Writing Less Platform-Dependent SIMD Code in Databases

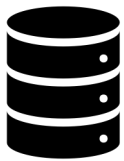
Lawrence Benson
lawrence.benson@hpi.de
Collaboration with Richard Ebeling and Tilmann Rabl

LLVM Meetup | 21.06.23

Agenda

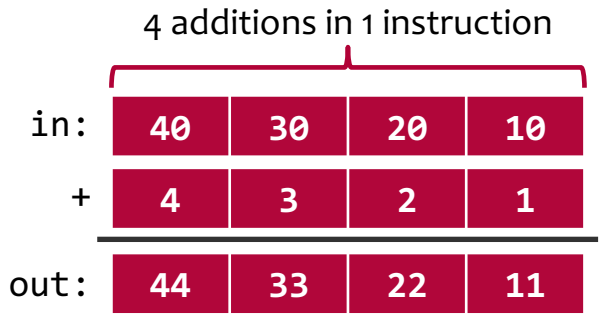
1. SIMD in Databases
2. Compiler-Intrinsics
3. NEON → Bitmask in LLVM
4. Benchmarks
5. Outlook

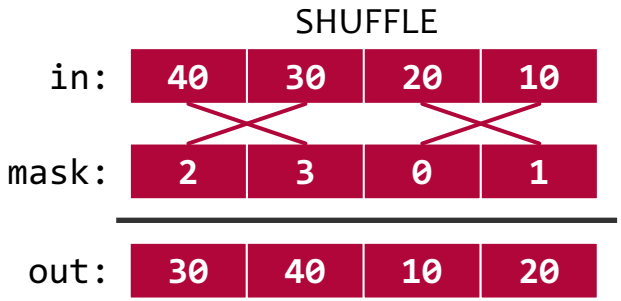
Disclaimer: I'm a database guy, not a compiler guy.



SIMD in a Nutshell

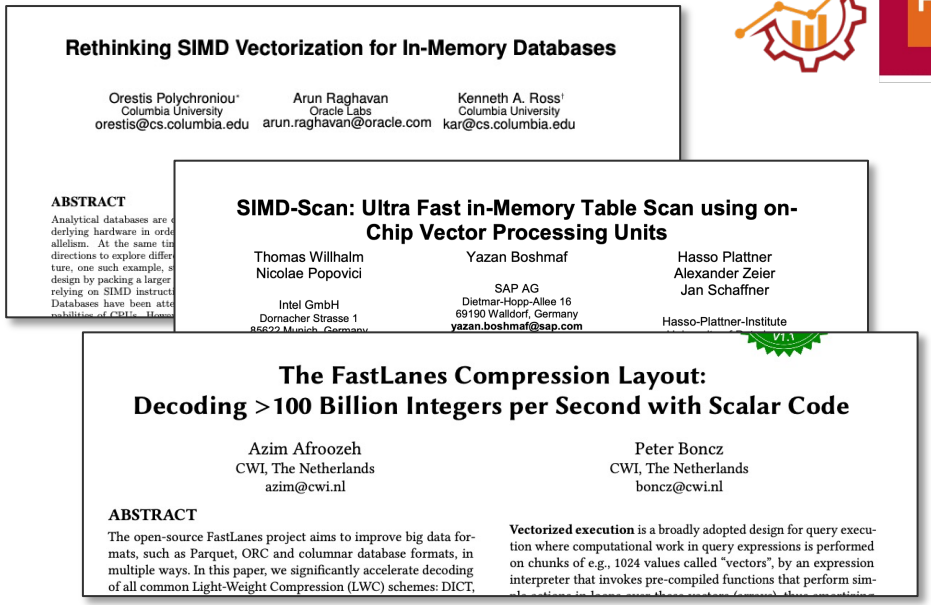
- Single Instruction Multiple Data (= SIMD)
- Most common instruction sets
  - x86: SSE, AVX, AVX2, AVX512 (> 6k instructions)
  - ARM: Neon (> 4k instructions), SVE
  - Power: Altivec
- Arithmetic, Logical, Shuffle, Shift, Load, Store, …
- Focus on x86 and Neon
  - x86: 128 – 512 Bit registers
  - Neon: 128 Bit registers
- Stored in Registers:
  - General: %eax (32-bit), %rdx (64-bit)
  - SIMD: %xmm (128-bit), %ymm (256-bit), %zmm (512-bit)
- Registers don't have types
  - One operation on 4x 32-bit, next on 16x 8-bit
  - Entries are called "lanes"





