Evaluating SIMD Compiler-Intrinsics for Database Systems

Lawrence Benson, Richard Ebeling, Tilmann Rabl

lawrence.benson@hpi.de

ADMS | 28.08.23
SIMD in a Nutshell

» Single Instruction Multiple Data (= SIMD)

4 additions in 1 instruction

\[
\begin{array}{cccc}
\text{in:} & 10 & 20 & 30 & 40 \\
+ & 1 & 2 & 3 & 4 \\
\text{out:} & 11 & 22 & 33 & 44 \\
\end{array}
\]
SIMD in a Nutshell

» Single Instruction Multiple Data (= SIMD)

» Most common instruction sets
  › x86: SSE, AVX, AVX2, AVX512 (> 6k instructions)
  › ARM: Neon (> 4k instructions), SVE
  › PowerPC AltiVec, RISC-V

4 additions in 1 instruction
SIMD in a Nutshell

» Single Instruction Multiple Data (= SIMD)

» Most common instruction sets
  › x86: SSE, AVX, AVX2, AVX512 (> 6k instructions)
  › ARM: Neon (> 4k instructions), SVE
  › PowerPC AltiVec, RISC-V V

» Arithmetic, Logical, Shuffle, Shift, Load, Store, …
SIMD in a Nutshell

» Single Instruction Multiple Data (= SIMD)

» Most common instruction sets
  › x86: SSE, AVX, AVX2, AVX512 (> 6k instructions)
  › ARM: Neon (> 4k instructions), SVE
  › PowerPC AltiVec, RISC-V

» Arithmetic, Logical, Shuffle, Shift, Load, Store, ...

» Focus on x86 and Neon
  › x86: 128 – 512 Bit registers
  › Neon: 128 Bit registers
SIMD in Databases

» Used to speed up, e.g.,
  › Table scans
  › Hash tables
  › Sorting

SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units

Thomas Wilhelm
Nicola Popovici
Intel GmbH
Dresden, Germany

Yazan Bosnian
SAP AG
Dietmar-Hopp-Allee 16
66150 Walldorf, Germany

Hasso Plattner
Alexander Zeier
Jan Schaffner
Hasso-Plattner-Institute

The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code

Azim Afrozeh
CWI, The Netherlands
azim@cwi.nl

Peter Boncz
CWI, The Netherlands
boncz@cwi.nl

ABSTRACT
The open-source FastLanes project aims to improve big data formats, such as Parquet, ORC, and columnar database formats, in multiple ways. In this paper, we significantly accelerate decoding of all common Light-Weight Compression (LWC) schemes without using SIMD vectorization.
SIMD in Databases

» Used to speed up, e.g.,
  › Table scans
  › Hash tables
  › Sorting

» SIMD code is …
  › ... hard to develop
  › ... hard to test
  › ... hard to benchmark
SIMD in Databases

» Used to speed up, e.g.,
  › Table scans
  › Hash tables
  › Sorting

» SIMD code is …
  › ... hard to develop
  › ... hard to test
  › ... hard to benchmark

» Non-x86 CPUs on the rise
  › How to translate x86 SIMD code?
SIMD in Databases

» Used to speed up, e.g.,
  » Table scans
  » Hash tables
  » Sorting

» SIMD code is ...
  » ... hard to develop
  » ... hard to test
  » ... hard to benchmark

» Non-x86 CPUs on the rise
  » How to translate x86 SIMD code?

What do these functions do?

```c
_mm_add_epi32()
_mm512_srl_epi64()
vaddq_s32()
```
SIMD in Databases

» Used to speed up, e.g.,
  › Table scans
  › Hash tables
  › Sorting

» SIMD code is ...
  › ... hard to develop
  › ... hard to test
  › ... hard to benchmark

» Non-x86 CPUs on the rise
  › How to translate x86 SIMD code?

What do these functions do?

