OPTIMIZING RELATIONAL QUERIES OVER BIT-PARALLEL DATABASE LAYOUT

Andrin Bertschi, Nicolas Wicki

Department of Computer Science
ETH Zurich, Switzerland

Isaak Hanimann, Carl Friess

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

In light of the advances in machine learning and its dependence on relational data, hardware acceleration in databases is becoming increasingly prevalent. Given the highly parallel nature of arithmetic operations executed in machine learning accelerators, specialized data layouts such as MLWeaving have been proposed as indexes in order to improve the performance of bit-parallel arithmetic.

Unfortunately, these data layouts add large performance overheads to queries executed on general purpose CPUs, leading to data duplication, as a traditional data layout needs to be maintained in order to allow for generic relational queries. To address this issue, we examine various optimizations to the execution of simple predicate and join queries operating directly on the MLWeaving format. Experimental results show that our optimized algorithms achieve runtime improvements of up to 31.9x compared to baseline implementations.

1 Introduction

MLWeaving is a specialized data representation that facilitates in-memory integer data to be loaded at arbitrary levels of precision, while optimizing the layout for cache-based memory hierarchies and hardware accelerators that make use of bit-parallel arithmetic units. In machine learning applications this is particularly useful, as it allows models to be trained using limited precision versions of relational training data, thus saving significant memory bandwidth, without incurring the costly overhead of storing data in multiple discrete levels of precision. MLWeaving provides a linear speedup inversely proportional to the number of bits of precision used on hardware accelerated training using FPGAs.

Motivation. Although the MLWeaving format used as an index alleviates the overheads that would be incurred by storing data in multiple levels of precision, it is not suitable as a primary storage format in relational databases with conventional execution of generic predicate-based queries. Given that general purpose processors employ bit-serial arithmetic, specialized algorithms are required to improve the performance of query execution and make MLWeaving viable as a primary storage format for extended applications.

Contribution. We evaluate various optimizations on the CPU-based execution of two types of queries, directly operating on the MLWeaving data layout. We focus on a predicate-based query (Query 1.0), combined with an optional aggregate function (Query 1.1), as well as a join query between two relations based on a predicate including a multiplication operation (Query 2). We introduce baseline implementations and discuss optimization techniques, ranging from sequential optimizations to vectorized implementations based on the Intel® AVX2 extension. Finally, we include experimental performance evaluations on relations up to 2 GB in size.

Related work. An early attempt on a bit parallel database layout is BitWeaving introduced by Li et al. [1]. BitWeaving proposes a way to rearrange data in memory in order to exploit bulk bit-wise operations to achieve higher performance on predicate-based database queries. The algorithms and optimizations discussed in this work are based on the MLWeaving format, originally introduced by Wang et al. in [2]. MLWeaving is a more recent bit-parallel database layout that builds on BitWeaving but is specifically optimized to enable hardware accelerated machine learning on FPGAs. To the author's knowledge, there are no other contemporary query execution algorithms that operate directly on the MLWeaving memory layout.

2 Background

In this section, we introduce the MLWeaving format at a high-level. We then proceed to introduce the SQL queries targeted by our algorithms, state assumptions and motivate our cost metrics.

MLWeaving. While we refer to [2] for an elaborate analysis of the data format, we now briefly introduce the key structure of MLWeaving. In the scope of this introduction, we consider a relation in conventional tabular format depicted on the left side in Figure 1. The relation has two rows, four columns and an integer precision of 4 bits. MLWeaving partitions the relation into banks and cache lines. In Figure 1 we consider a cache line of 8 bits. The cell **ABCD** at index (0, 0) in the conventional data format spans

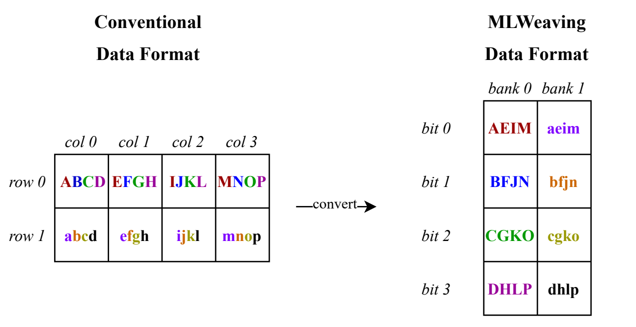

Fig. 1. Conventional Data Format converted to MLWeaving Data Format: Note that banks represent the number of rows loaded in a cache line.

over four cache lines in MLWeaving. Reading the first cache line in MLWeaving retrieves **AEIM** and **aeim**, which are the first bits of the entries in row 0 and 1. Using this format, we can read specific bits of multiple rows at once.

We use the term *block* to consider all data values in full precision, spanning over the supported bank count. In the example in Figure 1 a *block* is 32 bits in size.

