`std::simd` in Rust and C++ — A Practical Comparative Analysis

kleines Filmröllchen

IPVS

Abstract. While SIMD capabilities have existed in consumer and high-performance computers for several decades, their explicit use has often been relegated to architecture-specific intrinsic functions or handwritten assembly code. Recent improvements in ergonomics and possible utility of SIMD programming have therefore been introduced via the `std::simd` libraries in the systems programming languages C++ and Rust. This paper aims to explore how the two libraries differ, and how their performance, ergonomics, and utility compares to auto-vectorization. For this purpose, two example problems, Pixelflut and FLAC, are implemented.

1 Introduction

Single Instruction Multiple Data (SIMD) hardware provides a capability for improving the performance of certain kinds of linear-algebra like workloads. However, the best architecture features provide no benefit if programmers cannot make effective use of them, as demonstrated by Intel's Itanium architecture. Apart from SIMD libraries, three primary techniques are available: Assembly programming of SIMD instructions as an expensive [1] method of last resort [26]; intrinsics making instructions available as functions in higher-level languages [18]; and OpenMP [4] vectorization hints. While the first two are low-level and hardware specific, the latter is so high-level it barely improves over a compiler's ability to auto-vectorize certain code [19].

Hardware-agnostic SIMD libraries, on the other hand, attempt to fill this gap by providing high-level SIMD types, increased memory safety, and ergonomic SIMD operations, while still compiling to efficient machine code. While a programmer needs to be familiar with SIMD principles, they need not have knowledge of specific hardware implementations, and can easily write code that will run on any supported platform present and future.

Recently, SIMD libraries have been added to the standard libraries of C++ and Rust, both called `std::simd`. While C++'s SIMD library has been mostly approved for the ISO C++ 26 standard, Rust's SIMD library is still in an experimental phase. This presents a unique opportunity for comparison, as these libraries have been developed at about the same time.

When newly adding manual SIMD optimization to a program, the programmer must first gauge whether manual optimization is even worth the effort. Programmers might not be aware that their code is already being auto-vectorized

well enough, or they might only need certain compiler flags to get there. Additionally, choice of programming language may also be influenced by the available SIMD support and its performance. C++ and Rust, two languages seen as direct competitors, are both generally suitable for HPC and other applications that require intense use of SIMD.

The aim of this paper is therefore to answer questions around the choice of using the new SIMD libraries, and whether picking Rust over C++ (or C++ over Rust) is worth it just for `std::simd`. The main focus will be on performance, utilizing a four-way comparison of the two languages in auto-vectorization as well as in `std::simd`. For this we use two example problems, Pixelflut parsing and selected FLAC algorithms.

2 Methods

2.1 Single Instruction Multiple Data

"Single Instruction Multiple Data" (SIMD), coined by Flynn [7], is part of a four-way distinction between systems based on the number of instruction and data streams. Most single-processor systems fit into the default Single Instruction Single Data(SISD) paradigm, where one stream of instructions acts on one stream of data. Multiple Instruction Multiple Data (MIMD) represents multiple independent instruction-data-stream-pairs, such as multiprocessor systems. Hence SIMD occupies a middle ground: while only having one instruction stream, multiple data streams exist and have identical operations applied to them. In practice, SIMD capability in general-purpose microprocessors is provided as additional instructions, usually with dedicated registers. In this context, the specific terms vector instructions and vector registers are used, as can be seen in extension names such as RISC-V's "V" extension, AArch64's SVE (scalable *vector* extension), and x86-64's AVX (advanced *vector* extensions) family.

Each vector register, usually 128 to 512 bits in size, consists of a number of lanes, each of which hold one independent data element. The data type used for a lane is flexible (e.g. 16-bit or 64-bit signed/unsigned integers, single or double precision floating point), and its size determines how many lanes there are. For example, a 256-bit vector (as seen in ARM NEON and AVX) can hold 4 64-bit integer or double-precision floating-point lanes, but 32 8-bit lanes. Vector instructions can perform calculations independently within a lane (vertical operations) or across lanes (horizontal operations or reductions).

Masks are used to exclude and include certain lanes from calculations. Mask registers are bitfields, holding a single bit per lane, and many vector operations can indicate that only the lanes who have their corresponding bit set in a mask are to be processed by the operation. Masks are a powerful feature allowing for fast, branchless SIMD programming. The program can first construct a mask register from a boolean-valued calculation with vectors, then use the mask to only process elements that match the condition.

