Efficient SIMD Vectorization for Hashing in OpenCL

Tobias Behrens1 Viktor Rosenfeld1 Jonas Traub2 Sebastian Breß1,2 Volker Markl1,2

1DFKI GmbH 2Technische Universität Berlin

ABSTRACT

Hashing is at the core of many efficient database operators such as hash-based joins and aggregations. Vectorization is a technique that uses Single Instruction Multiple Data (SIMD) instructions to process multiple data elements at once. Applying vectorization to hash tables results in promising speedups for build and probe operations. However, vectorization typically requires intrinsics – low-level APIs in which functions map to processor-specific SIMD instructions. Intrinsics are specific to a processor architecture and result in complex and difficult to maintain code.

OpenCL is a parallel programming framework which provides a higher abstraction level than intrinsics and is portable to different processors. Thus, OpenCL avoids processor dependencies, which results in improved code maintainability. In this paper, we add efficient, vectorized hashing primitives to OpenCL. Our results show that OpenCL-based vectorization is competitive to intrinsics on CPUs but not on Xeon Phi coprocessors.

1 INTRODUCTION

Modern processors support Single Instruction Multiple Data (SIMD) extensions. These vectorized instructions process multiple data values in a single instruction to increase the computational efficiency of a program. Database operators that use SIMD instructions are several times faster than scalar operators, because they process multiple tuples at once [8, 10, 11, 13].

Compilers expose SIMD instructions through function-like primitives called intrinsics [5]. Since intrinsics correspond directly to SIMD instructions of a processor, they are processor-dependent. Different processor architectures use specific instruction sets and each processor generation typically adds new instructions. Consequently, supporting vectorized database operators on different processors requires continuous maintenance of a growing code base and increases development costs.

Parallel computing frameworks such as OpenCL abstract from low level intrinsics and enable programmers to write code in a restricted dialect of C. The major advantage of OpenCL is its portability. Processor-specific compilers translate OpenCL programs to efficient machine code. OpenCL natively supports vectorized data types, which are directly compiled to the native SIMD instructions of a particular processor. However, OpenCL’s vectorized instruction set is limited to arithmetic, logical, and permutation operations. Therefore, we need to emulate more complex SIMD instructions such as Gather and Scatter [8].

In this paper, we leverage OpenCL to provide vectorized implementations of database operators which are portable to different instruction set architectures and processors (e.g., Intel CPUs and Xeon Phi coprocessors). OpenCL programs are implemented in special functions called kernels. We provide vectorized kernels for the data movement primitives selective load, selective store, gather, and scatter [8]. These primitives are essential building blocks of hash-based operators. We use vectorized hashing operations for

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-073-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Figure 1: Vectorized data movement primitives. Grey boxes indicate values that are neither read nor written.

our case study because they are at the core of many database operators. Our results show that portable OpenCL-based hashing is competitive to processor-specific vectorized implementations. Specifically, we make the following contributions:

1) We adapt vectorized data movement primitives to the OpenCL computation model. Using these primitives, we formulate explicitly vectorized algorithms of different data processing operations (Section 3).

2) We compare our OpenCL-based approach with intrinsics-based SIMD instruction sets – namely, AVX2 on a Haswell CPU and AVX512 on a Xeon Phi coprocessor (Section 4).

2 BACKGROUND

2.1 Vectorized Data Movement Primitives

Vectorized data movement primitives move data between SIMD lanes (i.e., the components of SIMD registers) and memory locations [8]. Selective Load, Selective Store, Gather, and Scatter are such data movement primitives. Selective Load (Figure 1a) selects data from contiguous memory (starting at an offset) and copies it into SIMD lanes specified by a bitmask. Selective Store is the inverse operation of Selective Load, which copies data from SIMD lanes into contiguous memory. Gather (Figure 1b) selects data from discontiguous memory and copies it into SIMD lanes. A separate SIMD register provides the pointers to data elements. Scatter is the inverse operations of Gather which copies data from SIMD lanes to discontiguous memory. Modern processors support these operations natively to a certain extent: The Intel Xeon Phi coprocessor, which uses the AVX512 SIMD instruction set, supports all four primitives, Intel Haswell CPUs, which use the AVX2 SIMD instruction set, support Gather operations only. However, Polychroniou et al. emulate these primitives using basic SIMD permutation instructions at a small performance penalty [8].

