Make Larger Vector Register Sizes New Challenges?

Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms

Dirk Habich, Patrick Damme, Annett Ungethüm, Wolfgang Lehner

DBTest Workshop, 2018-06-15
Vectorization using SIMD

**Single Instruction Multiple Data (SIMD)**
- same instruction on multiple data elements simultaneously

- Heavily used in Database Systems
  - to increase (single-thread) performance

---

**Rethinking SIMD Vectorization for In-Memory Databases**

**Boosting Data Filtering on Columnar Encoding with SIMD**

**Make the Most out of Your SIMD Investments: Counter Control Flow Divergence in Compiled Query Pipelines**

**Authors:**
- Hao Jiang
  - The University of Chicago
  - hajiang@cs.ucla.edu
- Arun Raghavan
  - Oracle Labs
- Kenneth A. Ross
  - Columbia University
- Harald Lang
  - Technical University of Munich
  - Munich, Germany
  - harald.lang@in.tum.de
- Andreas Kipf
  - Technical University of Munich
  - Munich, Germany
  - andreas.kipf@in.tum.de
- Peter Boncz
  - Centrum Wiskunde & Informatica
  - Amsterdam, The Netherlands
  - boncz@cwii.nl
- Thomas Neumann
  - Technical University of Munich
  - Munich, Germany
  - thomas.neumann@in.tum.de
- Linnea Passing
  - Technical University of Munich
  - Munich, Germany
  - linnea.passing@in.tum.de
- Alfons Kemper
  - Technical University of Munich
  - Munich, Germany
  - alfons.kemper@in.tum.de
Trend of Increasing Vector Register Sizes

**Single Instruction Multiple Data (SIMD)**
- same instruction on multiple data elements simultaneously

**Heavily used in Database Systems**
- to increase (single-thread) performance

**Rethinking SIMD Vectorization for In-Memory Databases**

**INTEL SIMD Development**
- SSE (128 bit) → 4
- AVX(2) (256 bit) → 8
- AVX-512 (512 bit) → 16
Data Compression

**Technique** = abstract idea of how compression works

- **RLE** (Run Length Encoding)
  - Replace run by value & length

- **DELTA** (Differential Coding)
  - Replace data elem. by difference to predecessor

- **FOR** (Frame-of-Reference)
  - Replace data elem. by difference to reference value

- **DICT** (Dictionary Coding)
  - Replace data elem. by 0-based key in dictionary

- **NS** (Null Suppression)
  - Eliminate leading zeroes in binary representation

Logical natural numbers

- **preprocessing**

Physical bits and bytes

- **actual compression**

- Level data role
Data Compression

**Technique** = abstract idea of how compression works

- **RLE**
  - Run Length Encoding
  - Replace run by value & length

- **DELTA**
  - Differential Coding
  - Replace data elem. by difference to predecessor

- **FOR**
  - Frame-of-Reference
  - Replace data elem. by difference to reference value

- **DICT**
  - Dictionary Coding
  - Replace data elem. by 0-based key in dictionary

- **NS**
  - Null Suppression
  - Eliminate leading zeroes in binary representation

**Algorithm** = concrete combination of one or more of these techniques

Two algorithms for the same technique might differ in, e.g.
- their **data layout**
- their use of **vectorization** using SIMD instruction set extensions
Vectorization and Compression

SSE (128 bit)

Literature on lightweight compression mainly focuses on SSE
Vectorization and Compression

- **SSE (128 bit)**
  - Literature on lightweight compression mainly focuses on SSE

- **AVX(2) (256 bit)**
  - Increasing vector sizes promise performance improvements (hopefully speedup of 2 respectively 4 compared to 128-bit)

- **AVX-512 (512 bit)**
Vectorization and Compression

- **SSE (128 bit)**
- **AVX(2) (256 bit)**
- **AVX-512 (512 bit)**

**Literature on lightweight compression mainly focuses on SSE**

**Increasing vector sizes promise performance improvements (hopefully speedup of 2 respectively 4 compared to 128-bit)**

**How to employ recent SIMD extensions for lightweight data compression?**
Vectorization and Compression

Literature on lightweight compression mainly focuses on SSE. Increasing vector sizes promise performance improvements (hopefully speedup of 2 respectively 4 compared to 128-bit).

How to employ recent SIMD extensions for lightweight data compression?

- Substitute SSE instructions for their AVX(2) and AVX-512 counterparts
- Slightly adapt memory layout where necessary
- Easy to do (if possible)
SIMD-BP128* – Basic Idea

Null Suppression Algorithm

- very efficient from a performance as well as compression ratio perspective

Subdivide data into blocks of 128 data elements each,

→ Determine bit width for largest data element per block

Pack all data elements in the block using that bit width

SIMD-BP128 – Compression

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)
SIMD-BP128 – Compression

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)
SIMD-BP128 – Compression

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)
SIMD-BP128 – Compression

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)

Processing

_mm_load_si128()
SIMD-BP128 – Compression

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)

Processing

_mm_load_si128()  

_mm_slli_epi32()
SIMD-BP128 – Compression

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)

Processing

_mm_load_si128()
_mm_slli_epi32()
_mm_or_si128()
SIMD-BP128 – Compression

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)

