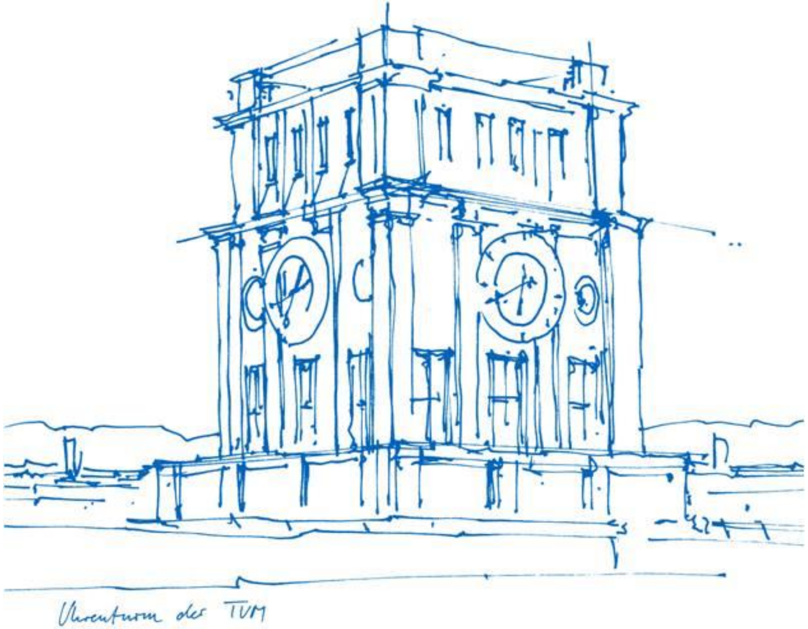
Portable Compiler-Centric SIMD Code in Databases

Lawrence Benson

Technische Universität München

HPTS '24



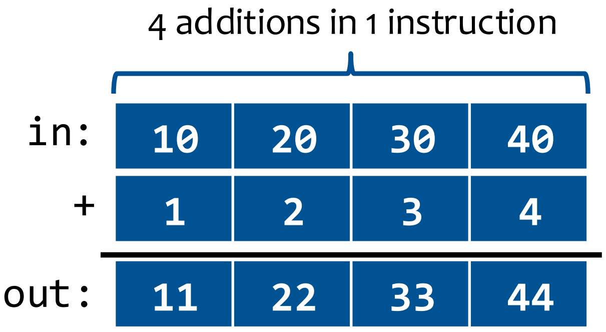
SIMD in a Nutshell

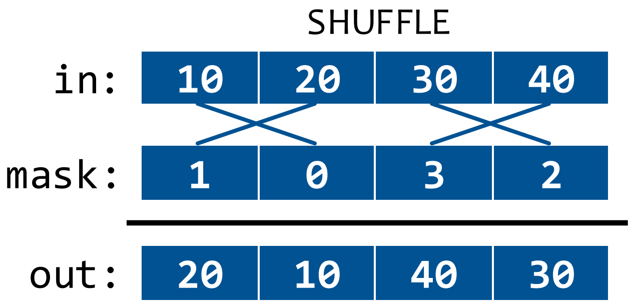
- Single Instruction Multiple Data (= SIMD)

