point with Highway is more complicated. First, the library must be compiled, linked using the typical compiler flags, and included. The library offers multiple header files, allowing users to choose desired additional features dynamically. However, for some includes a specific order is important. Another requirement is the proper setup of the Highway namespaces, which is relatively intricate [Goo22e]. These steps can make it more difficult for first-time users to get started with this library compared to starting with intrinsics. However, due to this intricate process, the user has flexibility regarding which features they want to include. This greater flexibility can be necessary or favorable for experienced users. Therefore, it is not necessarily a negative.

However, once the setup process is finished, using Highway is significantly easier than using intrinsics. Contributing to this are the previously mentioned overloaded functions which make it easy to remember names and the ability to change the lane datatype quickly.

Another advantage of Highway compared to other libraries and compiler intrinsics is that inexperienced users do not need actively select a SIMD extension set because Highway will choose the best available SIMD extension dynamically when using `-march=native` and only resort to a different extension if specified by the user [Goo22c]. In most other SIMD libraries like Libsimdpp 4.4.3, a macro is required at compile time to actively select the desired SIMD Extension.

4.4.2 Vc

The Vc library [Vc22a] was one of the first C++ SIMD libraries. It was developed at the Goethe University Frankfurt, Germany. This library aims, like Highway, to provide an

efficient way to vectorize code across platforms while introducing no additional overhead compared to code written with compiler intrinsics. The library requires C++11 and above and is published under a BSD 3-Clause license.

The development of Vc appears to have stopped, but pull requests with bug fixes from the community are still being reviewed. However, there seem to be plans to create Vc 2.0, based on the experimental std-simd library [SS21, Vc22a], which has been included in GCC/stdlibc++ from GCC version 11 onwards. It is unclear if these plans are still being pursued. The library currently lacks support for many newer SIMD Extension sets, e.g., SVE and AVX512, and only supports x86 SIMD extensions (compare Table 4.1) [Vc22a].

Functions

The library is highly advanced from a functionality standpoint because it provides all functions necessary to vectorize code, that could also be vectorized using intrinsics. Additionally, the library includes a couple of high-level math and utility functions, improving the usability for math and other problems profiting from vector parallelism [Vc22b].

Some examined libraries, one of them being Vc, allow the users to use operators such as `+` or `*` instead of using intrinsics functions like `_mm256_add_ps` or `_mm256_mul_ps` [Vc22b]. This functionality has the benefit that users unfamiliar with the typical syntax of vectorized code will have little trouble understanding the operations because they are the same as for scalar code.

Google's Highway library also offers this functionality, although this feature does not work on SVE and RISC-V V machines for some operations [Goo22e]. Therefore, using it should be reconsidered in order to ensure maximum portability.

Ease of Use

Starting with Vc is more straightforward than starting with Highway because the program structure is simpler than Highway since it does not require a complex include sequence or namespaces. Additionally, the library's GitHub page [Vc22a] includes various valuable examples and a detailed setup guide, making it easy to get started.

Documentation

The documentation for the Vc library is also superior to the documentation of Google's Highway library because, unlike Highway, it relies on a dedicated webpage (see [Vc22b]) to efficiently present all required information in a more structured form.

4.4.3 Libsimdpp

Unlike the previous two libraries, Libsimdpp [Lib22a] is not developed by a company or university but rather by the open-source community. Nevertheless, the goals are similar to the already mentioned libraries. Libsimdpp aims to provide a zero-overhead header-only API, allowing users to create portable vectorized code for the most popular SIMD extensions (compare Table 4.1) by providing a wrapper for the underlying compiler intrinsics functions. Libsimdpp offers a C++11 and a C++98 version. The library is published under the Boost Software License 1.0.

The active development of the library appears to have finished for now, but work on bug fixes continues.

Functions

Libsimdpp provides a great variety of overloaded functions and operators for the custom vector types provided by the library. Making it possible to solve most, if not all, problems by employing this library.

However, not all functions are implemented as nicely as in Highway. For example, Libsimdpp is missing an `Iota` and `PopulationCount` function as well as others, which we will discuss more closely when looking at the implementation of the Mandelbrot Benchmark in Section 5.4.6. To replicate an `Iota` function, the use of, for example, the `make_float` function is necessary [Lib22b]. The problem with this function is that it is vector length specific. So it is necessary to provide multiple implementations for different vector lengths. Therefore, the use of macros and if-directives is required.

Ease of Use

The library has a simple layout making it very easy to get started. Since Libsimdpp is a header-only library, no build process is required. Thus the first steps with it are straightforward. Moreover, no complicated include sequences or multiple namespaces are necessary, unlike with Highway. Additionally, the vector types and functions are clearly and intuitively named. Thus, Libsimdpp joins Vc in the list of libraries that have an easy and quick setup process.

Documentation

Of all the libraries discussed, the documentation of Libsimdpp [Lib22b] is by far the best. Like Vc, this library has a dedicated web page. However, the design of this web page is superior to that of the Vc documentation because it matches the design of the popular cppreference.com [Cpp23] page many C++ programmers are familiar with, making it easy to navigate.

4.4.4 NSIMD

Like the previous libraries, NSIMD [NSI21a] offers the ability to abstract SIMD programming and aims to provide a zero-cost SIMD abstraction library by relying on the inline optimization pass of the compiler. The core of this library originates from the boost.simd library [NSI18], which is no longer available but was benchmarked using a Mandelbrot benchmark in Pohl et al.'s paper [PCM+16] on SIMD C++ programming models. NSIMD is unique among the libraries because it also supports GPU programming models like NVIDIA CUDA. Besides the GPU support, the library offers support for most of the popular extensions on x86 and ARM systems, see Table 4.1.

Another distinct difference between this and the other libraries is that multiple APIs are included. There is a base and an advanced API version for C and C++. However, both C++ APIs wrap the C calls but offer operator overloading and higher-level type definitions [NSI21a]. The APIs support different C and C++ standards, with support for C89, C11, C++98, C++11, C++14, and C++20. The library is available under an MIT license [NSI21a].

Functions

This library is also well-equipped with a wide variety of functions that can handle most tasks. However, although NSIMD supports SVE, there is no population count function which is a standard SVE feature. Nonetheless, the library offers classic library functions such as `iota` or more advanced mathematical functions, such as a base-ten logarithm function [NSI21b]. Overall, the variety of functions will be more than enough for most projects, though problems could arise if, for example, a population count function is required as in the algorithm described in Chapter 6.

Ease of Use

Getting started with this library is quick and easy and requires less work than, e.g., starting with Highway. However, one major drawback is that no examples are included with the library. Thus, the setup process may be more challenging for first time users of SIMD libraries. Additionally, we have been unable to use the library on an ARM CPU supporting SVE. It remains unclear whether the issue lies with the library itself or if we made an error. Due to the brief usage section, we were unable to identify the cause of the issue.

Documentation

The library offers a dedicated web page [NSI21b] for its documentation. Overall, the documentation provides incredible detail on the library's functions. However, with the different APIs and the detailed description of each function, it can become problematic for users to understand what input a function needs. Therefore, providing the ability to hide some implementation details from the user may be beneficial for better readability and easier use. Although the function section of the library is very detailed, the usage section is relatively thin and may have indirectly led to the issue with SVE discussed in the previous section (see Section 4.4.4).

4.4.5 SIMD Everywhere

SIMD Everywhere is a C99 header-only library [SE23], which follows a different design principle than the previously discussed libraries. SIMD Everywhere aims to map all compiler intrinsics between the various extensions and not condense down the extension sets to their commonalities. Thus, this library's scope is much larger than the others. However, this also shows in the supported SIMD extensions. As seen in Table 4.1, many of the more recent extension sets are not fully supported yet, since they are still under development. The library is published under an MIT license, like NSIMD [SE23].

Functions

The library does not offer custom functions but relies on the functions provided through compiler intrinsics. So, for example, code utilizing AVX2 compiler intrinsics can be compiled to use NEON intrinsics. The process of doing this is relatively easy. First, we need to include a header file provided by the library for the compiler intrinsics we use instead of the native library like, for example, `immintrin.h`, which includes most of the intrinsics available on x86 CPUs [SE23, Int22a]. Additionally, we must define `SIMDE_ENABLE_NATIVE_ALIASES` so that

SIMD Everywhere can use native compiler intrinsic function names. Otherwise, all intrinsic functions and types must be prefixed with `simde_`, which is generally recommended due to possible portability issues in the API [SE23].

Ease of Use

The ability to use preexisting compiler intrinsics code with this library makes it incredibly easy for developers already familiar with them. Otherwise, the developers will have to learn the intrinsic functions and types of one SIMD extension to use this library because the code can be compiled from one base implementation to any other supported SIMD extension. Overall, using this library is simple and differs only marginally from programming with intrinsics.

Documentation

The documentation on the library's GitHub page [SE23] is relatively short because all the library's functions are the same as the intrinsic ones. Therefore, the documentation provided by hardware manufacturers can be used as guidance, for example, the Intel intrinsics guide, see Ref. [Int22a].

The rest of the library's documentation provides details on how to use it and opportunities for contribution, besides detailed information on the development state of the various SIMD extensions supported [Int22a].

4.4.6 Pure SIMD

Pure SIMD [PS21] is a header-only library that differs significantly from the other libraries because it does not wrap the underlying compiler intrinsics. It rather unrolls loops introduced through vector operations at compile time and uses code patterns the compiler can then easily vectorize [PS21]. The main advantage of this concept is that the library supports vectorization for every SIMD extension, with a compiler supporting auto-vectorization. Therefore, this library is by far the most portable. However, at the time of writing, this library remains largely unfinished.

Nonetheless, this approach to code vectorization is quite compelling and will, therefore, be included in the benchmarks. The library uses C++17 and requires a compiler that supports SLP (superword-level parallelism) vectorization, which combines similar independent instructions into vector instructions [PS21, LLV23]. The library is published under a BSD 3-Clause license.

Functions

As discussed in the previous section, the library still needs to be completed, and it is unclear if it will ever be. Moreover, in the current state, the library is missing various functions required in most programs. So most algorithms will have to rely on scalar sections in their implementation, limiting the upside the vectorization process can provide. Thus, we cannot endorse the functional extent of this library.

Ease of Use

The setup process is uncomplicated since it is a header-only library. Therefore, no complex build process is required, making the first steps with this library straightforward [PS21]. Furthermore, the existing functions are named intuitively. Additionally, the library includes overloaded operators, e.g., + and * (compare Listing 5.3) for vector types, similar to Vc (compare Section 4.4.2). Overall, getting started is easy and quick, further aided by numerous examples in the documentation.

Documentation

The documentation is relatively short, but that is to be expected due to the limited number of features. Overall it offers a good overview of the library's functionality and addresses areas where further development is necessary [PS21]. Moreover, it includes elaborate examples for all the functions and advice on structuring code. In summary, the documentation is sufficient for what the library offers.

4.5 Review Results

After reviewing the six libraries on their extensions, functions, documentation, and ease of use, it becomes clear that while there are many similarities, there are also some distinct differences. Thus, we highlight the strengths and weaknesses of each library in this section. Table 4.2 presents an overview indicating which criteria a library satisfies.

Library	Supported Extensions	Functions	Documentation	Ease Of Use
Highway	✓	✓	✓	✗
Vc	✗	✓	✓	✓
Libsimdpp	✓	✗	✓	✓
NSIMD	✓	✓	✗	✓
SIMD Everywhere	✓	✓	✓	✓
Pure SIMD	✓	✗	✓	✓

Table 4.2: Summary of the different review categories for each library. The ✓ symbol indicates that the library fulfills the expectations in the category; on the other hand, an ✗ symbol indicates the opposite.

For Google's Highway library, we critiqued that getting started was more challenging than with the other libraries due to the modular approach, which required more intricate knowledge of the library. However, when the initial hurdles are overcome, Highway's more flexible approach can become helpful, especially if more advanced functionality is required. Another critique was the documentation since it consists of multiple markdown files rather than a dedicated webpage. Though, it is necessary to note that the information provided in the files is more than sufficient. Thus, the library satisfies the criteria.

The drawback for Vc is relatively straightforward. As a result of its age, it does not support newer extension sets, such as AVX512 or SVE. Thus, the library is limited to older x86 CPUs.

Libsimdpp impressed in many of the categories. The only critique we have for this library is concerned with the available functions. While the library has most of the required functions, it is missing a handful of more advanced functions, which would be highly desirable, such as an iota and population count function.

The fourth library, NSIMD, has many convincing aspects, but its documentation could be more optimal. It is convoluted and overly specific in many places, while other areas need to be more detailed. Therefore, the documentation would benefit from a more concise approach in many areas as well as some additional information in its usage section.

SIMD Everywhere fulfilled the expectations in all categories.

The final library we reviewed is Pure SIMD, which is the only library that solely relies on auto-vectorization features of the compiler for its vectorization. Many aspects of the library are well executed. However, it lacks many functions, which can be considered essential for a SIMD library.

In summary, all libraries could satisfy most of the review criteria. However, all of them had different strengths. Some shined due to their excellent documentation, others due to their well-thought-out functions. We have found Highway, Libsimdpp, NSIMD, and SIMD Everywhere especially convincing.