2.2 Vectorized Linear Probing in Hash Tables

A probe operation iterates over many keys. Vectorized hash tables use a SIMD register (k) to probe multiple keys (ki) at once. We show the initial iteration step of vectorized hashing in Figure 2a:

1) We load probe keys (ki) into a SIMD register (k) with Selective Load. In the first iteration, we load all SIMD lanes as indicated by the green bitmask.

2) For each probe key ki in the SIMD register, we compute the hash hi and store it in a SIMD register h. We keep separate SIMD registers for probe keys (k) and their hashes (h).

3) We use the hash values as position pointers in a Gather operation to load buckets from the hash table. We store the found keys in a new SIMD register k’.
We compare the original probe keys ($k$) with the keys we retrieved from the hash table ($k'$). This comparison results in a collision mask ($c$). Light green boxes in the mask indicate matches and dark red boxes indicate collisions. In our example, we find three expected keys: $k_1$, $k_3$, and $k_4$. However, the hash table contains the key $k_5$ at position $h_5$ instead of $k_2$, which indicates a collision (i.e., $k_2$ and $k_5$ have the same hash). In general, there are three possible cases per probe key: (1) The bucket contains the key. (2) The hash bucket is empty. (3) The key in the hash bucket and the probe key are different.

In cases one and two, we replace the matched probe key in $k$ with a new probe key. In case three, we keep the probe key $k_1$, but increment its hash value $h_1$ in $h$ to probe the next bucket (Figure 2b, step ⑤). We now continue with the next iteration.

- We use the collision mask to load new keys into the SIMD lanes for which there was no collision in the previous iteration, leaving key $k_2$ unchanged as described above.
- We compute hashes for the new keys in $k$, leaving the value $h_2+1$ unchanged. Steps ⑤ to ⑦ repeat until all probe keys are processed.

Note that we need to write the payload (matched keys) into a temporary vector (Line 7). (3) It copies the left-most values back to their original positions (Lines 11–12).

To implement data movement primitives, we use several built-in functions. OpenCL natively supports vector data types which represent SIMD registers. OpenCL also provides arithmetic and logical operations on vector types. For example, we compare two vectors containing four values in Listing 1. We can also access individual vector components by their indices which increase from left to right. For example, probeKeys.5 selects the left-most component containing the value $k_1$.

Listing 1: Vectorized data types and operations in OpenCL

```c
uint4 probeKeys = {k1, k2, k3, k4};
uint4 foundKeys = {k1, k2, k3, k4};
uint4 mask = probeKeys == foundKeys; // [1, 0, -1, -1]
```

3 PORTABLE VECTORIZED HASHING

In this section, we review vectorization support in OpenCL and present the internals of our OpenCL-based primitives Selective Load, Selective Store, Gather, and Scatter.

3.1 Vectorization Support in OpenCL

To implement data movement primitives, we use several built-in functions. OpenCL natively supports vector data types which represent SIMD registers. OpenCL also provides arithmetic and logical operations on vector types. For example, we compare two vectors containing four values in Listing 1. We can also access individual vector components by their indices which increase from left to right. For example, probeKeys.5 selects the left-most component containing the value $k_1$.

3.2 Implementation of Primitives

Selective Load. Listing 2 shows the internals of the Selective Load primitive which we introduce with the other primitives in Section 2.1 (Figure 2a). The algorithm has four parameters: (1) input: source memory buffer, (2) offset: read offset on input, (3) vector: target vector, and (4) mask: indicates the components in vector which will be overwritten. The algorithm uses the parameters as follows: (1) It moves components which will be overwritten to the left of the target vector and adjusts the mask accordingly (Lines 3–6). (2) The algorithm loads the input data into a temporary vector (Line 7). (3) It copies the left-most values from the temporary vector into the target vector according to the mask (Line 8). (4) The algorithm moves the components of the target vector back to their original positions (Lines 11–12).

