Evaluating SIMD Compiler-Intrinsics for Database Systems

Lawrence Benson¹, Richard Ebeling¹ and Tilmann Rabl¹

¹Hasso Plattner Institute, University of Potsdam

Abstract
Modern engines often use SIMD instructions to speed up query performance. As these instructions are heavily CPU-specific, developers must write multiple variants of the same code to support multiple target platforms such as AVX2, AVX512, and ARM NEON. This process leads to logical code duplication, which is cumbersome, hard to test, and hard to benchmark. In this paper, we make the case for writing less platform-specific SIMD code by leveraging the compiler’s own platform-independent SIMD vector abstraction. This allows developers to write a single code variant for all platforms as with a SIMD library, without the library’s redundant layers of abstraction. Clang and GCC implement the platforms’ SIMD intrinsics on top of their own abstraction, so code written in it is optimized for the underlying vector instructions by the compiler. We conduct four database operation microbenchmarks based on code in real systems on x86 and ARM and show that compiler-intrinsic variants achieve the same or even better performance than platform-intrinsics in most cases. In addition, we completely replace the SIMD library in the state-of-the-art query engine Velox with compiler-intrinsics. Our results show that query engines can achieve the same performance with platform-independent code while requiring significantly less SIMD code and fewer variants.

1. Introduction

Numerous databases and query processing engines make use of vector instructions (SIMD = single instruction, multiple data) to speed up query processing. By performing the same operation on multiple data items at once, SIMD instructions increase the performance of these systems significantly [1, 2, 3]. However, SIMD instructions are CPU-specific and come with various register widths and capabilities. Most modern x86 servers support SSE, AVX, and AVX2, while a wide range of Intel servers also support AVX-512, only the newest AMD servers support it. With the rise of ARM in the cloud, query engines now also target NEON vector instructions and will soon likely target the new scalable vector extension (SVE), which is available in AWS’ latest ARM servers. Supporting all of these instruction sets requires many set-specific code variants, resulting in significant logical code duplication and checks to select the correct version. Optimizing performance-critical SIMD code in this setup is cumbersome, hard to benchmark, and hard to test.

Various SIMD libraries exist to reduce the complexity of supporting multiple vector instruction sets [4, 5, 6]. These libraries often provide a thin abstraction on top of the platform-specific SIMD intrinsics (platform-intrinsics). However, the explicit use of intrinsics directly or via a library can hinder compiler optimizations, as we show in our benchmarks. If an operation cannot be expressed using the library’s abstractions, systems still require multiple platform-specific methods to handle special cases and optimizations. This is the case, e.g., in the query engine Velox [7], which uses a SIMD library to structure the vectorized code, but still contains dozens of platform-specific functions for AVX2 and NEON that implement the same logic but for different types and sizes.

In this paper, we make the case for writing less platform-specific SIMD code in databases while also removing redundant layers of abstraction. We show the redundant layers of abstraction by inspecting how x86’s and NEON’s SIMD-intrinsics to add two 128-bit vectors of four 32-bit integers (__m_add_epi32 and vaddq_s32) are implemented in compilers and abstracted from in SIMD libraries. Both Clang and GCC provide platform-independent vector intrinsics (compiler-intrinsics) [8, 9], based on C/C++’s primitive types and compiler attributes. The vector-addition platform-intrinsics are implemented on top of these compiler-intrinsics, as outlined below.

// Simpilfied from Clang’s <emmintrin.h> header. // Public 16-byte __m128i type in x86 SIMD API. typedef long __m128i __attribute__((vector_size(16))); // Internal 16-byte vector of four integers. typedef int __v4su __attribute__((vector_size(16))); __m128i __m128i __attribute__((vector_size(16))); __m128i __attribute__((vector_size(16))); // Simplified from Clang’s <arm_neon.h> header. int32x4_t is defined analogously to __v4su. int32x4_t int32x4_t __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute__((vector_size(16))); __attribute((}
In this paper, we conduct four microbenchmarks on multiple x86 and ARM servers with both Clang and GCC. The benchmarks represent query processing operations (hashing, fingerprinted hash bucket lookups, bit-packing, column filter scan), based on implementations found in real systems. We find that in 7 out of 8 benchmark setups, compiler-intrinsics perform on par with or better than hand-written platform-intrinsics. We also demonstrate our approach in the state-of-the-art query engine Velox [7], from which we remove all platform-specific SIMD code by replacing their SIMD library dependency with compiler-intrinsics. Our results running Velox TPC-H on x86 and ARM show that we can remove hundreds of lines of platform-specific SIMD code while maintaining equal performance. In summary, we make the following contributions:

1) We compare the performance of compiler-intrinsics to scalar, auto-vectorized, and platform-specific SIMD code in four database operation microbenchmarks.

2) We replace all platform-specific SIMD code in the query engine Velox with compiler-intrinsics and show the impact on end-to-end TPC-H workloads.

3) Based on our results, we make the case for platform-independent SIMD code in databases and discuss open challenges.

In Section 2, we discuss vectorized processing concepts, before introducing compiler-intrinsics in Section 3. In Sections 4 and 5, we present our benchmark results on various machines, compiled with Clang and GCC. We discuss our findings in Section 6. We conclude our work in Section 8 after presenting related work in Section 7.