Loading and storing SIMD vectors is considered the most performance-critical aspect of SIMD calculations, as the required memory bandwidth is high. Usually,

vectors must be aligned to the vector size (as opposed to the element size). Most SIMD instruction sets support loads and stores with varying strides, thereby allowing efficient access to interleaved and structured data.

2.2 `std::simd` in Rust

Rust [13] is a systems programming language developed since 2006 by Graydon Hoare at Mozilla. Since its first stable release ten years ago, Rust has achieved prominence and is currently one of the most widely used systems programming languages. As a significant deviation from older languages, Rust provides high-level abstractions inherited from functional and concurrent paradigms. Rust has an advanced formally verified [12] type system that disallows entire categories of safety and correctness bugs. Nevertheless, Rust achieves significant performance by using LLVM as a code-generating backend, making it generally equivalent to C++ and a suitable option for high-performance computing.

`std::simd` in Rust was proposed with RFC 2948 [22] in 2020, and work is presently continuing towards a library that can be released as part of the stable Rust language. An unstable "nightly" compiler is currently required to use it. Rust centrally provides the `std::simd::Simd<T, N>` datastructure for a single SIMD vector. This type is parameterized around the element type `T` and the lane count `N`. The regular array type `[T; N]` is compatible with `Simd<T, N>` and conversion is facilitated by the `to_array` function. Furthermore, SIMD vectors can be loaded from slices (non-owned array spans), optionally with fill values in case the slice is not large enough to fill the entire vector. Basic operations are provided via overloaded operators. Many more operations are provided as member functions on `Simd`.

Masking operations are supported via the `std::simd::Mask<T, N>`. This type can be constructed from boolean arrays or bitmasks, but also from SIMD comparisons, such as `simd_lt` for element-wise "$<$". In turn, many operations on `Simd` provide a masked version, such as `store_select`. The `Mask` type is specific to a certain element type, though lossless type conversions are available for equal lane count.

2.3 `std::simd` in C++

C++ [24] is a popular systems programming language. Originally developed by Bjarne Stroustrup as an object-oriented layer on top of C in the late 1980's, it has become one of the most widely used programming languages for all kinds of applications, and it is very prevalent in high-performance computing. C++ is standardized by the International Standards Organization (ISO) [11], with the upcoming standard iteration known as C++26.

Matthias Kretz proposed the `std::simd`[1] library [14] and it was accepted as P1928R12 [15] into the C++26 standard as of November 2024. It

[1] As currently accepted, the name of the library is planned to be `std::datapar`, but a recent paper by the authors requests to change the name to `std::simd` [16].

is not yet available by default in compilers, but many, including the GNU Compiler Collection [23] (GCC), and LLVM [17] (via Clang), already provide `std::experimental::simd`[2]. In this work, we will use the Clang compiler, as it shares its backend (LLVM) with Rust and is therefore well-comparable, but its support of `std::experimental::simd` (in libc++) proved to be insufficient for the tasks at hand. Therefore, libstdc++ from GCC is used as the C++ standard library instead, as it provides near-complete support.

`simd<T, Abi>` is the central SIMD vector type. While the first template parameter determines the lane type, the second determines the application binary interface (ABI), which is responsible for a flexible lane count choice. For instance, while `simd_abi::fixed_size<N>` always chooses a fixed number of lanes, `simd_abi::native<T>` chooses a number of lanes that is most efficient on the target architecture. There are further ABI tags available for special cases. In this paper we will be using both the fixed-size and native ABI tags.

Operations on SIMD types are provided via operator overloads (e.g. `+` for vertical addition) or (mostly) free functions. Masking is provided with the `simd_mask<T, Abi>` struct, which contains boolean elements and has a few basic free functions available. Loading and storing masks or vectors is possible from any iterator (e.g. spans or pointers), and these functions additionally take an aligment tag that allows the function to assume certain alignments of the memory (e.g. vector-size-aligned or element-aligned).