Assumptions on MLWeaving. In this work, we consider a cache line to consist of 512 bits, as is the case for our target architecture (Intel® Coffee Lake). We work on unsigned integer data relations with a precision of at most 32 bits. Further constraints on the MLWeaving memory layout include:

- $banks = 2^i$, for $i \in \{3, 4, 5, 6\}$,
- $rows = 2^j$, for $j \in \mathbb{N}_{>0}$,
- $rows \equiv 0 \pmod{banks}$,
- $columns \equiv 0 \pmod{bits\_per\_bank}$.

The above assumptions remove corner cases from the query implementations. However, they can easily be relaxed in practice.

SQL Queries. Let R, $S \subseteq \mathbb{N}^{nxm}$ be two relations of n rows and m columns stored in MLWeaving format, with a precision of 32 bits. Let a, b, c be columns in the relations. This work examines the three queries presented in Listing 1.

```
-- Query 1.0
SELECT * FROM R WHERE R.a < R.b
-- Query 1.1
SELECT SUM(R.c) FROM R WHERE R.a < R.b
-- Query 2
SELECT * FROM R, S WHERE R.a * S.b = S.c
```

Listing 1. Implemented SQL Queries

Query 1.0 returns a bit vector $r = \{0, 1\}^n$, where $r_i = 1 \iff R[i].a < R[i].b$, i.e. row i fulfills the `WHERE` clause of the query. Query 1.1 is an aggregation of Query 1.0 and uses summation on column $R.c$. Hence, its result is a scalar value. For Query 2 we consider a `JOIN` on two relations where we output a set of matching tuples $\{(i, j) \mid i, j \in \{1, .., n\},\ R[i].a * S[j].b = S[j].c\}$.

Cost Analysis. Since we use MLWeaving to store integer data and our arithmetic operations operate entirely on integers, flops are not a relevant cost metric in our case. Furthermore, a non-negligible number of integer operations are used for address and index computations, making them difficult to isolate from other operations.

Thus, we choose runtime as our primary cost measure. From runtime, we are able to derive throughput in MB/s and input tuples per second for join operations. We prefer to visualize performance using throughput metrics since this allows us to more easily compare performance on data sets of various sizes, especially in view of runtimes that scale quadratically with input size.

3 Method

In this section, we present a detailed outline of the different optimizations applied to the query executions.

3.1 Query 1.0

Recall from Section 2 that Query 1.0 is a `SELECT` query on a single relation. We first introduce a baseline version, then proceed with sequential optimizations and finally uplift the best performing version to AVX2 intrinsics.

Implementing a Baseline. For the baseline version, we consider two variants of possible implementations. These variants are imposed by the data layout of MLWeaving. The first variant processes the relation in a *bank major* order, whereas the second variant uses *precision major* order. The access pattern for *bank major* order is identified in Listing 2.

```
for each block in R:
    for each bank in banks:  ①
        for each bit in precision: ②
            read bit of R.a and R.b;
        if a < b:
            output match;
```

Listing 2. Query 1.0: Pseudo code for the baseline implementation in bank major access pattern.

The second variant is analogous to Listing 2 but switches the loop ordering of ① and ② to first process all banks of the same precision.

Sequential Optimizations. We start with the standard procedure to apply code motion, strength reduction, and sharing of common sub-expressions. However, these optimizations are already applied when compiling with optimization flags, as they did not result in a significant speedup.

An important optimization is what we refer to as *early termination*. Recall the access pattern from Listing 2. The *bank major* iteration allows for early termination of the precision loop at ② as soon as the first bits of columns R.a and

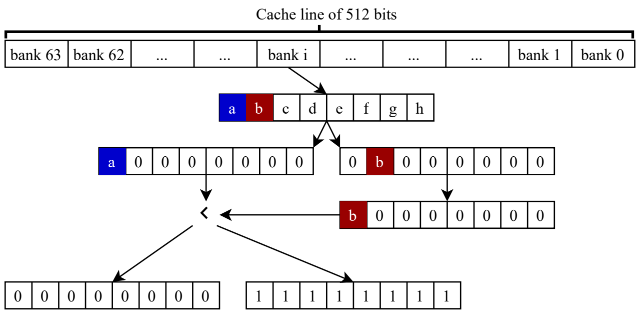

Fig. 2. Bit extraction and comparison on 8 bit chunks using 64 banks.

R.b differ. A similar idea can be applied to the *precision major* access pattern.

Vectorization. We follow up on our scalar optimizations with an implementation using Intel® AVX2. The loop order of this implementation is based on the *precision major* access pattern of the baseline. We choose this access pattern as the foundation for vectorization because it allows us to directly load a cache line of 512 bits into two 256-bit AVX2 vectors and proceed to operate on all banks at once. We hereby provide specialized implementations for different bank sizes[1]. The key idea here is to extract bits of columns R.a and R.b in each bank, shift them to the same position before applying a vectorized comparison of the bits for inequality. This process is depicted in Figure 2. If the current precision bit of R.a is smaller than R.b, we output a match, while bookkeeping ensures that consecutive bits no longer determine the result for that bank. The comparison returns a mask which can be accumulated and subsequently aggregated to a bit vector, where each bit indicates if a row is in the result set.