The shuffle functions used in steps 1 and 4 of the algorithm require permutation masks (left and back) to reorder the target vector. To speed up execution, we precompute these masks and store them in two lookup tables (one per step).
Inputs: input, offset, vector, mask

offset += popcount(index);
uchar8 left = convert_u16tmove_left_masks[index];
vector = shuffle(vector, left);

// Outputs: offset, vector

offset += popcount(index);
uchar8 back = convert_u16tmove_back_masks[index];
vector = shuffle(vector, back);

Listing 2: OpenCL implementation of Selective Load.

Inputs: input, offset, vector, mask

uchar index = \sum_{i=0}^{n-1} x_i * mask_i;
uchar8 left = convert_u16tmove_left_masks[index];
uchar8 back = convert_u16tmove_back_masks[index];

Listing 3: OpenCL implementation of Selective Store.

Listing 4: OpenCL implementation of Gather.

We select the required permutation mask depending on the mask parameter (Line 3). The lookup tables together consume 4 KB and fit comfortably in the L1 cache.

Step 4 of the algorithm is only required if the calling code expects the components of target vector to remain in the original order. In many cases the calling code does not have this expectation. For example, the hashing scheme in Section 2.2 does not require the original order as long as reordering is mirrored between probe keys and the collision mask. Therefore, we optimize the general implementation shown above by omitting step 4 of the algorithm (Lines 11 and 12). This optimization discards one lookup table and saves the respective space in the L1 cache.

Selective store. The implementation of Selective Store is provided in Listing 3. It is similar to Selective Load and has the same parameters. Again, we move the components of vector that will be stored according to mask to the left (line 5). The values are then written to output (Line 6). Since the original vector is unchanged, we do not have to shuffle it back.

Gather and Scatter. We provide the implementations for the Gather and Scatter primitives in Listings 4 and 5. We access the components of vector and mask by their indices (see Section 3.1).

Overall, we replace a complex and non-portable implementation based on intrinsics (Listing 6) with a fairly simple and portable implementation in OpenCL. Our implementation offers the same functionality, while improving maintainability and portability.

4 EVALUATION

4.1 Experimental Setup

Execution Environment. We evaluate our implementation on two processors, an Intel Core i7-6700K CPU\textsuperscript{2} and a Xeon Phi 7120P coprocessor\textsuperscript{3} based on Intel’s MIC architecture. As of writing, the newer Xeon Phi Knights Landing (KNL) product line does not yet support OpenCL. On the Xeon Phi, we utilize 60 threads for our measurements to emphasize call overheads. On the CPU, we utilize eight threads to emphasize attainable throughput. We use native implementations based on intrinsics \cite{8} as a baseline. First, we perform microbenchmarks to measure the performance of our vectorized data movement primitives in isolation. We then evaluate the complete hash table implementation by executing a hash join. We separately measure building the hash table on the inner join table, and probing it with keys from the outer table.

Every experiment is executed 20 times. We prevent autovectorization of the scalar and intrinsics implementation. For each experiment we generate new test data to obtain unbiased results.

Load and Store. To evaluate both primitives, we stream 100 million keys (10^8, 32-bit int) from memory into a SIMD register (or vice versa) and measure the throughput. For each invocation, we change the mask indicating which SIMD lanes are accessed, accessing four out of eight lanes on average. The large number of keys simulates a large outer table of a hash join.

Gather and Scatter. To evaluate Gather and Scatter, we read 100 million keys from contiguous memory locations into a SIMD register. We use memory regions of different sizes, from 4 KB to 64 MB, to simulate hash tables built on inner tables used in a hash join. The stride size between the memory locations depends on the size of the memory region. Especially for large memory regions, we chose the indices so that the accessed data does not fit into the processor cache.