_ mm_add_epi32()
_mm512_srl_epi64()
vaddq_s32()
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>+ B</td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
</tr>
</tbody>
</table>

Compiler SIMD @ ADMS | 28.08.2023
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ B</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>C:</td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
</tr>
</tbody>
</table>

Application code

```
vec C = A + B;
```
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

<table>
<thead>
<tr>
<th>A</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ B</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
</tr>
</tbody>
</table>

Application code

```cpp
vec C = A + B;
```

SIMD library

```cpp
struct vec {
    vec operator+(vec, vec);
};
```
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

<table>
<thead>
<tr>
<th>A</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ B</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>C:</td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
</tr>
</tbody>
</table>

Application code

```
vec C = A + B;
```

SIMD library

```
struct vec {
    vec operator+(vec, vec);
}
```

SIMD intrinsics

```
_mm_add_epi32
vaddq_s32
```
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

<table>
<thead>
<tr>
<th>A</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ B</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>C:</td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
</tr>
</tbody>
</table>

Application code

```c
vec C = A + B;
```

SIMD library

```c
struct vec {
    vec operator+(vec, vec);
};
```

SIMD intrinsics

```c
_mm_add_epi32
vaddq_s32
```

Compiler representation

```c
__attribute__((vector_size(N)));
```
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

```c
// Simplified from Clang's <emmintrin.h>
typedef long long __m128i __attribute__((vector_size(16)));
```
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

```c
// Simplified from Clang's <emmintrin.h>
typedef long long __m128i __attribute__((vector_size(16)));
```
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

SIMD Types

16 Byte type in x86

// Simplified from Clang's <emmintrin.h>

typedef long long __m128i __attribute__((vector_size(16)));

Compiler Representation

GCC/Clang's "vector" type

Compiler SIMD @ ADMS | 28.08.2023
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

```c
// Simplified from Clang's <emmintrin.h>
typedef long long __m128i __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
}
```
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

```c
__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
    return (__m128i)(__v4su)__a + (__v4su)__b;
}

// Simplified from Clang's <arm_neon.h>
int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
    __ret = __p0 + __p1;
    return __ret;
}
```

Compiler Representation
GCC/Clang's "vector" type
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

// Simplified from Clang's <emmintrin.h>
typedef long long __m128i __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
    return (__m128i)((__v4su)__a + (__v4su)__b);
}

// Simplified from Clang's <arm_neon.h>
int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
    int32x4_t __ret = __p0 + __p1;
    return __ret;
}
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

// Simplified from Clang's <emmintrin.h>

typedef long long __m128i __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
    return (__m128i)((__v4su)__a + ((__v4su)__b);
}

// Simplified from Clang's <arm_neon.h>

int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
    int32x4_t __ret;
    __ret = __p0 + __p1;
    return __ret;
}
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

```c
// Simplified from Clang's <emmintrin.h>
typedef long long __m128i __attribute__((vector_size(16)));

// Internal 16-Byte vector of four unsigned integers.
typedef unsigned int __v4su __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
    return (__m128i)(__v4su)__a + (__v4su)__b;
}

// Simplified from Clang's <arm_neon.h>
// int32x4_t is defined analogously to __v4su.
int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
    int32x4_t __ret;
    __ret = __p0 + __p1;
    return __ret;
}
```
Abstractions on top of Abstractions

Add two 128-bit registers of 4x 32-bit integers

```
typedef long long __m128i __attribute__(((vector_size(16))));

// Internal 16-Byte vector of four unsigned integers.
typedef unsigned int __v4su __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
  return ((__m128i)(__v4su)__a + ((__v4su)__b);
}
```

```
// Simplified from Clang's <emmintrin.h>
int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
  int32x4_t __ret;
  __ret = __p0 + __p1;
  return __ret;
}
```

Compiler Representation
GCC/Clang's "vector" type

Platform-intrinsics
Platform- and type-dependent C API
x86 NEON

SIMD Types
16 Byte type in x86
Add two 128-bit registers of 4x 32-bit integers

```c
// Simplified from Clang's <emmintrin.h>
typedef long long __m128i __attribute__((vector_size(16)));

// Internal 16-Byte vector of four unsigned integers.
typedef unsigned int __v4su __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
    return (__m128i)(__v4su)__a + (__v4su)__b;
}

// Simplified from Clang's <arm_neon.h>
// int32x4_t is defined analogously to __v4su.
int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
    int32x4_t __ret;
    __ret = __p0 + __p1;
    return __ret;
}
```

Platform-intrinsics are abstractions on top of compiler-intrinsics
Add two 128-bit registers of 4x 32-bit integers

```c
typedef long long __m128i __attribute__((vector_size(16)));
typedef unsigned int __v4su __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
    return (__m128i)(__v4su)__a + (__v4su)__b;
}

int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
    return __p0 + __p1;
}
```
Abstractions on top of Abstractions

