Hardware-Oblivious SIMD Parallelism for In-Memory Column-Stores

Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner
TU Dresden, Germany

Erich Focht
High Performance Computing Europe
NEC Deutschland GmbH, Germany

CIDR 2020, Amsterdam, Netherlands, 2020/01/14
Modern In-Memory Column-Store

Single-Instruction Multiple Data (SIMD)
- State-of-the-art parallelism concept
- Execute same instruction on a set of values (vector unit, vector register) → vectorization
- Increases single-thread performance

Intel SIMD Extensions

Google Scholar trend for keywords "SIMD + database + operators"

Common ground

Explicit Vectorization using SIMD Intrinsics

Performance with SIMD
Explicit SIMD Programming: Running Example

Materialized result:

Filter: 3 5 8 1 ≥ 4

SSE

_vec1 = _mm_load_si128(ptr_in);
_result = _mm_cmpgt_epi32(vec1, vec_const);
_mask = _mm_movemask_epi8(result);

switch (mask){
    case 0:    return;
    case 1:    *ptr_out = _mm_extract_epi32(p_vec,0);
                return;
    ...  
    case 12: p_vec=_mm_shuffle_epi8(p_vec,
                  _mm_set_epi8(7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8));
             _mm_storeu_si128(ptr_out, p_vec);
             return;
    ...  
    case 15: _mm_storeu_si128(ptr_out, p_vec);
             return;
}

Load

Mem

Vector: 3 5 8 1

3 5 8 1

≥

4 4 4 4

Mem

Compare

Vector: 3 5 8 1

3 5 8 1

≥

4 4 4 4

Mem

Compress

Store
Porting to AVX512

Load

\[
\text{__m128i vec1 = _mm_load_si128(ptr_in);}
\]

Compare

\[
\text{__m128i result = _mm_cmpgt_epi32(vec1, vec_const);}
\]

\[
\text{int mask = _mm_movemask_epi8(result);}
\]

Compress

\[
\text{switch (mask) \{} \\
\text{case 0: return;}
\text{case 1: *ptr_out = _mm_extract_epi32(p_vec, 0); return;}
\text{...}
\text{case 12: p_vec = _mm_shuffle_epi8(p_vec, _mm_set_epi8(7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8));}
\text{_mm_storeu_si128(ptr_out, p_vec); return;}
\text{...}
\text{case 15: _mm_storeu_si128(ptr_out, p_vec); return;}
\text{\}}
\]

Store

\[
\text{\textbf{Major Challenge}}
\]

An Abstraction to enable \textbf{Portability} and \textbf{Extensibility}

But keep \textbf{explicit Vectorization}

\[
\text{Materialized result:}
\]

\[
\begin{array}{c}
\text{5} \\
\text{8} \\
\text{...}
\end{array}
\]

AVX512

\[
\text{__m512i vec1 = _mm512_load_si512(ptr_in);}
\]

\[
\text{int mask = _mm512_cmpgt_epi32_mask(vec1, vec_const);} \\
\text{_mm512_mask_compressstoreu_epi32(ptr_out, mask, vec1);}
\]
<table>
<thead>
<tr>
<th>#Operations</th>
<th>Vector Size</th>
<th>Element Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE, AVX, AVX2, different subsets of AVX512</td>
<td>Xeon Gold (Skylake)</td>
<td>Intel</td>
</tr>
<tr>
<td></td>
<td>Xeon Gold &amp; Phi</td>
<td>4750</td>
</tr>
<tr>
<td></td>
<td>Xeon Phi (Knights Landing)</td>
<td>3500</td>
</tr>
<tr>
<td></td>
<td>ARM Chips with NEON, NEON v.2</td>
<td>4500</td>
</tr>
<tr>
<td></td>
<td>Neon &amp; SVE</td>
<td>3500</td>
</tr>
<tr>
<td></td>
<td>NEC Aurora TSUBASA Vector Engine</td>
<td>3250</td>
</tr>
<tr>
<td></td>
<td>ARM Chips with SVE</td>
<td>2500</td>
</tr>
</tbody>
</table>

Porting across architectures is not trivial!  
- Architecture independent API
Template Vector Library (TVL)