SIMD in Databases

- Used to speed up, e.g.,
  - Table scans
  - Hash tables
  - Sorting
- No number crunching
  - Mainly data movement
- Cloud moving towards ARM
  - How to support non-x86?
- SIMD code is …
  - … hard to develop
  - … hard to test
  - … hard to benchmark



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()
```

Don't have AVX512?
→ *Can't compile*

Don't have NEON?
→ *Can't compile*

Writing SIMD Code with Platform-Intrinsics

Extract lower two 32-bit values and extend to 64-bit

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
```

NEON

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

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

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

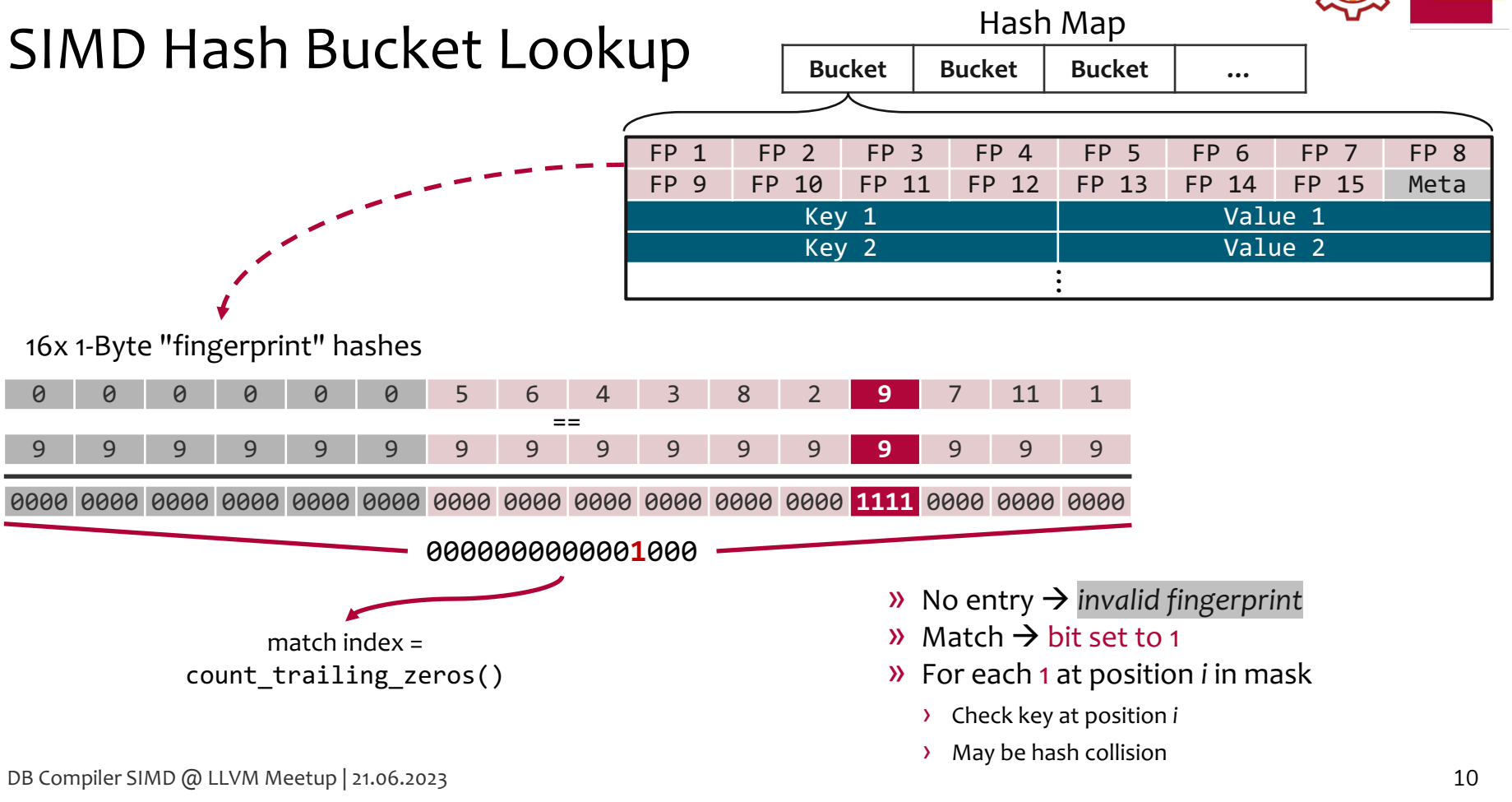
Compiler-Intrinsics

- 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()
- 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;
}
```

SIMD Bitmasks

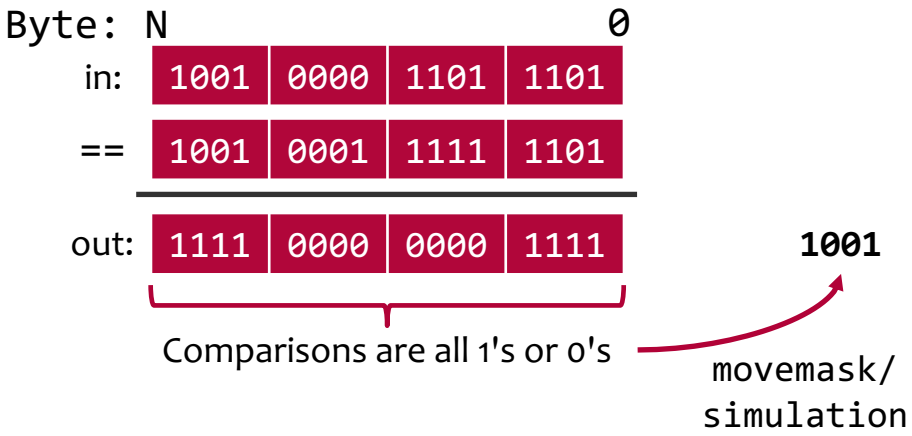
SIMD Hash Bucket Lookup



- No entry → *invalid fingerprint*
- Match → bit set to 1
- For each 1 at position *i* in mask
  - Check key at position *i*
  - May be hash collision

SIMD Compare to Bitmask