2. Vectorized Processing

In this section, we give a short overview of SIMD operations commonly used in databases. We then discuss the usage of SIMD instructions via intrinsics in C++ code. Finally, we briefly lay out auto-vectorization efforts in modern compilers.

SIMD. When using SIMD operations, the program code operates on vectors of elements and operations are applied to all elements of a vector. For example, binary operations such as addition are applied pairwise to the elements of the vectors. Since multiple vector elements are processed in a single operation, this yields a higher throughput in comparison to scalar operations. Databases often utilize this to speed up computations. In some cases, specialized data structures or layouts can help to utilize SIMD operations, e.g., columnar data storage or hash table buckets [10].
In hardware, a vector is stored as contiguous bits in a register, so element boundaries inside the vector depend on the operation. This allows for scenarios where input data is first shuffled as 8-bit values and then treated as 32-bit integers in the next operation. Common operations in database contexts are load and store, arithmetic, comparison, shuffling, gather/scatter, and widening/narrowing. For example, a column scan could load a vector of attribute values, compare them according to a filter predicate, and extract the indices of matching elements.

**Platform-Intrinsics.** There are multiple extensions to the x86 instruction set that add vector operations: SSE (Streaming SIMD Extensions) with four major versions introducing 128-bit registers, followed by AVX (Advanced Vector Extensions) with 256-bit registers, AVX2, and most recently AVX-512 with 512-bit registers. AVX-512 has recently introduced 128-bit registers, followed by AVX (Advanced Vector Extensions) with 256-bit registers, AVX2, and most recently AVX-512 with 512-bit registers. AVX-512 has 3. Compiler-Intrinsics

A disadvantage of platform-intrinsics is that they encode the type in the function name, as C does not support function overloading. This makes it hard to generically implement SIMD operations, as developers must use different functions for, e.g., `int` and `long`. Additionally, using platform-intrinsics does not guarantee that the corresponding instruction is actually chosen by the compiler. For example, Clang may still produce the same scalar code that can be vectorized and replace the code with a vectorized version [13]. Auto-vectorization may fail, typically either because the pattern is not supported or because some prerequisite for vectorization cannot be satisfied. There are commonly supported and documented approaches to circumvent these problems and help with auto-vectorization [14].

**Auto-Vectorization.** Apart from designated vector operations in the source code, major compilers such as GCC, Clang, and ICC also perform auto-vectorization as an optimization step. They attempt to detect patterns in scalar code that can be vectorized and replace the code with a vectorized version [13]. Auto-vectorization may fail, typically either because the pattern is not supported or because some prerequisite for vectorization cannot be satisfied. There are commonly supported and documented approaches to circumvent these problems and help with auto-vectorization [14].

**3. Compiler-Intrinsics**

A disadvantage of platform-intrinsics is that they encode the type in the function name, as C does not support function overloading. This makes it hard to generically implement SIMD operations, as developers must use different functions for, e.g., `int` and `long`. Additionally, using platform-intrinsics does not guarantee that the corresponding instruction is actually chosen by the compiler. For example, Clang may still produce the same scalar code that can be vectorized and replace the code with a vectorized version [13]. Auto-vectorization may fail, typically either because the pattern is not supported or because some prerequisite for vectorization cannot be satisfied. There are commonly supported and documented approaches to circumvent these problems and help with auto-vectorization [14].

---

1. We use godbolt.org to support our claims with generated assembly.
2. https://godbolt.org/z/M8zaqj3t
GCC and Clang provide vector extensions that allow declaring a vector type of \( n \) elements of a language base type, e.g., \( \text{int} [8, 9] \). A code example that extracts the lower two 32-bit integers from a 128-bit register and zero-extends them to two 64-bit values is shown in Listing 1. For both x86 and NEON, we have to define one function per type, as the intrinsics differ for signed and unsigned integer. Overall, we end up with four distinct implementations. With compiler intrinsics, we use templating and the compiler’s ability to choose appropriate instructions for each platform. This allows for a single implementation that can be used on both platforms.

Vectors are defined using GCC-style __attribute__, specifying the size in bytes and the element’s type. In this example, these are 16 byte for Vec and a template type \( T \), defining a vector of \( n = 16/\text{sizeof}(T) \) elements. The input vector is cast to a HalfVec to extract the lower \( n/2 \) values. We use the compiler’s built-in convert method to widen the values to the requested output type, e.g., \((u)\text{int64}_t\) in our example (specified in \texttt{out} template argument). In Listing 1d) and e), we see that the compiler-intrinsics produce the same assembly as the platform-intrinsics, without type- or platform-dependent code.

In Listing 2, we show a more complete code example from our filtering scan benchmark (Section 4.5). This code scans an integer column, filters by less-than some value, and writes matching row ids to an output array.

In Line 2, we first define a vector of four 32-bit integers. As vectors assume alignment equal to their size, we specify an unaligned version in Line 4, which we use for unaligned stores. Clang supports special Boolean vectors, where each entry is only 1 bit (Line 6). Vectors are loaded and stored using casts (Line 15). Operations are expressed using common C++ operators, such as arithmetic or comparison operators (+, -, <, ==, …), on these types. Operators also support scalar operands, e.g., < \texttt{filter_val} in Line 16. The ternary operator can be used for element-wise selection. For some complex operations that have no matching C++ operator, Clang and GCC provide helper functions, e.g., \texttt{__builtin_convertvector}() for narrowing or widening (Line 19). We cast the Boolean vector to a scalar bitmask and mask off the high bits (Line 22). We then use that bitmask to get row offsets from a lookup table and add them to the base row id (Lines 25–28). Finally, we use a cast to perform an unaligned store of the matching row ids (Lines 31–33).

