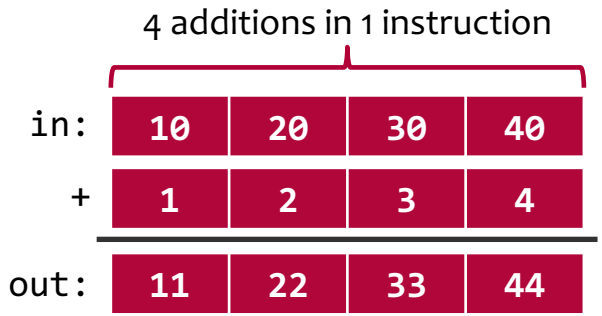
Evaluating SIMD Compiler-Intrinsics for Database Systems

Lawrence Benson, Richard Ebeling, Tilmann Rabl
lawrence.benson@hpi.de

ADMS | 28.08.23

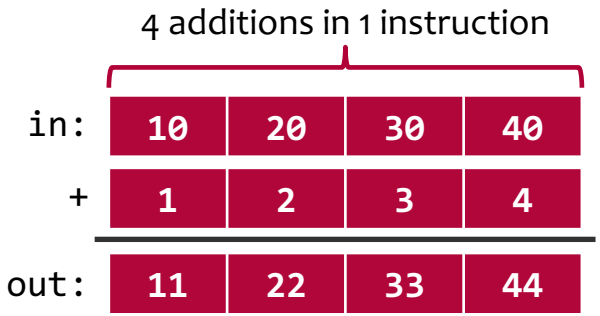
SIMD in a Nutshell

» Single Instruction Multiple Data (= SIMD)



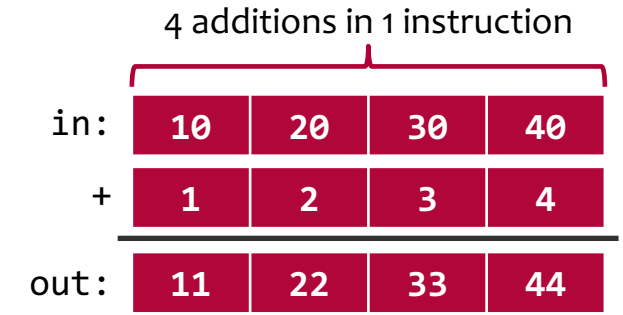
SIMD in a Nutshell

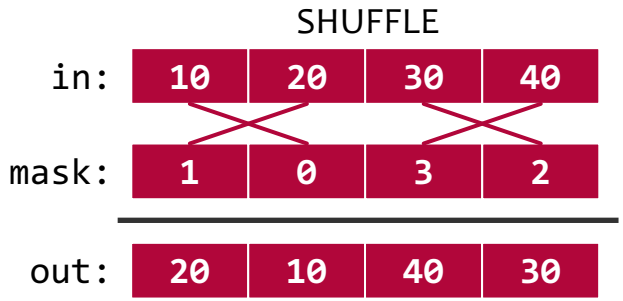
- Single Instruction Multiple Data (= SIMD)
- Most common instruction sets
  - x86: SSE, AVX, AVX2, AVX512 (> 6k instructions)
  - ARM: Neon (> 4k instructions), SVE
  - PowerPC AltiVec, RISC-V V



SIMD in a Nutshell

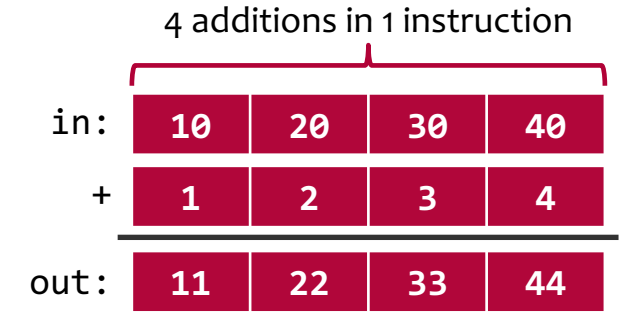
- Single Instruction Multiple Data (= SIMD)
- Most common instruction sets
  - x86: SSE, AVX, AVX2, AVX512 (> 6k instructions)
  - ARM: Neon (> 4k instructions), SVE
  - PowerPC AltiVec, RISC-V V
- Arithmetic, Logical, Shuffle, Shift, Load, Store, …