- Common operation: comparison to bitmask
  - Convert 128/256/512-Bit register to uint8/16/32/64
- x86 has dedicated instruction for this
- Neon needs multiple instructions



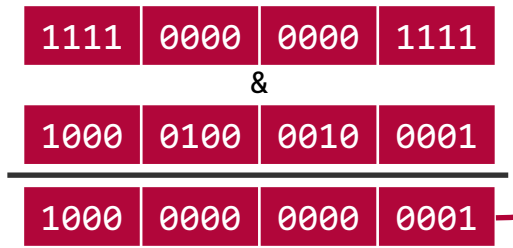
x86:

```
int _mm_movemask_epi8(__m128i a)
... and others
```

→ 16x: `get_most_significant_bit(a[i])`

NEON:

3 instructions: load, &, add



LLVM Patch: NEON to Bitmask

Patch: https://reviews.llvm.org/D145301 | Godbolt: https://godbolt.org/z/q4c5sY8aa

Compare two 4x 32-bit vectors and convert to bitmask

```
define i4 @compare_to_bitmask(<4 x i32> %vec1, <4 x i32> %vec2) {
  %cmp_result = icmp ne <4 x i32> %vec1, %vec2
  %bitmask = bitcast <4 x i1> %cmp_result to i4
  ret i4 %bitmask
}
```

LLVM <= 16

```
_compare_to_bitmask:
 sub      sp, sp, #16
 cmeq.4s  v0, v0, v1
 mvn.16b  v0, v0
 xtn.4h   v0, v0
 umov.h   w8, v0[1]
 umov.h   w9, v0[2]
 umov.h   w10, v0[0]
 umov.h   w11, v0[3]
 and      w8, w8, #0x1
 and      w9, w9, #0x1
 bfi      w10, w8, #1, #1
 bfi      w10, w9, #2, #1
 bfi      w10, w11, #3, #29
 and      w0, w10, #0xf
 add      sp, sp, #16
 ret
```

LLVM Patch: NEON to Bitmask

Patch: https://reviews.llvm.org/D145301

Compare two 4x 32-bit vectors and convert to bitmask