The compiler lowers these operations to the target platform. The goal of these compiler-intrinsics is to allow for platform-independent source code that performs well on all target platforms. Note that with minor templating, we could extend this method to support arbitrary vector sizes while maintaining only a single implementation. Depending on the given size, the compiler would generate the appropriate instructions for us. If developers want to use explicit platform-intrinsics, Clang allows using them with compiler-vector types. Operations that are not natively supported on the target platform can still be expressed in a canonical fashion. For example, up to AVX2, there is no intrinsic for the multiplication of vectors with 64-bit integer elements. Here, programmers need to fall back to multiplying and combining 32-bit halves of the input numbers. With compiler-intrinsics, programmers can use the C++ multiplication operator and the compiler inserts the required logic.

### 4. Benchmarks

In this section, we perform four microbenchmarks to compare platform-specific SIMD code with scalar, auto-vectorized, and platform-independent variants. We evaluate multiply-shift hashing (Section 4.2), a hash bucket key lookup (Section 4.3), bit-packed decompression (Section 4.4), and an integer column filter scan (Section 4.5). We then present the impact of platform- and compiler-intrinsics in end-to-end TPC-H benchmarks in Velox, a
state-of-the-art columnar query engine (Section 4.6). Our results show that in most benchmarks, vector code is needed to achieve peak performance and that when using vector code, compiler-intrinsics perform at least as well as platform-intrinsics. All of our code and all results can be found in our GitHub repository\(^1\).

4.1. Setup and Methodology

The results presented in this section are based on an x86 Intel Icelake CPU (Xeon Platinum 8352Y) and on an M1 MacBook Pro 14\(^{\text{\prime}}\) laptop with ARM NEON. We validate our results on various other x86 and ARM platforms and discuss this in Section 5. All experiments are performed using Clang/LLVM\(^2\) 15 with -march=arm11e-vfp-v2-neon and -O3 optimizations. All experiments are run single-threaded and repeated ten times. We report the average runtime, the variance for all benchmarks is below 5%. We show the relative speedup of each variant over a naive scalar implementation without explicit vectorization efforts. For all microbenchmarks, we show the naive baseline (naive), a scalar code version that is optimized for auto-vectorization (autovec), our compiler-intrinsics versions (vec-\(*\)), and a platform-intrinsics version (sse4-\(*\), avx2-\(*\), avx512-\(*\), neon-\(*\)). The platform-specific variants use manually selected intrinsics and also represent the usage of a SIMD library, which provides a thin abstraction on top of platform-intrinsics.

In some experiments, the platform-specific variants perform worse than the ones using compiler-intrinsics. We note that it is nearly always possible to adapt the platform-intrinsics to match the compiler’s code to achieve the same performance. However, we keep these worse results to demonstrate the complexity of manually selecting instructions from thousands of available ones. In these cases, we explicitly discuss the responsible instructions.

4.2. Multiply-Shift Hashing

Our first microbenchmark computes the hash values of the input numbers using multiply-shift hashing [15]. This is done, e.g., to determine the bucket in a hash index [16]. Each 64-bit input number is multiplied by a 64-bit constant, the result is truncated to 64-bits, and then shifted right by a run-time value that is known before the hot loop starts. Since each input number has the same arithmetic operations applied and produces exactly one output number without any data dependencies between loop iterations, programmers likely expect auto-vectorization to work well in a naive implementation. LLVM auto-vectorizes our naive implementation on x86, so we include an additional scalar bar that shows the naive implementation with disabled auto-vectorization. The results are shown in Figure 1.

On x86, the naive, autovec, and vec-512 variants produce optimal vectorized code. For smaller explicit vector widths, the performance degrades as we require more operations to process the same input. With 128-bit vectors, we observe a speedup of only \(-10\%\) over scalar.

The avx512 variant does not achieve peak performance because we use the shifting intrinsic \_mm512\_srli\_epi64 (compiled to vpsrlq), which requires an immediate operand that is known at compile-time. Compile-time constants can generally be used for more aggressive optimization. However, for the vec-512 variant, the compiler instead chooses the \_mm512\_sr1v\_epi64 (vpsrlq) instruction that shifts all elements in input vector \(v\_x\) by the amount specified by the corresponding element in input vector \(v\_y\). The second operand needed in the hot loop is moved outside the loop, and in this case, the instruction that the compiler chooses is \(-20\%\) faster than the instruction with the immediate operand.

With the avx2 and sse4 variants, we observe worse performance than with the corresponding compiler-variants using the same vector width. Our implementations only use instructions available in the instruction set that the code is targeted for. Vector multiplication with 64-bit integer elements was first introduced in AVX-512, so our code performs manual multiplication of 32-bit halves of the input using the approach that is also used in the vectorclass SIMD library [17]. LLVM is unable to optimize this to the proper multiplication instruction on the target architecture. This shows a core disadvantage of platform-specific code. Implementations tailored to a specific microarchitecture cannot benefit from new instructions and optimizations, while generic code can.

When repeating the measurements on a Skylake CPU without AVX-512 support, we observe that the autovec, vec-256 and vec-512 variants are still 25\% faster than the handwritten AVX2 implementation using the multiplication from the vectorclass library. The generated code only differs in how the multiplication is performed. In this case, using a specialized SIMD library even results in worse performance than the naive implementation.