Early Termination. Given the sequential order of precision bits in the MLWeaving format, we can exploit *early termination* on the basis of an AVX2 vector. Recall that we load a cache line into two 256 bit vectors. As soon as the query is determined for all banks in both vectors, no further precision bits need to be loaded, and we can proceed to the next rows in the dataset. This not only reduces unnecessary computation, but also decreases data movement.

Bank Extensions. To exploit further optimization avenues, we extend the implementation to different bank sizes including 8, 16, 32, and 64 banks. Thereby, we employ different AVX2 extensions, namely `*_epi8`, `*_epi16`, `*_epi32` and `*_epi64` for packed signed integers of different sizes. Increasing the bank count also increases the amount of rows processed simultaneously, which overall leads to less unnecessary data moved. However, not all instructions are available on packed signed integers of sizes 8 to 64. To name an example, there is no shift left 8-bit packed integers (`_mm256_sll_epi8`) instruction, which is why we emulate it using a sequence of vector instructions to work as a substitute.

[1]For instance, a bank size of 8 allows us to operate on 64 columns (one bit of each column) of 8 consecutive rows.

Column Index Extensions. We initially assumed that columns R.a and R.b inhabit indexes 0 and 1, which are strong restrictions. This is why we extend the column indexes to inhabit any position within the same bank. This allows for both columns to be positioned anywhere within the bit range of a bank[2]. This involves a more sophisticated approach to the shifting process used to align columns. For instance, if the columns inhabit the most significant bit of the bank, we need to avoid signed integer comparison, as the most significant bit determines the sign and hence leads to a wrong result.

Precision Extensions. With the precision extensions, we allow a query to only consider the first n result bits, starting from the most significant bit, where $n \in \{1, 2, .., 32\}$. This may, depending on the precision level, reduce data movement drastically. Recall, however, that reducing the result precision may lead to inaccurate results.

3.2 Query 1.1

Remember from Section 2 that Query 1.1 builds upon Query 1.0 by additionally aggregating a third column R.c using the `SUM` operator. We now proceed with a similar approach by starting with a baseline implementation, following up with sequential optimizations, and finally applying vectorization supporting different bank sizes, column indexes and result precision. However, due to similarities to Query 1.0 we dedicate most of this section to the aggregation process of the Query 1.1.

Baseline Implementation. We consider both access patterns in *bank major* and *precision major* order and keep track of an accumulator to capture the sum of R.c. The pattern for *bank major* is summarized in Listing 3.

```
result = 0;
for each block in R:
    for each bank in banks:
        for each bit in precision:
            read bit of R.a, R.b and R.c;
        if a < b:
            result += c;
```

Listing 3. Query 1.1: Pseudo code for the *bank major* access pattern.

Sequential Optimizations. Sequential optimizations include code motion, strength reduction, and sharing of common sub-expressions. Early termination as introduced in Query 1.0 cannot be exploited in this query because the full result precision of R.c is needed for aggregation. However,

[2]For a bank size of 8, the range is 64 bits.

we can stop data reads on R.a, and R.b as soon as their bits differ, which reduces the pressure on registers.

Vectorization and Aggregation. For vectorization, we consider specialized implementations for different bank sizes ranging from 8 to 64 banks. Analogous to Query 1.0, we "unroll" the bank loop in the *precision major* access pattern and operate on two 256-bit AVX2 vectors. We apply early termination on R.a, and R.b but still need to read out R.c in full result precision. The key idea behind aggregation is to employ two integer-packed AVX2 vectors as accumulators. They restore the integer values of R.c, where each precision bit is shifted to the bit position of the original integer in R.c.

Bank Extensions. To support aggregation on banks of size 32 and 64, two accumulators no longer have enough capacity to restore the original integer in R.c. For 32 banks, an accumulator only dedicates 16 bits to a single bank. In this case, we represent 32-bit integers using two 16-bit integers. This is implemented analogously in the case of 64 banks. Using shifts, we can reproduce the original value of R.c and in the end aggregate in a scalar.

3.3 Query 2

Query 2 introduces a `JOIN` condition used to join two relations. We will start by introducing our baseline implementation, then describe the applied sequential and vectorization optimizations. Finally, we also discuss early termination and an extension that allows for arbitrary result precision.

Baseline Implementation. Our algorithm implements a simple nested-loop join to evaluate the join condition of the query. For the baseline, we first iterate over each row of the relation S in order to read one value each from the columns b and c in full precision. Then, for each row of S, we iterate over each row of the relation R and read the corresponding value of the column a in full precision. Finally, we check the join condition as shown in Listing 4. If a match is found, we output a tuple containing the row indices of the matching rows in R and S.

```
for each row in S:
    read one S.b and S.c value;
    for each row in R:
        read one R.a value;
        if R.a * S.b == S.c:
            output match;
```

Listing 4. Query 2: Baseline implementation.

