Vectorized Bloom Filters for Advanced SIMD Processors

Orestis Polychroniou
Columbia University
orestis@cs.columbia.edu

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

ABSTRACT

Analytics are at the core of many business intelligence tasks. Efficient query execution is facilitated by advanced hardware features, such as multi-core parallelism, shared-nothing low-latency caches, and SIMD vector instructions. Only recently, the SIMD capabilities of mainstream hardware have been augmented with wider vectors and non-contiguous loads termed *gathers*. While analytical DBMSs minimize the use of indexes in favor of scans based on sequential memory accesses, some data structures remain crucial. The Bloom filter, one such example, is the most efficient structure for filtering tuples based on their existence in a set and its performance is critical when joining tables with vastly different cardinalities. We introduce a vectorized implementation for probing Bloom filters based on gathers that eliminates conditional control flow and is independent of the SIMD length. Our techniques are generic and can be reused for accelerating other database operations. Our evaluation indicates a significant performance improvement over scalar code that can exceed 3X when the Bloom filter is cache-resident.

1. INTRODUCTION

Advances in computer hardware have had a tremendous impact in the way software is written. The most profound is the inherent parallelism of the multi-core CPUs that forces all efficient applications to be re-written as parallel applications. In fact, thread parallelism is only the tip of the iceberg. Other hardware features that potentially influence performance are multi-level caches, both private and shared across the cores, cache consistency protocols augmented with hardware transactional memory support, and wide CPU registers supported by comprehensive SIMD instruction sets.

Due to the large main memory capacity of recent hardware, many workloads can be kept in RAM. Thus, DBMSs focus on in-memory execution of queries [18] in order to perform real-time analytics. Indexes have become less important as most queries access a large percentage of the data.

*Supported by NSF grant 0915956 and an Oracle Corp gift.

DaMoN'14, June 22 - 27 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2619228.2619234.

Common approaches on mainstream architectures include the use of SIMD instructions for scans [12, 19] and the use of partitioning for creating cache-resident sub-problems to avoid random memory accesses [13]. Data compression is also important [18, 19], as it allows us to process more tuples with the same number of instructions using the same registers. Besides scans, other database operations, such as sorting [5, 16], are also made faster using SIMD. However, these operations have the common property of sequential input access. The question whether SIMD is helpful remains open for operations that require random access patterns. To bridge the gap, mainstream hardware now offers non-contiguous SIMD load instructions, termed *gathers*, that allow random memory accesses through entirely SIMD code.

A problem that stands in the middle ground between random accesses and sequential scans is Bloom filter probing. A Bloom filter is a probabilistic data structure for testing whether an item belongs to a set. Bloom filters are crucial to analytical databases for performing joins between tables that have vastly different cardinalities. The keys of the small table are used to build the Bloom filter and the keys of the large table are probed through the filter to discard (most of) those that do not match. In distributed query execution, Bloom filters are used to filter tuples before sending them over the network. The process of filtering across tables, approximately or not, is termed semi-join. The items used to build the filter are relatively few compared to the items probed through the filter to test set membership. Thus, Bloom filter performance is typically dominated by probing.

Bloom filters are built using a pre-determined number k of hash functions. To test whether an item belongs to the set, we need to test k bits in different locations of the filter. The locations are determined by the k hash functions and do not need to be distinct. If any bit (out of k) is not set, the key is certainly not part of the qualifying set. A Bloom filter should be small to be cache-resident if possible. Increasing the number hash functions is not always helpful. If the Bloom filter size is m bits and is built using n distinct items, we can estimate the false positive error probability using the formula: $p = (1 - e^{-kn/m})^k$, which has a single (global) minimum for a small integer k. On the other hand, using more hash functions can be slower than using more bits. Overall, the DBMS should pick the fastest configuration. Each configuration includes a range of sizes and a number of functions and we can profile all cases for the underlying hardware off-line. Then, the optimizer can use the set cardinality and the desired error rate to decide the most suitable one to use for the target point of the query plan.