SIMD Libraries

```cpp
template <typename T>
struct vec {
  vec<T> operator+(vec<T> other);
};
```

Add two 128-bit registers of 4x 32-bit integers

```cpp
typedef long long __m128i __attribute__((vector_size(16)));
typedef unsigned int __v4su __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
  return (__m128i)((__v4su)__a + (__v4su)__b);
}

int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
  return __p0 + __p1;
}
```
## Abstractions on top of Abstractions

### SIMD Libraries

```cpp
template <typename T>
struct vec {
  vec<T> operator+(vec<T> other);
}

#if __x86_64__
vec<T> vec<T>::operator+(vec<T> other) {
  return _mm_add_epi32(data, other.data);
}
#elif __aarch64__
vec<T> vec<T>::operator+(vec<T> other) {
  return vaddq_s32(data, other.data);
}
#else ...
```

### Add two 128-bit registers of 4x 32-bit integers

```cpp
typedef long long __m128i __attribute__((vector_size(16)));
typedef unsigned int __v4su __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
  return (__m128i)(__v4su)__a + (__v4su)__b;
}

int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
  return __p0 + __p1;
}
```
Abstractions on top of Abstractions

SIMD Libraries

```
template <typename T>
struct vec {
    vec<T> operator+(vec<T> other);
};

#if __x86_64__
vec<T> vec<T>::operator+(vec<T> other) {
    return _mm_add_epi32(data, other.data);
}
#elif __aarch64__
vec<T> vec<T>::operator+(vec<T> other) {
    return vaddq_s32(data, other.data);
}
#else ...
```

SIMD libraries are abstractions on top of platform-intrinsics

Add two 128-bit registers of 4x 32-bit integers

```
typedef long long __m128i __attribute__((vector_size(16)));
typedef unsigned int __v4su __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
    return (__m128i)((__v4su)__a + ((__v4su)__b);
}

int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
    return __p0 + __p1;
}
```
Abstractions on top of Abstractions

**SIMD Libraries**

```cpp
template <typename T>
struct vec {
    vec<T> operator+(vec<T> other);
}

#if __x86_64__
vec<T> vec<T>::operator+(vec<T> other) {
    return _mm_add_epi32(data, other.data);
}
#elif __aarch64__
vec<T> vec<T>::operator+(vec<T> other) {
    return vaddq_s32(data, other.data);
}
#else ...
```

**Add two 128-bit registers of 4x 32-bit integers**

```cpp
typedef long long __m128i __attribute__((vector_size(16)));
typedef unsigned int __v4su __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {
    return (__m128i)(__v4su)__a + (__v4su)__b);
}

int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
    return __p0 + __p1;
}
```

Platform-intrinsics are abstractions on top of compiler-intrinsics

**SIMD libraries are abstractions on top of platform-intrinsics**
Abstractions on top of Abstractions

SIMD Libraries

```c
template <typename T>
struct vec {
    vec<T> operator+(vec<T> other);
};

#if __x86_64__
vec<T> vec<T>::operator+(vec<T> other) {
    return _mm_add_epi32(data, other.data);
}
#elif __aarch64__
vec<T> vec<T>::operator+(vec<T> other) {
    return vaddq_s32(data, other.data);
}
#else ...
```

**SIMD libraries are abstractions on top of platform-intrinsics**

Add two 128-bit registers of 4x 32-bit integers

```c
typedef long long __m128i __attribute__((vector_size(16)));
typedef unsigned int __v4su __attribute__((vector_size(16)));

__m128i _mm_add_epi32(__m128i _a, __m128i _b) {
    return (__m128i)((__v4su)_a + (__v4su)_b);
}

int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {
    return __p0 + __p1;
}
```

**Platform-intrinsics are abstractions on top of compiler-intrinsics**

```c
template <typename T>
using vec __attribute__((vector_size(16))) = T;

vec<T> foo(vec<T> a, vec<T> b) {
    // Do stuff
    vec<T> result = a + b;
    // ...
    return result;
}
```

**Use compiler-intrinsics to structure code**
Compiler-Intrinsics

» GCC/Clang have SIMD abstraction
  › via __attribute__((vector_size(SIZE)))
  › SIZE in bytes can be ... / 8 / 16 / 32 / 64 /...
Compiler-Intrinsics

» GCC/Clang have SIMD abstraction
  › Via __attribute__((vector_size(SIZE)))
  › SIZE in bytes can be .../8/16/32/64/...