Improved Cache Locality. Given the nature of the ML-Weaving memory layout, we were able to significantly improve the temporal locality of our algorithm by interleaving the loading of as many rows as there are banks (see ① and ② in Listing 5). A cache line is loaded into the cache when we access the first bank in the cache line. Given that in ML-Weaving a single cache line will contain bits from as many rows as there are banks, the bits from subsequent banks (and thereby subsequent rows) will already be available in the cache. The improvement in temporal locality comes from immediately reading the bits of subsequent rows prior to moving on to the next bit of precision of the same row. As a result, we now iterate through the relations linearly in blocks rather than non-linearly row-by-row. Beyond improving cache efficiency, this should also aid prefetching.

```
for each block in S:
    read 8 full precision S.b and S.c values; ①
    for each block in R:
        read 8 full precision R.a values; ②
        for each combination of rows in R and S:
            if any R.a * S.b == S.c:
                output match;
```

Listing 5. Query 2: Improved temporal cache locality.

Bit-parallel multiplication. Next, we attempted to fit all computations into the time spent waiting for memory requests by applying bit-parallel shift-and-add multiplication between each load. We do this by first reading eight S.b and S.c values ①, then reading R.a bit by bit ② while adding up the intermediate result of the multiplication in accumulators ③. We illustrate the implementation in Listing 6. Early benchmarking results revealed that performance was significantly worse. This is because multiplication already has quite high throughput (0.66 ops/cycle) and low latency (5 cycles) and the implementation of bit-parallel multiplication required much more instructions. Thus, we decided to abandon this approach.

```
for each block in S:
    read 8 full precision S.b and S.c values; ①
    initialize accumulators
    for block in R:
        for each bank in cache line:
            read one bit of each R.a; ②
            if bit == 1
                shift all S.b's and add to
                accumulators ③
        for each accumulator:
            if any accumulator == S.c:
                output match;
```

Listing 6. Query 2: Bit-parallel multiplication.

Further Sequential Optimizations. We achieved further performance improvements by applying code motion and strength reductions. In particular, we were able to significantly simplify memory address calculations.

Before vectorizing our implementation, we decided to fix the number of banks to eight and relation precision to 32 bits at compile time. These fixed values provided a basis for our vectorized implementation, as they were selected specifically given that AVX2 vectors consist of 256 bits, which in our case store eight 32-bit integer elements.

Vectorization. Similar to our sequential implementation, we load eight rows from each relations at a time, such that we obtain eight-element vectors for each column R.a, S.b and S.c. The key idea behind our vectorized implementation is to shift the elements in the R.a vector to produce eight different permutations before applying element wise multiplication resulting in the 64 different combinations of the rows loaded from both relation.

```
for each block in S:
    read 8 full precision S.b and S.c values;
    for each block in R:
        read 8 full precision R.a values;
        for i in 0..7:
            shift R.a elements by i;
            if any R.a * S.b == S.c:
                output matches;
```

Listing 7. Query 2: Vectorized implementation.

To load the operands from each relation, we read a full cache line (512 bits wide) into two AVX2 vectors (each 256 bits wide) and merge them using shuffle and blend operations such that we obtain a single AVX2 vector that contains the relevant bits for each of the eight rows contained in the cache line. As we load the next cache line, we shift each partial operand and combine them using an or operation.

In order to evaluate the join condition, we need to check all combinations of the rows loaded from R with the rows loaded from S. We shift the R.a vector seven times to obtain eight different permutations of the elements in the vector. By subsequently multiplying the S.b vector with all eight versions of the R.a vector and comparing the result to the S.c vector, we can evaluate all 64 combinations using only eight loop iterations.

We chose to shift the R.a vector rather than the S.b vector in order to prevent having to execute the shift operations on S.c as well.

Loop Unrolling. After vectorizing the implementation, we were able to further reduce data dependencies and apply some loop unrolling, yielding a moderate performance improvement. In particular, we fully unrolled the loop used to generate and evaluate the eight different versions of the R.a vector.

Early Termination. Given that each bit of precision is loaded sequentially, we have the opportunity to terminate this loading process early if we can determine that none of the values being loaded will lead to a match. By reversing the order in which we load bits and start with the least significant, we can rule out matches with a subset of the bits.

However, we observe that checking the join condition is expensive, as it involves a lot of arithmetic and logic operations and needs to be repeated eight times to cover all combinations. As such, it is impractical to check for matches every time we load another bit. Instead, we evaluate the join condition after s bits have been loaded (see ① in Listing 8). The probability of terminating is thus higher and the check is more likely to be worth it. If any of the 64 results indicate a possible match, we proceed to load the remaining bits for all operands ②.