5 Mandelbrot Benchmark

The Mandelbrot benchmark is based on the calculation of the Mandelbrot set, named after the french-american mathematician Benoît Mandelbrot. We will use this benchmark to examine and compare the performance of the different libraries we have previously discussed (see Chapter 4). First, we will look at the definition of the Mandelbrot set before going over the different implementations.

5.1 Introduction to the Mandelbrot Benchmark

The Mandelbrot set is the set of complex numbers c for which $f_c(z_n) = z_n^2 + c = z_{n+1}$ does not diverge when iterated from $z_0 = 0$. In other words, the distance of $f_c(z_n)$ to the origin $(0, 0)$ is bounded, meaning the distance of $f_c(z_n)$ to the origin does not approach infinity under iteration. For the Mandelbrot set, it can be assumed that if the Euclidean norm of z_n exceeds 2, then c is not in the Mandelbrot set.

Plotting the Mandelbrot set on the complex plane creates a distinct shape as can be seen in Figure 5.1. The calculations required to plot this image, are the Mandelbrot benchmark.

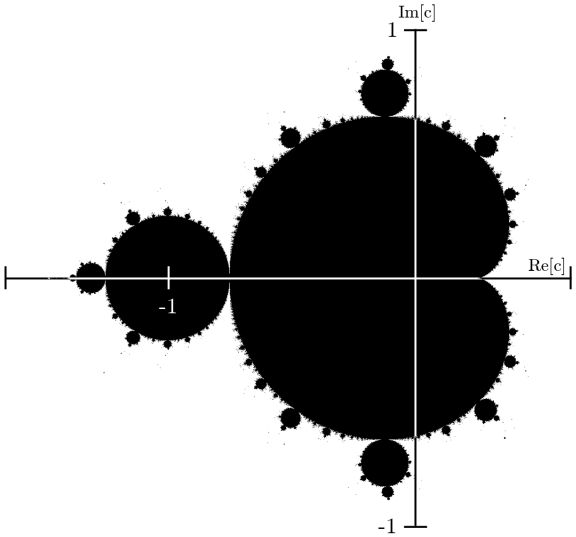

Figure 5.1: Graphical representation of the Mandelbrot set generated by the AVX2 implementation of the Mandelbrot benchmark overlaid onto a coordinate system of the complex plane.

Every pixel in Figure 5.1 represents a value of c in the complex plane; if c is in the Mandelbrot set, then the pixel is black. Otherwise, it is white. Some visualizations use more than two colors, each visualizing a specific behavior under iteration. The resulting images are famous for their intricate, fractal patterns, which exhibit self-similarity at different scales.

5.2 Pseudocode Implementation

The following section includes a pseudocode implementation in Algorithm 1 representing the Mandelbrot benchmark. The benchmark takes seven parameters, as shown in line one of the algorithm. The parameters $xBegin$, $xEnd$, $yBegin$, and $yEnd$ are the x and y dimensions of the complex plane area we want to test for numbers in the Mandelbrot set. The last three parameters, $width$, $height$, and $image$, provide information about the resolution of the image and a pointer to the image.

Algorithm 1 Mandelbrot benchmark

```
1:  procedure MandelBench(xBegin, xEnd, yBegin, yEnd, width, height, image)
2:      xScale ← (xEnd − xBegin)/width
3:      yScale ← (yEnd − yBegin)/height
4:
5:      for j ← 0, j < height, j ← j + 1 do
6:          for i ← 0, i < width, i ← i + 1 do
7:              c ← (xBegin + i ∗ xScale, yBegin + j ∗ yScale)
8:              z ← (0, 0)
9:
10:             iteration ← 0
11:             while true do
12:                 iteration ← iteration + 1
13:                 z ← (z ∗ z) + c                    ▷ Mandelbrot Iteration Rule
14:
15:                 if norm(z) > 2 then
16:                     image[j ∗ width + i] ← 0
17:                     break
18:                 else if iteration >= MAXITER then
19:                     image[j ∗ width + i] ← 1
20:                     break
21:                 end if
22:             end while
23:         end for
24:     end for
25: end procedure
```

For each pixel in our image, we choose a value for c and test if it is included in the set through iteration. Therefore, we need to calculate a different x and y value for each pixel corresponding to its position within the complex plane. In lines 2 and 3 of Algorithm 1, we calculate the distance of each pixel's position in the complex plane to the one next to it. This value is then used in line 7 to set the value for c.

The iteration rule for the Mandelbrot set can be found in line 13. After each iteration, we check if one of two possible cases has occurred; if so, we can stop the iteration and move to the next pixel. The first case is that z_n will approach infinity under iteration, which we can check by calculating the Euclidean norm of z and testing if it is greater than 2, which is done in line 15. If the norm is greater than 2, c is not in the Mandelbrot set. The other case is that we have reached the maximum number of iterations we have defined (see line 18).

When reaching that case, it is clear that the distance of z has stayed finite to the origin, and we include c in the Mandelbrot set. Consequently, the number of iterations chosen determines the accuracy of the calculated set and graphical representation because the norm of $z_{MAXITER}$ may be less than 2, but this cannot guarantee that z would continue to stay less if the iteration continued.

In this implementation of the Mandelbrot benchmark, we set the value of a pixel to zero if c is not in the set and to one if it is in the set. By collecting this data, we can create a bitmap image resembling Figure 5.1.

As seen in Algorithm 1 and the presented definition of the Mandelbrot set in Section 5.1, we can calculate the value of each pixel individually, i.e., the value of each pixel is independent of all other pixels. Hence, calculating the image is highly data parallel making it an excellent and common choice for a SIMD benchmark.

5.3 Scalar C++ Implementation

The scalar C++ implementation of the Mandelbrot benchmark in Figure 5.1 is quite similar to the pseudocode implementation discussed before. However, one significant difference is that we are not relying on a complex number type. Instead, we treat the complex number's real and imaginary parts as two separate floating point values. We used this simplification because the std::complex class provides a variety of assurances, which we do not need in this application, and which also affects the performance negatively [PCM+16]. Furthermore, testing has shown that using the std::complex class or a custom class for complex numbers can generally decrease the compiler's ability to perform auto-vectorization, leading to a slightly worse performance due to the more complex code structure. Additionally, most SIMD extension sets have no support for complex number types. Therefore, relying on a 32-bit floating point type makes sense across all benchmarks.

```
1  void mandelbrot_scalar(float xBegin, float xEnd,
2                         float yBegin, float yEnd,
3                         int width, int height, float * image) {
4
5      // calculate horizontal and vertical distance between pixels
6      float xScale = (xEnd - xBegin) / width;
7      float yScale = (yEnd - yBegin) / height;
8
9      float c_real, c_imag, z_real, z_imag, temp, z_real_squared,
           z_imag_squared, norm_squared;
10
11     for (int j = 0; j < height; j++) {
12         // imaginary part of c
13         c_imag = yBegin + j * yScale;
14         for (int i = 0; i < width; i++) {
15             // real part of c
16             c_real = xBegin + i * xScale;
17
18             //initialization of z
19             z_real = 0.0f;
20             z_imag = 0.0f;
21
22             int iteration = 0;
23             while (1) {
24                 iteration++;
```

```
25
26                  // iteration of f(z) = z^2 + c
27                  temp = z_real * z_imag;
28                  z_real_squared = z_real * z_real;
29                  z_imag_squared = z_imag * z_imag;
30                  z_real = (z_real_squared - z_imag_squared) + c_real;
31                  z_imag = temp + temp + c_imag;
32
33                  // no square root to reduce number of instructions
34                  norm_squared = z_imag_squared + z_real_squared;
35
36                  // breakout condition 1: squared norm larger than 4.0
37                  if (norm_squared > 4.0f) {
38                    *image++ = 0.0f;
39                    break;
40                  }
41
42                  // breakout condition 2: reached max iterations
43                  if (iteration >= MAX_ITERATIONS) {
44                    *image++ = 1.0f;
45                    break;
46                  }
47              }
48          }
49      }
50  }
```

Listing 5.1: Scalar C++ Mandelbrot benchmark implementation.

The implementation in Figure 5.1 is very similar to the pseudocode implementation discussed in Section 5.2. However, one crucial difference exists in how the Euclidean norm is calculated. The Euclidean norm is defined as $||z|| = \sqrt{Re(z)^2 + Im(z)^2}$. We already calculated the square of the real and imaginary part of z during the iteration of $f_c(z_n)$ (compare line 28 and 29). Thus, we would only need to calculate the square root of the sum of the two numbers to receive the Euclidean norm. However, we square the bailout value instead of calculating the square root. I.e., we use 4 instead of 2 in the first breakout condition's comparison (compare line 37). As a result, we do not need an additional square root operation.

5.4 Vectorized Implementations

The algorithm's vectorization occurs along the x-axis, with the vector register's lanes computing results for adjacent pixels in a single row. Overall, the vectorized implementations all share the same core elements and structure. Therefore, we will focus on specific aspects of each library's code rather than presenting the entire codebase. However, to ensure comprehension and maintain coherence, we have organized the code snippets chronologically, making it easy to understand where each piece fits within the algorithm.

5.4.1 Intrinsics

The first implementation we will look at is one of the compiler intrinsics versions. In total, there are four versions of the benchmark using intrinsics explicitly. The first two versions are written for x86 CPUs using the AVX2 and AVX512 SIMD extensions. The other two

implementations use SVE and NEON as their SIMD extension set for ARM CPUs. The following code snippet is part of the AVX2 implementation, see Listing 5.2.

```
1  ...
2  // for-loops and other variable initializations
3  ...
4
5  __m256 c_imag = _mm256_set1_ps(yBegin + (j * yScale));
6  __m256 c_real = _mm256_set_ps(8+i, 7+i, 6+i, 5+i, 4+i, 3+i, 2+i, 1+i);
7  c_real = _mm256_fmadd_ps(c_real, xScaleVec, xBeginVec);
8
9  __m256 z_real = _mm256_setzero_ps();
10 __m256 z_imag = _mm256_setzero_ps();
11
12 ...
13 // while-loop and end of algorithm
14 ...
```

Listing 5.2: AVX2 compiler intrinsics Mandelbrot benchmark; initialization of c and z_0.

The snippet shows the initialization of the real and imaginary parts of c and z, which corresponds to lines 13 to 20 in the scalar C++ implementation in Listing 5.1. Setting the initial value of z is easily achieved by setting the values of the z vectors to zero using a so-called splat operation, which broadcasts the value 0 across the vector register. This splat operation, `_mm256_setzero_ps`, can be seen in lines 9 and 10.

It is equally simple for the imaginary part of c, the value of c only changes when a new line of the image's pixels is calculated. Therefore, we can move line 5 out of the innermost for-loop (compare Listing 5.1), which has been done for the benchmark implementations to avoid unnecessary redefinitions of `c_imag`. The calculation of `c_imag` is identical to the pseudocode and scalar implementation (compare Sections 5.2 and 5.3). However, the initialization is different because the intrinsics function `_mm256_set1_ps` broadcasts the calculated value to all lanes.

The initialization of `c_real` is more complex than that of `c_imag`. Because we no longer work with scalar values, we do not increment `i` by 1 in every iteration of the for-loop. Instead, we increment `i` with the number of lanes of the vector. Thus, the function `_mm256_set_ps` is needed, which allows setting the value of each lane in a vector individually from the back to the front (compare line 6). After initializing the vector's values with its x-axis position in the image, we need to multiply `c_real` with our x-axis scaling to receive a vector with correctly spaced real parts of c. Now, we only need to add the x-position of the beginning on the complex plane to shift the `c_real` vector into place. On many AVX2-enabled machines, there is an FMA unit, which allows multiplication followed by an addition in a single instruction, which is more efficient than a separate multiplication and addition instruction [Int22a]. We utilize the function, `_mm256_fmadd_ps` in line 7 to extract the best possible performance.

5.4.2 Pure SIMD

The Pure SIMD library offers an `iota` function like a few of the other libraries [PS21, Goo22e]. Listing 5.3 displays the initialization of the real part of c, which is similar to the process in Listing 5.2. However, the existence of an `iota` function simplifies this because it is only necessary to give the function a start value and a step size between the scalar values of each lane, compare line 5 in Listing 5.3 and line 6 in Listing 5.2.

The subsequent scaling and shifting of `c_real` into position, displayed in line 6 is comparable to line 7 in Listing 5.2. As seen in the listing, the library offers overloaded operators, making it possible to use `+` and `*` on vector types.

```
1  ...
2  // for-loops and other initializations
3  ...
4
5  auto c_real = iota<TargetVec, size_t>(i, 1.0f);
6  c_real = (c_real * xScaleVec) + xBeginVec;
7
8  ...
9  // z initialization, while-Loop, and end of algorithm
10 ...
```

Listing 5.3: Pure SIMD Mandelbrot benchmark; initialization of *c_real*.

5.4.3 NSIMD

The NSIMD library offers multiple APIs for C and C++. We have implemented the benchmark using the C++ base API and the C++ advanced API. Therefore, we will compare the same snippet from both implementations to see their differences. The two code snippets in Listing 5.4 and 5.5 both show the initialization of z_0, which can also be seen in Listing 5.2.

```
1  typedef pack<float> floatv_t;
2  typedef packl<float> maskv_t;
3
4  ...
5  // c initialization and for-loops
6  ...
7
8  floatv_t z_real = set1<floatv_t>(0.0f);
9  floatv_t z_imag = set1<floatv_t>(0.0f);
10
11 ...
12 // while-loop and end of algorithm
13 ...
```

Listing 5.4: NSIMD C++ advanced API Mandelbrot benchmark; initialization of z_0.