In this paper, we present an efficient implementation of Bloom filter probing using SIMD instructions. On top of using gather instructions, we show how conditional control-flow can be transformed to data-flow using sophisticated techniques, which are generalizable for transforming other database operations into SIMD code. We chose the Bloom filter algorithm as it combines three properties: it (i) relies on non-contiguous accesses in the filter, so gathers become crucial; (ii) relies on branching logic to early abort keys that failed a function; (iii) is likely to remain cache-resident due to its optimized size and scope of use, so is less likely to be bound by the bottleneck of random memory accesses.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 describes the implementation of Bloom filters and the process of transforming control-flow scalar code into data-flow SIMD code. Section 4 is our experimental evaluation and we conclude in Section 5.

2. RELATED WORK

Bloom filters [3] have been widely studied. Fan et.al. [9] proposed *counting* Bloom filters that can delete certain items without re-creating the entire filter. Cohen et.al. [6] proposed *spectral* Bloom filters that can filter items whose multiplicity is less than a certain threshold. Chazelle et.al. [4] proposed *Bloomier* filters that allow the evaluation of functions beyond the set membership. Pagh et.al. [14] proposed an optimal replacement for Bloom filters that has better complexity and requires $\approx 30\%$ less space. Deng et.al. [7] proposed *stable* Bloom filters that are used for approximate duplicate elimination in data streams. Almeida et.al. [1] proposed *scalable* Bloom filters that can adapt dynamically to the number of elements stored. Beyer et.al. [2] proposed partitioning into cache-line sized Bloom filters and Putze et.al. [17] proposed faster and more accurate configurations. Our vectorized implementation can be applied to several of these techniques. In this paper, we use the original Bloom filter algorithm [3] as it is sufficient for analytical databases.

Zhou et.al. [20] showed that SIMD instructions can accelerate several database operations. Wassenberg et.al. [19] used SIMD to efficiently decompress n-bit-wide tuples that are dictionary compressed. Lemire et.al. [12] discussed a comprehensive menu of techniques for integer compression using SIMD. Inoue et.al. [10] proposed an algorithm that uses SIMD comb-sort in cache and merges using SIMD out of cache. Chhugani et.al. [5] accelerated merge-sort by using bitonic networks to sort on SIMD registers. Kim et.al. [11] improved main-memory indexes by designing multi-way trees accessed through SIMD. Polychroniou et.al. [15] proposed updating the aggregates of heavy hitters using SIMD and in recent work [16] designed a SIMD-based range index to accelerate range partitioning and comparison sorting.

3. IMPLEMENTATION

First, we present a brief overview of the Bloom filter algorithm and show an optimized scalar implementation. Then, we proceed step by step to implement a vectorized version of the code that converts control flow into data flow.

The hash functions we use are based on multiplicative hashing. Multiplicative hashing has been proved universal [8] and is among the fastest known universal hash functions. Assuming that the Bloom filter has 2^n bits, we compute the hash function using one multiplication and one shift. We ensure that the hash functions are pairwise independent by picking multiplicative factors whose differences are not divisible by high powers of two. More expensive hash functions slow down the Bloom filter without offering any benefits.

In the scalar implementation of Bloom filters, we load one key at a time and check hash functions iteratively. If a bit is not set, we discard the key and proceed to the next. To test a specific bit in the bitmap, we use the x86 instruction `bt` and avoid doing shift and bitwise-and to separate the word index and the bit offset. The `bt` instruction is not accessible directly in C/C++; thus, we use (GCC) inline assembly.

```
for (o = i = 0 ; i != size ; ++i) {
  // load key and check one function at a time
  key = keys[i];
  fun = 0;  // the hash function index
  do {
    hash = key * factors[fun];
    hash >>= shift;
    // jump to failure label if bit "hash" is not set
    asm goto ("btl  %1, (%0)\n\t" // test target bit
              "jnc  %l[failure]"  // jump if not set
    :: "r"(filter),     // the bitmap address
       "r"(hash)        // the bit index
    : "cc" : failure);  // the jump label
  } while (++fun != functions);
  // tuple qualifies since all bits are set
  pays_out[o] = pays[i];
  keys_out[o++] = key;
  failure:; }
```