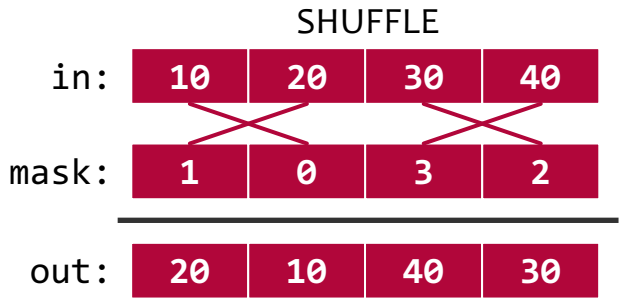




SIMD in a Nutshell

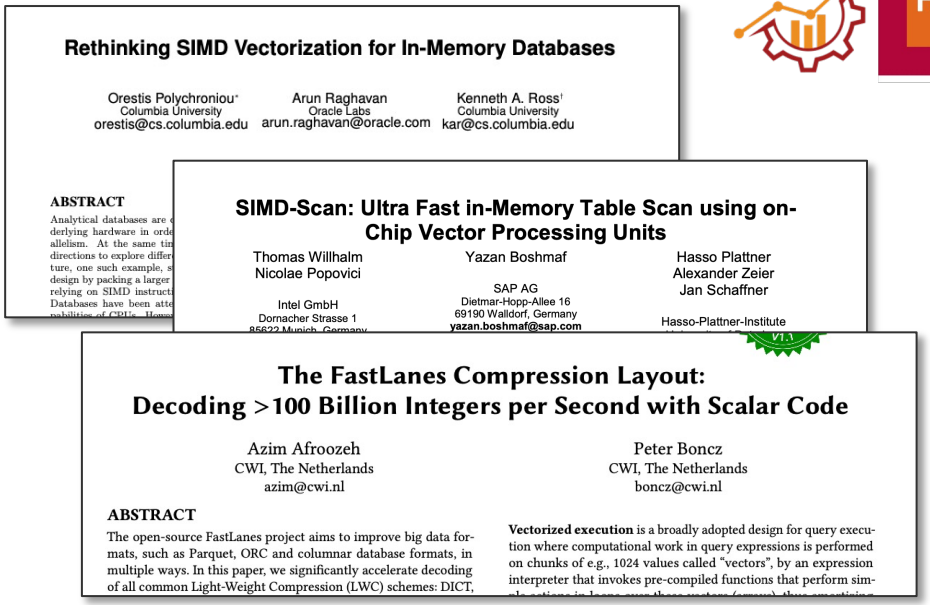
- Single Instruction Multiple Data (= SIMD)
- Most common instruction sets
  - x86: SSE, AVX, AVX2, AVX512 (> 6k instructions)
  - ARM: Neon (> 4k instructions), SVE
  - PowerPC AltiVec, RISC-V V
- Arithmetic, Logical, Shuffle, Shift, Load, Store, …
- Focus on x86 and Neon
  - x86: 128 – 512 Bit registers
  - Neon: 128 Bit registers





SIMD in Databases

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



Rethinking SIMD Vectorization for In-Memory Databases

Orestis Polychroniou* Columbia University orestis@cs.columbia.edu
Arun Raghavan Oracle Labs arun.raghavan@oracle.com
Kenneth A. Ross† Columbia University kar@cs.columbia.edu

ABSTRACT

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

Thomas Willhalm
Nicolae Popovici
Intel GmbH
Dornacher Strasse 1

Yazan Boshmaf
SAP AG
Dietmar-Hopp-Allee 16
69190 Walldorf, Germany
yazan.boshmaf@sap.com

Hasso Plattner
Alexander Zeier
Jan Schaffner
Hasso-Plattner-Institute

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

Azim Afroozeh
CWI, The Netherlands
azim@cwi.nl

Peter Boncz
CWI, The Netherlands
boncz@cwi.nl

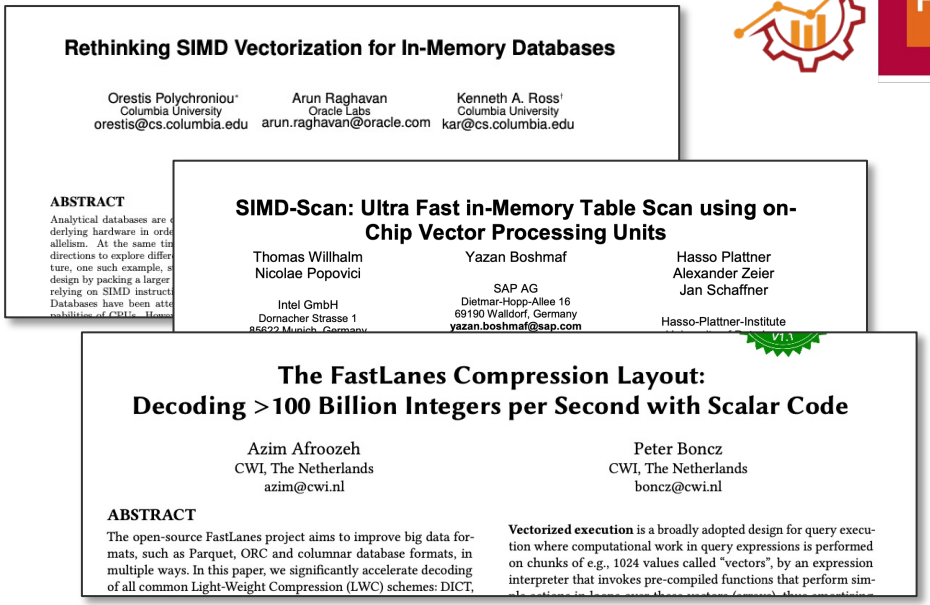
ABSTRACT

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: DICT,

Vectorized execution is a broadly adopted design for query execution where computational work in query expressions is performed on chunks of e.g., 1024 values called "vectors", by an expression interpreter that invokes pre-compiled functions that perform sim-