```
define i4 @compare_to_bitmask(<4 x i32> %vec1, <4 x i32> %vec2) {
  %cmp_result = icmp ne <4 x i32> %vec1, %vec2
  %bitmask = bitcast <4 x i1> %cmp_result to i4
  ret i4 %bitmask
}
```

- It's a journey :)
  - i.e., I had no idea where to fix this
  - Lots of debugging and stepping through code

- `AArch64TargetLowering` *(lower LLVM code to valid operators)*

```
...
t6: v4i1 = setcc t2, t4, setne:ch
...
Legalizing node: t7: i4 = bitcast t6
Promote integer result: t7: i4 = bitcast t6
```

→ `v4i1` is our comparison bit vector (but illegal type)

→ `i4` is not a legal type
→ must make `i4` legal (e.g., `i32`)

  - `::PromoteIntegerResult() > ::CustomLowerNode() > ::ReplaceNodeResults() > ::ReplaceBITCASTResults() > replaceBoolVectorBitcast() > ... add new code here`

LLVM Patch: NEON to Bitmask

Patch: https://reviews.llvm.org/D145301

Compare two 4x 32-bit vectors and convert to bitmask

```
define i4 @compare_to_bitmask(<4 x i32> %vec1, <4 x i32> %vec2) {
  %cmp_result = icmp ne <4 x i32> %vec1, %vec2
  %bitmask = bitcast <4 x i1> %cmp_result to i4
  ret i4 %bitmask
}
```

- Use NEON "trick" to create bitmask
  - Needs legal vectors
  - `v4i1` is not legal :/
- Try to get original vector type
  - e.g., `v4i32` in our example
  - Small node graph traversal

```
...
t6: v4i1 = setcc t2, t4, setne:ch
...
Legalizing node: t7: i4 = bitcast t6
Promote integer result: t7: i4 = bitcast t6
```

```
if (Op.getOpcode() == ISD::SETCC || Op.getOpcode() == ISD::TRUNCATE)
  return Op.getOperand(0).getValueType();  // t2: v4i32 ...
```

- Without original type → always 1 extra instruction to truncate to 8-Byte vector :(

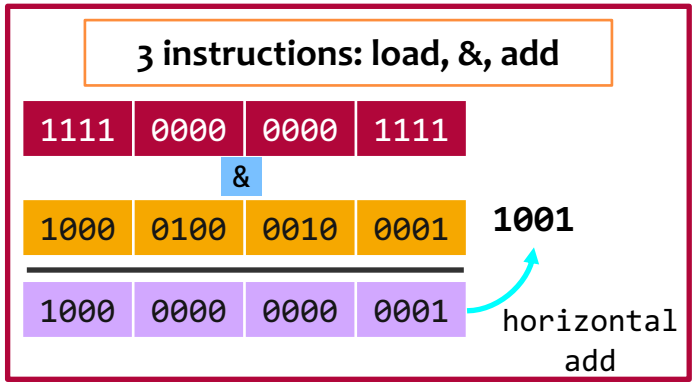
LLVM Patch: NEON to Bitmask

Patch: https://reviews.llvm.org/D145301

- Use NEON "trick" to create bitmask
  - Each lane must be all 1 or all 0
- Sign-extend vector
  - Fun fact: done via `(vec << 31) < 0`
- Finally do actual "trick" implementation

Compare two 4x 32-bit vectors and convert to bitmask

```
define i4 @compare_to_bitmask(<4 x i32> %vec1, <4 x i32> %vec2) {
  %cmp_result = icmp ne <4 x i32> %vec1, %vec2
  %bitmask = bitcast <4 x i1> %cmp_result to i4
  ret i4 %bitmask
}
```




```
unsigned MaxBitMask = 1u << (VecVT.getVectorNumElements() - 1);
for (unsigned MaskBit = 1; MaskBit <= MaxBitMask; MaskBit *= 2) {
  MaskConstants.push_back(DAG.getConstant(MaskBit, DL, MVT::i64));
}
SDValue Mask = DAG.getNode(ISD::BUILD_VECTOR, DL, VecVT, MaskConstants);
SDValue RepresentativeBits = DAG.getNode(ISD::AND, DL, VecVT, ComparisonResult, Mask);
return DAG.getNode(ISD::VECREDUCE_ADD, DL, ResultVT, RepresentativeBits);
```

LLVM Patch: NEON to Bitmask

Patch: https://reviews.llvm.org/D145301 | Godbolt: https://godbolt.org/z/q4c5sY8aa

Compare two 4x 32-bit vectors and convert to bitmask