To avoid the second branch associated with testing the number of hash functions, we can remove the inner loop and put k branches that test the different hash functions. This approach increases the code size but has two benefits. First, all multiplicative factors are kept in registers and are not loaded from the L1 cache. Second, each function uses a separate branch instruction for which the hardware keeps a separate branch prediction state. We expect the n-th function to fail more often than the $(n+1)$-th function and the prediction rate increases as a key passes more functions.

```
for (o = i = 0 ; i != size ; ++i) {
  // load key and check one function at a time
  key = keys[i];
  hash = key * factor_1;  // 1st function
  hash >>= shift;
  // 1st function branch (1st prediction state)
  asm goto ("btl  %1, (%0)\n\t"
            "jnc  %l[failure]"
  :: "r"(filter), "r"(hash) : "cc" : failure);
  hash = key * factor_2;  // 2nd function
  hash >>= shift;
  // 2nd function branch (2nd prediction state)
  asm goto ("btl  %1, (%0)\n\t"
            "jnc  %l[failure]"
  :: "r"(filter), "r"(hash) : "cc" : failure);
  [...]  // hard-code more functions if applicable
  // tuples qualifies since all bits are set
  pays_out[o] = pays[i];
  keys_out[o++] = key;
  failure:; }
```

Bloom filter probing is based on random accesses in the filter. The fundamental mismatch with existing SIMD approaches is that in the Bloom filter we need only one bit per access, while a sequential SIMD load would give us a large vector of bits. Instead, to perform random accesses we use SIMD *gather* instructions, shown in Figure 1, that allow loading data from non-contiguous memory locations.

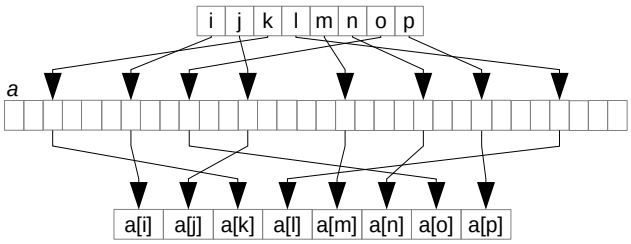

Figure 1: SIMD gather instruction illustration

To perform a bit-test with SIMD code, we gather the minimum number of bytes possible per access and we create a bit offset vector to test only one bit per loaded item. For instance, if each access is a 32-bit load, we first divide the location by 32 and use the address to load 32-bit words. Also, we create a vector where every 32-bit word has only a single set bit, based on the location modulo 32. We bitwise-and the two vectors to isolate the target bit and compare the resulting 32-bit masks with zero. The comparison generates a mask that determines which items are to be aborted.

To re-write Bloom filters using SIMD code, we need to transform the control flow logic into data flow logic. Testing all hash functions per key, before deciding whether the tuple qualifies or not, is a wasteful approach; it precludes early aborting tuples that failed a bit test. To maintain this property, we must never vectorize accesses for the same key.

To vectorize accesses across keys, every gather instruction should check one hash function for W different keys, given that our SIMD vectors hold W keys, but not necessarily the same function across the keys. Since some keys will fail earlier than others and some keys will not fail at all, we do not distinguish cases using control-flow based code. We cannot assume a specific pattern of failures, as the flow is dependent on both the input data and the configuration.

Multiplicative hashing, on top of being fast, is also convenient due to its simplicity for computing arbitrary hash functions on each SIMD lane. By changing the multiplicative factor we change the hash function. Having different multiplicative factors across the W SIMD lanes, allows us to compute different hash functions. In order to choose which hash function to use per key, we keep a function index vector. If $k \leq W$, we create the multiplicative factor mask using a SIMD permutation instead of a gather.