The code displayed in Listing 5.4 uses the C++ advanced API of the library. The one in Listing 5.5 uses the C++ base API. The type definition of the advanced API is similar to Highway and allows the creation of a datatype that carries most of the information, compare lines 1, 8, and 9. For the base API, we must choose from various vector data types and input the datatype into every function.

```
1  ...
2  // c initialization and for-loops
3  ...
4
5  vf32 z_real = set1(0.0f, f32());
6  vf32 z_imag = set1(0.0f, f32());
7
8  ...
9  // while-loop and end of algorithm
10 ...
```

Listing 5.5: NSIMD C++ base API Mandelbrot benchmark; initialization of z_0.

5.4.4 Vc

The code snippet displayed in Listing 5.6 shows the iteration of z. These instructions are equivalent to a complex multiplication and a subsequent addition. By relying on Vc's feature to use operators such as + and - on vectors, this code is almost identical to lines 26 to 31 in the scalar C++ implementation in Listing 5.1, with the only difference being the data types.

```
1  ...
2  // while-loop and initializations
3  ...
4
5  float_v z_real_squared = z_real * z_real;
6  float_v z_imag_squared = z_imag * z_imag;
7  float_v temp = z_real * z_imag;
8
9  z_real = (z_real_squared - z_imag_squared) + c_real;
10 z_imag = temp + temp + c_imag;
11
12 ...
13 // breakout conditions and store routine
14 ...
```

Listing 5.6: Vc Mandelbrot benchmark; iteration of $f_c(z_n)$.

5.4.5 Highway

In the Highway implementation (see Listing 5.7), we will look at the breakout condition of the while-loop (comparable to lines 33 to 46 in Listing 5.1). As discussed before, there are two conditions under which we store the results and break out of the while loop. The first is the condition that we have reached the maximum number of iterations set. The second and more challenging one is checking if the squared norm of z is greater than 4. Because we operate on vectors, the breakout condition changes slightly from the scalar version. The while-loop will only stop when all lanes of the vector unit report a squared norm greater than 4 or the maximum number of iterations has been reached.

```
1  ...
2  // while-loop
3  // f(z) iteration
4  ...
5
6  auto norm_squared = Add(z_real_squared, z_imag_squared);
7  auto mask = Lt(norm_squared, bailoutVec);
8
9  if (iteration >= MAX_ITERATIONS || AllFalse(d, mask)) {
10     ...
11     // create and save result
12     ...
13 }
14
15 ...
16 // end of algorithm
17 ...
```

Listing 5.7: Highway Mandelbrot benchmark; calculation of the squared norm and breakout condition.

First, we need to calculate the squared norm, which can be done simply by adding the square of the real and the square of the imaginary part of the complex number z using the function `Add`, as shown in line 6. The second step requires the creation of a mask. Masks are also vectors and consist of the same number of lanes as their corresponding vector, but they usually use a different kind of register on the hardware [Int22a]. Each lane of a mask can only have two possible values, 0 or 1, which makes masks comparable to a vector filled with booleans. The value 1 indicates that the corresponding vector lane satisfies a condition and 0 when it does not. In line 7, we initialize the mask with the condition that the squared norm is less than the bailout value of 4, using the function `Lt`. The following if-condition checks whether the maximum number of iterations has been exceeded or the mask only contains the false value, using the function `AllFalse`. If one of the conditions is true, we create a result vector, save it in the `image` array, and break out of the while loop.

5.4.6 Libsimdpp

The extracted code in Listing 5.8 for the Libsimdpp library displays the same part of the benchmark as the discussed Highway code in Section 5.4.5. However, due to the less extensive number of functions available in this library, a combination of operations must be used to achieve the same result.

```
1  ...
2  // while-loop
3  // f(z) iteration
4  ...
5
6  float32<N> norm_squared = add(z_real_squared, z_imag_squared);
7  mask_float32<N> mask = cmp_lt(norm_squared, bailoutVec);
8  float32<N> result = blend(oneVec, zeroVec, mask);
9
10 if (!test_bits_any(result) || iteration >= MAX_ITERATIONS) {
11     store(image + i + j*width, result);
12     break;
13 }
14 ...
15 // end of algorithm
16 ...
```

Listing 5.8: Libsimdpp Mandelbrot benchmark; calculation of the squared norm and breakout condition.

The calculation of the squared norm with the Libsimdpp function `add` and the resulting mask, calculated with `cmp_lt`, is similar to the Highway implementation in Section 5.4.5. However, unlike Highway, Libsimdpp does not offer an `AllFalse` function to evaluate a mask [Lib22b]. Though, it does offer the function `test_bits_any`, which returns `false` if no bits are set and `true` if any bits are set. By negating the result of this function, it performs like `AllFalse`, except that `test_bits_any` does not accept a mask type as its input [Lib22b, Goo22e]. Thus, we need to create a vector from the mask. Due to the lack of a type conversion function from mask to vector, we must use the `blend` function, which combines two vectors according to the provided mask [Lib22b]. By blending two vectors, of which one is 0 on all lanes, and the other contains the value 1 across all lanes, we receive an equivalent vector to our mask, compare line 8 of the code snippet. The created vector is also our result if it satisfies the breakout conditions.

This snippet shows nicely how a missing library function can sometimes lead to more complex code and the necessity for more instructions to be performed, thus hurting the performance. For example, the `blend` function in line 8, which we did not need in the Highway version, must be executed every iteration of the while loop, impacting performance negatively compared to the Highway implementation in Section 5.4.5.

5.4.7 SIMD Everywhere

As discussed in Section 4.4.5, the SIMD Everywhere library differs from most other SIMD libraries. Because it can run code written with, for example, AVX2 compiler intrinsics on a machine that does not natively support the extension [SE23].

As a result, the SIMD Everywhere implementation is identical to the code of the AVX2 version from Section 5.4.1, except for including the definition `SIMDE_ENABLE_NATIVE_ALIASES` and including the `simde/x86/avx2.h` header instead of `immintrin.h`. Without the definition, we would have had to prefix every AVX2 type and function with `simde_`, altering the AVX2 implementation [SE23].

6 Case Study Vectorization of Epistasis Detection Algorithm with Highway

In this chapter, we conduct a case study using Google's Highway library to vectorize an algorithm for epistasis detection. The algorithm was developed by Marques et al. at the Instituto Superior Técnico, Universidade de Lisboa, Portugal [MCSJ+22].

Marques et al. benchmarked and optimized the algorithm for CPUs and GPUs. On the CPU side, they tested an AVX2, and two AVX512 implementations [MCSJ+22]. For the case study, we will use Google's Highway SIMD library to create two vectorized implementations and evaluate the performance when targeting AVX2, AVX512, and SVE instructions. We will also explore how Highway manages different AVX512 versions and population count capabilities.

6.1 Problem Introduction

The epistasis detection algorithm can identify gene combinations in the sample data of patients linked to an increased risk of developing certain diseases. Helping healthcare services diagnose them early and provide personalized treatments to prevent or reduce the risk of the disease developing [MCSJ+22].

The process of epistasis describes the interaction of Single-Nucleotide Polymorphisms (SNPs) causing the onset of diseases such as Crohn's. For some diseases, the interaction of two SNPs (second-order epistasis) is enough to cause and identify them. Identifying others can, however, require the consideration of higher-order epistasis. Therefore, it is necessary to evaluate all possible genetic combinations in a data set [MCSJ+22]. The number of gene interactions to examine grows exponentially with the number of gene interactions. As a result, the epistasis detection algorithm, especially when considering higher interaction orders, becomes highly computationally demanding [MCSJ+22].

6.2 Pseudocode Implementation

Marques et al.'s paper [MCSJ+22] describes the epistasis detection algorithm in great detail. Therefore, we will only briefly touch on the most critical aspects of the vectorization of the algorithm. In addition, a pseudocode implementation is displayed in Algorithm 2 as well as a visualization in Figure 6.1, which presents the data sets and main steps of the algorithm.

The vectorization process is relatively straightforward. As seen in Algorithm 2, lines 8 to 13, LOAD operations, which are an essential part of all examined SIMD libraries, are used to retrieve the SNP data. The NOR operations in lines 14 to 16 allow for a memory optimization because every SNP consists of three values (compare Figure 6.1) of which the third can be inferred from the first two by using a NOR operation. We can create the NOR operations by combining the `Or` and `Xor` Highway functions, making them relatively

trivial to vectorize. After the NOR operations, only two other operations are required to fill the frequency table, an AND operation, and a POPCNT (Population Count) operation (compare Figure 6.1). These two instructions evaluate the gene interactions for all the possible genotype combinations. Bitwise AND operations are an essential function of all SIMD libraries and, therefore, trivial to vectorize. The real challenge is the vectorization of the POPCNT operation in line 19 of the algorithm.

Algorithm 2 Epistasis detection on each CPU core, as seen in Ref. [MCSJ+22]

```
1:  procedure EPISTASIS DETECTION(D_{0|1})
2:      for i_0, i_1, i_2 ← 1 to M/B_S do
3:          ft_{0|1} ← 0
4:          for p_0 ← 1 to N_{0|1}/B_P do
5:              for ii_0, ii_1, ii_2 ← 1 to B_S do
6:                  if ii_2 > ii_1 > ii_0 then
7:                      for p ← 1 to B_P do
8:                          X_{0|1}(0) ←LOAD(D_{0|1}[i_0, ii_0, p_0, p, 0])
9:                          X_{0|1}(1) ←LOAD(D_{0|1}[i_0, ii_0, p_0, p, 1])
10:                         Y_{0|1}(0) ←LOAD(D_{0|1}[i_1, ii_1, p_0, p, 0])
11:                         Y_{0|1}(1) ←LOAD(D_{0|1}[i_1, ii_1, p_0, p, 1])
12:                         Z_{0|1}(0) ←LOAD(D_{0|1}[i_2, ii_2, p_0, p, 0])
13:                         Z_{0|1}(1) ←LOAD(D_{0|1}[i_2, ii_2, p_0, p, 1])
14:                         X_{0|1}(2) ←NOR(X_{0|1}(0), X_{0|1}(1))
15:                         Y_{0|1}(2) ←NOR(Y_{0|1}(0), Y_{0|1}(1))
16:                         Z_{0|1}(2) ←NOR(Z_{0|1}(0), Z_{0|1}(1))
17:                         for g_X, g_Y, g_Z ← 0 to 2 do
18:                             ymm_{0|1}(g_X, g_Y, g_Z) ←AND(X_{0|1}(g_X), Y_{0|1}(g_Y), Z_{0|1}(g_Z))
19:                             ft_{0|1}(g_X, g_Y, g_Z) ← ft_{0|1}(g_X, g_Y, g_Z)
                                                          + POPCNT(ymm_{0|1}(g_X, g_Y, g_Z))
20:                         end for
21:                     end for
22:                 end if
23:             end for
24:         end for
25:         return get_score(ft_{0|1})
26:     end for
27: end procedure
```

Population count operations count the number of bits in an integer with the value 1. This function is not always called population count. Sometimes it is called Hamming Weight or the number of set bits. Population count functions have different implementations; some are hardware-based, others algorithmic, i.e., a sequence of instructions. Choosing the correct implementation depends on the use case and the available hardware. Thus, it is of particular importance when vectorizing portable code. Therefore, we will discuss some possible implementations in the following section.

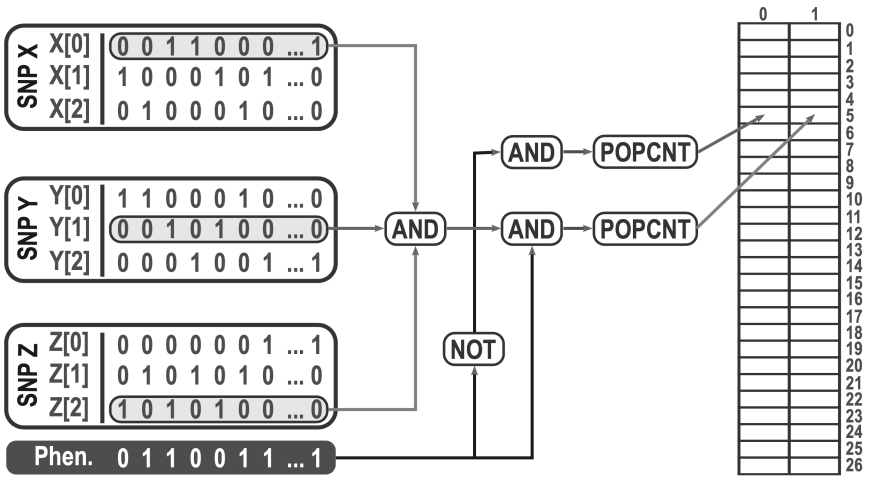

Figure 6.1: Binary representation of the SNP data sets, and the calculation of the frequency table. Visualization from Ref. [MCSJ+22].

6.3 Different Population Count Approaches

There are multiple ways of obtaining a population count on an integer. Some rely on special hardware instructions others use highly optimized algorithms to gather the count.