On M1, the LLVM cost model does not consider loop vectorization desirable for the naive implementation. For the autovec variant, we explicitly instruct the compiler to vectorize the loop via an annotation. LLVM generates very similar code for the vec-\(*\) variants, with nearly iden-

\(^1\)https://github.com/hpides/autovec-db
\(^2\)We refer to Clang for the C++ frontend and to LLVM for backend optimizations.
with the bytemask-logic \[22\]. The approach of narrowing variants perform a vector-comparison on the fingerprints 16 bytes as a is better on x86 due to simpler loop logic and a saved ciently, so the autovec expected, we are unable to use the comparison result effi-

Figure 2: Fingerprint + key hash bucket lookup (1024 values).

4.3. Hash Bucket Match

In this benchmark, we measure the performance of finding an entry in a hash bucket that stores additional fingerprints in a metadata block at the beginning of the hash bucket. A fingerprint is a 1-byte hash value of a key. The block of fingerprints can then be used to quickly skip over keys that do not match the search key. This design is common in index structures and used in, e.g., Facebook’s F14 hash table, Google’s Abseil hash map, Velox, Dash, and ART \[18, 19, 7, 20, 21\]. With vector operations, all fingerprints can be compared to the fingerprint of the search key in a single operation, allowing a direct jump to the first key with a fingerprint match. Our hash bucket holds 15 entries, each with a fingerprint, an 8-byte key, and an 8-byte value. Fingerprints are stored in a contiguous 16-byte array that is processed as a 128-bit vector. The benchmark repeatedly performs key lookups in the hash bucket with a 50% chance of a successful lookup.

Our benchmark results are shown in Figure 2. The naïve-key-only variant ignores the fingerprints and simply loops over the keys to find a match. The \*bytemask variants perform a vector-comparison on the fingerprints array, resulting in a vector of 16 mask-bytes with either all one-bits or all zero-bits. They then loop over the all-one-bytes in this vector and compare the correspond-

iii) loads a 128-bit vector, \(j\) shuffles to move the required bytes into the lanes, \(ii\) shifts right to move the relevant bits for each number to the beginning of the lane, and \(iii\) masks off leftover bits using bitwise-and. The extended 256 and 512-bit variants use the same approach with wider registers. The \(avx2-256\) variant uses two 128-bit registers since 256-bit registers are separated into two 128-bit lanes with no support for bytewise shuffles across lanes. The \(pdep\) variant implements the algorithm used

\[\text{https://godbolt.org/z/hTjnWMr3n}\]

\[\text{https://godbolt.org/z/PeWGt6rsj}\]

\[\text{Merged, available upstream since cd68e17.}\]
in Velox with the x86 pdep instruction \[27\] to extract multiple compressed n-bit values out of 64-bit data.

LLVM auto-vectorizes better for x86 than for NEON\[^4\], although both achieve the same \(\times 5\) speedup. It detects the pattern on x86 and auto-vectorizes it with gather, shift, and bitwise and instructions. For NEON, LLVM emits unrolled scalar code.

On x86, comparing the vec variants with the platform-intrinsic variants shows multiple things. The 128-bit variants perform equally, as they produce identical assembly. Vec-256 is faster than vec-128 due to higher parallelism. It is also faster than our avx2 variant, as the compiler selects more efficient instructions than we do. The avx2-256 variant processes two 128-bit registers to solve the problem that no cross-lane byte-level shuffle instruction is available for 256-bit registers in AVX2. For this reason, it performs the shifting and bit-masking twice as often compared to an algorithm using a 256-bit register. The vec-256 variant, on the other hand, is compiled to use 256-bit registers by first performing a cross-lane immediate permute operation and then using an in-lane shuffle. Here, LLVM utilizes the fact that for each of our shuffle operations, all selected indices are from a contiguous 16-byte range, so it is able to store all required source-bytes in both 128-bit lanes. Overall, both variants require the same number of instructions for shuffling, but the vec-256 variant uses half as many for shifting and masking.

The avx512 variant outperforms its vec-512 counterpart, as Clang inserts an and-instruction to mask the shuffle input in the frontend, which the backend cannot remove, resulting in more instructions in the tight loop. The vec-256 and avx512 variants perform equally, as the CPU retires twice as many 256-bit instructions per cycle as 512-bit, compensating for the lower parallelism. The vec implementation of SIMD-Scan outperforms sse4-pdep found in Velox, while not using any platform-specific code. Overall, we again see that the vec-* variants perform on par with the platform-intrinsic variants.

With NEON, we see that the 128- and 256-bit compiler-intrinsics outperform the NEON-intrinsics, as the compiler combines the two shift instructions with compile-time constants that are executed independently in the neon variant\[^5\]. During instruction selection, LLVM recognizes the vector shift operations in our vec variant and combines them with a platform-independent optimization. However, when using the NEON-intrinsics, these map to different LLVM op-codes, which are not detected in this optimization step, resulting in worse code. This shows that the compiler can detect the optimization for two shifts in general, but it currently cannot detect it when the developer explicitly requests NEON-intrinsics.

The 256-bit variant is compiled to two 128-bit operations with equal performance, while the 512-bit variant produces inefficient scalar code and performs significantly worse on NEON.