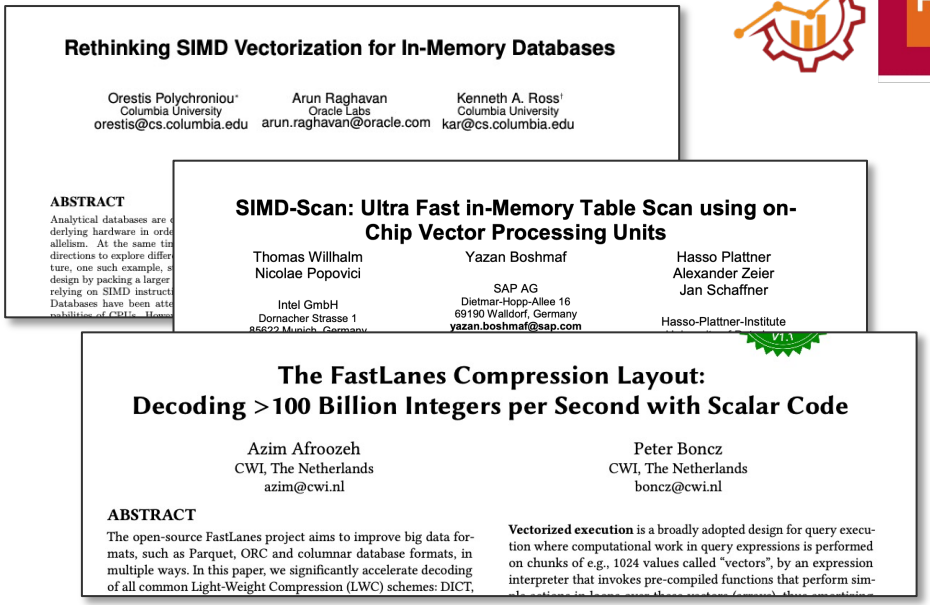
SIMD in Databases

- Used to speed up, e.g.,
  - Table scans
  - Hash tables
  - Sorting
- SIMD code is …
  - … hard to develop
  - … hard to test
  - … hard to benchmark



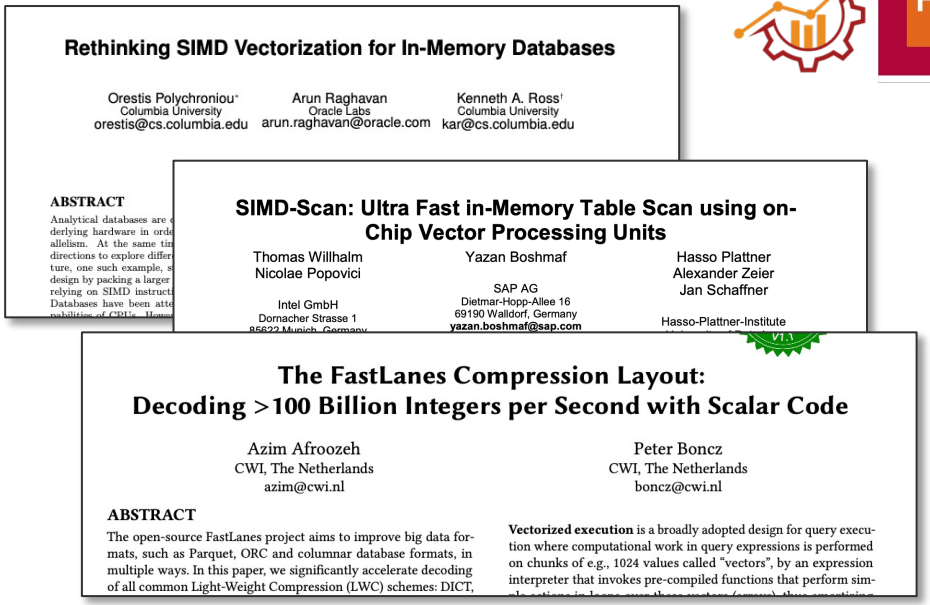
SIMD in Databases

- 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?



Rethinking SIMD Vectorization for In-Memory Databases

Orestis Polychroniou* Columbia University orestis@cs.columbia.edu
Arun Raghavan Oracle Labs arun.raghavan@oracle.com
Kenneth A. Ross† Columbia University kar@cs.columbia.edu

ABSTRACT

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

Thomas Willhalm, Nicolae Popovici — Intel GmbH, Dornacher Strasse 1
Yazan Boshmaf — SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany, yazan.boshmaf@sap.com
Hasso Plattner, Alexander Zeier, Jan Schaffner — Hasso-Plattner-Institute

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

Azim Afroozeh, 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: DICT,

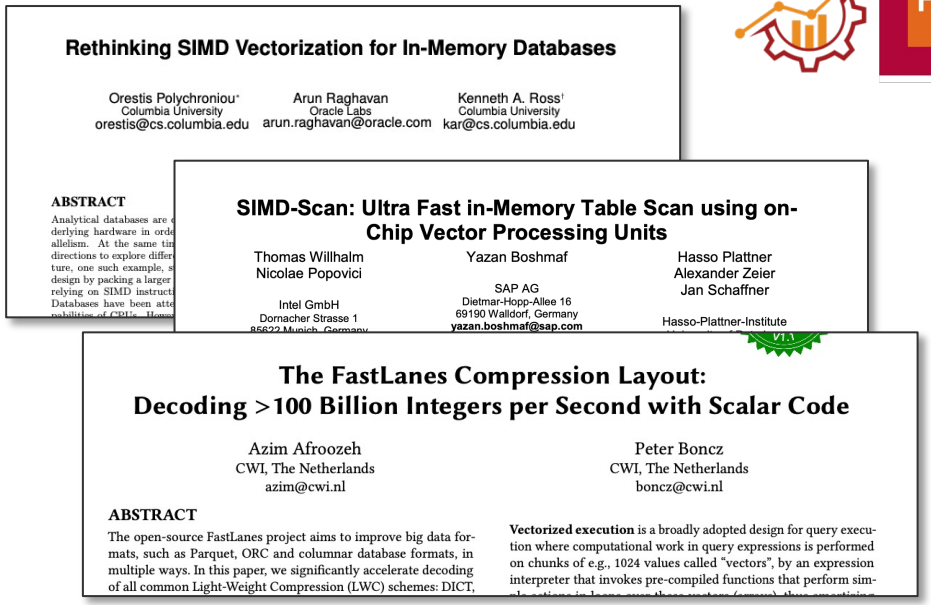
Vectorized execution is a broadly adopted design for query execution where computational work in query expressions is performed on chunks of e.g., 1024 values called "vectors", by an expression interpreter that invokes pre-compiled functions that perform sim-