```
define i4 @compare_to_bitmask(<4 x i32> %vec1, <4 x i32> %vec2) {
  %cmp_result = icmp ne <4 x i32> %vec1, %vec2
  %bitmask = bitcast <4 x i1> %cmp_result to i4
  ret i4 %bitmask
}
```

LLVM after patch

```
_compare_to_bitmask:
 adrp    x8, lCPI0_0@PAGE
 cmeq.4s v0, v0, v1
 ldr     q2, [x8,  lCPI0_0@PAGEOFF]
 bic.16b v0, v2, v0
 addv.4s s0, v0
 fmov    w0, s0
 ret
```

LLVM <= 16

```
_compare_to_bitmask:
 sub     sp, sp, #16
 cmeq.4s v0, v0, v1
 mvn.16b v0, v0
 xtn.4h  v0, v0
 umov.h  w8, v0[1]
 umov.h  w9, v0[2]
 umov.h  w10, v0[0]
 umov.h  w11, v0[3]
 and     w8, w8, #0x1
 and     w9, w9, #0x1
 bfi     w10, w8, #1, #1
 bfi     w10, w9, #2, #1
 bfi     w10, w11, #3, #29
 and     w0, w10, #0xf
 add     sp, sp, #16
 ret
```

Benchmarks

Benchmarks

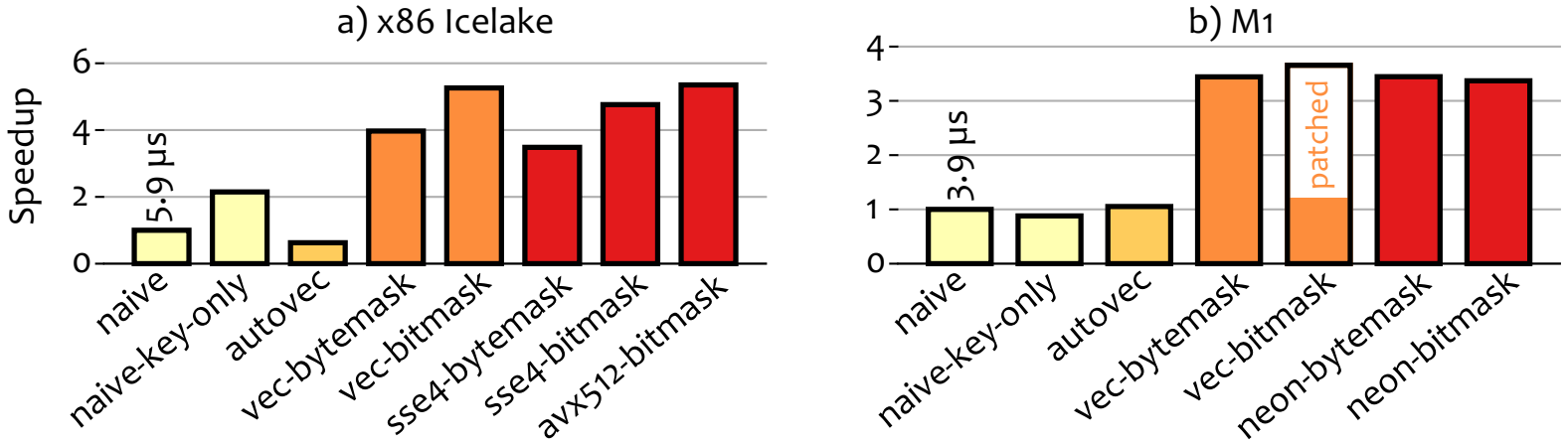
- M1: Apple Macbook Pro 14" M1 2021
- x86 Icelake: Intel Xeon Platinum 8352Y

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

- Single-threaded
- Average of 10 runs
- Show relative speedup over scalar version

Hash Bucket Lookup

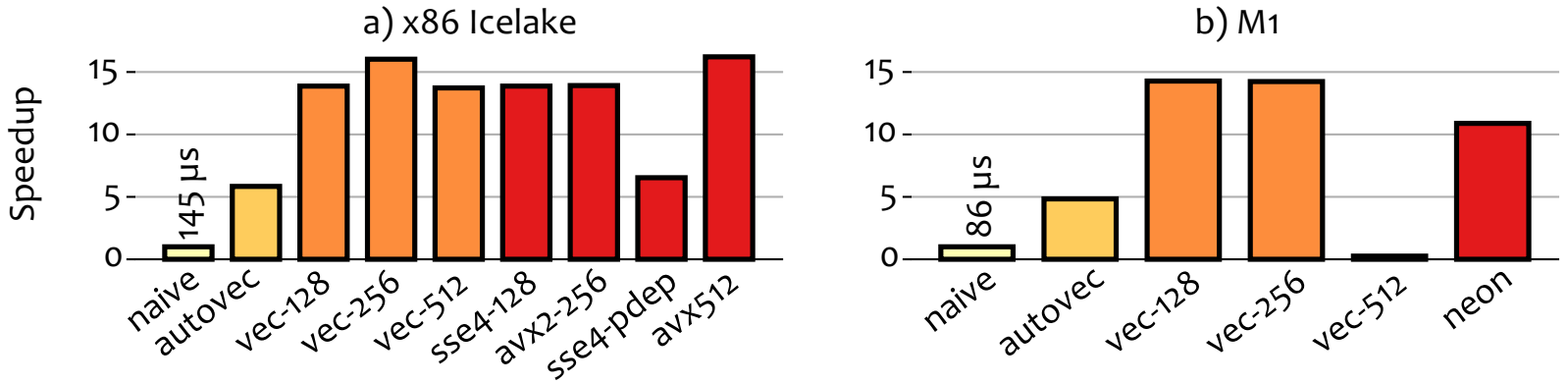
- Vectorized >> scalar lookup
- Compiler vec == hand-written SIMD