» Supports common operations
  › Arithmetic: +, -, *, /, >>, ...
  › Comparison: >=, <, !=, ...
  › Bitwise: &, |
  › Logical: &&, ||
Compiler-Intrinsics

» GCC/Clang have SIMD abstraction
  › Via __attribute__((vector_size(SIZE)))
  › SIZE in bytes can be .../8/16/32/64/...

» Supports common operations
  › Arithmetic: +, -, *, /, >>, ...
  › Comparison: >=, <, !=, ...
  › Bitwise: &, |
  › Logical: &&, ||

» Special built-in functions
  › convertvector()
  › shufflevector()
Compiler-Intrinsics

» GCC/Clang have SIMD abstraction
  › Via __attribute__((vector_size(SIZE)))
  › SIZE in bytes can be .../8/16/32/64/...

» Supports common operations
  › Arithmetic: +, -, *, /, >>,...
  › Comparison: >=, <, !=, ...
  › Bitwise: &, |
  › Logical: &&, ||

» Special built-in functions
  › convertvector()
  › shufflevector()

» Guaranteed to compile and be correct
  › Easier development + testing
Compiler-Intrinsics

» GCC/Clang have SIMD abstraction
  › Via __attribute__((vector_size(SIZE)))
  › SIZE in bytes can be .../8/16/32/64/...

» Supports common operations
  › Arithmetic: +, -, *, /, >>, ...
  › Comparison: >=, <, !=, ...
  › Bitwise: &, | 
  › Logical: &&, ||

» Special built-in functions
  › convertvector()
  › shufflevector()

» Guaranteed to compile and be correct
  › Easier development + testing

// 16-Byte vector of 4x uint32_t.
using Vec __attribute__((vector_size(16))) = uint32_t;
// Same as Vec but without 16-Byte alignment.
using UnalignedVec __attribute__((aligned(1))) = Vec;
// Vector of 4 bools (only available in LLVM).
using BitVec __attribute__((ext_vector_type(4))) = bool;
Compiler-Intrinsics

GCC/Clang have SIMD abstraction
  » Via __attribute__((vector_size(SIZE))))
  » SIZE in bytes can be .../8/16/32/64/...

Supports common operations
  » Arithmetic: +, -, *, /, >>, ...
  » Comparison: >=, <, !=, ...
  » Bitwise: &, |
  » Logical: &&, ||

Special built-in functions
  » convertvector()
  » shufflevector()

Guaranteed to compile and be correct
  » Easier development + testing

// 16-Byte vector of 4x uint32_t.
using Vec __attribute__((vector_size(16))) = uint32_t;
// Same as Vec but without 16-Byte alignment.
using UnalignedVec __attribute__((aligned(1))) = Vec;
// Vector of 4 bools (only available in LLVM).
using BitVec __attribute__((ext_vector_type(4))) = bool;
// Scan integer column and write matching row ids.
uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val, uint32_t* __restrict output) {
  uint32_t num_matches = 0;
  for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
    // Load data and compare.
    Vec values = *(Vec*) (column + row);
    Vec matches = values < filter_val;
    // Convert comparison to scalar bitmask using built-in.
    BitVec bitvec = __builtin_convertvector(matches, BitVec);
    uint8_t bitmask = (uint8_t&) bitvec;
    // Get ids from lookup table and add to current base row.
    Vec row_offsets = *(Vec*) MATCHES_TO_ROW_OFFSETS[bitmask];
    Vec compressed_rows = row + row_offsets;
    // Write matching row ids to output.
    *(UnalignedVec*) (output + num_matches) = compressed_rows;
    num_matches += std::popcount(bitmask);
  }
  return num_matches;
}
Compiler-Intrinsics

 GCC/Clang have SIMD abstraction
  - Via __attribute__((vector_size(SIZE)))
  - SIZE in bytes can be .../8/16/32/64/...