2.4 Pixelflut

Pixelflut [9] ("pixel flood") is an experimental networking protocol and multiplayer game popular in the Chaos Computer Club (CCC). Pixelflut servers hold a shared pixel canvas, usually projected onto a venue wall, and many clients can send arbitrary pixels to the server via TCP. To have any kind of visual object displayed permanently, a client must rapidly and continuously send all pixels of the object. "Winning" at Pixelflut, having one's visuals displayed prominently, consists of sending pixel data as fast as possible, often achieved via exotic means such as TCP abuse. Servers are in turn confronted with upwards of 100 Gbit/s continuous traffic. SIMD implementation ideas for this paper were taken from the notable high-performance server breakwater [2].

A significant bottleneck in Pixelflut servers is the parsing of Pixelflut text-based commands, which must be translated to actions on the canvas. Pixelflut commands span one line of text each, and may be one of the following:

- `PX X Y RRGGBB` or `PX X Y RRGGBBAA`: Set a pixel at coordinates (X, Y) to the given 24-bit/32-bit (hexadecimal) color with optional alpha.
- `OFFSET X Y`: Have the server offset all future drawing commands from this client appropriately, allowing smaller X and Y offsets and lower bandwidth per pixel.

[2] The differences to C++ 26 `std::simd` are minute and irrelevant here; see P1928R12 Section 4 for details.

- `SIZE`: Prompt the server to return the size of the canvas, as Pixelflut allows for any size of canvas to be used.
- `QUIT`: Ask the server to terminate the connection.

To increase the variety of command types, the less common `HELP` command is additionally recognized, which informs the client of the available commands.

For the implementation in this paper, a simplified but realistic parsing task is utilized. The program receives a slightly augmented stream of ≈ 250 MB of Pixelflut commands, captured at one of the author's recent Pixelflut events. The program then runs the parser on this data, converting it to an internal representation, and writing back a serialized version. This allows for easy correctness checking. We only benchmark parsing, not I/O or serialization.

2.5 FLAC

The Free Lossless Audio Codec (FLAC) [3] by Coalson, van Beurden, Weaver et al. is the currently most widely used lossless audio compression codec, expanding on ideas propagated by SHORTEN [21] and AudioPak [8] and utilizing fundamental algorithms like Rice coding [20]. FLAC was developed around 2001 with an open-source reference implementation and published as an Internet Engineering Task Force (IETF) standard in 2024.

For this paper, we implement three small aspects of FLAC encoding and decoding that are both performance-critical and lend themselves to auto-vectorization:

- When stereo audio is encoded, channel decorrelation can be employed, as the left and right channels of one sample contain almost the same information. They are therefore combined into a middle channel $m = \left\lfloor \frac{l+r}{2} \right\rfloor$ and a side channel $s = l - r$, where the side channel will have lower entropy and can therefore be encoded more efficiently.
- Central to FLAC's coding is linear predictive coding (LPC) [6], where sample values are approximated using a linear combination of up to 32 previous samples. As each sample's prediction depends on all the samples just before it, this problem does not appear to be trivially auto-vectorizable. However, the linear combination should lend itself to SIMD computing like matrix multiplications.
- In order to find out whether certain residual values are efficient to encode, we want to calculate the per-element cost under Rice-Golomb coding with certain parameters. An algorithm for this is given in the FLAC reference encoder source code, which includes an independent per-residual calculation.

All data used for these tasks was derived from FLAC specification test suite [10] files.

3 Results

All tests were performed on machines graciously provided by the Institute for Parallel and Distributed Systems (IPVS). For a better performance analysis,

Name	CPU	Architecture	SIMD extension	Cache (L1/L2/L3)	RAM
xenon	2 × 64 AMD EPYC 7543	x86-64	AVX2	64 KiB / 512 KiB / 512 MiB	3.93TB
pcsgs09	1 × 12 Intel Core i9-10920X	x86-64	AVX512	64 KiB / 1 MiB / 19.3 MiB	64GB
krypton	2 × 40 IBM POWER9	PowerPC 64 LE	VSX (Power 2.2)	64 KiB / 512 KiB / 10 MiB	381GB

Table 1. System specifications for benchmarking machines. Only physical cores are listed. SIMD extension is the highest available. Cache is given per-core.

x86 CPUs from different manufacturers (Intel and AMD) in addition to an IBM POWER CPU were used. Relevant specifications are provided in Table 1.

The author's previous experience with writing explicitly vectorized code was limited, although they are well-familiar with SIMD concepts. The author is an experienced C++ and Rust programmer. The author has experience with Pixelflut and is the main author of a C++ FLAC codec [25]; thus they have solid previous experience with both problem sets.