  - LLVM generates nearly identical code
  - M1: vec-bitmask does bitmask == 0 check before bitmask conversion



Bit-Packed Integer Decompression

- Based on VLDB '09 SIMD-Scan paper
- Unpack packed 9-Bit integers to 32 bits → shuffling + shifting

- Compiler vec >= hand-written SIMD
- Neon: vec-128 generates better code than translated x86 intrinsics
  - Implementation artifact on NEON

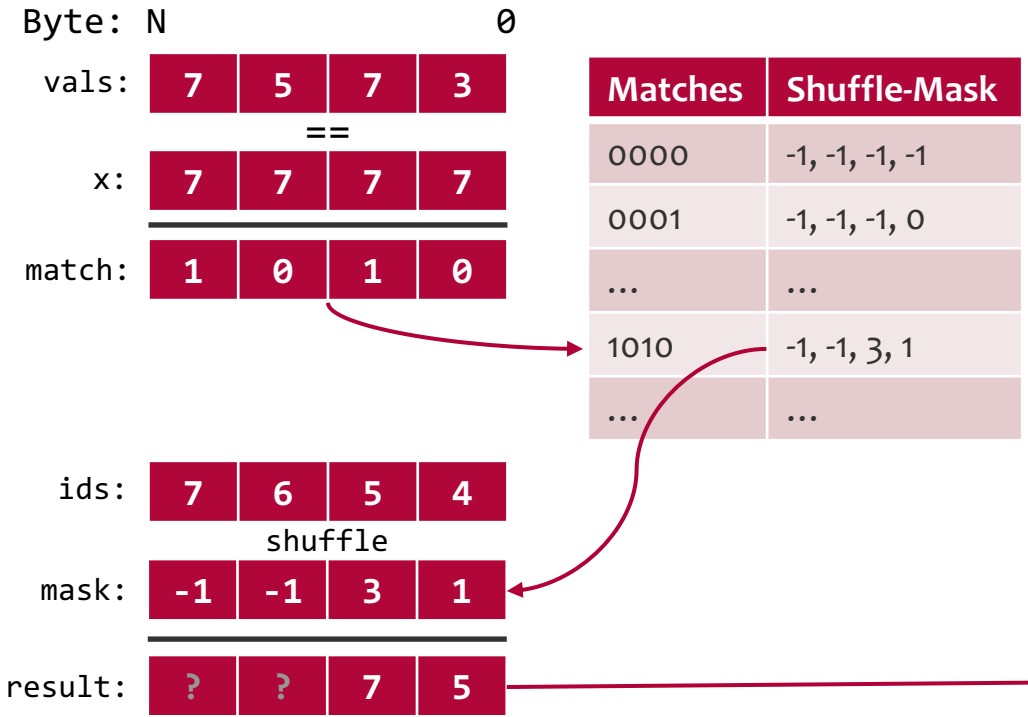


SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units
Thomas Willhalm et al. @ VLDB 2009

Missed Optimization with Platform-Intrinsics

- Compiler-intrinsics: `vec << X` and `vec >> Y`
  - Based on logic from original paper
  - Represented as `lshr`/`shl` in LLVM
- NEON: `vshlq_u32() (vector shift left)`
  - Internally: `__builtin_neon_vshlq_v()`
  - Represented as `llvm.aarch64.neon.ushl` in LLVM
- `lshr` is combined, `llvm.aarch64.neon.ushl` is not
  - `InstCombiner` covers many generic shift cases
  - Not implemented for `neon.ushl` in AArch64 lowering
- Advantage of staying in compiler's domain

Dictionary Table Scan



Matches	Shuffle-Mask
0000	-1, -1, -1, -1
0001	-1, -1, -1, 0
…	…
1010	-1, -1, 3, 1
…	…

Scalar