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

- Arithmetic, Logical, Shuffle, Shift, Load, Store, …

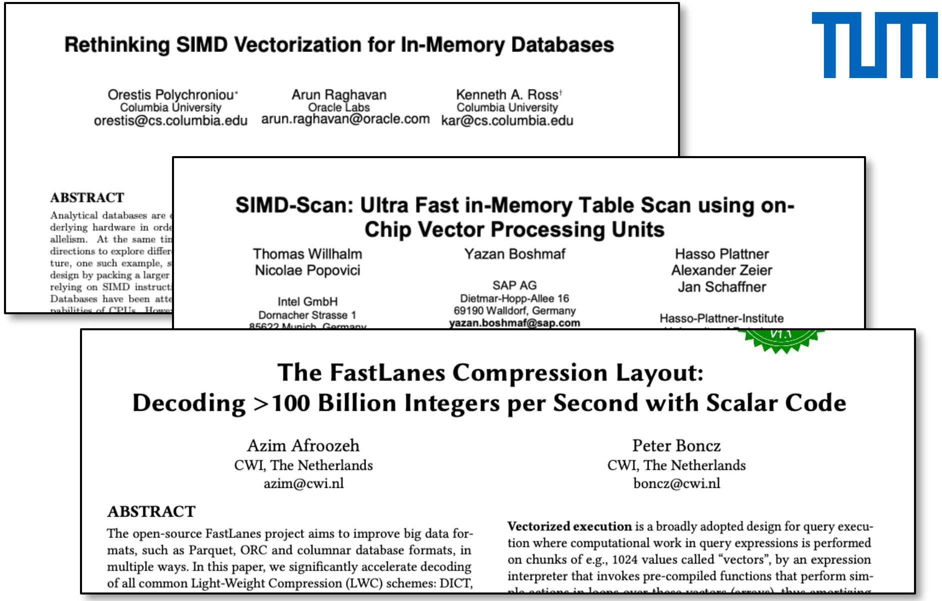
- Focus on x86 and Neon
  - x86: 128 – 512 Bit registers
  - Neon: 128 Bit registers





SIMD in Databases

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

Analytical databases are c
derlying hardware in orde
allelism. At the same tin
directions to explore differ
ture, one such example, s
design by packing a larger
relying on SIMD instructi
Databases have been atte
pabilities of CPUs. Howe

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

Thomas Willhalm
Nicolae Popovici

Intel GmbH
Dornacher Strasse 1
85622 Munich, Germany

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

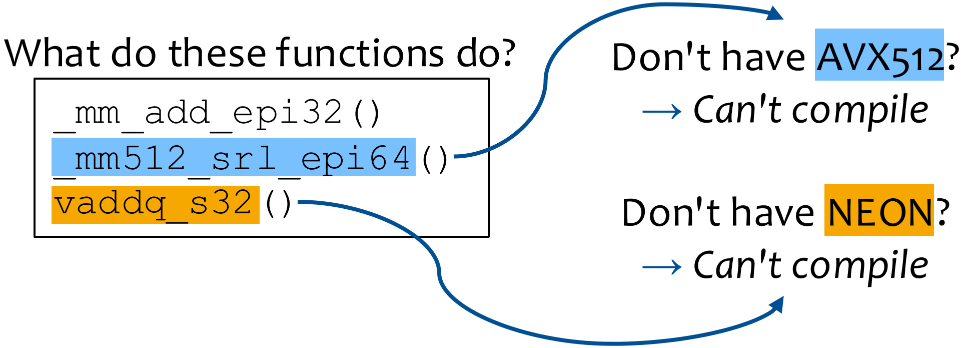
Azim Afroozeh
CWI, The Netherlands
azim@cwi.nl

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?