4.5. Filtering Integer Column Scan

In our final microbenchmark, we show the performance of an integer scan with a filter on 32k values, inspired by implementations in Velox \[28\] and Hyrise \[29\]. All values in the column are represented as 32-bit integers and we filter by \(val < x\) with a selectivity of 50%. The matching 32-bit RowIDs are written to an output vector. The results are shown in Figure 4.

The naive variant implements a for-loop and writes to the output vector if the filter matches. The autovec variant does the same but uses a branchless implementation, i.e., \(out\_idx += val < x\). This always writes to the output but overwrites the current value in the next iteration if it did not match. The *-shuffle variants create a bitmask (4-bit in 128, 16-bit in 512) from the comparison result vector and perform a table lookup (2\(^7\) or 2\(^{16}\) entries) with that mask to retrieve the indices with which the current iteration’s RowIDs are shuffled. If positions 1 and 3 match, \((ids[1], ids[3], \circ, \circ)\) is written to the output, where the \(\circ\)’s are overwritten in the next iteration. The *-add variants do not shuffle but use the bitmask to look up which RowIDs must be added to the current iteration’s start RowID \(r\). If positions 1 and 3 match, \((r + 1, r + 3, \circ, \circ)\) is written. This saves one shuffle instruction but only works for contiguous input RowIDs. The vpcompressd variants use the AVX-512 vpcompressd \[30\] instruction that shifts all matching values to the beginning of the SIMD register and stores them to memory. We also evaluate a version with the SSE4 pext \[31\] instruction, but this is slower than the more general approaches.

\[^4\]https://godbolt.org/z/e7WMPMT4
\[^5\]https://godbolt.org/z/5bvs3qWT6

---

Figure 3: Bit-packed integer decompression from 9- to 32-bit integers (100 000 values).

Figure 4: Filtering integer column scan (32 000 values).
compiler-intrinsics do not achieve the performance of platform-intrinsics on x86. The special vpcmpxsdd instruction performs the required logic and achieves significantly better results by combining the lookup and shuffle in a single instruction. However, there is currently no way to make LLVM produce this instruction using auto-vectorization or compiler-intrinsics [32, 33].

autovec significantly outperforms naive because of branchless execution. The compiler unrolls the naive loop five times but converts the write inside an if-branch into five branches in assembly, resulting in high runtime cost for branch mispredictions.

For the -shuffle/add variants, the 512-bit variants outperform the 128-bit ones by only 2×. The higher parallelism is counterbalanced by the need for a bigger lookup table of up to 4 MB that causes cache misses. With the compress variants, we observe a 3.4× speedup from 128- to 512-bits, as there is no lookup table and the 128- and 512-bit instructions have the same latency and throughput [30], benefiting the higher parallelism of the 512-bit variant. For the 128-bit add-based variants, LLVM generates slightly better assembly with three fewer instructions when using compiler-intrinsics instead of SSE.

For NEON, we see that the vec-128 variants achieve the same performance as the NEON implementation with our patched LLVM [25]. Without the patch, both suffer from the conversion-to-bitmask problem as in the hash bucket lookup (Section 4.3). For the shuffle variant, LLVM also generates worse shuffle code that uses element-wise insertion. We submitted an LLVM patch to fix this using NEON’s TBL instruction12 (dashed part of bar) [34]. As NEON does not have 512-bit registers, the improved conversion to bitmask and shuffle cannot be used in vec-512, resulting in worse assembly.

### 4.6. SIMD in Velox

In this benchmark, we evaluate the performance of compiler-intrinsics in an end-to-end system setup, running TPC-H queries in Velox [7]. To show the difference between platform- and compiler-intrinsics, we replace Velox’ xsimd [4] dependency with Clang’s vector intrinsics. We run 18 out of 19 supported TPC-H queries in Velox13 with scale factor 1. All queries are executed ten times with Velox’s default TPC-H configuration of four threads and we report the average runtime. We compile with Clang/LLVM 15 on x86 and with our patched version on M1, using -O3 and -march=mtune=native flags. The results are shown in Figure 5.

For both x86 and M1, our vec implementation performs equally to Velox’s original xsimd code. The average runtime differs by 1.3% on Icelake and around 0.13% on M1. However, the difference on Icelake is not caused by SIMD code changes but by code alignment issues [36]. Explicitly adding eight nop padding bytes before a single hot loop in a scan operator reduces the performance difference to 0.1%. Code-alignment optimizations are highly platform-specific and out of scope of this work. We use them to illustrate that performance differences at this level have various causes that must be considered when optimizing. Overall, there is no performance impact when switching from platform- to compiler-intrinsics.

We also perform a run without any explicit SIMD processing by replacing all compiler-intrinsics with fixed-sized array operations. This decreases the overall performance by 4%, i.e., explicit SIMD code in Velox achieves 4% performance gains for TPC-H on average. However, these gains are not distributed uniformly. While some queries are not affected at all, one drops by 24.9%. In Table 1, we show Q15, Q16, and Q18, which exhibit the largest differences to the xsimd baseline when disabling all SIMD code (novec).