Many CPUs support hardware instructions, allowing the execution of highly efficient population counts. There are multiple ways of targeting these instructions with different levels of portability. One solution is using a function built into the compiler. For example, GCC and Clang support using `__builtin_popcountl` on 64-bit unsigned integers to obtain a population count. This built-in function will target hardware instructions if they exist [GCC23a, Cla23]. Otherwise, it will rely on a highly optimized algorithm to perform the count, which is part of the libgcc library and named `__popcountdi2` when using GCC, see Ref. [GCC23d]. However, `__builtin_popcountl` is a scalar function and can not operate on vectors. Furthermore, few CPUs offer an intrinsic function for vector population counts. For example, Intel Corporation's Ice Lake-SP CPUs support a 512-bit vector population count, but CPUs using the Skylake-SP micro-architecture do not [Int22a].

As a result, there are multiple ways to perform a population count on a vector. The first and most efficient way is to use a compiler intrinsics function like the AVX512 function `_mm512_popcnt_epi32` available on Ice Lake-SP CPUs. Another approach is implementing a vector algorithm consisting of a combination of compiler intrinsics functions to perform a population count. The third possibility only makes sense in very few cases; here, we extract each vector lane and perform a population count on the scalar values separately. Unfortunately, this approach will make that portion of the code scalar and present a considerable bottleneck.

6.3.1 Pseudocode Vector Population Count with Reduction

The epistasis detection algorithm requires a population count across a vector. Afterward, a horizontal addition sums up all lanes of the vector to a scalar result to save it in the frequency table (see Figure 6.1). A basic pseudocode implementation can be seen in Algorithm 3. The algorithm begins with a vector population count in line 2 and is followed up by an add

reduction, which creates a sum across the lanes. The result of this function is added to a scalar accumulator, which is subsequently used to fill the frequency table (compare Figure 6.1).

Algorithm 3 Vector population count followed by horizontal addition

```
1: procedure POPCNTREDUCE(vec_input)
2:     vec_popcnt ← VEC_POPCNT(vec_input)
3:     result ← REDUCE_ADD(vec_popcnt)
4:     return result
5: end procedure
```

Horizontal additions or add-reductions, like the REDUCE_ADD function in line 3, break with the parallel nature of SIMD operations because they require a sequence of instructions with a narrowing number of vector lanes to reduce a vector of integers to a single scalar value. As a result, it is typically recommended to avoid using horizontal additions in performance-critical sections [Int22a]. Therefore, it would be better for the performance if we used vector accumulators instead of scalar accumulators to accumulate the result of this function, so that we must only perform a single reduction for each accumulator right before filling the frequency table.

6.3.2 AVX2 Extract Population Count

AVX2 does not include a hardware vector population count instruction, though CPUs supporting AVX2 do support a scalar population count instruction. Therefore, extracting each lane and performing the scalar population count on them can be beneficial when a reduction would be necessary, regardless. This approach can be seen in Listing 6.1 and relies on a scalar accumulator like Algorithm 3. However, this solution is only viable when a scalar population count hardware instruction exists. Otherwise, an algorithmic vector population count should be used because reducing a vector to a scalar value by extracting each lane is inefficient and only viable due to the strong performance of the hardware population count.

This exact implementation was used by Marques et al. in the AVX2 implementation of the epistasis detection algorithm [MCSJ+22].

```
1  uint32_t ExtractPopcnt(__m256i vec_input){
2      uint64_t t = 0;
3
4      // extract each lane and perform scalar hardware population count
5      t += __builtin_popcountl(_mm256_extract_epi64(vec_input, 0));
6      t += __builtin_popcountl(_mm256_extract_epi64(vec_input, 1));
7      t += __builtin_popcountl(_mm256_extract_epi64(vec_input, 2));
8      t += __builtin_popcountl(_mm256_extract_epi64(vec_input, 3));
9
10     return (uint32_t) t;
11 }
```

Listing 6.1: AVX2 ExtractPopcnt function. Extract each lane and perform scalar hardware population count on them.

6.3.3 Highway Extract Population Count

The implementation displayed in Listing 6.2 is the Highway equivalent to the AVX2 implementation in Listing 6.1, which was used by Marques et al. [MCSJ+22]. While also representing the equivalent to the AVX512 implementation used on Intel Skylake-SP CPUs by Marques et al. [MCSJ+22]. However, a few adjustments were made to ensure portability between extensions and functionality.

Since the order of bits is irrelevant to the result of a population count, it becomes possible to append the bits of multiple integers and perform a single population count on the new integer. For example, performing a population count separately on two 32-bit integers and adding the result together is equivalent to appending the bits of the two 32-bit integers and turning them into one 64-bit integer on which we perform a population count. The single 64-bit population count is more efficient than two separate 32-bit counts. Therefore, we want to be able to extract 64-bit integers from `vec_input`. However, unlike AVX2 intrinsics, Highway is type-safe. Thus, we cannot simply extract 64-bit unsigned integers from a 32-bit unsigned integer vector, like we have done in Listing 6.1. Therefore, we must first cast `vec_input` to an unsigned 64-bit integer vector, `vec_cast`, as seen in line 4.

The second adjustment was the inclusion of a for-loop in lines 7 to 9. Due to the unknown number of lanes, we require a for-loop to extract the lanes of the current target dynamically. In comparison, for the AVX implementation in Listing 6.1, we knew that the 256-bit vector had four lanes of 64-bit unsigned integers.

```
1  HWY_ATTR uint32_t ExtractPopcnt(V vec_input) {
2      // cast uint32_t vector to uint64_t vector
3      const ScalableTag<uint64_t> d;
4      auto vec_cast = BitCast(d, vec_input);
5
6      uint64_t t = 0;
7      for(int i = 0; i  < Lanes(d); i++) {
8          t += __builtin_popcountl(ExtractLane(vec_cast, i));
9      }
10
11     return (uint32_t) t;
12 }
```

Listing 6.2: Highway ExtractPopcnt function. Extract each lane and perform scalar hardware population count on them.

6.3.4 AVX512 Population Count Accumulate

This implementation follows the same principles as Algorithm 3, but with a slight tweak to optimize the performance. The first important fact about this implementation is that it requires an AVX512 version supporting the `vpopcntd` hardware instruction, which enables a hardware-based vector population count. This instruction is supported by Intel's Ice Lake-SP processors, and therefore only a subset of AVX512 capable machines have this functionality [Int22a, MCSJ+22]. The instruction can be targeted using the AVX512 intrinsic function `_mm512_popcnt_epi32`, which can be seen in line 2.

Unlike the implementation presented in Algorithm 3, we do not immediately follow up the vector population count with a reduction. We do this since we want to avoid using reductions in performance-critical sections due to their poor performance, as mentioned in Section 6.3.1. Therefore, we rely on vector accumulators instead of a scalar accumulator,

as required with the `ExtractPopcnt` functions from Section 6.3.2 and Section 6.3.3, which needed a reduction for every entry in the frequency table for every iteration. By using vector accumulators, we only have to perform the reduction once on every accumulator for the corresponding frequency table entries, using the AVX512 function `_mm512_reduce_add_epi32`. This approach ensures the best performance when a hardware vector population count is available.

```
1    __m512i PopcntAccumulate(__m512i vec_acc, __m512i vec_input) {
2        vec_popcnt = _mm512_popcnt_epi32(vec_input);
3        return _mm512_add_epi32(vec_acc, vec_popcnt);
4    }
```

Listing 6.3: AVX512 PopcntAccumulate function. Hardware vector population count with vector accumulator.

6.3.5 Highway Population Count Accumulate

The Google Highway implementation in Listing 6.4 is the Highway equivalent to the AVX512 implementation in Listing 6.3. The population count function in line 2, `PopulationCount`, behaves differently, depending on the target it is compiled for. As mentioned before, AVX2, for example, does not support a hardware-based vector population count. In this case, `PopulationCount` compiles to an algorithmic implementation of a population count. On other CPUs, for example, ARM CPUs that support SVE or Ice Lake-SP CPUs, the function will compile to the machine instruction for the vector population count, thus, providing excellent performance.

```
1  HWY_ATTR V PopcntAccumulate(V vec_acc, V vec_input) {
2      V vec_popcnt = PopulationCount(vec_input);
3      return Add(vec_acc, vec_popcnt);
4  }
```

Listing 6.4: Highway PopcntAccumulate function. Vector population count with vector accumulator.

7 Evaluation

After having gone over the various SIMD libraries, we will examine the results of the Mandelbrot benchmark, discussed in Chapter 5, and the epistasis detection algorithm, discussed in Chapter 6.

7.1 Experimental Setup

Before reviewing the results of the Mandelbrot benchmark in Section 7.2 and the epistasis detection case study in Section 7.3, we will go over the experimental setup we are using in order to ensure the result's reproducibility.

7.1.1 Hardware

All measurements were conducted on the Bavarian Energy, Architecture and Software Testbed (BEAST) of the Leibniz Supercomputing Center (LRZ). On this system, we have had access to some of the newest hardware available in the high-performance CPU market. We conducted the benchmarks using the hardware and SIMD extension sets specified in Table 7.1.

System	CPU	Vector Size (Extension)
(ICE)	Intel® Xeon® Platinum 8360Y (Ice Lake-SP)	256-bit (AVX2) 512-bit (AVX512)
(AFX)	Fujitsu A64FX (Armv8.2-A + SVE)	512-bit (SVE)
(THX)	Cavium Thunder-X2 (Armv8.1-A + SIMD)	128-bit (NEON)

Table 7.1: List of CPUs and SIMD extensions used on the LRZ BEAST platform.

7.1.2 SIMD Library Versions

We always aimed to use the latest version of the libraries when running the benchmarks. For Highway, we used version 1.0.4, which was used for all Mandelbrot benchmarks and the epistasis detection algorithm. We used version 1.4.3 of the Vc library. Libsimdpp and NSIMD were evaluated using version 2.1 and 3.0.1, respectively. Pure SIMD and SIMD Everywhere do not have official release versions, so we used the most recent build available on their GitHub pages at the time [PS21, SE23]. For Pure SIMD, that build was from October 2021, and for SIMD Everywhere, it was from February 2023.

7.1.3 Measurement Method

We used Version 1.7.1 of Google's Benchmark library to take precise measurements [Goo23]. For the Mandelbrot benchmark, multiple hundred repetitions were performed. However, the epistasis detection algorithm was only repeated ten times for every setting due to the high resource demands of the algorithm. The library then calculates the mean, median, standard deviation, coefficient of variation, max duration, and min duration from the gathered data. All measurements and benchmarks were done with frequency scaling turned off. The Benchmark library currently does not support processor affinity (thread pinning) [Goo21]. Thus, there is no guarantee that a thread will always be executed on the same core.

Throughout the following sections, we will often mention a maximum theoretical speed-up for the algorithms. A theoretical speed-up in SIMD programming is typically the number of lanes and represents the maximum speed-up that can be expected by using vector instructions in comparison to scalar instructions. However, it is essential to note that a theoretical speed-up is a theoretical concept that can not always be reached or may even be exceeded due to factors such as the latency and throughput of the scalar instructions compared to the vector instructions and other factors such as memory bandwidth.

7.2 Mandelbrot Benchmark Evaluation

In this section, we will evaluate the Mandelbrot benchmark introduced in Chapter 5. The benchmark will be conducted on the systems and the corresponding extensions displayed in Table 7.1. We will examine all the libraries presented in Chapter 4 to evaluate their performance on the different CPU architectures and SIMD extensions.

System	Target	Compiler Optimization Flags
(ICE), (AFX), (THX)	SCALAR	`-O2 -fno-tree-vectorize -ffast-math`
(ICE)	AVX2	`-O2 -fopenmp -ftree-vectorize -ffast-math -march=haswell -mtune=haswell -maes`
(ICE)	AVX512	`-O2 -fopenmp -ftree-vectorize -ffast-math -march=icelake-client -mavx512vpopcntdq`
(AFX)	SVE	`-O2 -fopenmp -ftree-vectorize -ffast-math -march=armv8.2-a+sve -mcpu=a64fx+sve`
(THX)	NEON	`-O2 -fopenmp -ftree-vectorize -ffast-math -march=armv8-a+simd -mtune=thunderx2t99`

Table 7.2: Compiler optimization flags used on each system and target for the Mandelbrot benchmark.

Due to the hardware-close nature of SIMD programming, using the correct optimizations is highly important because the compiler needs to know the type of CPU it generates instructions for. Therefore, we had to use different optimization flags on each system. Table 7.2 presents an overview of the set flags. The `-march=<CPU>` flag is often required by libraries to select the best SIMD extension the CPU supports. Some libraries also require additional confirmation like Libsimdpp, which needs, for example, `-DSIMDPP_ARCH_X86_AVX512F`

`-mavx512f -DSIMDPP_ARCH_X86_FMA3 -mfma` flags to compile to AVX512 with FMA units enabled.

We will evaluate the libraries' performance by comparing the speed-up of the execution times going from a scalar implementation to a library implementation while also comparing them to a compiler intrinsics implementation. Through this, we will see how close the library comes to theoretical maximum speed-up and how it compares to a dedicated compiler intrinsics implementation.

7.2.1 Results AVX2

First, we will look at the performance of the Mandelbrot benchmark on the ICE system. We will evaluate the libraries' performance for this system when targeting AVX2 and AVX512. We used g++ version 11.2.0 to compile all libraries and benchmarks on the ICE system.