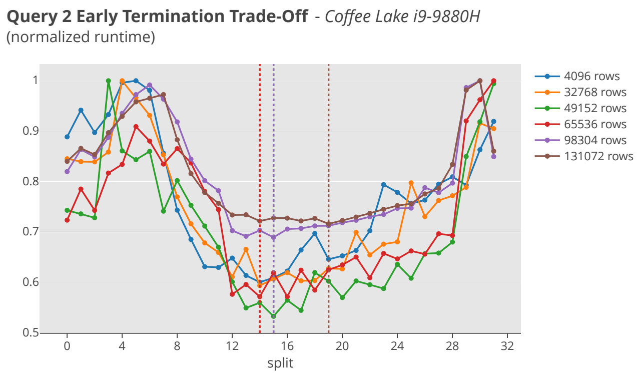

Fig. 3. Normalized runtime for each split factor. *The vertical lines show the best split factor for each of the data set sizes. Lower is better.*

```
for each block in S:
    read 8 full precision S.b and S.c values;
    for each block in R:
        read lowest s bits of 8 R.a values; ①
        for each shifted R.a vector:
            if any mask(R.a * S.b) == mask(S.c):
                read remaining bits for R.a; ②
                for each shifted R.a vector:
                    if any R.a * S.b == S.c:
                        output match;
                break;
```

Listing 8. Query 2: Early Termination.

To determine the right split factor, s we benchmarked different split factors for data sets of various sizes. The results are shown in Figure 3. As one can see, if we check too early (small split factor) the probability of being able to terminate is low and thus the early termination check wasn't worth it. Conversely, if we check too late (large split factor) too many bits are loaded unnecessarily.

In our data set distribution, the probability of a match is dependent on the number of rows. As expected, we observe a general trend: the more rows, the higher the ideal split factor. For smaller data sets the best split factor is 14, while for larger data sets it is 19.

Arbitrary Input Precision. An important observation for implementing arbitrary precision for Query 2, is that results change based on precision, and you need to generate custom data to keep result size under control with low input precisions.

The implementation is straight forward: one can simply stop reading R.a's, S.b's and S.c's after desired precision many cache lines instead of always reading 32 cache lines.

4 Experimental Results

In this section, we turn to three series of experiments which we conducted to evaluate the different optimizations. We state the experimental setup and detail the data generation process. We first present benchmark results for Query 1.0, proceed with Query 1.1 and finally discuss Query 2.

Experimental Setup. We execute all benchmarks on a MacBook Pro (Darwin 20.5.0, Big Sur 11.3, 16-inch, 2019) using an Intel® Coffee Lake i9-9880H, clocked at 2.3 GHz with an L3 cache of 16 MB, and 16 GB RAM. We disable Turbo Boost and restart the device before executing all benchmarks. All queries are compiled with Apple® clang version 12.0.5, using flags `-O3` and `-march=native`. Benchmarks are executed using the Google Benchmark [3] framework (version 1.5.4). Google Benchmark features automatic iteration estimation, which dynamically ensures a statistically stable result [4].

Generating Data. For Queries 1.0 and 1.1, we gather benchmark results on relations of different row counts up to 2 GB in relation size, with 512 columns. Tuple data is generated uniformly at random in 32 bit precision. For Query 2, we generate two relations with 64 columns each and the same number of rows. Given the quadratic runtime associated with nested-loop join algorithms, we caped the data set size at 64 MB. Although this is a relatively small data set, it is still much larger than the L3 cache on the target processor.

Evaluation Metric. As motivated in Section 2, plots presented in the preceding sections use throughput in either MB/s or input tuples/s. The x-axes detail different sizes of the data set, while the y-axes indicate throughput of the various implementations. When stating speedup relative to a baseline, we refer to dataset sizes of 2 GB for Query 1.0 and 1.1. In the case of Query 2, we use a dataset size of 32 MB.

4.1 Results Query 1.0

In this section, we discuss the performance results of the first query when applying optimizations discussed in Section 3.1. If not otherwise stated, all experiments use column index 0 for R.a and 1 for R.b. Banks and precision are fixed at 8 and 32 respectively.