What do these functions do?

```
_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?

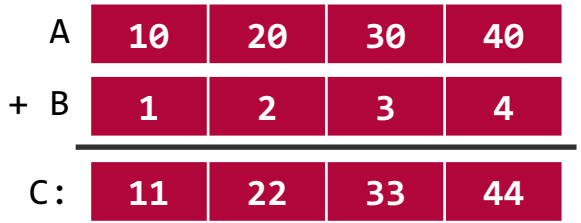
```
_mm_add_epi32()
_mm512_srl_epi64()
vaddq_s32()
```

Don't have AVX512?
→ *Can't compile*

Don't have NEON?
→ *Can't compile*

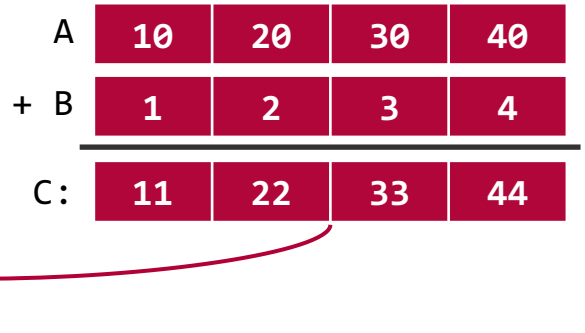
Abstractions on top of Abstractions

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

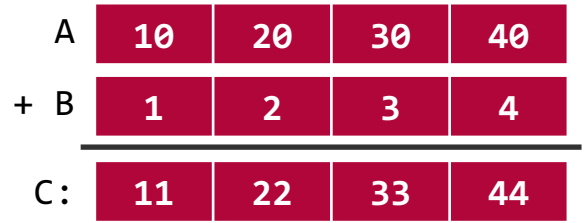


Application code

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

Abstractions on top of Abstractions

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



Application code

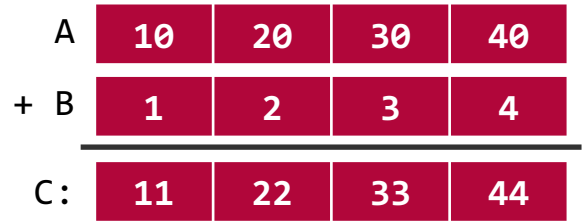
```
vec C = A + B;
```

SIMD library

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

Abstractions on top of Abstractions

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



Application code

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

SIMD intrinsics

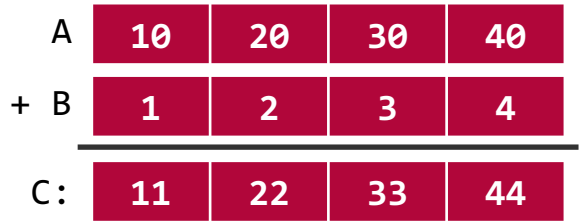
```
_mm_add_epi32
vaddq_s32
```

SIMD library

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

Abstractions on top of Abstractions

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



Application code

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

SIMD intrinsics

```
_mm_add_epi32
vaddq_s32
```

SIMD library

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

Compiler representation

```
__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

```
// 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)));
```

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

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)));



__m128i _mm_add_epi32(__m128i __a, __m128i __b) {

}
```

Compiler Representation
GCC/Clang's "vector" type

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)));



__m128i _mm_add_epi32(__m128i __a, __m128i __b) {

}

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

int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {



}
```

Compiler Representation
GCC/Clang's "vector" type

Abstractions on top of Abstractions

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

SIMD Types
16 Byte type in x86

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

Compiler Representation
GCC/Clang's "vector" type

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

__m128i _mm_add_epi32(__m128i __a, __m128i __b) {

}

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

int32x4_t vaddq_s32(int32x4_t __p0, int32x4_t __p1) {


}
```

Abstractions on top of Abstractions

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

SIMD Types
16 Byte type in x86

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

Compiler Representation
GCC/Clang's "vector" type

```
// 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

SIMD Types
16 Byte type in x86

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

Compiler Representation
GCC/Clang's "vector" type

```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

SIMD Types
16 Byte type in x86

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

```
// 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;
}
```

Compiler Representation
GCC/Clang's "vector" type

Operators
`operator+()` on vector type

Abstractions on top of Abstractions

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

SIMD Types
16 Byte type in x86

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

```
// 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;
}
```

Compiler Representation
GCC/Clang's "vector" type

Operators
operator+() on vector type

Platform-intrinsics are abstractions on top of compiler-intrinsics

Abstractions on top of Abstractions

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

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

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

```
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

```
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

```
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;
}
```

Platform-intrinsics are abstractions on top of compiler-intrinsics

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;
}
```

Platform-intrinsics are abstractions on top of compiler-intrinsics

```
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

```
01 // 16-Byte vector of 4x uint32_t.
02 using Vec __attribute__((vector_size(16))) = uint32_t;
03 // Same as Vec but without 16-Byte alignment.
04 using UnalignedVec __attribute__((aligned(1))) = Vec;
05 // Vector of 4 bools (only available in LLVM).
06 using BitVec __attribute__((ext_vector_type(4))) = bool;
07
08
09
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