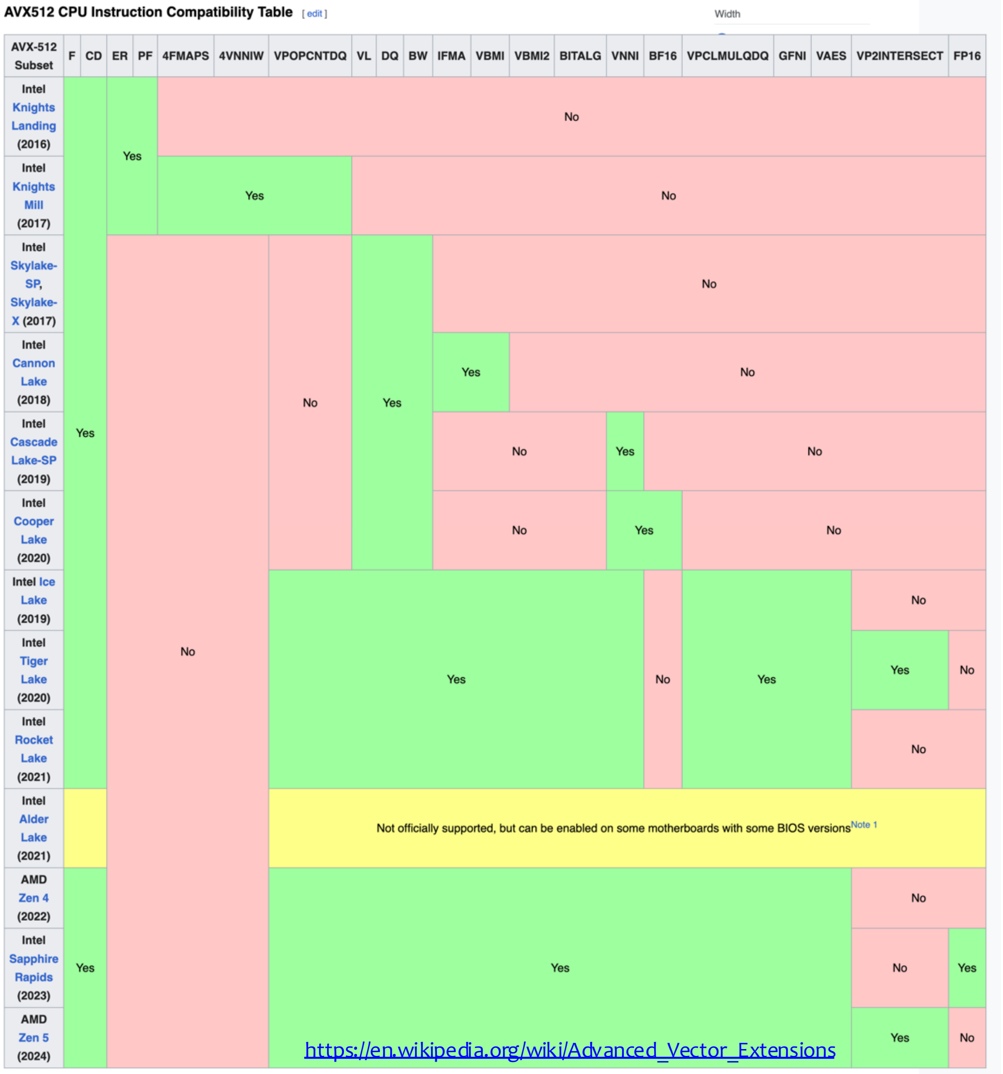
```
_mm_add_epi32()
_mm512_srl_epi64()
vaddq_s32()
```

Don't have AVX512?
→ *Can't compile*

Don't have NEON?
→ *Can't compile*

- Hand-picking instructions for AVX512 is *rather complicated*

- Let the compiler do it for you :)



AVX512 CPU Instruction Compatibility Table [edit]