Hash Join Build and Probe. In general, we adopt the experimental setup of Polychroniou et al. \cite{8} in order to obtain comparable results. We build a hash table on the inner table with a load factor of 50% and evaluate hash tables of different sizes, from 4 KB to 64 MB, to simulate hash tables built on inner tables used in a hash join. We separately measure building the hash table on the inner join table, and probing it with keys from the outer table.

To simplify partitioning the workload to different threads, the number of keys in the outer table depends on the processor. On the CPU, the outer table contains 100 million keys. On the Xeon Phi, it contains 245.76 million keys. We chose the keys in the outer table so that on average every tenth key is found in the hash table. Note, that our hash join implementation includes writing the matched probe keys to an output buffer.

491
4.2 Results

Selective Load and Store. Figure 3a shows the results of the Selective Load and Store microbenchmarks. The intrinsics-based version of Selective Load outperforms the portable OpenCL implementation by a factor of 1.75. The native Selective Store implementation is 4.6 times faster than the OpenCL version.

Gather and Scatter. Figure 3d shows the results of the Gather and Scatter microbenchmark. If the hash table fits into the L1 cache, the intrinsics implementation of Scatter is 1.5 times faster than the OpenCL version. However, if the hash table size exceeds the L1 cache, the performance of both implementations drops. The native implementations of Gather is between 1.75 and 1.9 times faster than the OpenCL-based implementation.

Hash Join Build. We present the results of the hash build experiment for the CPU in Figure 3b and for the Xeon Phi in Figure 3e. On the CPU, the OpenCL-based vectorized implementation is marginally slower than the OpenCL-based scalar implementation. Both are significantly slower than the C-based build and exhibit a curious rising trend up to a hash table size of 1 MB. This trend is due to overheads of OpenCL kernel invocations. To illustrate this, we also show scalar implementations which build 10000 hash tables inside a single function to minimize call overhead (dashed lines). These curves follow the expected shape. On the Xeon Phi, all implementations show a rising trend due to function call overhead. These overheads cause the bowl-shaped form of the curves which is most visible for the C-based scalar implementation. However, we cannot fully explain why the curves of the C-based and OpenCL-based scalar implementations cross between the hash table sizes of 64 kB and 128 kB.

Hash Join Probe. Figures 3c and 3f show the result of the hash probe experiment on the CPU and the Xeon Phi. We compare the vectorized OpenCL-based implementation with an intrinsics-based implementation [8] and a scalar implementation. On the CPU, both vectorized implementations outperform the scalar version as long as the hash table fits into L2 cache. For 4 kB hash tables, the intrinsics-based implementation is twice as fast as the scalar implementation, whereas the OpenCL-based implementation is 1.3 times faster. For small hash tables on the Xeon Phi, the intrinsics-based implementation greatly outperforms the scalar version whereas the OpenCL-based vectorized implementation is slower. On this processor, the data movement primitives are implemented directly as SIMD instructions which perform much faster than implementations that rely on an emulation.

5 CONCLUSION

Vectorized database operators improve performance but require processor-specific APIs. In this paper, we vectorize the essential primitives Gather, Scatter, Selective Load and Selective Store in OpenCL to reduce code complexity and to ensure portability.

We conduct an evaluation on CPUs and Xeon Phi coprocessors. In general, vectorized hashing based on intrinsics outperforms OpenCL-based hashing. Hash tables usually exceed processor caches. In this case, both variants are memory-bound and perform similarly. However, on CPUs, OpenCL-based vectorized hashing outperforms scalar hashing for moderately sized hash tables that fit into the L2 cache. In this case, our OpenCL-based hashing scheme is competitive to intrinsics-based hashing.

Acknowledgements: This work was funded by the EU projects SAGE (671190) and E2Data (780245), DFG Stratosphere (606902), and the German Ministry for Education and Research as BBDC (01IS1013A) and Software Campus (01IS12056).

REFERENCES