![Figure 5: End-to-end TPC-H benchmark (19 supported queries) with SF1 from Parquet in Velox on x86 and M1.](image)

#### Table 1: Performance difference from xsimd TPC-H baseline to no SIMD code (novec) and compiler-intrinsics (vec).

<table>
<thead>
<tr>
<th>Query</th>
<th>x86</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q15</td>
<td>novec</td>
<td>vec</td>
</tr>
<tr>
<td>x86</td>
<td>5.0%</td>
<td>3.4%</td>
</tr>
<tr>
<td>M1</td>
<td>+0.1%</td>
<td>0.2%</td>
</tr>
</tbody>
</table>

12https://godbolt.org/z/h96P7s173
13Merged, available upstream since 267d6d6.
14We omit Q21 as Velox produces wrong results with scale factor 10 [35]. Overall, SF10 shows the same performance trends as SF1.
This highlights the complexity of correctly handling mul-

Table 2: Bit-packed integer decompression results on multiple platforms to avoid the impact of measurement bias. While developing the benchmarks, results across multiple platforms to avoid the impact of platform-specific macro was not included correctly [37]. In this section, we discuss the generalizability of our approach to a wider range of systems. We see that most vec-∗ variants achieve the same or better performance than the x86-specific ones. Systems that do not have the necessary AVX512 instructions perform poorly with vec-512, as the logic is not easily translatable to AVX2 instructions. On all servers, explicit vectorization outperforms naive and autovec. On AMD Rome, the sse4-pdep variant, as used in Velox, performs significantly worse than any other variant. This is caused by the pdep instruction being significantly slower on AMD than on Intel CPUs [38]. On ARM, we see a wider absolute performance range. The Raspberry Pi naive baseline is 4× slower than M1, while the Graviton CPUs are 1.5–2× slower. Regardless of absolute performance, all machines benefit from vectorization. As NEON does not have 256-bit registers, the vec-256 code is split into multiple 128-bit instructions. Graviton 2 does not scale well here, as it has only two SIMD units per core, while M1 and Graviton 3 have four. Overall, the results show that our insights and conclusions hold on a wider range of systems. We see that in nearly all cases, our compiler-intrinsics approach performs on par with or outperforms the platform-intrinsics.

5.2. Across Compilers

In our evaluation, we focus on Clang/LLVM. While many projects focus on one compiler, it is common to use multiple compilers for testing. If features and built-in functions diverge, developers have to write multiple compiler-
Table 3: Results for compilation with GCC on multiple x86 and NEON platforms. Speedup relative to naive.

<table>
<thead>
<tr>
<th></th>
<th>Intel Icelake</th>
<th>AMD Rome</th>
<th>M1 Graviton 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>1 (112 µs)</td>
<td>1 (123 µs)</td>
<td>1 (61 µs)</td>
</tr>
<tr>
<td>autovec</td>
<td>3.4x</td>
<td>1.4x</td>
<td>1.2x</td>
</tr>
<tr>
<td>vec-128</td>
<td>7.3x</td>
<td>5.2x</td>
<td>7.9x</td>
</tr>
<tr>
<td>vec-256</td>
<td>10.9x</td>
<td>10.9x</td>
<td>0.5x</td>
</tr>
<tr>
<td>vec-512</td>
<td>8.7x</td>
<td>0.4x</td>
<td>0.5x</td>
</tr>
<tr>
<td>sse4-128</td>
<td>6.0x</td>
<td>7.7x</td>
<td>-</td>
</tr>
<tr>
<td>avx2-256</td>
<td>7.3x</td>
<td>5.2x</td>
<td>-</td>
</tr>
<tr>
<td>sse4-pdep</td>
<td>5.6x</td>
<td>0.1x</td>
<td>-</td>
</tr>
<tr>
<td>avx512</td>
<td>8.7x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>neon</td>
<td>-</td>
<td>-</td>
<td>7.8x</td>
</tr>
</tbody>
</table>

(a) Bit-packed integer decompression.

<table>
<thead>
<tr>
<th></th>
<th>Intel Icelake</th>
<th>AMD Rome</th>
<th>M1 Graviton 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>1 (156 µs)</td>
<td>1 (143 µs)</td>
<td>1 (97 µs)</td>
</tr>
<tr>
<td>autovec</td>
<td>8.2x</td>
<td>7.4x</td>
<td>8.6x</td>
</tr>
<tr>
<td>vec-128-shuffle</td>
<td>7.9x</td>
<td>8.1x</td>
<td>9.1x</td>
</tr>
<tr>
<td>vec-128-add</td>
<td>8.1x</td>
<td>8.7x</td>
<td>10.2x</td>
</tr>
<tr>
<td>vec-512-shuffle</td>
<td>7.8x</td>
<td>1.9x</td>
<td>3.4x</td>
</tr>
<tr>
<td>vec-512-add</td>
<td>7.9x</td>
<td>4.7x</td>
<td>5.6x</td>
</tr>
<tr>
<td>sse4-128-shuffle</td>
<td>14.0x</td>
<td>18.0x</td>
<td>-</td>
</tr>
<tr>
<td>sse4-128-add</td>
<td>14.1x</td>
<td>19.8x</td>
<td>-</td>
</tr>
<tr>
<td>vpcompressd-512</td>
<td>38.2x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>neon-shuffle</td>
<td>-</td>
<td>-</td>
<td>12.3x</td>
</tr>
<tr>
<td>neon-add</td>
<td>-</td>
<td>-</td>
<td>12.8x</td>
</tr>
</tbody>
</table>

(b) Filtering column scan.

Also, GCC produces slower code for the vec-256 variants on the ARM platforms.