AVX2 is one of the most common SIMD extension sets and can be found in many consumer-level x86 CPUs nowadays. As a result, all examined libraries support this extension set (compare Table 4.1).

In Figure 7.1, we have visualized the results of the benchmark targeting AVX2. The y-axis displays the factor by which the implementation was faster than a scalar implementation. On the x-axis, the different implementations are presented. We conducted three benchmarks for each implementation, which differ in the number of maximum while-loop iterations (compare Algorithm 1 line 18). The maximum while-loop iteration settings were 10, 100, and 1000.

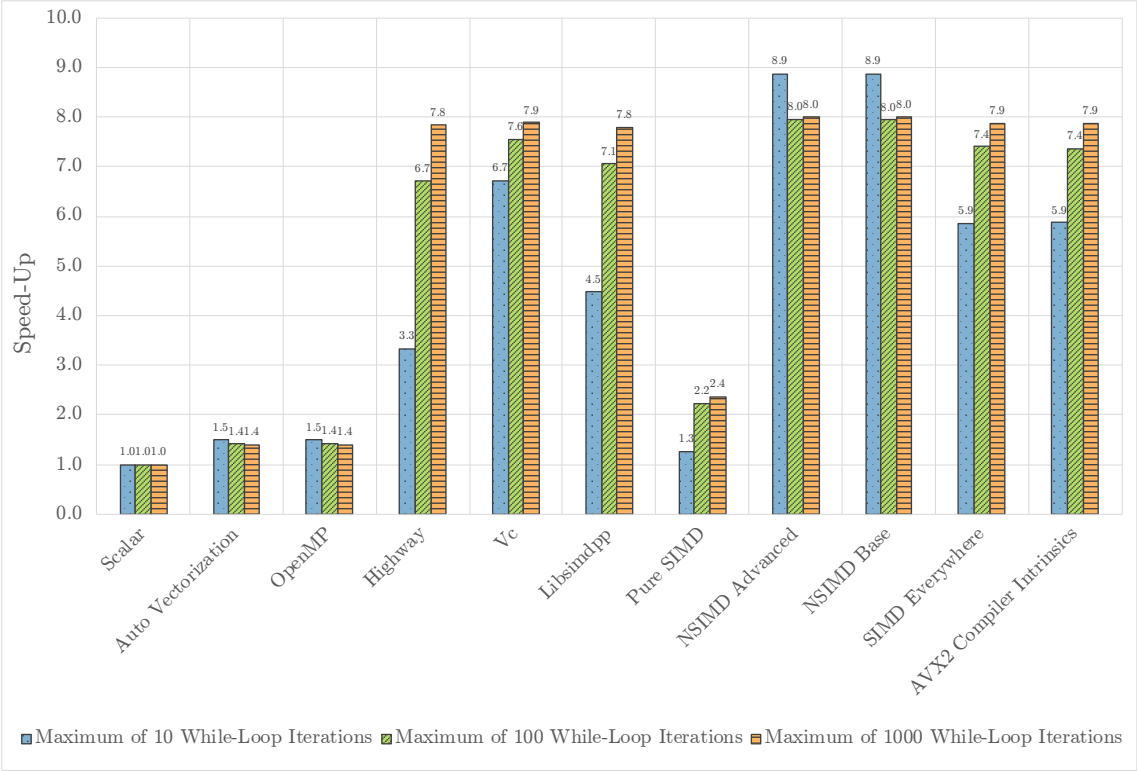

Figure 7.1: Mandelbrot benchmark targeting AVX2 compiled with g++ 11.2.0 and run on the ICE system.

Implicit Vectorization

First, we will examine the performance of implicit vectorization approaches (compare Figure 7.1). The auto-vectorization and the OpenMP approach behave very similarly. Both implementations achieve a maximum speed-up of 1.5x when running with a maximum of 10 while-loop iterations. Notably, the performance decreases slightly with an increase in iterations, which is the opposite of almost all other implementations. Overall, achieving a speed-up of only 1.5x falls short of expectations when almost all explicit implementations reach a speed-up of close to a factor of 8x. However, this result is consistent with other studies [PJ15]. It is also noteworthy that the additional information provided by pragmas in the OpenMP implementation did not result in a relevant performance increase for this implementation.

Implementation	Mean [ms]	Median [ms]	SD [ns]	CV [%]	Max [ms]	Min [ms]
Scalar	111.63	111.62	36262	0.03	111.73	111.60
Auto Vectorization	78.59	78.59	6163	0.01	78.60	78.58
OpenMP	78.54	78.54	10536	0.01	78.56	78.53
Highway	16.64	16.63	3010	0.02	16.64	16.63
Vc	14.77	14.77	1605	0.01	14.78	14.77
Libsimdpp	15.83	15.83	1540	0.01	15.83	15.83
Pure SIMD	50.00	50.01	4894	0.01	50.02	50.00
NSIMD Advanced	14.04	14.04	1460	0.01	14.04	14.04
NSIMD Base	14.04	14.04	1810	0.01	14.04	14.04
SIMD Everywhere	15.08	15.08	4146	0.03	15.10	15.08
AVX2 Intrinsics	15.15	15.15	1940	0.01	15.16	15.15

Table 7.3: Detailed data of the AVX2 Mandelbrot benchmark with a maximum of 100 while-loop iterations, compiled with g++ 11.2.0 and run on the ICE system, visualization in Figure 7.1. SD is an abbreviation for Standard Deviation, CV for Coefficient of Variation.

Explicit Vectorization

All explicit libraries achieve a speed-up comparable to or better than a compiler intrinsics implementation, except Pure SIMD, which relies on auto-vectorization.

All libraries compiling to intrinsics perform nearly the same with 100 and 1000 While-Loop iterations. For 1000 iterations, the speed-up is between 7.8x - 8.0x, reaching the maximum theoretical speed-up, which is very impressive. When running 100 iterations, performance drops slightly in almost all libraries. However, the drop in the speed-up is most significant in Google's Highway library, which only achieves a speed-up of 6.7x. This drop in performance is even more pronounced when looking at the results when running with a maximum of 10 while-loop iterations. Here Highway delivers a speed-up of 3.3x, while the other libraries provide a speed-up of 4.5x - 8.9x. The speed-up of 8.9x observed with the NSIMD library is extraordinary because 8.0x is the theoretical maximum. Therefore, this library and some others must use code patterns that allow the compiler to perform additional optimizations to reach and surpass the speed-up achieved with AVX2 compiler intrinsics.

Overall, the NSIMD library and its two different APIs performed the best, providing a

speed-up of around 8.0x in all benchmarks.

We used the data displayed in Table 7.3 for the visualization in Figure 7.1. From the table, it becomes clear that the benchmarks were very consistent with little deviation, because the coefficient of variation stays between 0.01% and 0.03% for all implementations.

7.2.2 Results AVX512

For the evaluation of the libraries targeting AVX512, we also used the ICE system and g++ 11.2.0. However, the Vc library does not support AVX512, making it the only library we have examined that does not support this extension (see Table 4.1).

One of the critical differences between AVX2 and AVX512 is the larger vector register size of 512-bit compared to AVX2's 256-bit vector registers. As a result, the maximum theoretical speed-up shifts from 8x to 16x because AVX512 can calculate 16 pixels simultaneously.

Figure 7.2 follows the same layout as the visualization of the results of the Mandelbrot benchmarks targeting AVX2 in Figure 7.1.

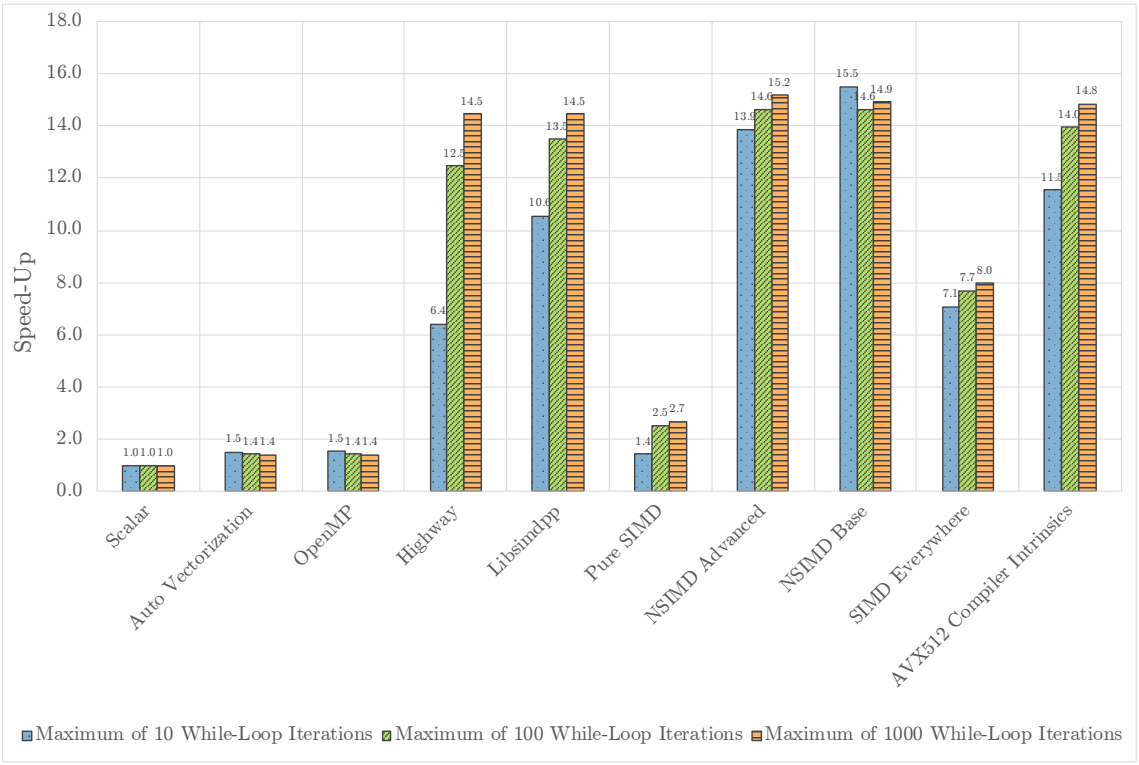

Figure 7.2: Mandelbrot benchmark targeting AVX512 compiled with g++ 11.2.0 and run on the ICE system.

Implicit Vectorization

When examining Figure 7.2, it becomes clear that the auto-vectorization for AVX512 cannot improve past the speed-up achieved in the AVX2 benchmark (compare Figure 7.1), and so provides the same speed-up of only 1.4x - 1.5x. This is the case for both the auto-vectorization version and the version aided by OpenMP.

Explicit Vectorization

The Pure SIMD implementation fails, just like in the AVX2 benchmark, to reach the performance of other explicit vectorization approaches. However, Pure SIMD did perform slightly better in the AVX512 benchmark, see Figure 7.2, with a speed-up of 2.7x at 1000 iterations compared to 2.4x at 1000 iterations in the AVX2 Mandelbrot Benchmark (compare Figure 7.1).

Another library that falls behind the others is SIMD Everywhere because it fails to surpass a speed-up of 8x. As mentioned in Section 5.4.7 this library relies on a slightly altered AVX2 compiler intrinsics implementation. When inspecting the assembly code generated when compiling the implementation to AVX512, it becomes clear that the library cannot scale up the implementation to 16 lanes and take advantage of the larger register size. Instead, the generated assembly code includes no significant changes compared to the AVX2 implementation. The slight increase in performance of SIMD Everywhere seen in Figure 7.2 compared to Figure 7.1 is likely primarily a result of better compiler optimization for the ICE system when using the flags specified in Table 7.2 for AVX512 compared to AVX2. However, it must be stated that the AVX512 support of SIMD Everywhere is still in active development and unfinished. Therefore, the speed-up measured here might improve significantly in the future [SE23].

All other libraries can match the speed-up observed in the compiler intrinsics implementation. Similar to the results seen in the AVX2 benchmark, the two NSIMD implementations performed best in the AVX512 benchmark. The advanced API of NSIMD reached a speed-up of 15.2x at 1000 iterations, and the base API achieved a speed-up of 14.9x, which is equal to the AVX512 compiler intrinsics implementation. Libsimdpp and Highway have the same speed-up of 14.5x at 1000 iterations. However, Highway performs slightly worse than Libsimdpp at lower while-loop iteration counts.

Overall, the explicit libraries performed very well and generally came very close to the theoretical maximum speed-up of 16x. These results solidify the usage of those libraries on x86 Systems with AVX512 support.

7.2.3 Results SVE

Coming from the two x86 SIMD extensions, we will now look at the first ARM extension. On the AFX system, we have access to the Scalable Vector Extension (SVE) set. SVE follows a different design philosophy than all the other extensions examined in this paper because SVE intrinsics do not define the SIMD register size in the underlying hardware. They instead support SIMD register sizes ranging from 128-bit to 2048-bit [ARM23]. The AFX system has SIMD registers with a maximum size of 512-bit [Fuj23]. As a result, the maximum theoretical speed-up is 16x with 32-bit floats.

SVE is the newest among the extensions examined and less widespread than other recent extensions, such as AVX512. Therefore, only a limited number of libraries currently support it. NSIMD states that it supports SVE but only using the base API when compiling with GCC. However, we have been unable to compile the base NSIMD implementation for SVE, so we sadly have to exclude it [NSI21b]. Furthermore, the SVE support of SIMD Everywhere is unfinished, similar to the AVX512 support. Nonetheless, we will include it in the benchmark and analysis [SE23].