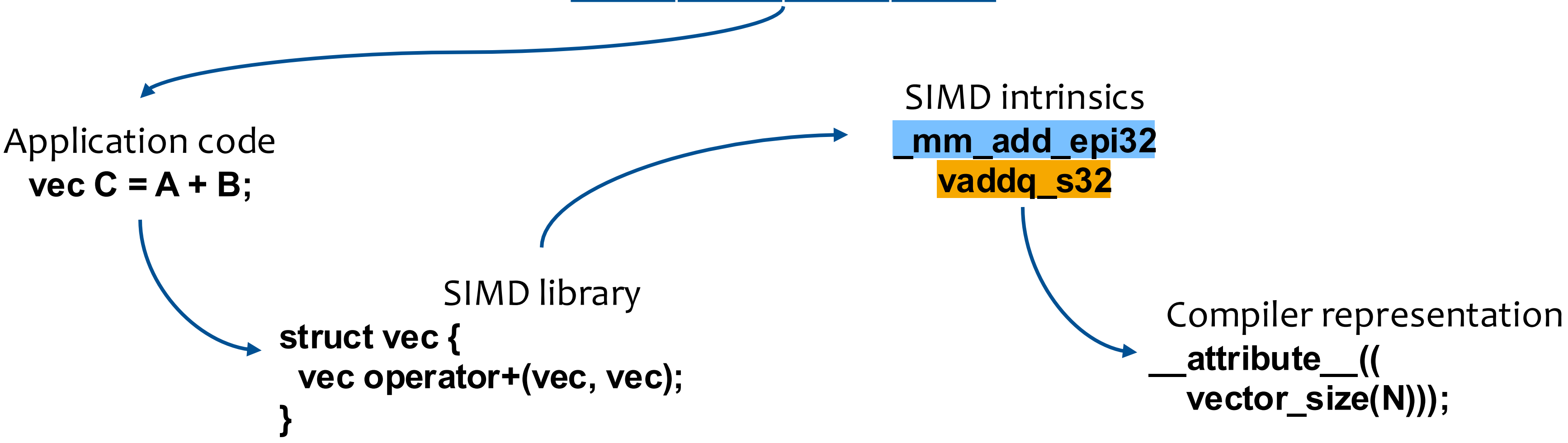
AVX-512 Subset	F	CD	ER	PF	4FMAPS	4VNNIW	VPOPCNTDQ	VL	DQ	BW	IFMA	VBMI	VBMI2	BITALG	VNNI	BF16	VPCLMULQDQ	GFNI	VAES	VP2INTERSECT	FP16
Intel Knights Landing (2016)	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
Intel Knights Mill (2017)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No
Intel Skylake-SP, Skylake-X (2017)	Yes	Yes	No	No	No	No	No	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No
Intel Cannon Lake (2018)	Yes	Yes	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No
Intel Cascade Lake-SP (2019)	Yes	Yes	No	No	No	No	No	Yes	Yes	Yes	No	No	No	No	Yes	No	No	No	No	No	No
Intel Cooper Lake (2020)	Yes	Yes	No	No	No	No	No	Yes	Yes	Yes	No	No	No	No	Yes	Yes	No	No	No	No	No
Intel Ice Lake (2019)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No
Intel Tiger Lake (2020)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No
Intel Rocket Lake (2021)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No
Intel Alder Lake (2021)			No	No	No	No	Not officially supported, but can be enabled on some motherboards with some BIOS versions[Note 1]														
AMD Zen 4 (2022)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No
Intel Sapphire Rapids (2023)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
AMD Zen 5 (2024)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No