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

```
01 // 16-Byte vector of 4x uint32_t.
02 using Vec __attribute__((vector_size(16))) = uint32_t;
03 // Same as Vec but without 16-Byte alignment.
04 using UnalignedVec __attribute__((aligned(1))) = Vec;
05 // Vector of 4 bools (only available in LLVM).
06 using BitVec __attribute__((ext_vector_type(4))) = bool;
07
08 // Scan integer column and write matching row ids.
09 uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val,
10                            uint32_t* __restrict output) {
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30 }
```

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

```
01 // 16-Byte vector of 4x uint32_t.
02 using Vec __attribute__((vector_size(16))) = uint32_t;
03 // Same as Vec but without 16-Byte alignment.
04 using UnalignedVec __attribute__((aligned(1))) = Vec;
05 // Vector of 4 bools (only available in LLVM).
06 using BitVec __attribute__((ext_vector_type(4))) = bool;
07
08 // Scan integer column and write matching row ids.
09 uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val,
10                            uint32_t* __restrict output) {
11   uint32_t num_matches = 0;
12   for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28   }
29
30 }
```

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

```
01 // 16-Byte vector of 4x uint32_t.
02 using Vec __attribute__((vector_size(16))) = uint32_t;
03 // Same as Vec but without 16-Byte alignment.
04 using UnalignedVec __attribute__((aligned(1))) = Vec;
05 // Vector of 4 bools (only available in LLVM).
06 using BitVec __attribute__((ext_vector_type(4))) = bool;
07
08 // Scan integer column and write matching row ids.
09 uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val,
10                            uint32_t* __restrict output) {
11   uint32_t num_matches = 0;
12   for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
13     // Load data and compare.
14     Vec values = *(Vec*)(column + row);
15     Vec matches = values < filter_val;
16
17
18
19
20
21
22
23
24
25
26
27
28   }
29
30 }
```

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

```
01 // 16-Byte vector of 4x uint32_t.
02 using Vec __attribute__((vector_size(16))) = uint32_t;
03 // Same as Vec but without 16-Byte alignment.
04 using UnalignedVec __attribute__((aligned(1))) = Vec;
05 // Vector of 4 bools (only available in LLVM).
06 using BitVec __attribute__((ext_vector_type(4))) = bool;
07
08 // Scan integer column and write matching row ids.
09 uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val,
10                            uint32_t* __restrict output) {
11   uint32_t num_matches = 0;
12   for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
13     // Load data and compare.
14     Vec values = *(Vec*)(column + row);
15     Vec matches = values < filter_val;
16
17     // Convert comparison to scalar bitmask using built-in.
18     BitVec bitvec = __builtin_convertvector(matches, BitVec);
19     uint8_t bitmask = (uint8_t&) bitvec;
20
21
22
23
24
25
26
27
28   }
29
30 }
```

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

```
01 // 16-Byte vector of 4x uint32_t.
02 using Vec __attribute__((vector_size(16))) = uint32_t;
03 // Same as Vec but without 16-Byte alignment.
04 using UnalignedVec __attribute__((aligned(1))) = Vec;
05 // Vector of 4 bools (only available in LLVM).
06 using BitVec __attribute__((ext_vector_type(4))) = bool;
07
08 // Scan integer column and write matching row ids.
09 uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val,
10                            uint32_t* __restrict output) {
11   uint32_t num_matches = 0;
12   for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
13     // Load data and compare.
14     Vec values = *(Vec*)(column + row);
15     Vec matches = values < filter_val;
16
17     // Convert comparison to scalar bitmask using built-in.
18     BitVec bitvec = __builtin_convertvector(matches, BitVec);
19     uint8_t bitmask = (uint8_t&) bitvec;
20
21     // Get ids from lookup table and add to current base row.
22     Vec row_offsets = *(Vec*) MATCHES_TO_ROW_OFFSETS[bitmask];
23     Vec compressed_rows = row + row_offsets;
24
25
26
27
28   }
29
30 }
```

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

```
01 // 16-Byte vector of 4x uint32_t.
02 using Vec __attribute__((vector_size(16))) = uint32_t;
03 // Same as Vec but without 16-Byte alignment.
04 using UnalignedVec __attribute__((aligned(1))) = Vec;
05 // Vector of 4 bools (only available in LLVM).
06 using BitVec __attribute__((ext_vector_type(4))) = bool;
07
08 // Scan integer column and write matching row ids.
09 uint32_t dense_column_scan(uint32_t* column, uint32_t filter_val,
10                            uint32_t* __restrict output) {
11   uint32_t num_matches = 0;
12   for (uint32_t row = 0; row < NUM_ROWS; row += 4) {
13     // Load data and compare.
14     Vec values = *(Vec*)(column + row);
15     Vec matches = values < filter_val;
16
17     // Convert comparison to scalar bitmask using built-in.
18     BitVec bitvec = __builtin_convertvector(matches, BitVec);
19     uint8_t bitmask = (uint8_t&) bitvec;
20
21     // Get ids from lookup table and add to current base row.
22     Vec row_offsets = *(Vec*) MATCHES_TO_ROW_OFFSETS[bitmask];
23     Vec compressed_rows = row + row_offsets;
24
25     // Write matching row ids to output.
26     *(UnalignedVec*)(output + num_matches) = compressed_rows;
27     num_matches += std::popcount(bitmask);
28   }
29   return num_matches;
30 }
```