To operate on W different keys, we fill the key vector with new keys if some keys failed in the previous loop. The basic SIMD instruction we need is the ability to shuffle a register using lane indexes from another register. This instruction is provided, either directly or indirectly, by all SIMD instruction sets. After we shuffle the key register, the left part of the register will contain all qualifying tuples, and the right part of the register will contain non-qualifying tuples that will be overwritten with new tuples from the input.

For example, if our vector is `[A,B,C,D,E,F,G,H]`, and the condition result (boolean) vector is `,[T,F,T,F,T,F,F,F]`, we re-order the input vector to `[B,D,F,G,H,A,C,E]`. Note that `T` represents the keys that fail the filter, not the intuitive opposite. For this operation, we need the permutation mask `[1,3,5,6,7,0,2,4]`. This case is also shown in Figure 2. The permutation mask is loaded from a lookup table using the boolean vector bit-mask as the index. If the SIMD vector holds W items, the lookup table must have 2^W entries and remains L1 cache-resident for small SIMD lengths ($W \leq 8$).

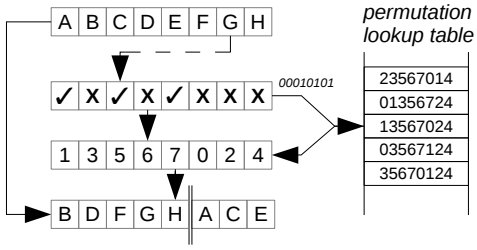

Figure 2: SIMD permutation process illustration

The keys that fail are shuffled to the right end of the SIMD register and are overwritten with new keys read from the input in the next loop. The same permutation is applied for payloads, the boolean vector of aborts, and the hash function indexes. The hash function index of the new keys is reset. The vectorized (Intel AVX 2) code is shown below, assuming a 32-bit key column and a 32-bit payload column:

```
inv = _mm256_cmpeq_epi32(inv, inv);
for (o = i = 0 ; i + 8 <= size ;) {
  // load new items (right side has new items)
  new_key = _mm256_maskload_epi32(&keys[i], inv);
  new_pay = _mm256_maskload_epi32(&pays[i], inv);
  // invalidate aborted keys and reset functions
  key = _mm256_andnot_si256(inv, key);
  pay = _mm256_andnot_si256(inv, pay);
  fun = _mm256_andnot_si256(inv, fun);
  // mix new tuples with old tuples
  key = _mm256_or_si256(key, new_key);
  pay = _mm256_or_si256(pay, new_pay);
  // hash each key with its current hash function
  fact = _mm256_permutevar8x32_epi32(factors, fun);
  hash = _mm256_mullo_epi32(key, fact);
  hash = _mm256_srl_epi32(hash, shift);
  // increment hash function index and check if done
  fun = _mm256_add_epi32(fun, mask_1);
  last_fun = _mm256_cmpeq_epi32(fun, functions);
  // access bitmap and determine which keys fail
  div_32 = _mm256_srli_epi32(hash, 5);
  div_32 = _mm256_i32gather_epi32(filter, div_32, 4);
  mod_32 = _mm256_and_si256(hash, mask_31);
  mod_32 = _mm256_sllv_epi32(mask_1, mod_32);
  res = _mm256_and_si256(div_32, mod_32);
  inv = _mm256_cmpeq_epi32(res, mask_0);
  // branch to output qualifying tuples (rare branch)
  if (!_mm256_testz_si256(res, last_fun)) { [...] }
  inv = _mm256_or_si256(inv, last_fun);
  // load permutation mask
  m = _mm256_movemask_ps(_mm256_castsi256_ps(inv));
  perm_comp = _mm_loadl_epi64(&perm_table[m]);
  perm = _mm256_cvtepi8_epi32(perm_comp);
  // permute keys, payloads, aborts, and functions
  inv = _mm256_permutevar8x32_epi32(inv, perm);
  fun = _mm256_permutevar8x32_epi32(fun, perm);
  key = _mm256_permutevar8x32_epi32(key, perm);
  pay = _mm256_permutevar8x32_epi32(pay, perm);
  // update the input index (the only scalar part)
  i += _mm_popcnt_u64(m); }
```