Processing:

- \_mm\_load\_si128() (loads 128-bit data)
- \_mm\_slli\_epi32() (shifts left by a fixed number of bits)
- \_mm\_or\_si128() (logical OR of two 128-bit values)
# SIMD-BP128 – Compression

## Processing

<table>
<thead>
<tr>
<th>Uncompressed input (32 bits/int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compressed output (10 bits/int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_load_si128()</td>
</tr>
<tr>
<td>_mm_slli_epi32()</td>
</tr>
<tr>
<td>_mm_or_si128()</td>
</tr>
</tbody>
</table>

Processed:

```
    7 3 6 2 5 1 4 0

    B  A  3  8

    7 6 5 4

    7 3 6 2 5 1 4 0
```
SIMD-BP128 – Compression

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)

Processing

_mm_load_si128()  
_mm_slli_epi32()  
_mm_or_si128()
SIMD-BP128 – Compression

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)

 Processing

_mm_load_si128()

_mm_load_si128()

_mm_load_si128()

_mm_slli_epi32()

_mm_or_si128()

_mm_store_si128()
**SIMD-BP128 – Compression**

Uncompressed input (32 bits/int)

Compressed output (10 bits/int)

Trivially Portable

All employed **SSE**-intrinsics have equivalents in **AVX2** and **AVX-512**, e.g.

- `_mm_load_si128()`
- `_mm256_load_si256()`
- `_mm512_load_si512()`
- `_mm_slli_epi32()`
- `_mm256_slli_epi32()`
- `_mm_or_si128()`
- `_mm256_or_si256()`
- `_mm512_or_si512()`
**SIMD-BP128** – Idea

1. Subdivide data into blocks of 128 data elements each,
2. Determine bit width for largest data element per block
3. Pack all data elements in the block using that bit width

**SIMD-BP128** – Idea

- Subdivide data into blocks of 128 data elements each,
- Determine bit width for largest data element per block
- Pack all data elements in the block using that bit width

Ported versions need blocks of
- 256 elements (AVX2)
- 512 elements (AVX-512)

Evaluation Setup

**Algorithms**
- Implemented in C/C++
  - Some by the original authors
  - Some implemented by us
- Compiled using g++-7.0.1 –O3

**Synthetic Data**
- Allows to vary the data properties carefully

**Evaluation System**
- Intel Xeon Phi 7250
  - 1.4 GHz
  - L1-cache: 32 KB (data)
  - L2-cache: 1 MB
- 6x32 GB DDR4 @ 2400 MHz

**Measurements**
- All experiments completely in-memory
- disk not touched during time measurements
- Compression ratio reported in bits/int
  - Lower is better
  - Uncompressed data has 32 bits/int
- Speeds reported in million integers per second (mis)
  - Higher is better
  - Only single-thread performance
SIMD-BP128 – Evaluation

Data

- 100 M 32-bit integers
- Unsorted
- 4-bit values vs. 28-bit outliers
- We vary the outlier ratio
**SIMD-BP128 – Evaluation**

**Data**
- 100 M 32-bit integers
- Unsorted
- 4-bit values vs. 28-bit outliers
- We vary the outlier ratio

**Insights**
- Increasing vector size → increasing vulnerability to outliers

![Graphs showing compression rate and relative compression rate](image)
**SIMD-BP 128 – Evaluation**

**Data**
- 100 M 32-bit integers
- Unsorted
- 4-bit values vs. 28-bit outliers
- We vary the outlier ratio

**Insights**
- Increasing vector size → increasing vulnerability to outliers
- Suboptimal speed ups
  - Far away from speedups of 2 or 4

![Graphs showing compression rate, speed, relative rate, and speedup comparison for few and many outliers.](image)
Evaluation Second Example
Run-Length Encoding

**Basic Idea**
- View subsequent occurrences of the same value as a run
- Each run representable by its value and length → just two integers

**RLE-SIMD**
- Uses SIMD instructions to parallelize comparisons
- Proposed for 128-bit vectorization

**Porting to Larger Vector Sizes**
- easily portable using a straightforward approach
Evaluation using Different Vector Sizes

**Compression Speed**
- Measured in million integers per second (mis)

**Speedup**
- Compared to baseline of 128-bit

---

![Graphs showing Compression Speed and Speedup for RLE512, RLE256, and RLE128 with different avg. run length.](image)
Evaluation using Different Vector Sizes

Conflict Detection-based Run-Length Encoding — AVX-512 CD Instruction Set in Action

Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Dirk Habich, Wolfgang Lehner

Database Systems Group, Technische Universität Dresden
Dresden, Germany
{firstname.lastname}@tu-dresden.de

Compression Speed
▪ Measured in million integers per second (mis)

Speedup
▪ Compared to baseline of 128-bit

non-well-performing area

well-performing area
Conclusion

Increased vector sizes promise performance improvements.

Literature on lightweight compression mainly focuses on SSE.

**Advantage**
- Straightforward porting usually feasible

**Disadvantages**
- Desired speedups usually not achieved
- Negative effect on compression ratio

CONCLUSION

NOT THE RIGHT WAY