3.1 Pixelflut

The first implementation was written in Rust. Significant time was spent on a SIMD-based decimal number parser that turned out to be slower than a naive implementation. More software engineering metrics have been evaluated in Table 2. Vectorized aspects of the parser now include finding spaces and newlines along which to split commands and their arguments (Listing 1.1), determining which command is present by bitmasking, and parsing hexadecimal numbers (Listing 1.2).

Metric	Rust plain	Rust SIMD	C++ plain	C++ SIMD
Implementation time	1h	6h	3h	8h
Implementation difficulty	Trivial	Medium	Easy	Hard
Debugging difficulty	None	Easy	Easy	Medium
Length (lines of code)	86	101	130	194

Table 2. Software engineering metrics relating to the Pixelflut implementations. Implementation difficulty was measured according to (1) frequency of needing to rewrite code according to compiler errors and (2) frequency of consulting documentation.

We then ported the parser to C++, utilizing `fixed_simd`. Significant hurdles were encountered here. Notably, (1) masks cannot be converted to bitfield integers easily, complicating some operations on masks, (2) memory load with fallback values (in case the underlying memory array is too short) is missing, (3) shifting entire elements around in a vector is missing, though can be imple-

```
const NEWLINES: u8x32 = u8x32::splat(b'\n');
const SPACES: u8x32 = u8x32::splat(b' ');
let newline_positions = NEWLINES.simd_eq(input).to_bitmask();
// arguments = vector containing command args
// offset = whether this is "OFFSET"
let space_positions = SPACES.simd_eq(arguments).to_bitmask()
    | (newline_positions >> if offset { 7 } else { 3 });
```

Listing 1.1. Finding spaces and newlines with SIMD instructions in Rust.

```
const SHIFT_PATTERN: u32x8 = u32x8::from_array([4, 0, 12, 8, 20, 16, 28, 24]);
const SIMD_6: u32x8 = u32x8::from_array([6; 8]);
const SIMD_F: u32x8 = u32x8::from_array([0xf; 8]);
const SIMD_9: u32x8 = u32x8::from_array([9; 8]);
fn parse_hex(rgb_vec: u32x8) -> u32 {
    let sr6 = rgb_vec >> SIMD_6;
    let and15 = rgb_vec & SIMD_F;
    let mul = sr6 * SIMD_9;
    let hexed = and15 + mul;
    let shifted = hexed << SHIFT_PATTERN;
    shifted.reduce_or()
}
```

Listing 1.2. Parsing hexadecimal 32-bit integers with SIMD in Rust.

mented in a few lines of code via `split`. Debugging was hindered by the inability to easily print out SIMD vectors.

Initially, the C++ and Rust plain parsers had vastly differing performance, with some simple optimizations based on *perf* [5] information first applied to the C++ parser and then the Rust parser to fix major performance oversights in both. A third implementation "hexsimd" only uses SIMD to parse hexadecimal numbers (and decimal numbers in a second iteration), which is the single most expensive operation in the parser. It should be noted that these hexsimd parsers are effectively equivalent to breakwater's parser [2].

Detailed performance data is available in Figure 1. While variance of benchmarks was measured, it is smaller than would be visible.

In general, the full-SIMD parser is the slowest Rust implementation but the fastest C++ implementation, though on AVX512, it performs well in Rust too. Generally, C++ full-SIMD performs better than Rust full-SIMD, while for hexsimd the situation is usually opposite. Heavily dependent on the machine is the relative placement of the normal parser, though its performance is consistent across machines and languages. Even for the here-used smaller-than-possible vectors (8×32), SIMD on AVX512 appears to perform better than on other architectures.

3.2 FLAC

The three FLAC problems were initially implemented in C++ and later ported to Rust. As opposed to Pixelflut, not many hurdles in either API were encountered. For Rice-Golomb bitcount and stereo decorrelation, the C++ implementation uses `native_simd` as seen in Listing 1.3, which we emulated in an inferior and platform-dependent way in Rust with a larger SIMD type for