We give a brief description of relevant SIMD instructions for convenience. `_mm256_mask{load,store}_epi32` conditionally loads and stores up to 8 32-bit integers. Bit operations are done by `_mm256_{andnot,and,or}_si256`. Fixed-stride right shifts are done by `_mm256_{srl,srli}_epi32` and `_mm256_sllv_epi32` performs variable-stride left shift. `_mm256_{add,mullo,cmpeq}_epi32` perform 32-bit addition, multiplication, and equality check. `_mm256_movemask_ps` extracts the bit-mask, `_mm_loadl_epi64` loads a 64-bit mask,

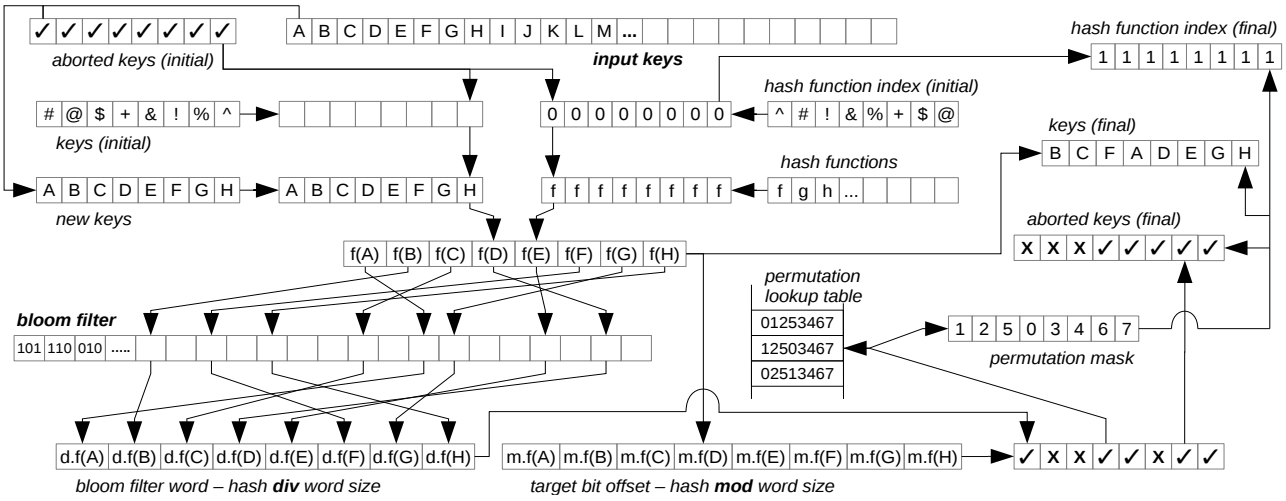

Figure 3: 1st loop of vectorized Bloom filter probing (32-bit key, no payload) without the output code

`extended by_mm256_cvtepi8_epi32` to a 256-bit mask, and `_mm256_permutevar8x32_epi32` performs the SIMD permutation. `_mm256_testz_si256` tests if all bits are zero, and `_mm_popcnt_u64` is a scalar instruction that counts set bits. More extensive descriptions of the instructions can be found in the corresponding instruction reference manuals.[1]

Vector `key` holds the probed keys and `pay` holds their payloads. `inv` is the boolean vector of aborts, and `fun` stores the hash function indexes. All other variables are either constant (`mask_{0,1,31}`, `functions`, `factors`) or temporary.

Figures 3 and 5 show a possible first and second loop of our approach. The example omits payloads and the output generation for simplicity. In both the code and the example, each 256-bit register holds eight 32-bit keys ($W = 8$). The tick and `X` marks represent boolean results. The permutation masks are stored in a hard-coded array with 256 entries of 64-bit numbers holding one permutation index per byte.