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

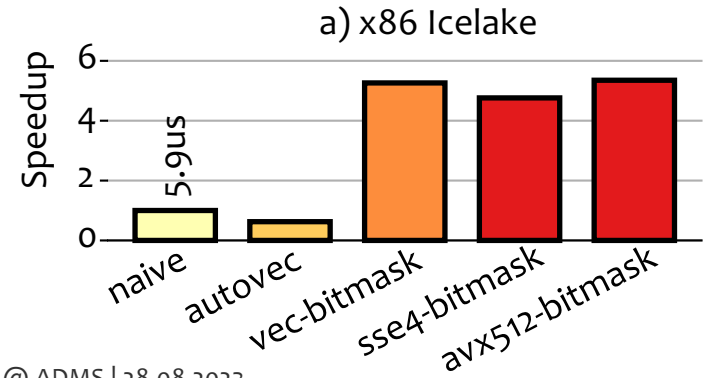
- Also other x86/ARM CPUs and GCC → see paper

Hash Bucket Lookup

» Vector comparison and conversion to scalar bitmask

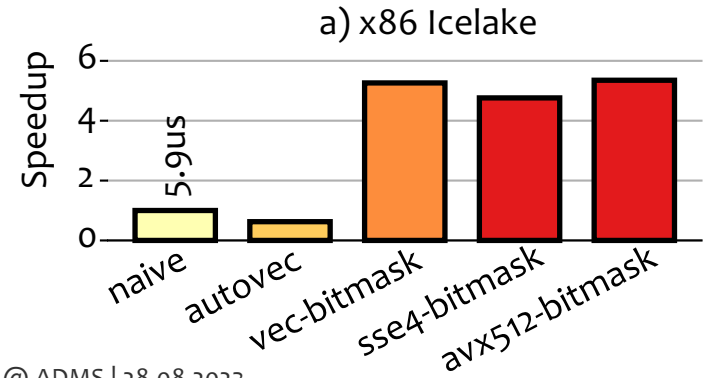
Hash Bucket Lookup

» Vector comparison and conversion to scalar bitmask



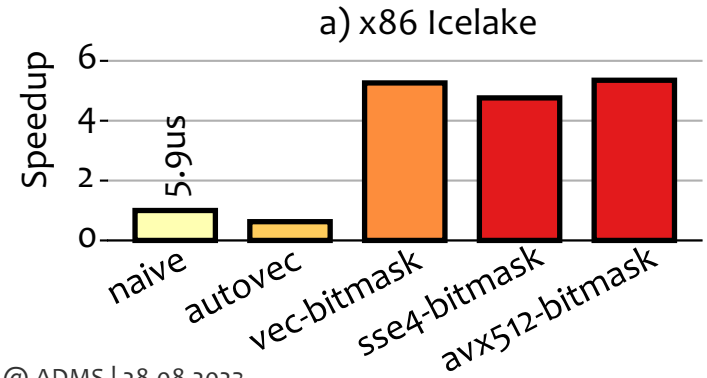
Hash Bucket Lookup

- Vector comparison and conversion to scalar bitmask
- Vectorized >> scalar lookup
  - LLVM does not auto-vectorize well



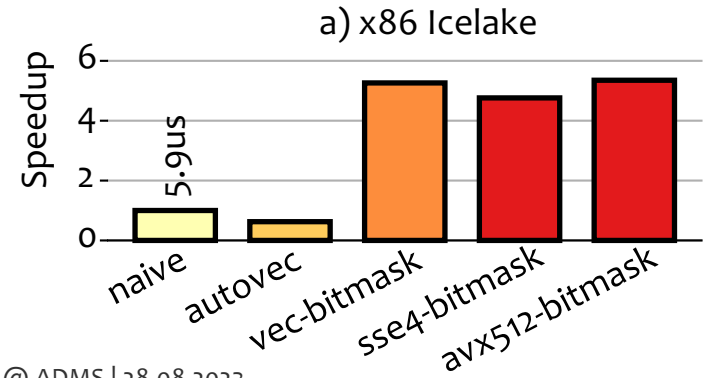
Hash Bucket Lookup

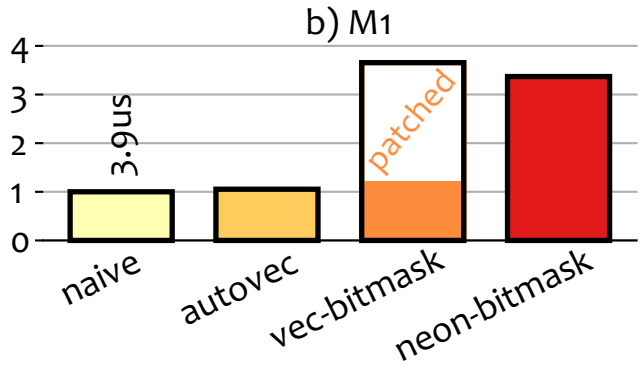
- Vector comparison and conversion to scalar bitmask
- Vectorized >> scalar lookup
  - LLVM does not auto-vectorize well
- Compiler vec == hand-written SIMD
  - LLVM generates nearly identical code