Classes extracted from column store operators

Hardware-Oblivious API

Hardware-Conscious Implementation (Plug-in)

Compress
Store

L/S
Class
Arithmetic
Class
Boolean
Logic
Class
Create
Class
Manipulate
Class
Comparison
Class
Extract
Class

Processing Style

- Instruction set
  - AVX2/Neon/NEC
- Vector size
  - 64/128/512 bit
- Element type & size
  - 64/128/256 bit int

Template Specialization with force inlined functions

Scalar, 64 bit values, 64 bit registers

*ptr_out=p_vec;

switch (mask){
  case 0: return;
  ...
  case 15: _mm_storeu_si128(ptr_out, p_vec);
  return;
}

Intel SSE
SSE, 32 bit values, 128 bit registers

Intel AVX2
...

Intel AVX512
AVX512, 64 bit values, 128/512 bit registers

_ARM Neon
...

NEC
...

_mm_mask_compressstoreu_epi64(ptr_out, mask, vec1);

_mm512_mask_compressstoreu_epi64(ptr_out, mask, vec1);
**TVL in Practice**

---

### Materialized result:

<table>
<thead>
<tr>
<th>Filter:</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>≥</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Derived Vector Properties

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Vector Size</th>
<th>Element Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>using processingStyle = avx512&lt;v512&lt;int32_t&gt;&gt;;</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Primitives**

```
__m128i vec1 = _mm_load_si128(ptr_in);

__m128i result = _mm_cmpgt_epi32(vec1, vec_const);
int mask = _mm_movemask_epi8(result);
```

```
switch (mask){
    case 0:  return;
    case 1:  *ptr_out = _mm_extract_epi32(p_vec,0);
             return;
    ...
    case 12: p_vec=_mm_shuffle_epi8(p_vec,
                       _mm_set_epi8(7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8));
             _mm_storeu_si128(ptr_out, p_vec);
             return;
    ...
    case 15: _mm_storeu_si128(ptr_out, p_vec);
             return;
}
```

**TVL Data Types**

```
vector_t vec1 = load <processingStyle, iov::ALIGNED, vector_size_bit>(ptr_in);
mask_t mask = greater <processingStyle, vector_base_t_granularity>
              (vec1, vec_const);
```

**Data alignment required for primitives in L/S class**

```
compressstore <processingStyle, iov::UNALIGNED, vector_size_bit>
               (ptr_out, vec1, mask);
```

**SSE**

**TVL**
End-to-End Evaluation

In-memory column store processing engine for analytical workloads

One code base for all operators

Backends for

- Intel SSE, AVX2, AVX512
- ARM Neon
- NEC SX-Aurora Tsubasa (partially)

D Habich, P Damme, A Ungethüm, J Pietrzyk, A Krause, J Hildebrandt, W Lehner
"MorphStore-In-Memory Query Processing based on Morphing Compressed Intermediates LIVE."
SSB runs on 5 different Instruction Sets and 4 different register sizes

Only ONE codebase for all Benchmarks

SIMD is beneficial in most cases
Microbenchmarks for Runtime Overhead

Test System: Intel Xeon Gold 5120
Comparison between TVL and hand-vectorized Operator (AVX2)
Virtually no overhead caused by TVL-library
Microbenchmarks for Runtime Overhead

Overhead of TVL over auto-vectorizer by NEC

TVL uses LLVM-VE backend providing intrinsics support

2 Cases:
- Filter only
- Filter followed by aggregation

Virtually no overhead

Test System: NEC SX-Aurora Tsubasa
Microbenchmarks for Performance

Same instruction set behaves differently on different systems
Longer vectors or newer instructions not always best choice
Future Work

Same instruction set behaves differently on different systems
Longer vectors or newer instructions not always best choice

- Exciting future work: Choice of Vector Extension as Optimization Knob
Hardware-Oblivious SIMD Parallelism for In-Memory Column-Stores

Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner
TU Dresden, Germany

Erich Focht
High Performance Computing Europe
NEC Deutschland GmbH, Germany

CIDR 2020, Amsterdam, Netherlands, 2020/01/14