Optimizations. In Figure 4, we show the optimizations for Query 1.0. The baselines *precision major* and *bank major* achieve a comparable throughput. *stdc* using early termination delivers a noticeable performance improvement of 2.17x when compared to the *bank major* baseline. By terminating the precision loop early, data movement is reduced significantly. The vectorized implementation *vect. 8 banks* increases the throughput up to a factor of 3.51. Supporting arbitrary *col. indexes* using index 383 for R.a and 320 for R.b introduces some overhead, however, it allows for more flexibility in real-world applications, while still maintaining a performance gain of 3.11x. Increasing the amount of

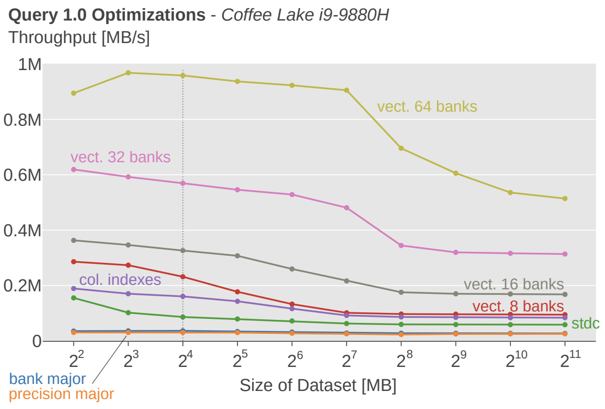

Fig. 4. Different optimizations applied for Query 1.0. *L3 cache of 16 MB is marked with a vertical line. Higher is better.*

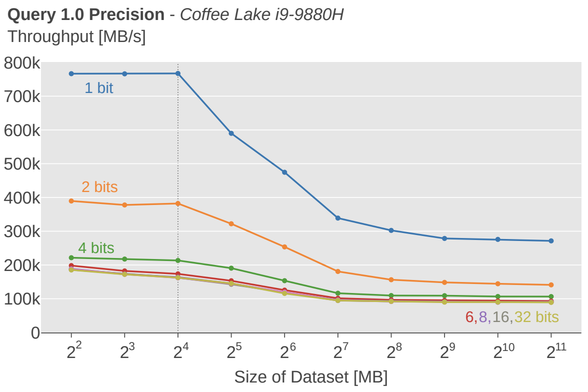

Fig. 5. Different result precisions applied to Query 1.0. *L3 cache of 16 MB is marked with a vertical line. Higher is better.*

banks delivers a linear speedup up to 32 banks with *vect. 32 banks* further improving the throughput up to a factor of 11.67x. The fully optimized version using 64 banks (*vect. 64 banks*) finally advances to a speedup of 19.13x with a slight deviation in the linear trend caused by the overhead of emulation as mentioned in Section 3.1. The throughput drops with increasing dataset size, and stabilizes at 128 MB.

Precision Extensions. In Figure 5, we observe that the fewer bits we use to evaluate the query, the higher the throughput. The throughput increases significantly once we reach fewer than 6 bits, as early termination already minimizes the amount of bits moved dependent on the distribution of the data[3]. Note the drastic drop in throughput, once the dataset reaches a size greater than 16 MB (L3 cache),

[3]Normal distribution increases the chance of termination by 50% for each bit read.

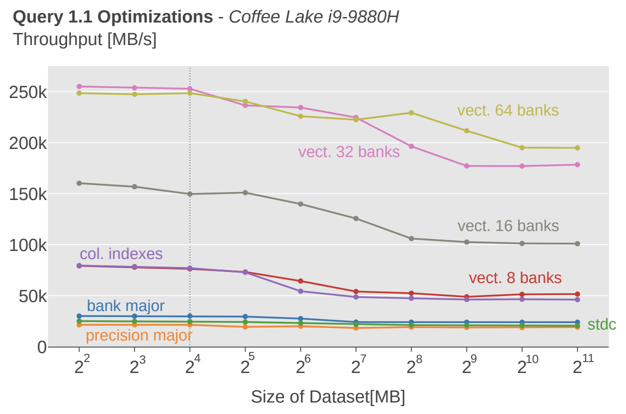

Fig. 6. Different optimizations applied for Query 1.1. *L3 cache of 16 MB is marked with a vertical line. Higher is better.*

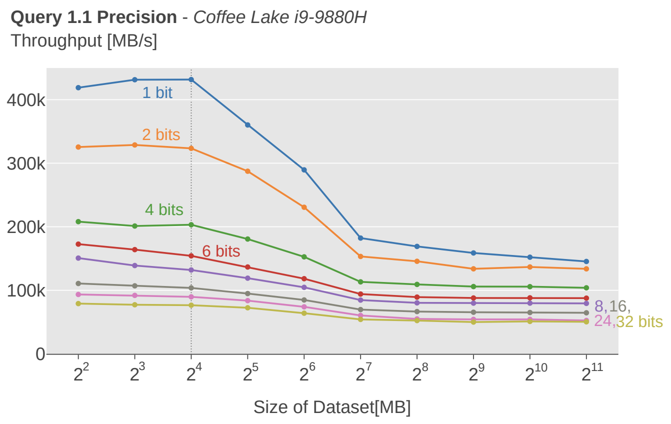

Fig. 7. Different result precisions applied to Query 1.1. *L3 cache of 16 MB is marked with a vertical line. Higher is better.*

which converges at around 256 MB.

4.2 Results Query 1.1

In this section, we assert the performance results of Query 1.1 when applying the optimizations discussed in the previous sections. If not otherwise stated, all experiments use column index 0 for R.a, 1 for R.b, and 2 for R.c. Banks and precision are again fixed at 8 and 32 respectively.