 Supports common operations
  - Arithmetic: +, -, *, /, >>, ...
  - Comparison: >=, <, !=, ...
  - Bitwise: &, | 
  - Logical: &&, ||

 Special built-in functions
  - convertvector()
  - shufflevector()

 Guaranteed to compile and be correct
  - Easier development + testing

```c
using Vec __attribute__((vector_size(16))) = uint32_t;
// Same as Vec but without 16-Byte alignment.
using UnalignedVec __attribute__((aligned(1))) = Vec;
// Vector of 4 bools (only available in LLVM).
using BitVec __attribute__((ext_vector_type(4))) = bool;

// Scan integer column and write matching row ids.
uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val, 
                          uint32_t* __restrict output)
{
  uint32_t num_matches = 0;
  for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
...}
```
Compiler-Intrinsics

 GCC/Clang have SIMD abstraction
  via __attribute__((vector_size(SIZE))))
  SIZE in bytes can be .../ 8 / 16 / 32 / 64/...

 Supports common operations
  Arithmetic: +, -, *, /, >>, ...
  Comparison: >=, <, !=, ...
  Bitwise: &,
  Logical: &&, ||

 Special built-in functions
  convertvector()
  shufflevector()

 Guaranteed to compile and be correct
  Easier development + testing

 // 16-Byte vector of 4x uint32_t.
 using Vec __attribute__((vector_size(16))) = uint32_t;
 // Same as Vec but without 16-Byte alignment.
 using UnalignedVec __attribute__((aligned(1))) = Vec;
 // Vector of 4 bools (only available in LLVM).
 using BitVec __attribute__((ext_vector_type(4))) = bool;

 // Scan integer column and write matching row ids.
 uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val,
                            uint32_t* __restrict output) {
  uint32_t num_matches = 0;
  for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
    // Load data and compare.
    Vec values = *(Vec*)(column + row);
    Vec matches = values < filter_val;
    // Convert comparison to scalar bitmask using built-in.
    BitVec bitvec = __builtin_convertvector(matches, BitVec);
    uint8_t bitmask = (uint8_t&)(bitvec);
    // Get ids from lookup table and add to current base row.
    Vec row_offsets = *(*(Vec*)MATCHES_TO_ROW_OFFSETS + bitmask);
    Vec compressed_rows = row + row_offsets;
    // Write matching row ids to output.
    *(UnalignedVec*)(output + num_matches) = compressed_rows;
    num_matches += std::popcount(bitmask);
  }
  return num_matches;
}
# Compiler-Intrinsics

» GCC/Clang have SIMD abstraction
   ‣ Via `__attribute__((vector_size(SIZE)))`
   ‣ SIZE in bytes can be .../8/16/32/64/...

» Supports common operations
   ‣ Arithmetic: +, -, *, /, >>,...
   ‣ Comparison: >=, <, !=, ...
   ‣ Bitwise: &,
   ‣ Logical: &&, ||

» Special built-in functions
   ‣ `convertvector()`
   ‣ `shufflevector()`

» Guaranteed to compile and be correct
   ‣ Easier development + testing

```c
using Vec __attribute__((vector_size(16))) = uint32_t;
// Same as Vec but without 16-Byte alignment.
using UnalignedVec __attribute__((aligned(1))) = Vec;
// Vector of 4 bools (only available in LLVM).
using BitVec __attribute__((ext_vector_type(4))) = bool;

uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val,
                           uint32_t* __restrict output) {
    uint32_t num_matches = 0;
    for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
        // Load data and compare.
        Vec values = *(Vec*)(column + row);
        Vec matches = values < filter_val;

        // Convert comparison to scalar bitmask using built-in.
        BitVec bitvec = __builtin_convertvector(matches, BitVec);
        uint8_t bitmask = (uint8_t&)(bitvec);
    }
    return num_matches;
}
```
// 16-Byte vector of 4x uint32_t.
using Vec __attribute__((vector_size(16))) = uint32_t;
// Same as Vec but without 16-Byte alignment.
using UnalignedVec __attribute__((aligned(1))) = Vec;
// Vector of 4 bools (only available in LLVM).
using BitVec __attribute__((ext_vector_type(4))) = bool;

// Scan integer column and write matching row ids.
uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val, 
                            uint32_t* __restrict output) {
    uint32_t num_matches = 0;
    for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
        // Load data and compare.
        Vec values = *(Vec*)(column + row);
        Vec matches = values < filter_val;

        // Convert comparison to scalar bitmask using built-in.
        BitVec bitvec = __builtin_convertvector(matches, BitVec);
        uint8_t bitmask = (uint8_t&bitvec);

        // Get ids from lookup table and add to current base row.
        Vec row_offsets = *(Vec*)MATCHES_TO_ROW_OFFSETS[bitmask];
        Vec compressed_rows = row + row_offsets;

        num_matches += std::popcount(bitmask);
    }
    return num_matches;
}
Compiler-Intrinsics

» GCC/Clang have SIMD abstraction
   ◦ via __attribute__((vector_size(SIZE)))
   ◦ SIZE in bytes can be .../8/16/32/64/...