https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

Abstractions on top of Abstractions

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

A	10	20	30	40
+ B	1	2	3	4
= C	11	22	33	44



Application code

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

SIMD library

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

SIMD intrinsics

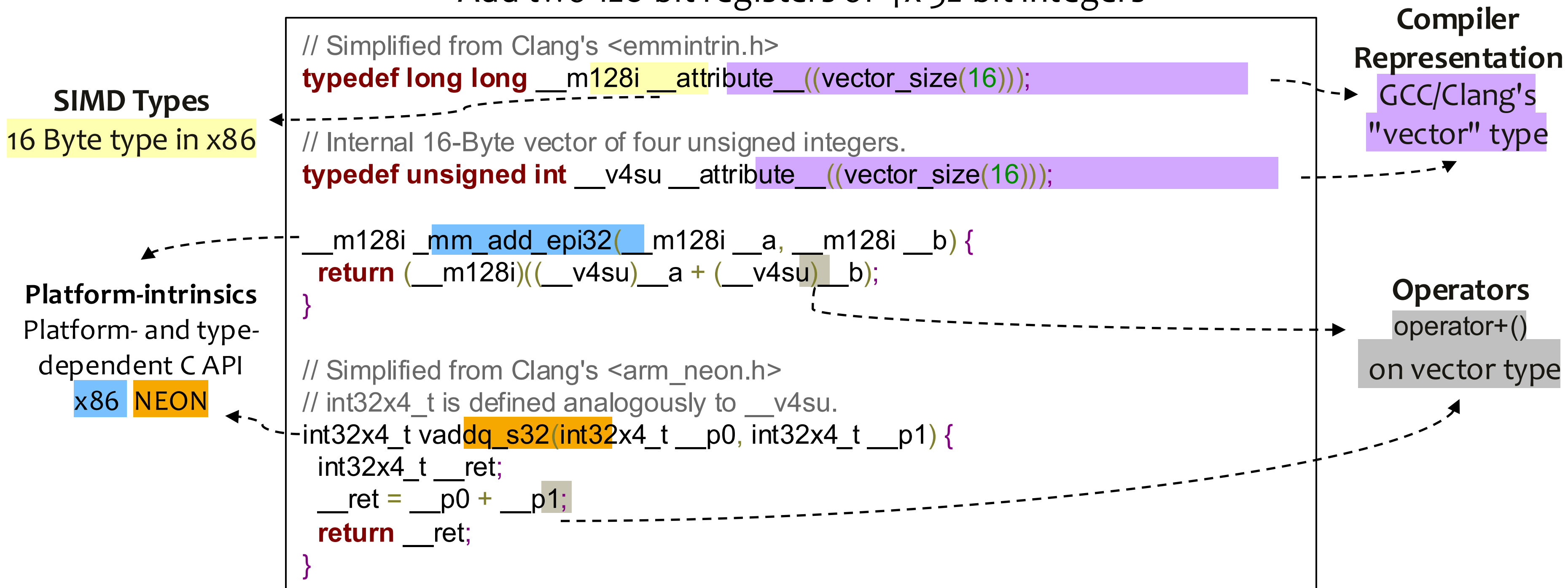
```
_mm_add_epi32
vaddq_s32
```

Compiler representation

```
__attribute__((
   vector_size(N)));
```

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)));

// 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;
}
```

SIMD Types

16 Byte type in x86

Platform-intrinsics

Platform- and type-dependent C API

x86 NEON

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

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 /…
- Supports common operations
  - Arithmetic: +, -, *, /, >>, …
  - Comparison: >=, <, !=, …
  - Bitwise: &, |
  - Logical: &&, ||
- Special built-in functions
  - convertvector()
  - shufflevector()
  - vectorelements()
- Guaranteed to compile and be correct
  - Easier development + testing

FastLanes Bitpacking Algorithm

```
using Vec __attribute__((vector_size(VECTOR_SIZE))) = LaneT;
...

auto* in = (VecT*)(compressed_in);
auto* out = (OutVecT*)(values_out);
VecT in_vec = *in;
LaneT overflow = 0;

for (uint32_t k = 0; k < NUM_BITS; ++k) {
  for (LaneT i = overflow; i <= BITS_IN_LANE; i += NUM_BITS) {
    VecT tmp = (in_vec >> i) & MASK;
    *(out++) = __builtin_convertvector(tmp, OutVecT);
  }

  if constexpr (LANE_WIDTH % NUM_BITS == 0) {
    in_vec = *(++in);
  } else if (k < NUM_BITS - 1) {
    LaneT tail = ((k + 1) * LANE_WIDTH) % NUM_BITS;
    VecT out_vec = tail > 0
                 ? in_vec >> (LANE_WIDTH - tail)
                         : ZERO_VEC;

    in_vec = *(++in);
    overflow = NUM_BITS - tail;