Optimizations. In Figure 6, the baselines for *bank major* and *precision major* perform similarly as in Figure 4. We further establish that *stdc* with sequential optimizations is outperformed by the `-O3` compiler optimizations. This can be attributed to manually introduced optimization blockers. However, when using the *vect. 8 banks* optimization we achieve a speedup of 2.16x over the *bank major* baseline. The support of various column indexes using indexes 383 for R.a, 321 for R.b, and 320 for R.c in *col. indexes* introduces some overhead, but still delivers a speedup of 1.92x. The throughput increases linearly with the increase in banks again up to *vect. 32 banks* with a speedup of 7.45x. However, the implementation *vect. 64 banks* interrupts this trend with a speedup of 8.14x where the emulation of vector instructions as described in Section 3.2 becomes prevalent. We experience a considerable performance drop when compared to Query 1.0 caused by the lack of *early termination*.

Precision Extension. In Figure 7, we identify a decreasing throughput the more bits we use to evaluate the query. However, in contrast to the results presented in Figure 5, we notice that the throughput linearly decreases with the increase in precision as *early termination* cannot be applied. Again, a drop in throughput is visible when exceeding a dataset size of 16 MB.

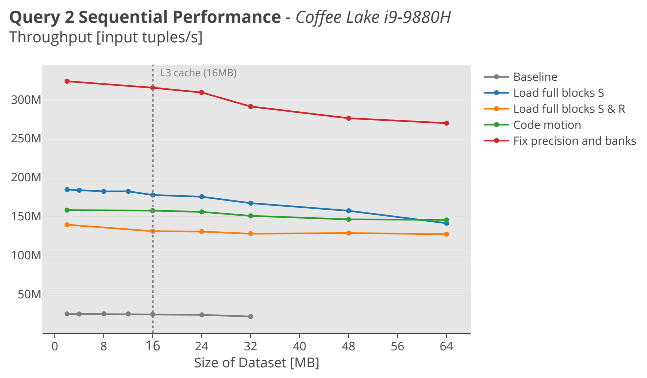

Fig. 8. Query 2: Throughput of all sequential implementations. Higher is better.

4.3 Results Query 2

Sequential Optimizations. Figure 8 shows that reading the entries in batches to increase temporal locality achieves a significant speedup of 7.4x. We also notice, that as the data set grows beyond the L3 cache size (16 MB) we experience a small drop in performance. Furthermore, defining the precision of the values stored in the relations and the number of banks at compile time as discussed above, significantly boosts throughput and is our best sequential implementation, as various optimization blockers for the compiler are removed.

Vectorization. Figure 9 compares multiple vectorized implementations to the best sequential implementation (red line). The plot illustrates well why choosing the ideal split factor is important. The early termination benchmarks in Figure 9 are run using a constant split factor of $s = 14$. For larger data sizes (e.g. 64 MB) the ideal split factor is 19, not 14. Therefore, the performance of early termination for this

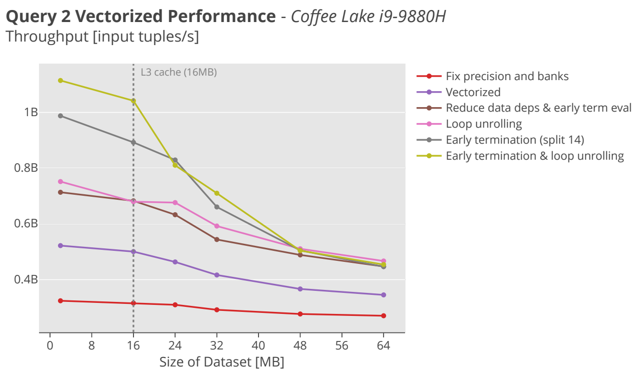

Fig. 9. Query 2: Throughput of vectorized implementations. *Constant split factor* $s = 14$ *for early termination.*

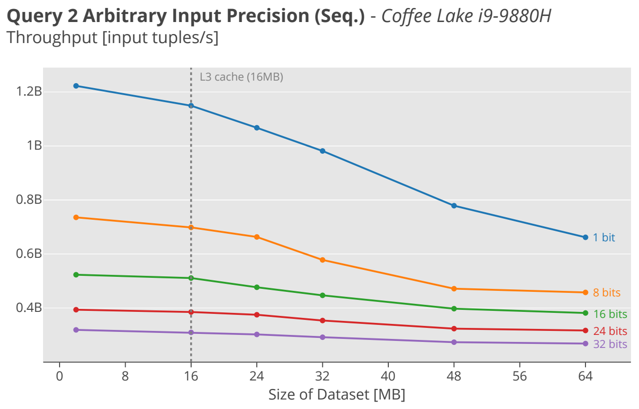

Fig. 10. Query 2: Throughput with arbitrary precision on our best sequential implementation. *Higher is better.*

data size turned out worse than not using early termination at all. Nonetheless, it is clear that when the split factor is well tuned, the implementation with early termination provides the best performance.

