Lecture #06: Vectorized Query Execution

15-721 Advanced Database Systems (Spring 2024)
https://15721.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Prof. Andy Pavlo

1 Introduction

Vectorization is the process of converting an algorithm's scalar implementation that processes a single pair of operands at a time to a vector implementation that performs the same operation on multiple pairs of operands at once. It is often also called **Data Parallelization**.

To take an extreme case, we can parallelize our algorithm over 32 cores, each with a 4-wide SIMD register. Then, we are looking at a theoretical speedup of 128x. Although such a drastic speedup is not feasible in real-world scenarios, vectorization is still essential for most OLAP systems. The reasons will be talked about more in the following sections as follows:

2. Background
3. Implementation Approaches
4. Vectorization Fundamentals
5. Vectorized DBMS Algorithms

2 Background

This section will cover the basics of how SIMD works, the operations supported by current architectures, and the techniques to utilize them in DBMS.

2.1 SIMD

SIMD (Single Instruction, Multiple Data) is A class of CPU instructions that allow the processor to perform the same operation on multiple data points simultaneously. All major ISAs have micro-architecture support for SIMD operations. The fundamental idea of SIMD vector processing is to keep the data in the SIMD registers for as long as possible, perform efficient operations on them, and only flush them out when necessary.

The following are several common SIMD Instructions. For more details on the differences between different architectures refer presentation by James Reinders:

- **Data Movement**: Moving data in and out of SIMD registers, to memory, other registers, and possibly even bypassing the cache.
- **Basic Operations**: Arithmetic, Logical and Comparison operations

We will talk more about AVX-512 and the new instructions it introduced in the following sections.

2.2 Vectorization Direction

Using these SIMD registers and operations, the DBMS can perform vectorized operations in two broad ways:

- **Horizontal**: Operate on all elements together in a single vector. Useful for aggregation style queries like Sum. It is not commonly used and is only supported in AVX2 and up.
- **Vertical**: Perform operation in an element-wise manner on elements of each vector. It is helpful for a variety of queries like comparison or arithmetic operations and is widely used.

3 Implementation Approaches

The following are the broad categories of approaches to convert scalar operations to their vectorized versions:

- **Automatic Vectorization**: The compiler can identify when instructions inside of a loop can be rewritten as a vectorized operation. It works for simple operations only and is rare for database operators. However, the compiler is very conservative and will avoid converting to vector operations if there is any chance of it causing a correctness issue, E.g., Possibly overlapping pointers.
- **Compiler Hints**: Provide the compiler with additional information about the code to inform it that it is safe to vectorize. The developer is responsible for providing correct hints/guidelines. The following are two approaches to this:
  - Give explicit information about memory locations. E.g., Using restrict keyword in C/C++. Very common in DBMS.
  - Force the compiler to ignore vector dependencies.
- **Explicit Vectorization**: Use CPU intrinsics to marshal data between SIMD registers and execute vectorized instructions manually. It is often not portable across ISA/versions unless you use an indirection library like Google Highway.

The results from [1] using ICC (intel's compiler) indicate that although manual often performs best, it does not always lead to the smallest lines of code. The hybrid approach of letting the compiler auto-vectorize with hints and then optimize whatever is left manually often gives a good middle ground in terms of the amount of effort, performance, and lines of code.

4 Vectorization Fundamentals

This section explains the new operations in AVX-512 in greater detail and how they can speed up computation [2].

- **Masking**: Almost all AVX-512 operations support predication variants whereby the CPU only performs operations on lanes specified by an input bitmask. This avoids wasteful computation.
- **Permute**: For each lane, copy values in the input vector specified by the offset in the index vector into the destination vector. Before AVX-512, the DBMS had to write data from the SIMD register to memory and then back to the SIMD register, thus polluting the CPU cache.
- **Selective Load**: Use a bitmask to load data from memory onto only the selected elements in the SIMD vector.
- **Selective Store**: Use a bitmask to store data in the memory from only the selected elements in the SIMD vector.

- **Compress**: Use a bitmask to store only the selected elements from an input vector onto a vector contiguously.
- **Expand**: Use a bitmask to store the contiguous elements of an input onto the selected locations of another vector.
- **Selective Gather**: Use an index vector to load the selected elements from memory onto a vector contiguously.
- **Selective Scatter**: Use an index vector to store the contiguous elements from a vector onto the selected elements on memory.

Although AVX-512 is extremely useful because of the new operations, the extension has been split into different groups, and CPUs can selectively provide some or all of the groups. Older extensions like AVX2 were all or nothing and thus consistent. This forces DBMS to check the exact AVX-512 group supported by the hardware and choose the optimal algorithm accordingly.