On Ice lake, the 512-bit variants vec-512 and avx512 perform 20% worse than vec-256. Code inspection shows that these variants produce identical assembly except for vector width and offset values. Thus, we attribute the lower performance of the 512-bit variants to microarchitectural details. With LLVM, these variants are compiled to a slightly different instruction order, where vec-256 and vec-512 achieve similar performance.

On Rome, vec-128 performs a bit worse than sse4-128. This is caused by differing instruction selection. In vec-128, we use a shift-operation on vectors to align the numbers inside the lanes, which GCC compiles to a vector-shift instruction. In handwritten code, this corresponds to the _mm_sllv_epi32 instruction, which is only available in AVX2. So we instead use a multiplication instruction (_mm_mullo_epi32) in sse4-128, which is slightly faster. An improvement of GCC’s instruction selection would remove this difference.

For the filtering column scan, computing the matching element bitmask is a fundamental operation. As described above, GCC does not support vector-bitmask types, and thus does not allow expressing bitmask-conversion instructions. We circumvent this by accessing all bitmask-related logic through a helper function that uses the NEON-approach of bitwise masking and horizontal combining if compiled with GCC. In isolation, this approach does not result in good performance on x86 platforms that have native _movemask operations\(^\text{1}\). In a performance critical path, programmers would have to implement bitmask conversion for each platform.

The code generated by our generic implementation causes a significant performance drop compared to the platform-specific variants. On x86, the sse4-128 variants outperform the vec-128 variants by 1.7× on Ice lake and 2.2× on Rome. Our function to compute bitmasks scales in complexity with the number of vector elements. For a 128-bit vector of 32-bit integers, GCC compiles it to 8 instructions, while requiring 40 for a 512-bit vector. Due to this additional overhead, the vec-512 variants do not outperform the vec-128 ones.

On ARM, the compiler-intrinsics and platform-intrinsics differ by a smaller, but still significant amount. The handwritten NEON-variants are 30% and 40% faster on M1 and Graviton 3, respectively. GCC correctly converts the shuffle to a TBL instruction, but struggles with the bitmask, as for x86.

6. Discussion

Our results show that in most of the evaluated database operations, platform-specific SIMD code does not achieve\(^\text{1}\)https://godbolt.org/z/azh9E3zW
better performance than platform-independent SIMD code. In 7/8 microbenchmarks, compiler-intrinsics perform on par with or better than platform-intrinsics. Auto-vectorization alone does not achieve this, providing good performance in only 2/8 benchmarks. We conclude that with state-of-the-art compilers, explicit SIMD code is necessary to achieve high performance, and we show that this is possible without implementing multiple versions of the same code for different platforms and data types.

6.1. Using Compiler-Intrinsics

There are cases in which platform-intrinsics are necessary. Therefore, we recommend approaching writing SIMD code as follows. First, developers should try to rely on auto-vectorization, because this code is the most portable and often easiest to understand. If this does not yield good results or is not possible, compiler-intrinsics should be used to a) structure the overall SIMD code and b) write portable SIMD algorithms. Only in cases where the desired logic cannot be expressed portably, developers should use platform-intrinsics as small, localized performance fixes.

We show this approach in a small example in Listing 3, in which we store matching values based on a filter predicate, similar to the benchmark in Section 4.5. In our benchmarks, we observe that AVX512’s compressstore is very efficient in storing only matching values. To leverage this via compiler-intrinsics, developers can write a single platform- and type-independent templated method for the general filtering logic (Lines 1–13). In our example, the same function can be used for integers, floats, and longs in 16-, 32-, or 64-byte vectors on x86, ARM, and other platforms. We distinguish between platforms and types only for the special compressstore instruction in Line 10. Depending on the platform, we choose the AVX512 implementation (Lines 16–26) or a fallback implementation (Lines 28–32). Even in this small example, we see that supporting all variants of compressstore leads to many code paths, i.e., three vector sizes and multiple element sizes. However, this complexity is localized and does not leak into the filtering operation.

6.2. Advantages of Compiler-Intrinsics

The approach described in Section 6.1 is also achievable via SIMD libraries due to their vector abstractions and implemented as such, e.g., in Velox. However, relying on compiler-intrinsics over libraries has three key advantages. First, development and testing becomes easier. When relying on a SIMD library that wraps platform-intrinsics, functions can only be compiled on machines that support these intrinsics, e.g., testing AVX512 code on an AVX2 server is not possible in general. When expressing the logic via compiler-intrinsics, it is possible to test the generic 512-bit “AVX512” vector logic on an AVX2 or even an ARM server. While the generated code may be inefficient, the compiler guarantees to generate valid code for the given input regardless of the available instructions. Second, as our benchmarks show, staying in the compiler’s own type domain allows for better optimization in some cases. Third, with minimal code overhead, developers can structure SIMD code according to their needs without adapting to the interface given by the library or relying on a third-party dependency at all.