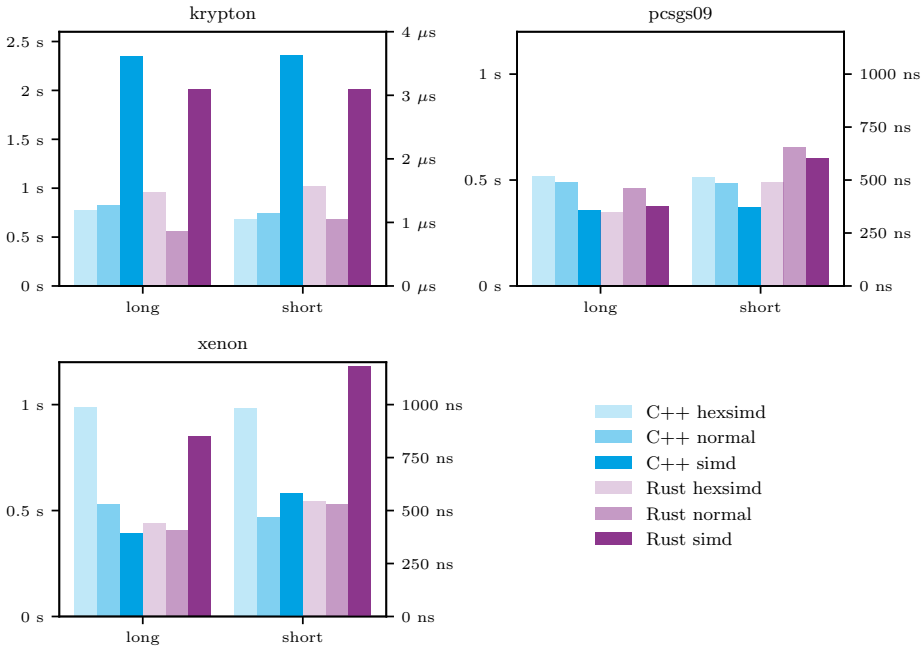

Fig. 1. Pixelflut parsers wall-time on short (1 KB) and long (250 MB) datasets, grouped per machine and dataset. Benchmark time is averaged over >60s of measurements.

AVX512. Convenient in Rust was the ability to split any slice into three chunks (`slice::as_simd`): a head, a list of aligned SIMD vectors, and a tail. This allows easy usage of aligned SIMD memory accesses, which are considered faster than unaligned memory accesses.

Performance data is available in Figure 2. The most striking difference between the two languages can be observed on PowerPC, where C++ appears to fail to properly vectorize even the SIMD implementations. While for Rust, the manual vectorization mostly yields faster code, for C++ this is never the case. The exception to this is Rice-Golomb bitcounting, which performs better in manual vectorization in both languages. LPC is the opposite, performing slower in its SIMD variant across the board.

4 Discussion

The most surprising observation for Pixelflut is that the plain Rust parser remains as the most performant parser. Furthermore, as expected, the Rust SIMD parser is very slow on PowerPC, as intrinsics are missing on this platform. Only on AVX512 does the SIMD parser perform well in both languages.

With the FLAC problems, we observe that for both languages, stereo decorrelation is the only problem which auto-vectorizes well. With both LLVM and

```
size_t partition_bits = 4 + (1 + order) * residuals.size();
using svec = native_simd<i16>;
auto const limit = residuals.size() - (residuals.size() % svec::size());
size_t i = 0;
for (; i < limit; i += svec::size()) {
    svec const residuals_vec_16{ &residuals[i], element_aligned_tag{} };
    auto const unary_bits = (residuals_vec_16 << 1) ^ (residuals_vec_16 >> 15);
    auto const unary_bps = static_simd_cast<u32>(unary_bits) >> order;
    auto const val = reduce(unary_bps, std::plus {});
    partition_bits += val;
}
if (i != residuals.size()) {
    i -= svec::size() - 1;
    for (; i < residuals.size(); ++i)
        partition_bits += static_cast<u32>((residuals[i] << 1) ^ (residuals[i] >> 15)) >> order;
}
```

Listing 1.3. Calculating required bits for Rice-Golomb coding in C++ using SIMD, simplified. While the first loop uses SIMD, the second loop handles the non-vectorizable tail.

GCC, compiler-generated code outperforms human-written code as usual. While LPC prediction does not lend itself well to vectorization in the first place, Rice-Golomb bitcounting serves as a good example where auto-vectorization fails and manually vectorized code performs better.