    out_vec |= (in_vec << (NUM_BITS - overflow)) & MASK;
    *(out++) = __builtin_convertvector(out_vec, OutVecT);
  }
}
```

Benchmarks

- Bitpacking FastLanes-style
  - Achieves state-of-the-art performance
- Dynamic Dispatch
  - Build once, run everywhere
- Filter with selection vector
  - Supports complex instructions

- Run with ~latest Clang (18/19/trunk)
  - Some features in active development
- -O3, -march/-mtune=native
- Single-threaded performance

Bitpacking

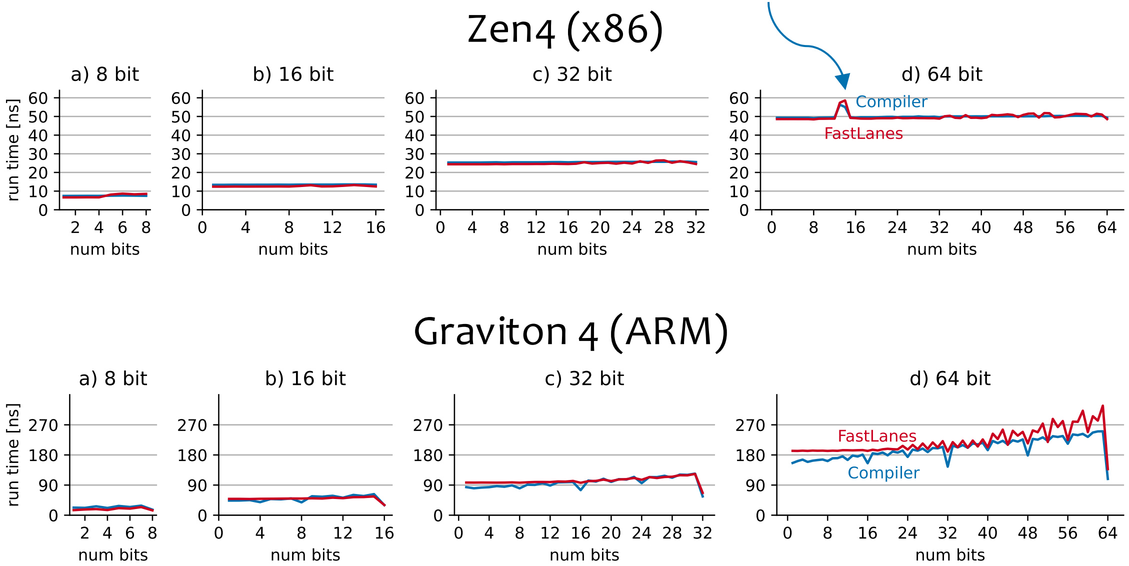
- » Based on FastLanes[1]
- » Uses 1024-bit "virtual" vector
- » Decompress 1024 values

- » Identical performance

- » FastLanes: ~18k generated LoC
- » Compiler: ~30 LoC

- » Unable to express directly in many SIMD libraries
  - › No support for 1024-bit vectors

Fun with µOp Cache



[1]: Afroozeh et al., 2023. The FastLanes Compression Layout, SIGMOD

Dynamic Dispatch

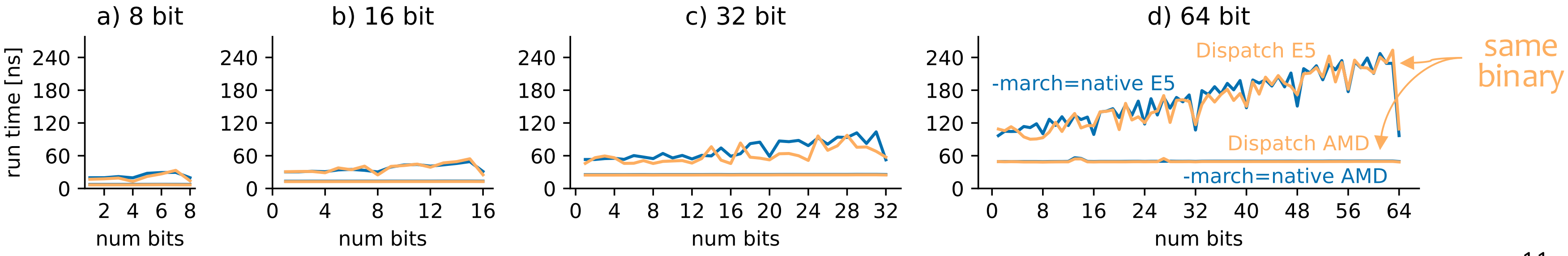
- Single binary
  - compile once, run everywhere
- via `__attribute__((target_clones("arch=x86-64-v3", "arch=x86-64-v4", …, "default"))`