We used g++ version 11.0.0 to compile the benchmarks for SVE on the AFX system.

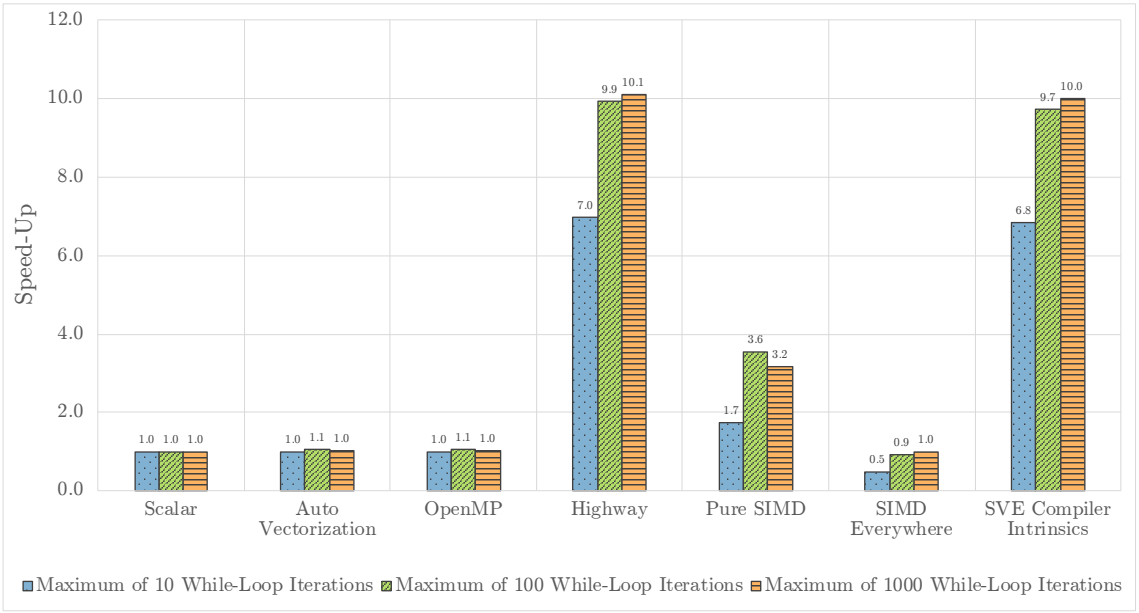

Figure 7.3: Mandelbrot benchmark targeting SVE compiled with g++ 11.0.0 and run on the AFX system.

Implicit Vectorization

As shown in Figure 7.3, the auto-vectorization approaches fail when targeting SVE. The pure auto-vectorization approach and the OpenMP-aided approach only provide a speed-up of 1.1x at 100 iterations; the other two benchmarks achieve the same performance as the scalar implementation. Due to the short existence of SVE, it may be the case that GCC 11.0.0 is not yet fully optimized for SVE because the version used was also the first GCC version to support SVE [GCC23c]. However, Pure SIMD also relies on the auto-vectorization feature of GCC and performs relatively well, see Figure 7.3.

Explicit Vectorization

Looking at the explicit vectorization approaches, only three libraries support SVE, excluding NSIMD (compare Figure 7.3). First, we will examine SIMD Everywhere. Although the library's SVE support is in the early stages, it could compile to SVE. However, the resulting implementation led to a slow-down of 0.5x compared to the scalar implementation when running at 10 and 100 while-loop iterations. When running 1000 iterations, it managed to match the scalar implementation. As mentioned when analyzing the library's AVX512 performance, it may very well be the case that these results improve when further support is added. However, from a compatibility standpoint, it is significant that it works.

The second library we will examine is Pure SIMD, which surprisingly had its best speed-up when targeting SVE with a speed-up of 3.6x at 100 iterations and 3.2x at 1000 iterations.

The last library of which we are testing its SVE support is Highway. This library is the only one that does an excellent job at matching the SVE compiler intrinsics implementation, even surpassing the speed-up of the intrinsics version at 100 and 1000 iterations by 0.2x and 0.1x, respectively, and overall achieving a speed-up of 10.1x at 1000 iterations. It is also

interesting to see how SVE with 512-bit registers compares to AVX512. While targeting AVX512, most libraries achieved a speed-up of 14.5x - 15.2x with 1000 iterations (see Figure 7.2), the SVE implementations did not come close to that, and only achieved a speed-up of 10x - 10.1x. We did not find any significant clues in the assembly code indicating the origin of the difference.

7.2.4 Results NEON

The fourth and final SIMD extension for which we will examine the libraries is another ARM extension called NEON. Unlike SVE, NEON was not developed for high-performance computing and only offers 128-bit SIMD registers. Therefore, the theoretical maximum speed-up on the THX system is only a factor of 4x. However, due to the older age of NEON, a lot more libraries support it.

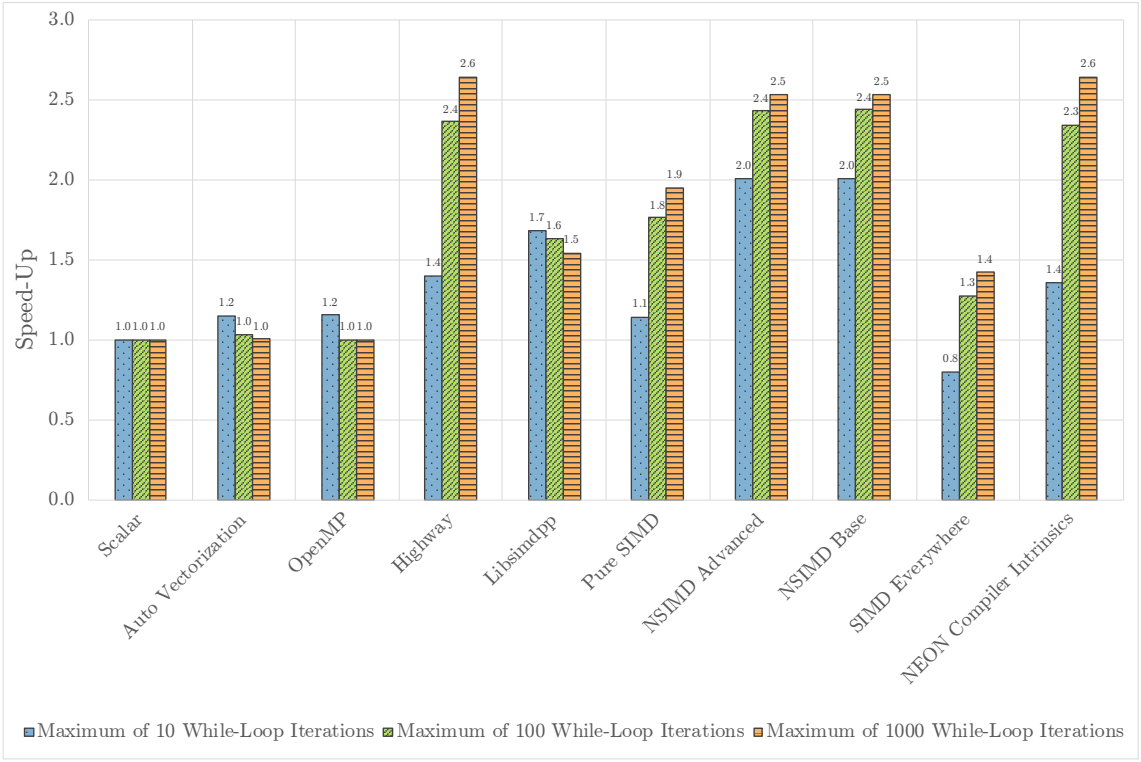

Figure 7.4: Mandelbrot benchmark targeting NEON compiled with g++ 8.3.1 and run on the THX system.

Implicit Vectorization

Similar to SVE, little to no speed-up could be achieved through the two auto-vectorization approaches, with both only achieving a minor speed-up of 1.2x at 10 iterations and matching the scalar implementation for the other two benchmarks (compare Figure 7.4).

Explicit Vectorization

For the results of the explicit vectorization, we will first look at SIMD Everywhere, which performs better for NEON than for SVE and achieves a speed-up of 1.3x and 1.4x with 100 and 1000 iterations (compare Figure 7.4 and Figure 7.3). However, it does slow down to 0.8x with 10 while loop iterations. Nevertheless, this result is solid considering that the NEON implementation is also still under development and unfinished.

Pure SIMD also performs well with a speed-up of 1.9x at 1000 iterations, which is faster than Libsimdpp, which only achieves 1.5x. This result is surprising because Libsimdpp performs so well when targeting AVX2 and AVX512 (compare Figure 7.1 and Figure 7.2). Highway and both NSIMD APIs manage to match the NEON compiler intrinsics implementation. Highway performs better than the NSIMD versions with a speed-up of 2.6x at 1000 iterations compared to 2.5x. Though, Highway performs significantly worse than NSIMD at 10 iterations, where it only achieves a speed-up of 1.4x compared to NSIMD's 2.0x.

Overall, no implementation came close to the theoretical maximum speed-up of 4x, indicating other bottlenecks, which is consistent with other studies, see Ref. [SJD+18]. However, all libraries brought a significant speed-up at higher iteration counts. Again, Highway matched the intrinsics implementation perfectly. The achieved results make Highway and NSIMD a good solution for NEON.

7.3 Case Study Epistasis Detection

In this section, we will evaluate the case study on the vectorization of the epistasis detection algorithm, introduced in Chapter 6, using Google's Highway library.

System	Target	Compiler	Compiler Optimization Flags
(ICE)	SCALAR	g++ 11.2.0	`-O2 -fopenmp -fno-tree-vectorize -mpopcnt`
(ICE)	AVX2	g++ 11.2.0	`-O2 -fopenmp -ftree-vectorize -march=haswell -mtune=haswell -maes`
(ICE)	AVX512	g++ 11.2.0	`-O2 -fopenmp -ftree-vectorize -march=skylake-avx512`
(ICE)	AVX512 - vpopcntdq	g++ 11.2.0	`-O2 -fopenmp -ftree-vectorize -march=icelake-client -mavx512vpopcntdq`
(AFX)	SCALAR	clang++ 11.0.0	`-O2 -fopenmp -fno-tree-vectorize`
(AFX)	SVE	clang++ 11.0.0	`-O2 -fopenmp -ftree-vectorize -mcpu=a64fx+sve`

Table 7.4: Compiler optimization flags for the epistasis detection algorithm used on each system, target, and compiler.

We will benchmark the algorithm on the ICE and AFX systems and compare the speed-up achieved using two different Highway implementations. The first is the Highway ExtractPopcnt implementation, which uses the ExtractPopcnt function examined in Section 6.3.3.

The second is the Highway PopcntAccumulate implementation, which uses the PopcntAccumulate function discussed in Section 6.3.5.

The epistasis detection algorithm is highly resource intensive; therefore, we have used threads via OpenMP to parallelize the execution in addition to vectorization. For each implementation, we benchmarked the algorithm with three settings for the number of SNPs (2048, 4096, 8192) and 16384 samples. On the AFX system, we did not run the algorithm with 8192 SNPs due to a slow scalar execution on the platform. However, we do not expect a significant performance difference with 8192 SNPs.

We used g++ and clang++ to compile the epistasis detection benchmarks. Unfortunately, we could not compile the Highway SVE implementation using g++ due to an error we could not fix. However, the error prohibiting us from compiling the SVE version of the algorithm does not occur when compiling the SVE implementation of the Mandelbrot benchmark. Accordingly, we only used the clang compiler for the epistasis detection benchmarks on the AFX system. A detailed summary of the systems, targets, compilers, and compiler optimization flags can be found in Table 7.4.

7.3.1 Results AVX2 and AVX512

We will start by examining the results of the Highway implementations when targeting AVX2, AVX512, and AVX512vpopcntdq, with 128 threads on the ICE system.

	Implementation	**Mean [s]**	**Median [s]**	**SD [ms]**	**CV [%]**	**Max [s]**	**Min [s]**
1.	Scalar	133	133	293	0.22	133	132
2.	Compiler Intrinsics ExtractPopcnt (AVX2)	61	62	1212	1.98	62	59
3.	Highway ExtractPopcnt (AVX2)	64	64	32	0.05	64	64
4.	Highway PopcntAccumulate (AVX2)	63	63	60	0.10	63	63
5.	Highway ExtractPopcnt (AVX512)	68	68	277	0.41	68	67
6.	Highway PopcntAccumulate (AVX512)	46	46	107	0.23	46	46
7.	Highway PopcntAccumulate (AVX512vpopcntdq)	16	16	70	0.45	16	16

Table 7.5: Detailed data on execution times of the epistasis detection algorithm with 2048 SNPs and 16384 samples, collected on the ICE system using 128 threads.

First, we will review the implementations listed in Table 7.5. The scalar implementation

in row 1 and compiler intrinsics implementation in row 2 of Table 7.5 were both provided to us by Marques et al. [MCSJ+22]. The Highway implementations with ExtractPopcnt in their name (see row 3 and row 5) use the population count implementation discussed in Section 6.3.3. The Highway implementations with PopcntAccumulate in their name (see rows 4, 6, and 7) use the population count function presented in Section 6.3.5.