Irrespective of observations about SIMD programming in general (memory alignment, cache usage, tail handling, lane type casting, vector size choice, and many more), the performance numbers generally favor Rust's `std::simd`. In many cases, manually vectorized Rust code outperforms plain Rust code, and even when this is not the case (e.g. Pixelflut with hexsimd), the performance is often not significantly worse than the plain code. This is an important factor for portable code, as platforms without SIMD available should be able to use the manually vectorized implementation without performance loss. Additionally, even on the not directly supported PowerPC architecture, Rust SIMD code often results in performance improvements regardless, while it only sometimes does for C++, where, based on our reading of libstdc++, GCC should support PowerPC vector instructions directly.

It appears that C++'s `std::simd` is not ready for production use. While `native_simd` is an incredibly useful feature missing from Rust, we found most of the APIs missing features in comparison, the debuggability slightly inferior, and the resulting code longer and more complex. Especially unconventional problems like Pixelflut that rely heavily on complex masking operations are not easily vectorized with C++'s APIs. Rust's implementation has a mature feature set, is comfortable to use, and appears largely ready for stabilization to us.

Regardless, both APIs are mostly a significant improvement over other options discussed in the introduction. As observed, auto-vectorization cannot be relied upon as is the case with many compiler optimizations, and utilizing them or vectorization pragmas is not an option in general. Using `std::simd` is hardware-independent unlike intrinsics, never requiring the programmer to think about a particular SIMD extension beyond its register width. While the `std::simd` libraries often provide a path to performance improvements, they still exhibit

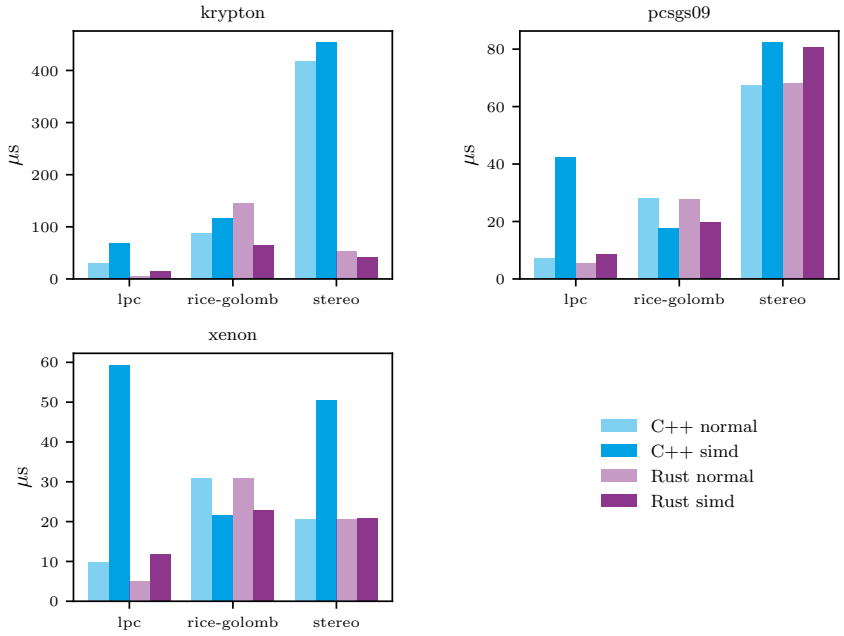

Fig. 2. FLAC benchmarks wall-time on data based off FLAC test file 17, grouped per machine and problem. Benchmark time is averaged over >60s of measurements.

vast differences in runtime on different SIMD extensions and should be evaluated carefully on each target platform.

4.1 Outlook

Whether `std::simd` will be stable in C++ before Rust remains an open question; C++26 will likely not be available for at least another year in major compilers, and Rust stabilization can sometimes move fast. While both libraries provide measurable benefits in terms of performance and software engineering already today, adoption by production projects only happens with stable compiler support.

Our work provided a good initial perspective on the two `std::simd` libraries, and based on this well-founded software engineering decisions can be made. However, several open avenues for research exist. We did not concern ourselves in-depth with the software engineering aspects of `std::simd`, such as in regards to maintainability and readability of high-level SIMD code. Additionally, no experienced SIMD developers were involved in this paper, skewing its performance results. Furthermore we did not analyze performance on several relevant SIMD-supporting architectures, such as ARM's AArch64 NEON and SVE extensions as well as RISC-V's V extension.