~AVX2 ~AVX512 base x86

- Creates copy of function for each target
- Function resolved at runtime when library is loaded via GNU's .ifunc feature

Intel Xeon E5-2686 *AMD EPYC 7950X*

-march=native vs. dynamic dispatch on x86 (AVX2 and AVX512)

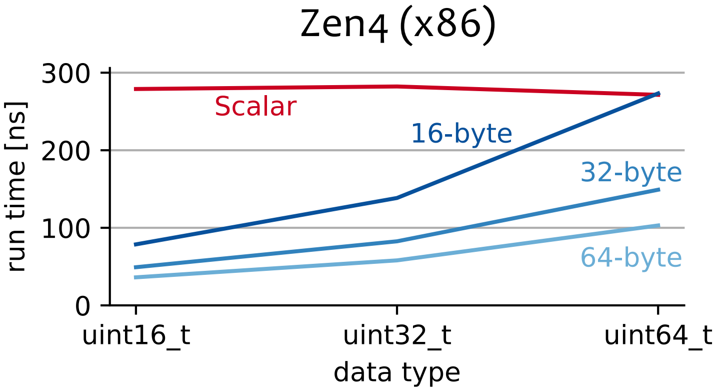


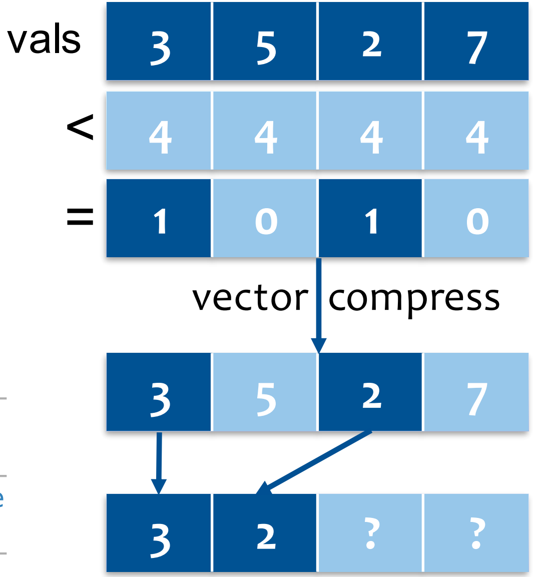
Filter with Compressing Store

- Specialized features for common SIMD patterns
- Added new @llvm.experimental.vector.compress intrinsic
  - `if (data[i] < filter_val) out[pos++] = data[i];`

- Translates to:
  - vpcompress in AVX512
  - compact in SVE

- Note: Work in progress
- Rest needs fallback
  - still slow :/





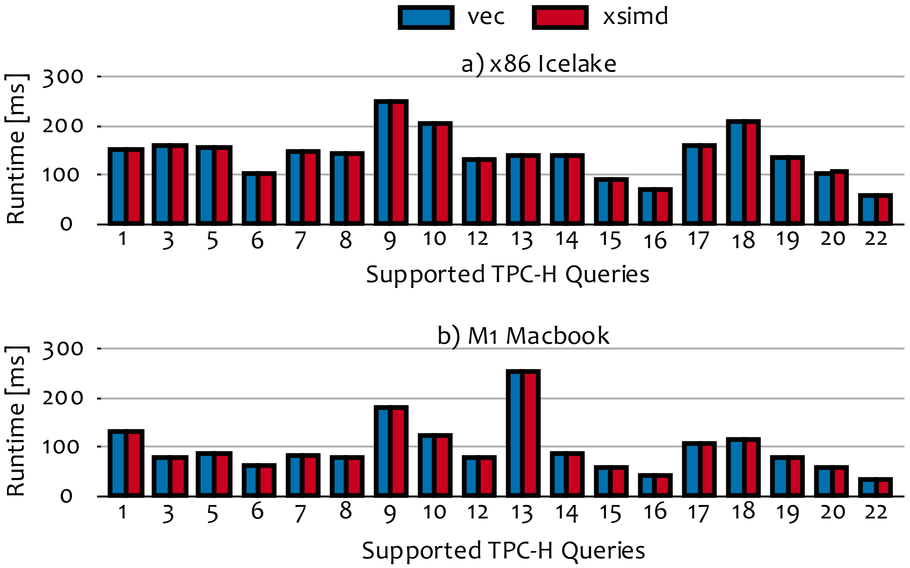
Compiler-Intrinsics in Velox

- Velox: Meta's new unified query engine

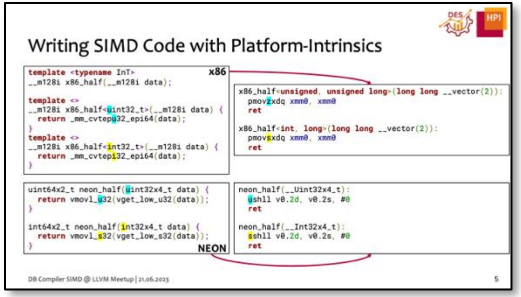
- Removed xSIMD dependency
- Use only compiler-intrinsics

- 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



Summary



Writing SIMD Code with Platform-Intrinsics

x86