As SIMD code is written for high performance, developers still need to benchmark and evaluate their code continuously. Even with compiler-intrinsics, the compiler should be instructed to generate optimized code via -march/mtune flags, and the code should be profiled and updated if new compiler versions offer new functionality. This is also the case when using platform-intrinsics but without the benefit of automatically adapted instructions on new platforms.

```c
1 template <typename T, typename FilterOp>
2 void filter(T* in, T filter_val, T* out, FilterOp op) {
3     int num_elements = sizeof(Vec<T>) / sizeof(T);
4     for (int i = 0; i < N; i += num_elements) {
5         Vec<T> vals = (Vec<T>&) in[i];
6         // Filter is platform-independent and templatable.
7         Vec<T> matches = op(val, filter_val);
8         // n variants only needed for special instructions.
9         uint64_t mask = compress_store(vals, matches, out);
10         out += std::popcount(mask);
11         }
12     }
13 }
14 #ifdef __AVX512F__ // We can use compressstore.
15 template <typename T> uint64_t
16     compress_store(Vec<T> vals, Vec<T> matches, T* out) {
17         uint64_t mask;
18         // When T is 32-bit
19         if constexpr (sizeof(Vec<T>) == 16) {
20             mask = _mm256_maskz_epi32(matches);
21         }
22         __mmask compress_storeu_epi32(out, mask, vals);
23         }
24         else if (sizeof(Vec<T>) == 64) { // _mm512...
25             return mask;
26         } #endif // We can use compressstore.
27 #else // We can’t use compressstore.
28     template <typename T> uint64_t
29         compress_store(Vec<T> vals, Vec<T> matches, T* out) {
30             // Fallback shuffle + store similar to Listing 2.
31             ... #endif
32 }
```

Listing 3: Use compiler-intrinsics to structure code, represent vectors, and for common operations. Only switch to localized platform-intrinsics for specific instructions without multiple variants for the same filtering logic.
6.3. Open Challenges

Compiler-intrinsics perform well in our experiments and Velox, but a few things still require improvement. GCC’s missing support for vector-to-bitmask conversion forces us to use workarounds with suboptimal performance. LLVM does not generate good vectorized code for some common (x86-) patterns. For example, an x86 gather is detected in some cases, but not in others, and an analogous scatter is not detected at all14. Clang supports the ternary operator for element-wise selection but LLVM does not yet combine it with mask-able instructions available in, e.g., AVX51215. Overall, compiler-intrinsics support x86 better than NEON due to wider adoption. The documentation of compiler-intrinsics is not as broad as that of platform-intrinsics, as they are mostly used internally. Better instruction selection and documentation as well as more features will increase the applicability of compiler-intrinsics, further reducing the need for platform-intrinsics.

SVE and RISC-V V implement vector length agnostic programming approaches where vector sizes may be determined at runtime. C++ code using the proposed compiler-intrinsics can conceptually be written to support arbitrary vector sizes. However, when this is done using templating, the compiler still typically computes constants such as loop increments at compile time. The programming model of variable length vectors also emphasizes usage of predicate/mask registers to decouple the available input element count from actual vector sizes. This allows removing any epilogue logic handling left-over elements. To our knowledge, there is currently no way to express runtime dynamic vector sizes or predicated vector operations with compiler intrinsics in GCC and Clang. With wider adoption of variable length vector instructions, we expect compilers to add more support for these concepts.

Platform-independent vector extensions provided by major compilers are not widely known in the database community, and thus, they are not widely used. We believe that higher adoption will lead to more features and better instruction selection, solving some of the open challenges. Our results show that platform-dependent SIMD code is often not needed, so we encourage database developers to use compiler-intrinsics in their code.

7. Related Work

In this section, we briefly discuss related work on SIMD programming in databases and on SIMD libraries.

SIMD in Databases. There is a wide range of research on vectorization in databases [5, 1, 39, 3, 40, 41, 2, 42].

This research shows that vectorization is beneficial and serves as a motivation for our work. Based on some of it, we conduct microbenchmarks and show that they can often be implemented with platform-independent SIMD code. In general, research on database vectorization is orthogonal to our work and we think that it should consider platform-independent optimizations in the future.

Some work makes use of many platform-specific instructions [43, 44], making it hard to fit into our platform-independent approach. There are cases where this is beneficial, but we recommend using these platform-specific algorithms only if they have shown to be better.

SIMD Libraries. Various SIMD libraries exist that help developers to write platform-independent code by hiding the platform-specific intrinsics behind an abstraction layer [4, 45, 46, 5, 6, 17]. To varying degrees, they cover the standard operations supported on all platforms and additional interfaces for platform-specific instructions. Some of these libraries also provide higher-level functions that combine instructions to offer more features, which is orthogonal to our work. However, all of these libraries add a layer of abstraction on top of the compiler’s platform-intrinsics. In contrast, we propose to remove a layer of abstraction and work directly with compiler-intrinsics instead. Our approach reduces code complexity and occasionally benefits from better compiler optimizations.

Another approach is to add a SIMD abstraction to the programming language’s standard library [47, 48, 49]. This supports our claim that SIMD programming should be platform-independent.

8. Conclusion

In this paper, we make the case for writing less platform-specific SIMD code in databases. Instead of requiring multiple platform-specific implementations for the same logic, compiler vector intrinsics allow developers to write a single version that is optimized by the compiler for every target platform. Our evaluation shows that this approach achieves the same or better performance in most of our microbenchmarks and a real system. While challenges remain, adopting compiler-intrinsics in databases leads to less logical code duplication and fewer platform-specific SIMD variants.

Acknowledgments

This work was partially funded by the German Ministry for Education and Research (ref. 01IS18025A and ref. 01IS18037A), the German Research Foundation (ref. 414984028), and the European Union’s Horizon 2020 research and innovation programme (ref. 957407).
References