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





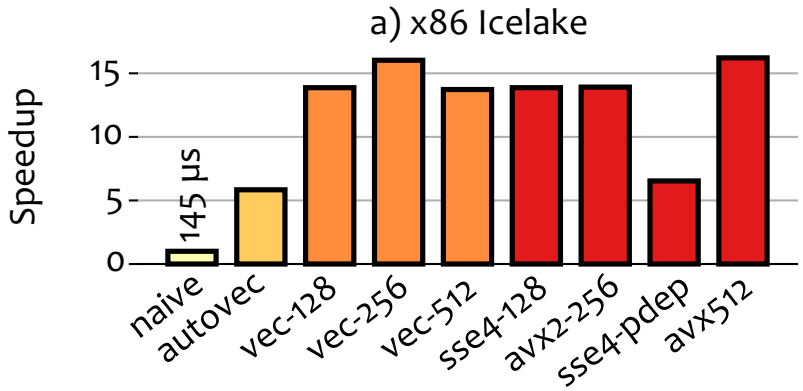
Bit-Packed Integer Decompression

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

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

Bit-Packed Integer Decompression

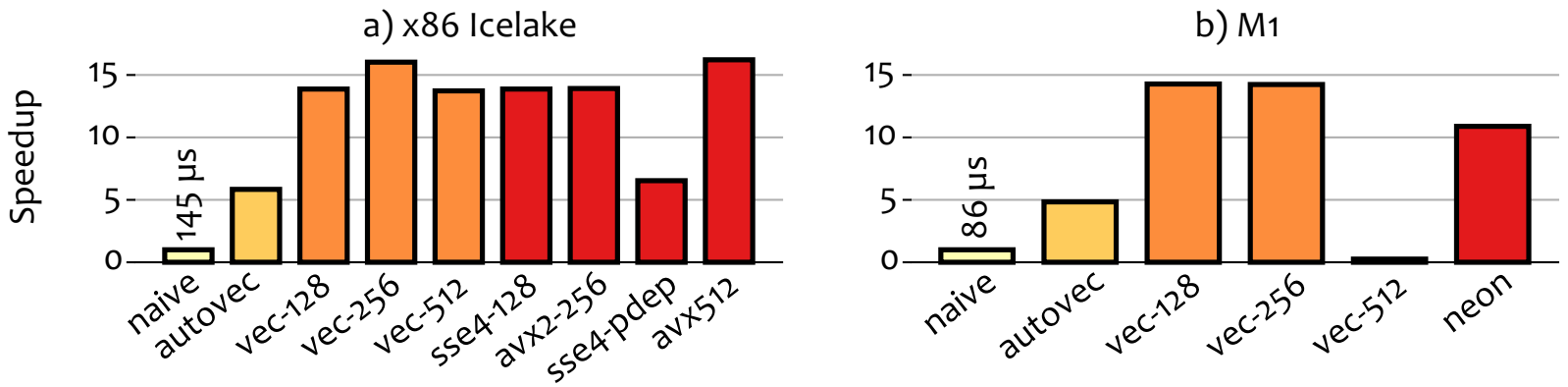
- Unpack packed 9-Bit integers to 32 bits → shuffling + shifting + masking
- Compiler vec >= hand-written SIMD
  - x86: *vec-512* LLVM adds AND instruction (not in *avx512*)



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

Bit-Packed Integer Decompression

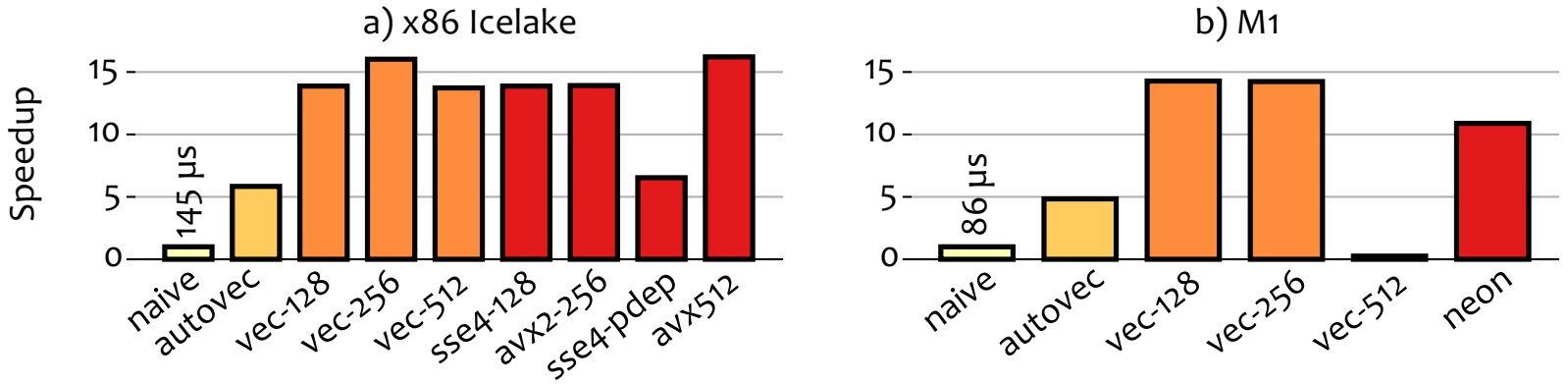
- Unpack packed 9-Bit integers to 32 bits → shuffling + shifting + masking
- Compiler vec >= hand-written SIMD
  - x86: *vec-512* LLVM adds AND instruction (not in *avx512*)
  - Neon: *vec-128* generates better code than translated x86 intrinsics