Additionally, some CPUs downclock [2] the processor to avoid heating issues when running some AVX-512 instructions, which may lead to overall higher processing time. Hence, some DBMS choose to stick to AVX2.

5 Vectorized DBMS Algorithms

This section will discuss the principles for efficient vectorization by using fundamental vector operations to construct more advanced functionality [4]. The two basic ideas are as follows:

- We typically favor vertical vectorization by processing different input data per lane. Horizontal will also be used but has fewer applications.
- Maximize lane utilization by executing unique data items per lane subset. Avoid useless computations on data known to be invalid.

In particular, we will talk about vectorizing the following algorithms:

5.1 Selection Scans

Load data onto a SIMD vector and perform the predicate evaluation on the vector in parallel. Use the resulting bitmask to perform further operations on only the relevant data. Once the final bitmask is ready, use it to store the result. Minor optimizations can also be implemented to check if the entire vector becomes invalid after the operations. Auto vectorization of complex selection predicates is not trivial and often needs human intervention to be optimal.

The results from [1] show that using SIMD instructions and hand-written optimal code over Scalar instructions can lead to a speedup of 10% - 130% based on the type of operator for individual element processing. However, when queries are compared end to end, there is barely a 10% improvement due to the limited portion of the query that can be vectorized and materialization overheads.

5.2 Relaxed Operator Fusion [3]

For each batch of tuples, the SIMD vectors may contain tuples that are no longer valid (they were disqualified by some previous check), which leads to underutilization.

The idea behind Relaxed Operator Fusion (ROF) is to decompose the pipelines into stages that operate on vectors of tuples with artificial pipeline breakers. Each stage may contain multiple operators, which communicate through cache-resident buffers. Only move on to the next stage when the buffer is full; otherwise, keep processing the current stage.

Another benefit of ROF is that it works well with software prefetching, thus hiding cache miss latency. The DBMS can tell the CPU to grab the next vector while it works on the current batch. Prefetch-enabled operators define the start of a new stage. Any prefetching technique, like group prefetching, works well with ROF and is simple to implement.

5.3 Vector Refill Algorithms

The following are two approaches to refilling vectors using AVX-512 (possible due to the higher number of registers):

- **Buffered**: Use additional SIMD registers to stage results within an operator and proceed with the next loop iteration to fill in underutilized lane vectors.
- **Partial**: Use additional SIMD registers to buffer results from underutilized vectors and then return to the previous operator to process the next vector. Requires fine-grained bookkeeping to make sure other operators do not clobber deferred vectors.

Not many systems do this because of the sheer complexity of implementing this and the overhead of the bookkeeping required.

5.4 Hash Tables

Linear probe hash tables are not typically suited to vector operations because they require arbitrary memory accesses. Two methods for vectorizing this is as follows:

- **Horizontal**: Store multiple keys and values at each offset in the hash table. Use SIMD to compare the single input key to the multiple keys in the hash table location to find a match. It is not guaranteed to fully utilize the vector since key slots in the hash table may be empty.
- **Vertical**: Store a single key per vector slot. Use a SIMD gather to get key slots from memory to match. Replace the input keys that did match and offset the others by 1. This almost always results in complete utilization.

The results indicate that vertical is better for most use cases, but both drop down to as slow as scalar when the hash table grows too large to fit in the cache.

5.5 Partitioning - Histogram

When building histograms for statistics, use scatter and gather to increment counts. However, since multiple keys may map to the same hash index and clobber each other, we replicate the histogram so that each SIMD vector slot updates only one column (essentially becomes a matrix where each column corresponds to one SIMD vector slot). Then, a SIMD add is done to get the final histogram values.

References

[1] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13): 2209–2222, sep 2018. ISSN 2150-8097. doi: 10.14778/3275366.3284966. URL https://doi.org/10.14778/3275366.3284966.

[2] H. Lang, A. Kipf, L. Passing, P. Boncz, T. Neumann, and A. Kemper. Make the most out of your simd investments: counter control flow divergence in compiled query pipelines. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358538. doi: 10.1145/3211922.3211928. URL https://doi.org/10.1145/3211922.3211928.

[3] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. 11(1):1–13, sep 2017. ISSN 2150-8097. doi: 10.14778/3151113.3151114. URL https://doi.org/10.14778/3151113.3151114.

[4] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1493–1508, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2747645. URL https://doi.org/10.1145/2723372.2747645.