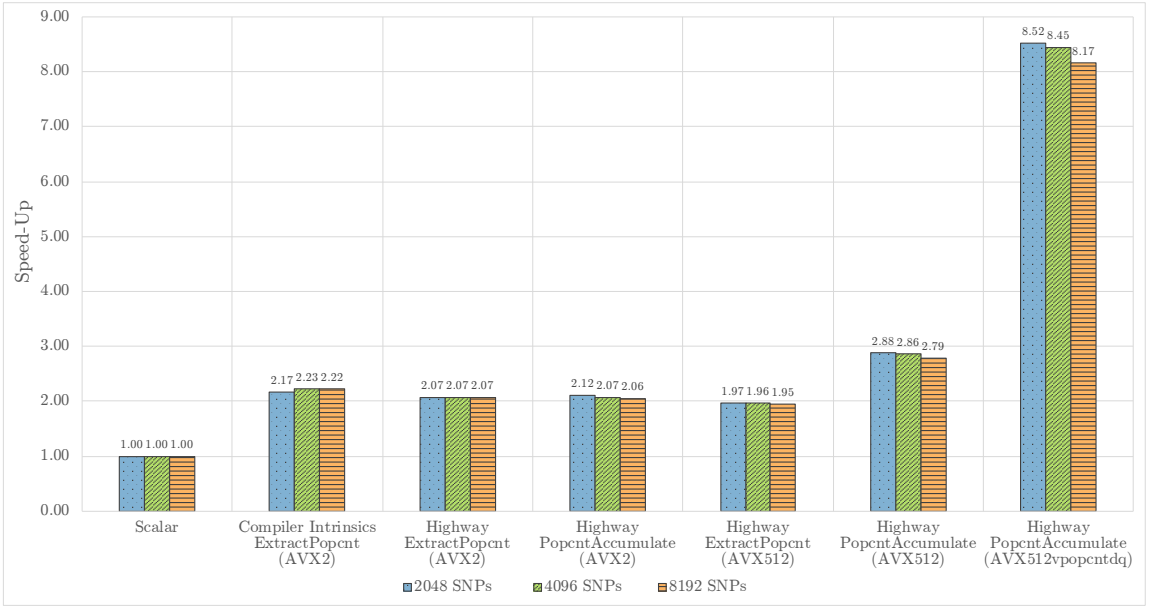

Figure 7.5: Epistasis detection algorithm targeting AVX2 and AVX512 compiled with g++ 11.2.0 and run on the ICE system using 128 threads.

Figure 7.5 includes a visualization of the data in Table 7.5. The y-axis presents the factor by which an implementation is faster than the scalar implementation. On the x-axis, the different implementations are presented.

The compiler intrinsics AVX2 implementation achieves a speed-up of 2.17x with 2048 SNPs, 2.23x with 4096 SNPs, and 2.22x with 8192 SNPs. The equivalent Highway ExtractPopcnt implementation targeting AVX2 achieves scores similar to the compiler intrinsics implementation, with a speed-up of 2.07x at 2048, 4096, and 8192 SNPs. Accordingly, the Highway implementation running with 4098 SNPs reaches 92.83% of the speed-up measured for the AVX2 compiler intrinsics implementation. Therefore, Highway fulfills the design goal not to deviate more than 10% to 20% from a dedicated intrinsics implementation for the epistasis detection algorithm targeting AVX2. In addition, the Highway implementation using the PopcntAccumulate function from Section 6.3.5 achieves slightly better results than the Highway ExtractPopcnt implementation with a speed-up of 2.12x at 2048 SNPs when targeting AVX2.

From Figure 7.5, it also becomes clear that the Highway ExtractPopcnt implementation targeting AVX512 does not provide a performance increase compared to the corresponding AVX2 Highway version but rather a slowdown. The worse performance of the implementation when targeting AVX512 is the result of the necessity for two extract instructions to retrieve a lane from an AVX512 `ZMM` register, which leads to a significant overhead. These results are consistent with findings made by Marques et al. using an implementation similar to this one for AVX512 [MCSJ+22].

The Highway implementation using the PopcntAccumulate function and targeting AVX512 did, however, achieve a speed-up of 2.88x with 2048 SNPs, 2.86x at 4096 SNPs, and 2.79x SNPs. Therefore, this is the most efficient implementation on CPUs that support AVX512 but do not support the `vpopcntd` instruction. The speed-up achieved with the algorithmic population count of Highway is better than the ExtractPopcnt implementation, which Marques et al. used for AVX512 on CPUs without support for the `vpopcntd` instruction [MCSJ+22]. In other words, we obtained a higher speed-up for AVX512 than Marques et al. due to the algorithmic population count Highway offers, which Marques et al. did not implement with compiler intrinsics.

All speed-ups achieved are minor compared to those achieved when utilizing the `vpopcntd` instruction supported on Ice Lake-SP CPUs. The Highway PopcntAccumulate implementation targeting AVX512vpopcntdq achieves a speed-up of 8.52x at 2048 SNPs, 8.45x at 4096 SNPs, and 8.17x at 8192 SNPs. These results are consistent with the speed-ups recorded by Marques et al. when using the `vpopcntd` instruction [MCSJ+22].

Therefore, it makes sense to rely on the Highway implementation using the PopcntAccumulate function from Section 6.3.5 over the ExtractPopcnt function from Section 6.3.3 when targeting AVX512. For AVX2, it is not as straightforward because the two Highway implementations targeting AVX2 achieve almost the same speed-up, with the ExtractPopcnt being slightly better at higher SNP counts and the PopcntAccumulate implementation better at 2048 SNPs. However, the difference is so marginal that we believe that it is reasonable to use the Highway PopcntAccumulate implementation for all targets.

7.3.2 Results SVE

The third SIMD extension which we want to test using the Highway implementations of the epistasis detection algorithm is SVE. We executed the epistasis detection algorithm with 48 threads on the AFX system.

Implementation	Mean [s]	Median [s]	SD [s]	CV [%]	Max [s]	Min [s]
Scalar	2873	2877	18	0.63	2891	2826
Highway ExtractPopcnt (SVE)	1999	2008	19	0.94	2021	1977
Highway PopcntAccumulate (SVE)	96	95	1.90	1.98	99	93

Table 7.6: Detailed data on execution times of the epistasis detection algorithm with 2048 SNPs and 16384 samples, collected on the AFX system using 48 threads, visualization included in Figure 7.6. SD is an abbreviation for Standard Deviation, CV for Coefficient of Variation.

The scalar implementation listed in Table 7.6 corresponds to the scalar implementation we received from Marques et al. [MCSJ+22]. However, Marques et al. did not test their algorithm on a CPU supporting SVE; therefore, we do not have an SVE compiler intrinsics reference implementation as we did for AVX2. The two Highway implementations are identical to the ones used in the AVX2 and AVX512 benchmarks of the algorithm. To recap, the Highway ExtractPopcnt implementation uses the ExtractPopcnt function, which we discussed in Section 6.3.3, and the Highway PopcntAccumulate implementation relies on the PopcntAccumulate function, which we examined in Section 6.3.5.

When comparing the data presented in Table 7.6 to the data from Table 7.5, it becomes

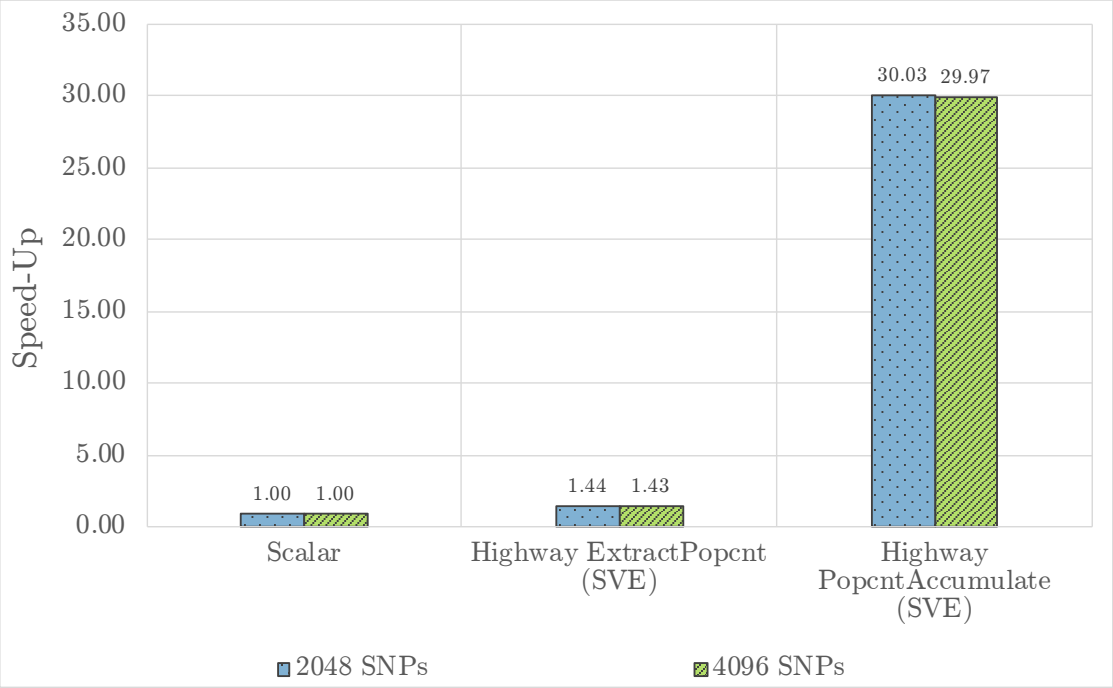

Figure 7.6: Epistasis detection algorithm targeting SVE compiled with clang++ 11.0.0 and run on the AFX system using 48 threads.

clear that the benchmarks took substantially longer on the AFX system compared to the ICE system. This finding is partly caused by the fact that the benchmarks on the ICE system used 128 threads instead of 48 and that the ICE CPU cores have a higher frequency. When considering these differences and running the benchmark on the ICE system with 48 threads with an average frequency of 3.384 GHz and the AFX system with 48 threads and a fixed frequency of 1.8 GHz, one could expect the ICE system to finish the calculation roughly twice as fast. However, this is not the case. Benchmarking the scalar execution has shown that the ICE system finishes 18.32x faster when running the algorithm with 48 threads, 2048 SNPs, and 16384 Samples. We will discuss a possible explanation for this later.

The visualization in Figure 7.6 follows the same structure as Figure 7.5. The first implementation we will examine is the Highway ExtractPopcnt, which only achieved a speed-up of 1.44x at 2048 SNPs. Testing has shown that extracting lanes from SVE registers is less efficient than extracting from AVX registers. Storing the entire vector instead and then applying the population counts on the scalar values could slightly improve the 1.44x speed-up. However, this approach can, in general, not be recommended because, as discussed in Section 6.3.5, Highway supports the hardware vector population count found on all SVE-supporting CPUs and does not rely on the algorithmic approach, therefore, enabling excellent performance when using the Highway `PopulationCount` function. As a result, the Highway PopcntAccumulate implementation should always be used when targeting SVE.

The AFX system has 512-bit SIMD registers, and we are using `uint32_t` integers, so we are working with 16 lanes, which leads to a theoretical maximum speed-up of 16x. Nonetheless, the Highway PopcntAccumulate implementation achieved a very impressive speed-up of 30.03x at 2048 SNPs and 29.97x at 4096 SNPs. This significant additional performance

compared to the scalar implementation may be related to a slow scalar population count on the AFX system, which may also cause the scalar implementation's poor performance on the AFX system compared to the ICE system. This theory is further supported by the fact that the Highway version using the ExtractPopcnt function only achieves a minor speed-up, and the Highway PopcntAccumulate implementation, which only changes a single function, has a significant speed-up. However, we could not confirm this theory because the A64FX Microarchitecture Manual [Fuj22] does neither state the throughput of the scalar population count hardware instruction nor that of the vector population count instruction.

8 Conclusion

Throughout this thesis, we have examined different approaches to SIMD programming and determined that libraries can be an essential tool to streamline the development of SIMD applications. We examined each of the six libraries regarding the supported extensions, functions, ease of use, documentation, and performance compared to a compiler intrinsics implementation. The results of these factors are summarized in Table 8.1. A ✓ indicates that we determined the library has fulfilled the expectations in the category, and an ✗ indicates the opposite.

Library	Supported Extensions	Functions	Documentation	Ease Of Use	Performance Per Extension			
					AVX2	AVX512	NEON	SVE
Highway	✓	✓	✓	✗	✓	✓	✓	✓
Vc	✗	✓	✓	✓	✓	✗	✗	✗
Libsimdpp	✓	✗	✓	✓	✓	✓	✗	✗
NSIMD	✓	✓	✗	✓	✓	✓	✓	✗
SIMD Everywhere	✓	✓	✓	✓	✓	✗	✗	✗
Pure SIMD	✓	✗	✓	✓	✗	✗	✗	✗

Table 8.1: Summary of all the examined factors and findings for each library. The ✓ symbol indicates that the library fulfills the expectations in the category; on the other hand, an ✗ symbol indicates the opposite.

The first library we will summarize the results of is Vc, which performed excellently when targeting AVX2. However, due to the relatively old age of the library and the end of active development, it does not support most of the newer common SIMD extensions. As a result, of this significant drawback, we can generally not recommend this library.