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

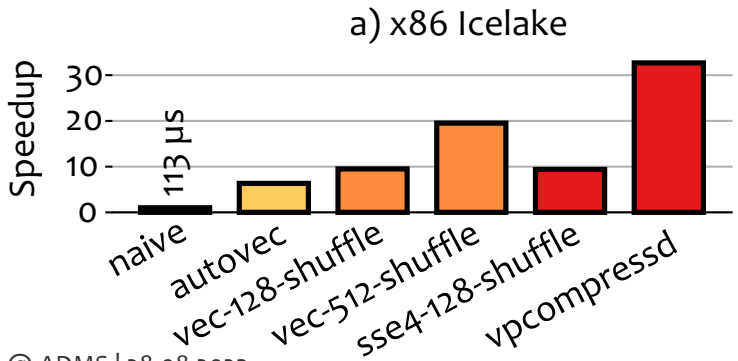
Dictionary Table Scan

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

Dictionary Table Scan

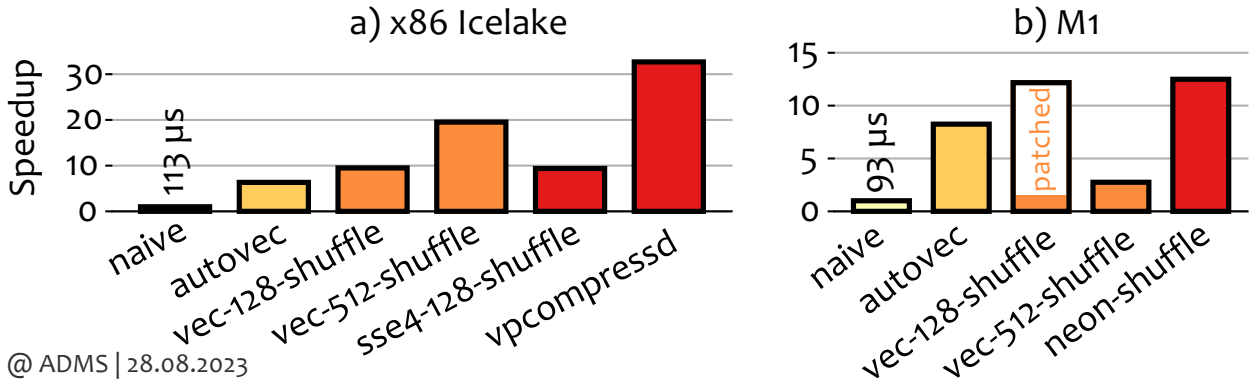
» Vector comparison, (conversion to bitmask + shuffle), store

» AVX512 `compresstore`
  › Clang doesn't generate it for auto-vectorization
  › Can't be expressed with compiler vectors



Dictionary Table Scan

- Vector comparison, (conversion to bitmask + shuffle), store
- AVX512 `compresstore`
  - 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



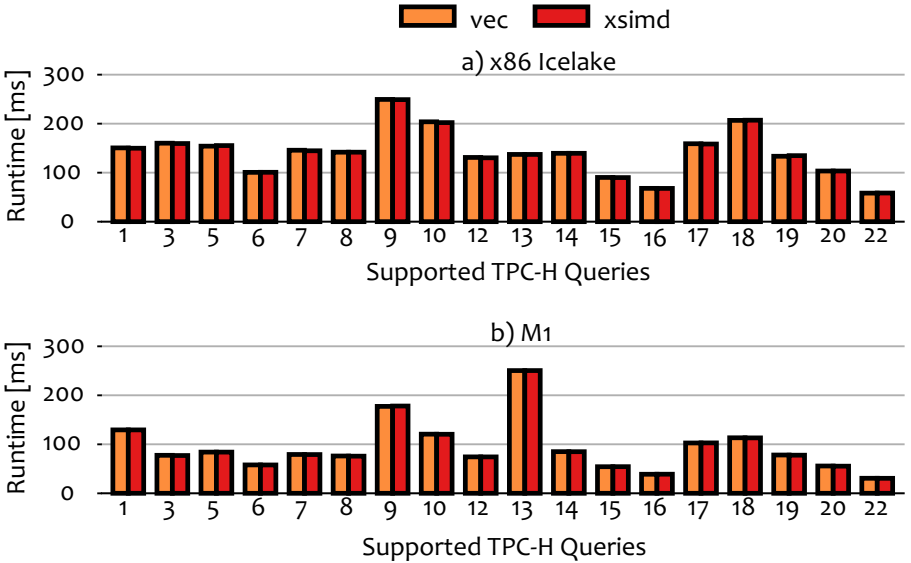
Compiler-Intrinsics in Velox

- 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 ≈≈ 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

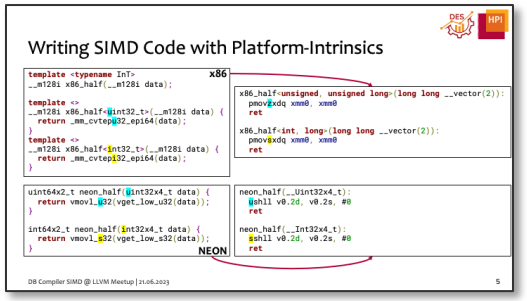
Use Compiler-Intrinsics

- 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

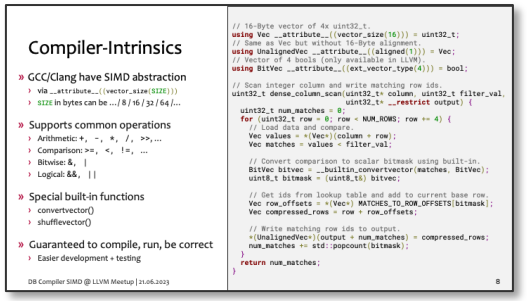
Use Compiler-Intrinsics

- 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

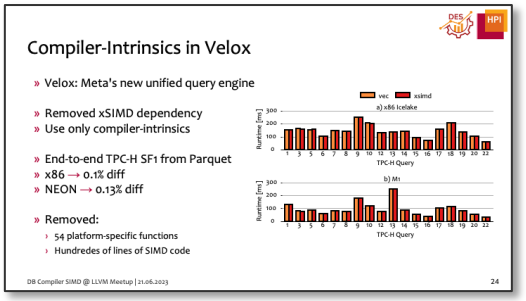
Summary



Writing platform-intrinsics code is hard and cumbersome



Compiler-intrinsics generic and cross-platform



Compiler-intrinsics achieve the same performance

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