» Supports common operations
   ◦ Arithmetic: +, -, *, /, >>, ...
   ◦ Comparison: >=, <, !=, ...
   ◦ Bitwise: &, |, 
   ◦ Logical: &&, ||

» Special built-in functions
   ◦ convertvector()
   ◦ shufflevector()

» Guaranteed to compile and be correct
   ◦ Easier development + testing

// 16-Byte vector of 4x uint32_t.
using Vec __attribute__((vector_size(16))) = uint32_t;
// Same as Vec but without 16-Byte alignment.
using UnalignedVec __attribute__((aligned(1))) = Vec;
// Vector of 4 bools (only available in LLVM).
using BitVec __attribute__((ext_vector_type(4))) = bool;

// Scan integer column and write matching row ids.
uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val, 
                           uint32_t* __restrict output) {
    uint32_t num_matches = 0;
    for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
        // Load data and compare.
        Vec values = *(Vec*)(column + row);
        Vec matches = values < filter_val;

        // Convert comparison to scalar bitmask using built-in.
        BitVec bitvec = __builtin_convertvector(matches, BitVec);
        uint8_t_t bitmask = (uint8_t_t)bitvec;

        // Get ids from lookup table and add to current base row.
        Vec row_offsets = *(Vec*)MATCHES_TO_ROW_OFFSETS[bitmask];
        Vec compressed_rows = row + row_offsets;

        // Write matching row ids to output.
        *(UnalignedVec*)(output + num_matches) = compressed_rows;
        num_matches += std::popcount(bitmask);
    }
    return num_matches;
}
Benchmarks

» M1: Apple Macbook Pro 14" M1 2021
» x86 Icelake: Intel Xeon Platinum 8352Y

» All experiments with:
  › Clang 15 (x86) and trunk Clang ~17 (ARM)
  › -O3, -march/-mtune=native

» Single-threaded
» Show relative speedup over scalar version

» Also other x86/ARM CPUs and GCC → see paper
Hash Bucket Lookup

» Vector comparison and conversion to scalar bitmask
Hash Bucket Lookup

» Vector comparison and conversion to scalar bitmask

a) x86 Icelake

Speedup

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>0</td>
</tr>
<tr>
<td>autovec</td>
<td>5.9us</td>
</tr>
<tr>
<td>vec-bitmask</td>
<td></td>
</tr>
<tr>
<td>sse4-bitmask</td>
<td></td>
</tr>
<tr>
<td>avx512-bitmask</td>
<td></td>
</tr>
</tbody>
</table>
Hash Bucket Lookup

» Vector comparison and conversion to scalar bitmask

» Vectorized >> scalar lookup
  › LLVM does not auto-vectorize well
Hash Bucket Lookup

» Vector comparison and conversion to scalar bitmask

» Vectorized >> scalar lookup
  › LLVM does not auto-vectorize well

» Compiler vec == hand-written SIMD
  › LLVM generates nearly identical code
Hash Bucket Lookup

» Vector comparison and conversion to scalar bitmask

» Vectorized >> scalar lookup
  › LLVM does not auto-vectorize well

» Compiler vec == hand-written SIMD
  › LLVM generates nearly identical code

» M1: submitted patch for bitmask conversion
  › vec-bitmask does bitmask == 0 check before bitmask conversion

---

**a) x86 Icelake**

- naive: 5.9us
- autovec: 4.9us
- vec-bitmask: 3.9us
- sse4-bitmask: 3.4us
- avx512-bitmask: 3.3us

**b) M1**

- naive: 3.9us
- autovec: 3.9us
- vec-bitmask: 3.3us (patched)
- neon-bitmask: 3.2us
Bit-Packed Integer Decompression

» Unpack packed 9-Bit integers to 32 bits → shuffling + shifting + masking
Bit-Packed Integer Decompression

» Unpack packed 9-Bit integers to 32 bits → shuffling + shifting + masking
» Compiler vec >= hand-written SIMD
  › x86: vec-512 LLVM adds AND instruction (not in avx512)

SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units
Thomas Willhalm et al. @ VLDB 2009
Bit-Packed Integer Decompression

» Unpack packed 9-Bit integers to 32 bits → shuffling + shifting + masking

» Compiler vec >= hand-written SIMD
  › x86: vec-512 LLVM adds AND instruction (not in avx512)
  › Neon: vec-128 generates better code than translated x86 intrinsics

![Bar chart showing speedup for different SIMD modes on two different architectures](chart.png)

SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units
Thomas Willhalm et al. @ VLDB 2009
Bit-Packed Integer Decompression

- Unpack packed 9-Bit integers to 32 bits → shuffling + shifting + masking
- Compiler vec ≥ hand-written SIMD
  - x86: vec-512 LLVM adds AND instruction (not in avx512)
  - Neon: vec-128 generates better code than translated x86 intrinsics
- Neon: vec-512 not physically supported → bad performance

SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units
Thomas Willhalm et al. @ VLDB 2009
Dictionary Table Scan

» Vector comparison, (conversion to bitmask + shuffle), store
Dictionary Table Scan

» Vector comparison, (conversion to bitmask + shuffle), store

» AVX512 compressstore
  › Clang doesn’t generate it for auto-vectorization
  › Can’t be expressed with compiler vectors

![Graph showing speedup comparison]

a) x86 Icelake

Compiler SIMD @ ADMS | 28.08.2023
Dictionary Table Scan

» Vector comparison, (conversion to bitmask + shuffle), store

» AVX512 compressstore
  › Clang doesn’t generate it for auto-vectorization
  › Can’t be expressed with compiler vectors

» NEON vec-128: Shuffling doesn’t use TBL instruction
  › With our LLVM patch → equal to NEON performance
Compiler-Intrinsics in Velox

» Velox: Meta's new unified query engine

» Removed xSIMD dependency
» Use only compiler-intrinsics

» End-to-end TPC-H SF1
» x86 → 0.1% diff
» NEON → 0.13% diff

» Removed:
» 54 platform-specific functions
» Hundreds of lines of SIMD code
Use Compiler-Intrinsics

» In 7/8 benchmarks + Velox: compiler-intrinsics $\approx$ platform-intrinsics
Use Compiler-Intrinsics

» In 7/8 benchmarks + Velox: compiler-intrinsics ≈ platform-intrinsics

» Approach to writing SIMD code
  › Try auto-vectorization
  › Use compiler-intrinsics
  › Use platform-intrinsics
Use Compiler-Intrinsics

» In 7/8 benchmarks + Velox: compiler-intrinsics ≈ platform-intrinsics

» Approach to writing SIMD code
  › Try auto-vectorization
  › Use compiler-intrinsics
  › Use platform-intrinsics

» Structure code around compiler-intrinsics
» Only localized platform-specific code
Use Compiler-Intrinsics

» In 7/8 benchmarks + Velox: compiler-intrinsics ≈ platform-intrinsics

» Approach to writing SIMD code
  › Try auto-vectorization
  › Use compiler-intrinsics
  › Use platform-intrinsics

» Structure code around compiler-intrinsics
» Only localized platform-specific code

» Easier development + testing
  › Don't need AVX512 CPU for 512-bit vector code

» Potentially better/more optimizations
» Adapt SIMD code to own need, not dependency
Summary

Writing SIMD Code with Platform-Intrinsics

- Writing platform-intrinsics code is hard and cumbersome

Compiler-Intrinsics

- GCC does not have SIMD abstraction
  - The intrinsics are very slow
  - Using them in the code is cumbersome

- Supports common operations
  - Addition, subtraction, multiplication, division
  - Comparison, logic, bit manipulation

- Special built-in functions
  - Comparison
  - Bit manipulation

- Guaranteed to compile, run, be correct
  - Basic development and testing

Compiler-Intrinsics achieve the same performance

- Compiler-Intrinsics in Velox
  - Velox: Meta’s new unified query engine
    - Removed SIMD dependency
    - Use only compiler-intrinsics
    - End-to-end TPC-H SFI from Parquet
  - X86 -> 0.5 s/TF
  - X64 -> 0.1 s/TF
  - Removed:
    - 54 platform-specific functions
    - Hundreds of lines of SIMD code

https://github.com/hpides/autovec-db