Arbitrary Precision. Figure 10 establishes that the throughput of our fastest sequential version improves linearly with decreasing precision. Similarly, Figure 11 shows that the throughput of our fastest vectorized version also improves with decreasing precision. However, it isn't quite linear.

Performance Summary. Table 1 summarizes the improvements in throughput for all major steps in our journey optimizing Query 2.

5 Conclusions

In this work, we explored and applied various sequential and vectorized optimizations to significantly improve the performance of queries executed directly on the MLWeaving data format. Our experimental evaluations show throughput improvements of up to 19.13x for Query 1.0, 8.14x for Query 1.1, and 31.9x for Query 2. Further performance improvements can be achieved when using reduced input precision in all cases.

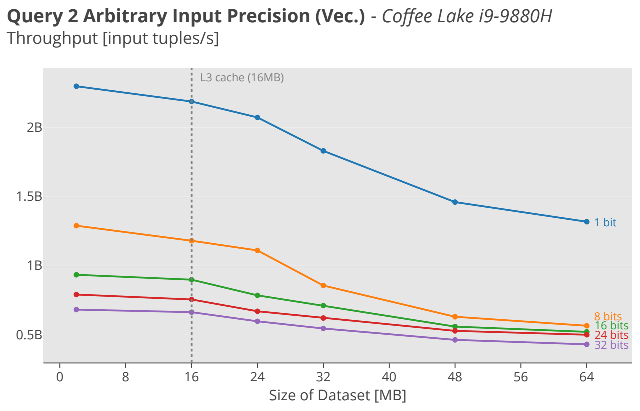

Fig. 11. Query 2: Throughput with arbitrary precision on our vectorized implementation. *Higher is better.*

Baseline	1.0x
Load full blocks S	7.4x
Fix precision and banks	13.1x
Vectorized	18.7x
Reduce data deps and early term eval	24.4x
Loop unrolling	26.6x
Early termination	29.7x
Early termination and loop unrolling	**31.9x**

Table 1. Speedup summary of Query 2.0 with a dataset size of 32 MB.

We consistently found leveraging the cache-aware nature of MLWeaving and early termination to be powerful optimizations in evaluating these queries. The predicate-based queries with additional aggregation benefit drastically from an increased amount of banks, while still providing reasonable flexibility in their column choices. However, an even more generalized approach would be of interest in real-world applications, and a potential involvement of early termination in the aggregation approach might nonetheless be viable. In the case of the join query, the success of early termination is dependent on accurate tuning of an additional parameter. We see room for future work to show how this parameter can be effectively set by heuristics based on query optimization information provided by a database engine.

Overall, this work successfully shows that with specialized query execution, data formats optimized for hardware accelerators can be used as primary storage formats in general purpose database applications with reasonable performance constraints. We leave the generalization to more complex query types for future work.

6 Contributions of Team Members

We split our team into two groups to more easily distribute the workload. We frequently updated our progress and exchanged acquired insights.

Andrin Bertschi. Worked together with Nicolas on optimizations for Query 1.0 up to 8 banks, proceeded to apply optimization for bank variant 32. Focused on result aggregation in Query 1.1. Implemented arbitrary columns for banks 16 and 32. Applied arbitrary result precision in Query 1.0.

Nicolas Wicki. Worked together with Andrin on optimizations for Query 1.0 up to 8 banks, and continued with implementing bank variants for 16 and 64 banks. Also, provided implementations for arbitrary columns with 8 and 64 banks. Applied arbitrary result precision for Query 1.1.

Isaak Hanimann. Focused on adapting the algorithm to use bit-parallel multiplication, as well as some sequential optimizations and early termination using the bit-parallel multiplication approach. Implemented the arbitrary precision extensions for the sequential and vectorized versions of Query 2.

Carl Friess. Implemented the baseline for Query 2, as well as some sequential optimizations. Mostly focused on the vectorized version of the algorithm and exploring avenues for making use of early termination in Query 2. Implemented the early termination approach using a split factor and evaluated the trade-off in selecting the optimal split factor.

7 References

[1] Yinan Li and Jignesh M. Patel, *BitWeaving: Fast Scans for Main Memory Data Processing*, SIGMOD, 2013.

[2] Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Onur Mutlu, and Ce Zhang, *Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning*, PVLDB, 2019.

[3] Google, “Google Benchmark, A library to benchmark code snippets, similar to unit tests,” `https://github.com/google/benchmark`, Accessed on 21-06-25.

[4] Google, “Runtime and Reporting Considerations in Google Benchmark,” `https://github.com/google/benchmark#runtime-and-reporting-considerations`, Accessed on 21-06-25.