Since we expect the number of output tuples to be significantly smaller than the input, executing the output generation code in every loop is wasteful. A branch that is rarely taken should not be eliminated. Since each tuple must go through all hash functions before it can pass the filter, we expect the branch to be taken in $(f + e) \cdot W/k$ loops, where f is the rate of qualifying tuples and e is the false positive rate. The code that generates the output (filling `[...]` in the code segment from the previous page) is shown below:

```
// who checked the last function and succeeded it
ok = _mm256_andnot_si256(inv, last_fun);
// load permutation mask using the inverse
m = _mm256_movemask_ps(_mm256_castsi256_ps(ok));
perm_comp = _mm_loadl_epi64(&perm_table[m ^ 255]);
perm = _mm256_cvtepi8_epi32(perm_comp);
// permute mask, keys, and payloads
ok = _mm256_permutevar8x32_epi32(ok, perm);
key_out = _mm256_permutevar8x32_epi32(key, perm);
pay_out = _mm256_permutevar8x32_epi32(pay, perm);
// write qualifying items to output
_mm256_maskstore_epi32(&keys_out[o], ok, key_out);
_mm256_maskstore_epi32(&pays_out[o], ok, pay_out);
o += _mm_popcnt_u64(m);
```

[1] `http://software.intel.com/sites/landingpage/IntrinsicsGuide/`

The loop stops when the remaining keys are not enough to fill the lanes of the input SIMD register. Some tuples may have not finished checking all hash functions, but these tuples are few ($\leq W$) and we can use scalar code for them.

Note that the SIMD code is still *stable*, which can be useful for not disturbing the order if the input is sorted. The qualifying tuples have the same order as in the input because our permutation masks always perform stable permutations.

The order of the SIMD instructions is important to minimize latencies as much as possible. While the order of instructions should be irrelevant for a sophisticated compiler that has access to instruction latencies, sometimes instruction re-ordering can affect performance by a small margin. Our code descriptions use the simplest, not the fastest order.

In some cases, even using the optimal instruction ordering and a CPU that supports out-of-order execution, we still cannot fully utilize the pipeline. In these cases, we can increase the instructions-per-cycle (IPC) by manually unrolling the loop. The SIMD code we presented has a number of data dependencies that forbid a simple unrolling of the loop. The best approach is to read the input from both the start and the end, as shown in Figure 4. This way, we are able to issue two instances of each instruction, one for the low-to-high data and one for the high-to-low data. The loop stops when the two input vectors overlap to avoid reading the same item twice. The number of items that we handle with scalar code is still small ($\leq 2W$). The overall operator is no longer stable, but can generate two stable output parts.

Loop unrolling allows us to issue multiple instances of each instruction in an interleaved fashion, minimizing latencies and increasing the IPC. On the other hand, we need to have enough CPU registers available to hold the multiple states before the compiler starts spilling registers.