```
for row in dict_column:
  if row.val == x:
    output[num_matches++] = row.id
```

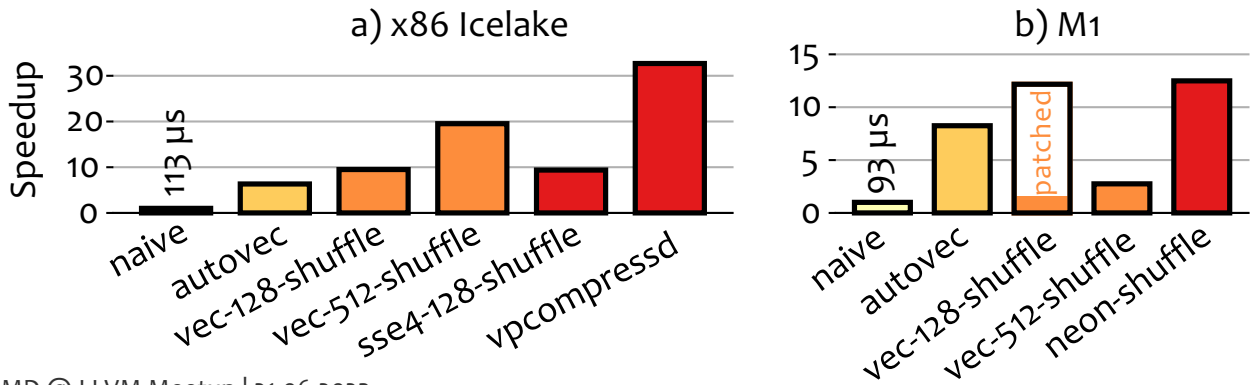
Vectorized

```
for rows in dict_column:
  matches = rows.vals == x
  row_ids = rows.ids

  // This is where the magic happens
  left_pack(row_ids, matches)

