FUSED TABLE SCANS: COMBINING AVX-512 AND JIT

Markus Dreseler, Jan Kossmann, Johannes Frohnhofen, Matthias Uflacker, Hasso Plattner

Joint Workshop of HardBD & Active @ ICDE
Paris, 16th of April 2018
FUSED TABLE SCANS - COMBINING AVX-512 AND JIT

- **AVX-512**: Intel’s newest instruction set for SIMD operations
- **Just-In-Time** compilation: Creating binary code at program runtime

- Efficient (multi-predicate) sequential scans are a necessity for relational database systems
- Secondary indexes can speed up such operations
  - Drawbacks: memory consumption and maintenance cost
- **Contribution**: Combine the above techniques to accelerate table scans
Optimizations of sequential scans can be grouped into two categories:

- Block-at-a-time: Evaluate multiple values (SIMD) of a column at a time
  - Store results in position list
  - Materialization between operators

- Data-centric compilation: Generate (JIT) a tight, optimized loop to process one tuple at a time
  - No utilization of SIMD until now
  - Suboptimal interplay with some hardware optimizations
WHY SHOULD WE COMBINE DATA-CENTRIC OPERATION & SIMD?

Assumptions: Data resides in-memory in column-major format with fixed size values

SELECT COUNT(*) FROM tbl WHERE a = 5 AND b = 2 could look similar to:

```c
int total_results = 0;
for (pos_t i = 0; i < col_a.size(); ++i) {
    if (col_a[i] == 5 && col_b[i] == 2) {
        ++total_results;
    }
}
```
WHY SHOULD WE COMBINE DATA-CENTRIC OPERATION & SIMD?

```c
int total_results = 0;
for (pos_t i = 0; i < col_a.size(); ++i) {
    if (col_a[i] == 5 && col_b[i] == 2) {
        ++total_results;
    }
}
```
WHY SHOULD WE COMBINE DATA-CENTRIC OPERATION & SIMD?

- Experiment: Does a single value at a time evaluation fully utilize the available memory bandwidth?
- Reduce the number of cpu operations, but still load all data from memory
IMPLEMENTATION

- Utilizing the new instruction set AVX-512
  - Wider (doubled) register sizes
  - New instructions offer efficiency advantages
    - We built equivalent functions using AVX2 (up to 32 lines)
- Basic idea
  - Keep data in the AVX-registers during whole scan
uint32_t

128 byte

Column A:

2 5 4 5 6 1 5 7 6 8 5 3 5 9 9 5

_mm_loadu_si128

__m128i

_mm_cmpeq_epi32_mask

_first search value

indexes of current block

_mm_mask_compress_epi32

_position list

_matching positions in column a

_second iteration

_final result: row 1 (the second entry) matches both conditions

_mm_i32gather_epi32

_mm_cmpeq_epi32_mask

_mm_mask_compress_epi32

_mm_mask_cmpeq_epi32_mask

_mm_mask_compress_epi32

_mm_permutex2var_epi32

_mm_permutex2var_epi32

_mm_mask_compress_epi32

uint32_t

128 byte

Column A:

2 5 4 5 6 1 5 7 6 8 5 3 5 9 9 5

_mm_loadu_si128

__m128i

_mm_cmpeq_epi32_mask

_first search value

indexes of current block

_mm_mask_compress_epi32

_position list

_matching positions in column a

_second iteration

_final result: row 1 (the second entry) matches both conditions
uint32_t 128 byte

Column A:

_ mm_loadu_si128

_mm_cmpeq_epi32_mask

_mm_mask_compress_epi32

_mm_mask_cmpeq_epi32_mask

_mm_permutex2var_epi32

_mm_permutex2var_epi32

_mm_permutex2var_epi32

_mm_mask_compress_epi32

first search value

indexes of current block

position list

matching positions in column a

final result: row 1 (the second entry) matches both conditions

Column B:

_mm_i32gather_epi32

_mm_cmpeq_epi32_mask

_mm_mask_compress_epi32

_mm_mask_cmpeq_epi32_mask

_mm_permutex2var_epi32

_mm_permutex2var_epi32

_mm_permutex2var_epi32

_mm_mask_compress_epi32

matching positions in column a
Column A:

1. Load from memory using `mm_loadu_si128`.
2. Compare values using `mm_cmpeq_epi32_mask`.
3. Apply compression mask using `mm_mask_compress_epi32`.
4. Permute values using `mm_permutex2var_epi32`.
5. Match positions in column A using `mm_i32gather_epi32`.

First search value
Indexes of current block
Position list
Matching positions in column A

Final result: row 1 (the second entry) matches both conditions

Column B:

1. Compare values using `mm_cmpeq_epi32_mask`.
2. Apply compression mask using `mm_mask_compress_epi32`.
3. Match positions in column A using `mm_i32gather_epi32`.

Matching positions in column A
Final result: row 1 (the second entry) matches both conditions
IMPLEMENTATION – ACHIEVEMENTS

- Fully utilize the CPU’s computation power
  - More comparisons/cycle by using SIMD instructions
- Avoid useless prefetches, only load necessary data of the second column
  - Increased memory bus efficiency
- Fewer and cheaper branch mispredictions
  - Reduced number of conditions in code
- Reduced memory transfers
  - Intermediary results are kept in AVX registers and are not materialized
FUSED TABLE SCANS

JIT - RUNTIME CODE GENERATION

Regular query plan

Plan with Fused Table Scan
JIT – RUNTIME CODE GENERATION

- **Problem:** Some parameters are only known at runtime
  - Size of scanned values
  - Exact data types
    - Signed & unsigned 1, 2, 4, or 8 byte int plus float & double
  - Type of comparison operators: ! =, ==, <, >, <=, >=

- Larger number of possible code paths
The query optimizer identifies fusible operator chains.
Parameters are determined by the translator during runtime.
Result: Specialized, monolithic function
Cached for efficiency.

https://github.com/hyrise/hyrise/
EVALUATION

- On current Skylake system
  - Intel Xeon Platinum @ 2.5 - 3.8 GHz with 2TB of PC4-2666 main memory
- Evaluated dimensions during experiments
  - Table Size
  - Selectivity
  - Implementations / Instruction sets
    - SISD, AVX2, and AVX-512, automatic compiler vectorization
    - AVX-Register width: 128, 256, and 512 Bit
  - Number of Predicates

FUSED TABLE SCANS
EVALUATION - PERFORMANCE RELATIVE TO SISD IMPLEMENTATION

![Graph showing performance relative to SISD implementation for different table sizes and selectivity percentages. Each bar represents a different table size in rows, and the performance is measured relative to a base SISD scan. The selectivity legend includes 50.0%, 10.0%, 1.0%, 0.1%, and 0.0001% selectivity levels.](image-url)
EVALUATION – INSTRUCTION SETS & REGISTER WIDTH

Table with 32M rows

- SISD (no vec)
- SISD (auto vec)
- AVX2 Fused (128)
- AVX-512 Fused (128)
- AVX-512 Fused (256)
- AVX-512 Fused (512)
EVALUATION - NUMBER OF PREDICATES

Table with 32M rows

- SISD (auto vec)
- AVX2 Fused (128)
- AVX-512 Fused (512)

Median Runtime (ms)

Number of Predicates

Table with 32M rows
SUMMARY & CONCLUSION

- Branch mispredictions and useless prefetches are a huge cost factor in multi-predicate scans
- Doubling the register size does not (yet) double the performance
- Bringing together AVX-512 with Just-In-Time compilation
  - Use new AVX-512 instructions to efficiently load and remove tuples from AVX-registers without leaving SIMD mode
  - Performance was at least doubled in 80% of test cases
- Future Work: Impact of other encoding methods