For the subsequent library, Libsimdpp, a distinction between users must be made. Libsimdpp is a versatile and easy-to-use library, perfect for simple and small applications, but users requiring more advanced functionality will not find it here. Additionally, the library's NEON performance could have been better, and it does not support SVE. Thus, it will not be the best choice when targeting ARM SIMD extensions. Accordingly, while the library can generally be recommended, it will be less suitable for large projects demanding complex SIMD functions.

The concept of SIMD Everywhere differs from all the other libraries we have examined because it aims to map all compiler intrinsics between extension sets. However, through our evaluation, we have found that although the code compiles to other extension sets, it falls substantially short of the extension's optimal performance. Therefore, we can only recommend this library under the circumstance that code using intrinsics must be run on a platform that does not support the extension set natively and where performance is of lesser concern. In summary, SIMD Everywhere can be a valuable tool to execute existing SIMD code on other platforms, but it will not achieve optimal performance.

Another library we cannot recommend is Pure SIMD. Overall, the concept of this library is compelling because it utilizes the auto-vectorization feature of the compiler, thus, delivering

optimal portability across extensions. Moreover, the results obtained with Pure SIMD were good, considering multiple code sections had to rely on scalar code because the library did not offer the necessary functions. Accordingly, since the library does currently not offer the necessary functions to vectorize most algorithms, we cannot recommend Pure SIMD.

NSIMD came close to Highway, with its outstanding performance for AVX2 and AVX512. However, we could not compile for SVE due to an error. Therefore, we were unable to conduct an SVE benchmark for this library. As discussed before, lacking documentation in the usage section made it hard to determine the cause of the error. In the future, additional tests could be beneficial to determine the cause of the error because NSIMD could be a contender to Highway if the compilation issues can be resolved. Another question surrounding NSIMD is the development state. The last update was over a year ago. Thus, it is unclear if or when new features and extensions will be added. Overall, we cannot unequivocally recommend the library.

When reviewing each library's performance, it becomes clear that Google's Highway library was the only one consistently performing excellently across all extensions. However, the ease of use could be further improved. Specifically, getting started with Highway was more complex compared to other libraries. These difficulties are caused by a more flexible approach to includes and namespaces. While the additional flexibility may be troublesome for beginners, these options can be highly beneficial for advanced users. Overall, Highway offers flexibility and advanced features like none of the other libraries. Furthermore, the library has a fast and active development cycle, making it currently the only library we tested with excellent performance across all extensions with a development team this active. Therefore, we can strongly recommend Highway when searching for a zero-overhead SIMD library capable of tackling more advanced problems.

There are numerous SIMD libraries we did not evaluate in this thesis. One of them that could be evaluated in the future is xSIMD [xSI23], which is an active open-source library with a strong developer community. The existence of so many SIMD libraries with a similar purpose highlights the need for standardization and a standard API for SIMD programming.

The semiconductor company ARM Ltd. has taken the first steps to standardize and future-proof SIMD features on ARM64 machines with the release of SVE. As mentioned, SVE stands for Scalable Vector Extension, meaning SVE intrinsics do not have to specify the underlying SIMD register size. Therefore, SVE intrinsics code can scale between SIMD registers with sizes ranging from 128-bit to 2048-bit [ARM23]. This scalability aspect will future-proof the SVE extension for the foreseeable future since current CPUs supporting SVE only have SIMD registers with sizes up to 512-bit [Fuj23]. It would be desirable to see other hardware manufacturers future-proof their SIMD approaches. However, in the end, hardware manufacturers must find a solution that facilitates portable SIMD code across architectures.

Developing a standardized SIMD API across different extensions presents the biggest challenge for SIMD to break out of the high-performance computing field into general consumer software. However, for many use cases, Google's Highway library may currently present the best available abstraction tool for developing software using explicit vectorization for the most commonly available SIMD extensions.

List of Figures

List of Listings

List of Tables

Bibliography

[ARM23] ARM: *ARM SVE*. https://developer.arm.com/Architectures/Scalable%20Vector%20Extensions. Version: 2023. – visited on March 4, 2023

[Bre20] Breshears, Clay P.: *OpenMP* Vectorization Support*. https://www.intel.com/content/www/us/en/developer/articles/technical/delve-into-mysteries-openmp-vectorization-support.html. Version: 2020. – visited on January 4, 2023

[CFCD17] Cardoso, Joã. P. ; Figueired Coutinho, José G. ; Diniz, Pedro C.: *Embedded computing for high performance: Efficient mapping of computations using customization, code transformations and compilation*. Morgan Kaufmann, 2017

[Cla23] Clang: *Clang Language Extensions*. https://clang.llvm.org/docs/LanguageExtensions.html. Version: 2023. – visited on January 26, 2023

[Cpp23] Cppreference: *Cppreference Web Page*. https://en.cppreference.com/w/. Version: 2023. – visited on January 19, 2023

[Fuj22] Fujitsu: *A64FX Microarchitecture Manual 1.8*. https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.8.pdf. Version: 2022. – visited on March 28, 2023

[Fuj23] Fujitsu: *Datasheet Fujitsu A64FX*. https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet_en.pdf. Version: 2023. – visited on March 24, 2023

[GCC05] GCC: *GCC Vectorizer Dump Reports*. https://gcc.gnu.org/legacy-ml/gcc-patches/2005-01/msg01247.html. Version: 2005. – visited on December 30, 2022

[GCC22a] GCC: *GCC Auto Vectorization*. https://gcc.gnu.org/projects/tree-ssa/vectorization.html. Version: 2022. – visited on December 30, 2022

[GCC22b] GCC: *GNU Compiler Collection Landing Page*. https://gcc.gnu.org. Version: 2022. – visited on December 27, 2022

[GCC23a] GCC: *Built-in Functions Provided by GCC*. https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. Version: 2023. – visited on January 26, 2023

[GCC23b] GCC: *Extended Asm - Assembler Instructions with C Expression Operands*. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html. Version: 2023. – visited on March 19, 2023

[GCC23c] GCC: *GCC 11 Release Series Changes, New Features, and Fixes*. https://www.gnu.org/software/gcc/gcc-11/changes.html. Version: 2023. – visited on March 4, 2023

[GCC23d] GCC: *GCC Libgcc2 Library*. https://github.com/gcc-mirror/gcc/blob/master/libgcc/libgcc2.c. Version: 2023. – visited on March 21, 2023

[GCC23e] GCC: *Options That Control Optimization*. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options. Version: 2023. – visited on March 20, 2023

[Goo21] Google: *Google Benchmark CPU Affinity Pull Request*. https://github.com/google/benchmark/pull/988. Version: 2021. – visited on April 12, 2023

[Goo22a] Google: *Google Highway Design Philosophy*. https://github.com/google/highway/blob/master/g3doc/design_philosophy.md. Version: 2022. – visited on January 18, 2023

[Goo22b] Google: *Google Highway Documentation*. https://github.com/google/highway/tree/master/g3doc. Version: 2022. – visited on December 3, 2022

[Goo22c] Google: *Google Highway GitHub*. https://github.com/google/highway. Version: 2022. – visited on November 22, 2022

[Goo22d] Google: *Google Highway Introduction*. https://github.com/google/highway/blob/master/g3doc/highway_intro.pdf. Version: 2022. – visited on December 4, 2022

[Goo22e] Google: *Google Highway Quick Reference*. https://github.com/google/highway/blob/master/g3doc/quick_reference.md. Version: 2022. – visited on December 3, 2022

[Goo23] Google: *Google Benchmark GitHub*. https://github.com/google/benchmark. Version: 2023. – visited on January 19, 2023

[Int22a] Intel: *Intel Intrinsics Guide*. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#. Version: 2022. – visited on November 8, 2022

[Int22b] Intel: *Using Auto Vectorization with Intel® C++ Compiler*. https://www.intel.com/content/www/us/en/docs/cpp-compiler/tutorial-auto-vectorization/18-0/overview.html. Version: 2022. – visited on March 19, 2023

[Int23] Intel: *Contiguous Memory Accesses*. https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/optimization-guide/2023-0/contiguous-memory-accesses.html. Version: 2023. – visited on March 6, 2023

[JR15] Jeffers, Jim ; Reinders, James: *High performance parallelism pearls volume two: multicore and many-core programming approaches*. Morgan Kaufmann, 2015

[KL12] Kretz, Matthias ; Lindenstruth, Volker: Vc: A C++ library for explicit vectorization. In: *Software: Practice and Experience* 42 (2012), Nr. 11, S. 1409–1430

[KMSZ15] Kaeli, David R. ; Mistry, Perhaad ; Schaa, Dana ; Zhang, Dong P.: *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann, 2015

[Lib22a] Libsimdpp: *Libsimdpp GitHub*. https://github.com/p12tic/libsimdpp. Version: 2022. – visited on Januar 19, 2023

[Lib22b] Libsimdpp: *Libsimdpp Web Documentation*. http://p12tic.github.io/libsimdpp/v2.2-dev/libsimdpp/w/index.html. Version: 2022. – visited on December 8, 2022

[Lir09] LiraNuna: *SSE intrinsics optimizations in popular compilers*. https://www.liranuna.com/sse-intrinsics-optimizations-in-popular-compilers/. Version: 2009. – visited on March 19, 2023

[LLV23] LLVM: *Auto-Vectorization in LLVM*. https://www.llvm.org/docs/Vectorizers.html. Version: 2023. – visited on March 19, 2023

[Mar18] Marinescu, Dan C.: *Cloud computing: theory and practice*. Morgan Kaufmann, 2018

[MCSJ+22] Marques, Diogo ; Campos, Rafael ; Santander-Jiménez, Sergio ; Matveev, Zakhar ; Sousa, Leonel ; Ilic, Aleksandar: Unlocking Personalized Healthcare on Modern CPUs/GPUs: Three-way Gene Interaction Study. In: *arXiv preprint arXiv:2201.10956* (2022)

[NDR+11] Nuzman, Dorit ; Dyshel, Sergei ; Rohou, Erven ; Rosen, Ira ; Williams, Kevin ; Yuste, David ; Cohen, Albert ; Zaks, Ayal: Vapor SIMD: Auto-vectorize once, run everywhere. In: *International Symposium on Code Generation and Optimization (CGO 2011)* IEEE, 2011, S. 151–160

[NSI18] NSIMD: *The state of boost.simd*. https://github.com/agenium-scale/boost.simd/issues/545. Version: 2018. – visited on March 9, 2023

[NSI21a] NSIMD: *NSIMD GitHub*. https://github.com/agenium-scale/nsimd. Version: 2021. – visited on March 9, 2023

[NSI21b] NSIMD: *NSIMD Web Documentation*. https://agenium-scale.github.io/nsimd/index.html. Version: 2021. – visited on March 4, 2023

[Ope18] OpenMP: *OpenMP Application Programming Interface*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Version: 2018. – visited on January 4, 2023

[Pac22] Pacheco, Peter: *An introduction to parallel programming*. Elsevier, 2022

[PCM+16] Pohl, Angela ; Cosenza, Biagio ; Mesa, Mauricio A. ; Chi, Chi C. ; Juurlink, Ben: An evaluation of current SIMD programming models for C++. In: *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, 2016, S. 1–8

[PJ15] Porpodas, Vasileios ; Jones, Timothy M.: Throttling automatic vectorization: When less is more. In: *2015 International Conference on Parallel Architecture and Compilation (PACT)* IEEE, 2015, S. 432–444

[PS21] Pure-SIMD: *Pure SIMD GitHub*. https://github.com/eatingtomatoes/pure_simd. Version: 2021. – visited on March 9, 2023

[Sch16] Schmitz, Andreas: *GCC Autovectorization*. https://hpac.cs.umu.se/teaching/sem-accg-16/slides/08.Schmitz-GGC_Autovec.pdf. Version: 2016. – visited on January 18, 2023

[SE23] SIMD-Everywhere: *SIMD Everywhere GitHub*. https://github.com/simd-everywhere/simde. Version: 2023. – visited on January 21, 2023

[Sie16] Siewert, Sam: *Algorithm Acceleration Using Single Instruction Multiple Data*. https://www.intel.com/content/www/us/en/developer/articles/technical/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitives-to-accelerate-algorithms.html. Version: 2016. – visited on November 8, 2022

[SJD+18] Sornet, Gauthier ; Jubertie, Sylvain ; Dupros, Fabrice ; De Martin, Florent ; Limet, Sebastien: Performance analysis of SIMD vectorization of high-order finite-element kernels. In: *2018 International Conference on High Performance Computing & Simulation (HPCS)* IEEE, 2018, S. 423–430

[SS21] Std-Simd: *Std-Simd GitHub*. https://github.com/VcDevel/std-simd. Version: 2021. – visited on January 18, 2023

[Vc22a] Vc: *Vc GitHub*. https://github.com/VcDevel/Vc. Version: 2022. – visited on January 18, 2023

[Vc22b] Vc: *Vc Web Documentation*. https://vcdevel.github.io/Vc-1.4.3/. Version: 2022. – visited on December 8, 2022

[Wik23] Wikipedia: *Discontinued Google Services*. https://en.wikipedia.org/wiki/List_of_Google_products#Discontinued_products_and_services. Version: 2023. – visited on February 18, 2023

[xSI23] xSIMD: *xSIMD GitHub*. https://github.com/xtensor-stack/xsimd. Version: 2023. – visited on March 29, 2023