  memcpy(output + num_matches, row_ids)
  num_matches += popcount(matches)
```

Dictionary Table Scan

- AVX512 `compresstore`
  - Clang doesn't generate it for autovec
  - Can't be expressed with compiler vectors

- NEON vec-X: Shuffling doesn't use TBL instruction
  - With our LLVM patch → equal to NEON performance



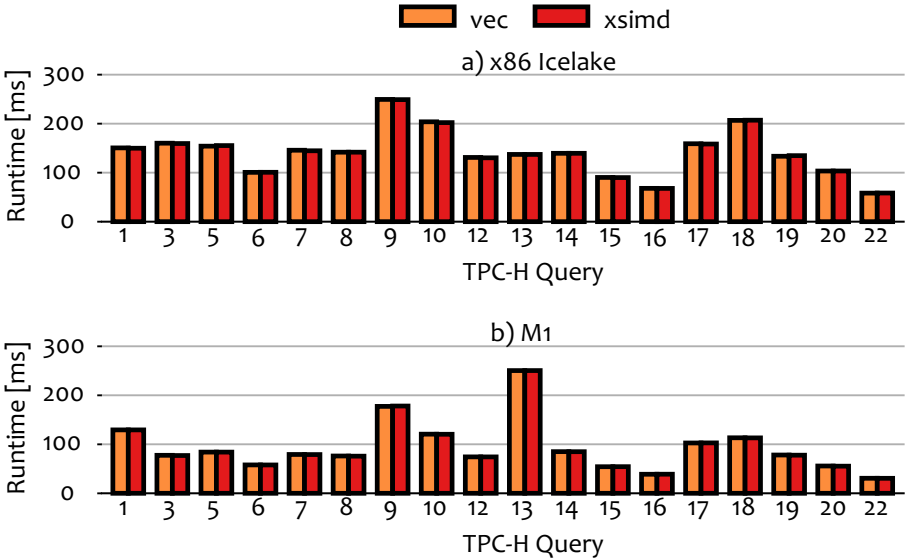
Compiler-Intrinsics in Velox

- Velox: Meta's new unified query engine

- Removed xSIMD dependency
- Use only compiler-intrinsics

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

- Removed:
  - 54 platform-specific functions
  - Hundredes of lines of SIMD code



Different Platforms, Different Compilers

- Also tested on
  - x86: Intel Cascadelake, Intel Skylake, and AMD Rome
  - NEON: Graviton 2, Graviton 3, Raspberry Pi
  - Results show same trends ✓
- Also tested with GCC
  - No bit-vector type → that part is slow ✗
  - Rest shows same trends ✓
  - Trageting multiple compilers at this level is hard
- Not possible with MSVC
  - Does not have `vector_size` attribute

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

Use Compiler-Intrinsics

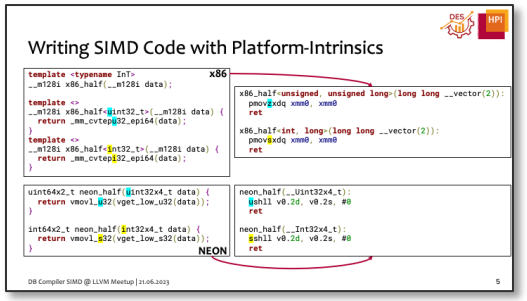
```
void filter(T* in, T filter_val, T* out, Op filter_op) {
  for (int i = 0; i < N; i += sizeof(Vec<T>) / sizeof(T)) {
    Vec<T> values = (Vec<T>&) in[i];
    // Filtering logic is platform-independent and generic.
    Vec<T> matches = filter_op(values, filter_val);
    uint64_t mask = compress_store(values, matches, out);
    out += std::popcount(mask);
  }
}
```

```
#ifdef __AVX512F__
// We can use compressstore.
template <typename T>
uint64_t compress_store(Vec<T> vals, Vec<T> matches, T* out) {
  uint64_t mask;
  if constexpr (sizeof(Vec<T>) == 16) { // and T is 32-bit
    mask = _mm_movepi32_mask((__m128i) matches);
    _mm_mask_compressstoreu_epi32(out, mask, (__m128i) vals);
  } else if constexpr (sizeof(Vec<T>) == 32) { // _mm256...
  } else if constexpr (sizeof(Vec<T>) == 64) { // _mm512...
  }
  return mask;
}
#else
// We can't use compressstore.
template <typename T>
uint64_t compress_store(Vec<T> vals, Vec<T> matches, T* out) {
  // Fallback shuffle + store similar to slide 22 ...
}
```

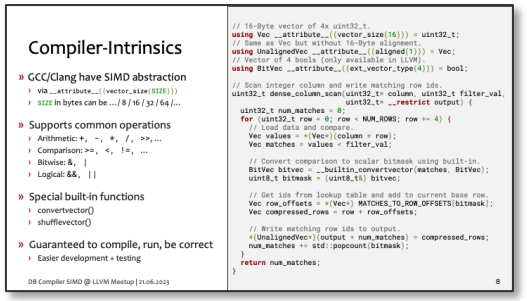
- Generic operator logic
- Platform-independent
- Type-indpendent

- Only n variants for specialized code

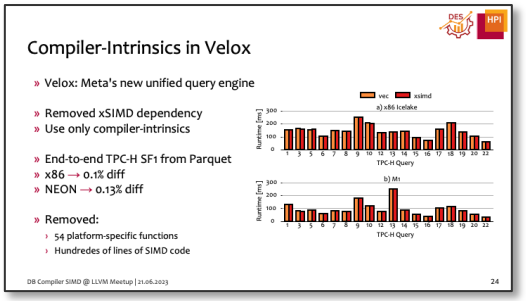
Summary



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