References

1. Backus, J.: The history of Fortran I, II, and III. IEEE Annals of the History of Computing **20**(4), 68–78 (1998). https://doi.org/10.1109/85.728232, `https://www.softwarepreservation.org/projects/FORTRAN/paper/p165-backus.pdf`
2. Bernauer, S.: breakwater – Fast Pixelflut server written in Rust (2022), `https://github.com/sbernauer/breakwater`
3. van Beurden, M.Q., Weaver, A.: Free Lossless Audio Codec (FLAC). RFC 9639 (Dec 2024). https://doi.org/10.17487/RFC9639, `https://www.rfc-editor.org/info/rfc9639`
4. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Computational Science and Engineering **5**(1), 46–55 (1998). https://doi.org/10.1109/99.660313
5. De Melo, A.C.: The new linux 'perf' tools. In: Slides from Linux Kongress. vol. 18 (2010)
6. Durbin, J.: The fitting of time-series models. Revue de l'Institut International de Statistique / Review of the International Statistical Institute **28**(3), 233–244 (1960), `http://www.jstor.org/stable/1401322`
7. Flynn, M.: Very high-speed computing systems. Proceedings of the IEEE **54**(12), 1901–1909 (1966). https://doi.org/10.1109/PROC.1966.5273
8. Hans, M., Schafer, R.: Lossless compression of digital audio. IEEE Signal Processing Magazine **18**(4), 21–32 (2001). https://doi.org/10.1109/79.939834
9. Hellkamp, M.: Pixelflut - multiplayer canvas, `https://github.com/defnull/pixelflut`
10. IETF Codec Encoding for LossLess Archiving and Realtime transmission Working Group, van Beurden, M.: FLAC decoder test suite. GitHub repository, `https://github.com/ietf-wg-cellar/flac-test-files`
11. ISO/IEC 14882:2024: Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH (Oct 2024)
12. Jung, R.: Understanding and evolving the rust programming language (2020). https://doi.org/http://dx.doi.org/10.22028/D291-31946
13. Klabnik, S., Nichols, C.: The Rust programming language. No Starch Press (2023)
14. Kretz, M.: Data-parallel vector types and operations. In: JTC1/SC22/WG21 Papers (2018), `https://wg21.link/p0214r9`
15. Kretz, M.: std::simd – merge data-parallel types from the parallelism ts 2. In: JTC1/SC22/WG21 Papers (2024-10-11), `https://wg21.link/p1928r12`
16. Kretz, M., Majumder, A., Lelbach, B.A., Towner, D., Burylov, I., Hoemmen, M., Arutyunyan, R.: Reconsider naming of the namespace for "std::simd". In: JTC1/SC22/WG21 Papers (2025), `https://wg21.link/p3691r0`
17. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (Mar 2004)
18. Lomont, C.: Introduction to intel advanced vector extensions. Intel white paper **23**(23), 1–21 (2011), `https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf`
19. Nuzman, D., Zaks, A.: Outer-loop vectorization: revisited for short SIMD architectures. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. pp. 2—11. PACT '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1454115.1454119

20. Rice, R., Plaunt, J.: Adaptive variable-length coding for efficient compression of spacecraft television data. IEEE Transactions on Communication Technology **19**(6), 889–897 (1971). https://doi.org/10.1109/TCOM.1971.1090789
21. Robinson, T.: SHORTEN: Simple lossless and near-lossless waveform compression. Tech. rep., Cambridge University Engineering Department (dec 1994)
22. Sivonen, H., gnzlbg: RFC 2948: Portable packed SIMD vector types. In: Rust Language Requests For Comments (2020), `https://github.com/rust-lang/rfcs/pull/2948`
23. Stallman, R.M., et al.: Using and porting the GNU compiler collection. Addison-Vesley Publishing, New York.–2000.–556 p (1988)
24. Stroustrup, B.: An overview of C++. In: Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming. pp. 7–18 (1986)
25. The SerenityOS Project: LibAudio FLAC codec. GitHub repository (2025), `https://github.com/SerenityOS/serenity/blob/master/Userland/Libraries/LibAudio/FlacLoader.h`
26. Wang, J., Wang, L., Wang, P.: Optimisation of x264 encoder acceleration based on RISC-V vector instructions. In: 2023 3rd International Symposium on Computer Technology and Information Science (ISCTIS). pp. 1128–1133 (2023). https://doi.org/10.1109/ISCTIS58954.2023.10213200