```
template <typename InT>
__m128i x86_half(__m128i data);

template <>
__m128i x86_half<uint32_t>(__m128i data) {
  return _mm_cvtepu32_epi64(data);
}
template <>
__m128i x86_half<int32_t>(__m128i data) {
  return _mm_cvtepi32_epi64(data);
}
```

```
x86_half<unsigned, unsigned long>(long long __vector(2)):
  pmovzxdq xmm0, xmm0
  ret

x86_half<int, long>(long long __vector(2)):
  pmovsxdq xmm0, xmm0
  ret
```

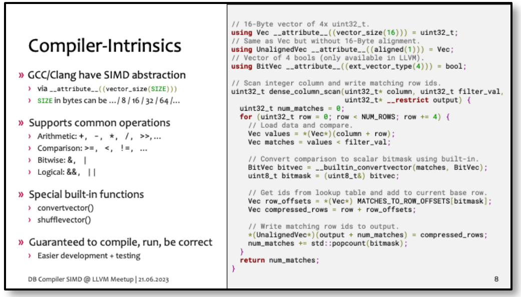
```
uint64x2_t neon_half(uint32x4_t data) {
  return vmovl_u32(vget_low_u32(data));
}

int64x2_t neon_half(int32x4_t data) {
  return vmovl_s32(vget_low_s32(data));
}
```

NEON

```
neon_half(__Uint32x4_t):
  ushll v0.2d, v0.2s, #0
  ret

neon_half(__Int32x4_t):
  sshll v0.2d, v0.2s, #0
  ret
```

DB Compiler SIMD @ LLVM Meetup | 21.06.2023 5

Writing platform-intrinsics code is hard and cumbersome



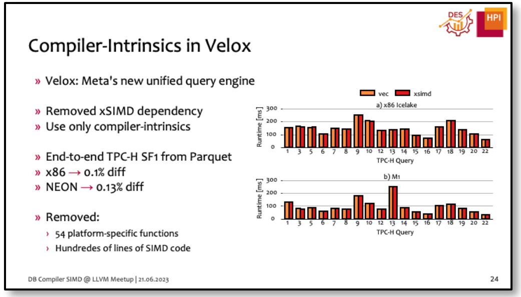
Compiler-Intrinsics

- 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, run, 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;
}
```

DB Compiler SIMD @ LLVM Meetup | 21.06.2023 8

Compiler-intrinsics are generic and cross-platform



Compiler-Intrinsics in Velox

- Velox: Meta's new unified query engine
- Removed xSIMD dependency
- Use only compiler-intrinsics
- End-to-end TPC-H SF1 from Parquet
- x86 → 0.1% diff
- NEON → 0.13% diff
- Removed:
  - 54 platform-specific functions
  - Hundredes of lines of SIMD code

DB Compiler SIMD @ LLVM Meetup | 21.06.2023 24

Compiler-intrinsics achieve the same performance

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