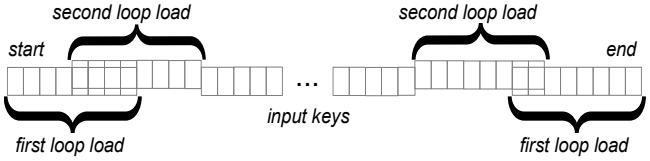

Figure 4: Processing the input from both directions

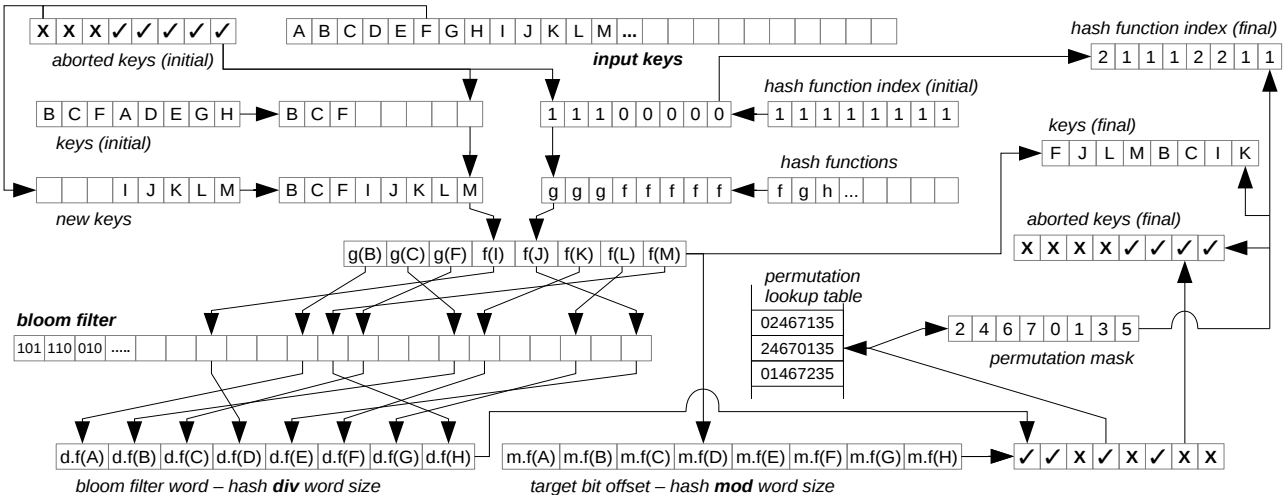

Figure 5: 2nd loop of vectorized Bloom filter probing (32-bit key, no payload) without the output code

4. EXPERIMENTAL EVALUATION

All experiments were executed on the same platform with one Intel Xeon E3-1275 v3 CPU at 3.5 GHz based on the Haswell micro-architecture with 32 GB dual-channel DDR3 ECC RAM at 1600 MHz. The CPU has 4 physical cores and supports 2-way simultaneous multi-threading (SMT). Each core has private access to a 32 KB L1 data cache and a 256 KB L2 cache. The L3 cache is 8 MB and is shared across the chip. We compile with GCC 4.8 using `-O3` optimization and the OS is Linux 3.11. All data used in experiments are synthetically generated and follow the uniform distribution, which represents an average (not especially favorable) case. The source code used for all experiments is available online.[2]

Figure 6 shows the throughput of Bloom filter probing when the Bloom filter is cache-resident. The figure is split in three parts and shows results for L1, L2, and L3 cache-resident filters separately and the filter is 16 KB, 128 KB and 2 MB respectively. The number of items in the filter is 1/10 of the filter bits. The percentage of tuples that qualify is 5% without including the false positives, representing a medium case between very small and large selectivities. The output is materialized to memory and is small compared to the input. The false positive rate depends on the number of bits per item and the number of hash functions. The false positive rates for 1–6 hash functions with 10 bits per item are 9.52%, 3.29%, 1.74%, 1.18%, 0.94%, and 0.84% respectively.

The scalar versions shown either loop over the hash functions (soft) or hard-code them (hard), as shown in Section 3. The SIMD versions either read the input from one direction (single) or from both directions (double) to unroll the loop. The bandwidth line shows the time required to read the input and materialize the same output (with errors) without probing and thus represents an upper bound on performance. We use all available hardware threads of the CPU.

The SIMD code is up to 3.3X faster than the scalar code when the Bloom filter is resident in either the L1 or the L2 cache. The L1-resident filter is up to 13% faster than the L2-resident filter. In all cases, we assume that the filter is half the capacity of the cache, because the probed tuples pollute the cache when read. The L3-resident filter is less than half as fast as the L2-resident filter, but the vectorized version remains up to 2.1X faster than the scalar version. Unrolling the loop by reading the input from both the start and the end helps by up to 30%. The speedup of SIMD over scalar code increases with the number of hash functions, when the filter becomes more accurate. For the vectorized filter, the performance decrease from using 6 functions instead of 1 is less than 30%, while the false positive rate can be prohibitive

[2] `http://www.cs.columbia.edu/~orestis/vbf.c`

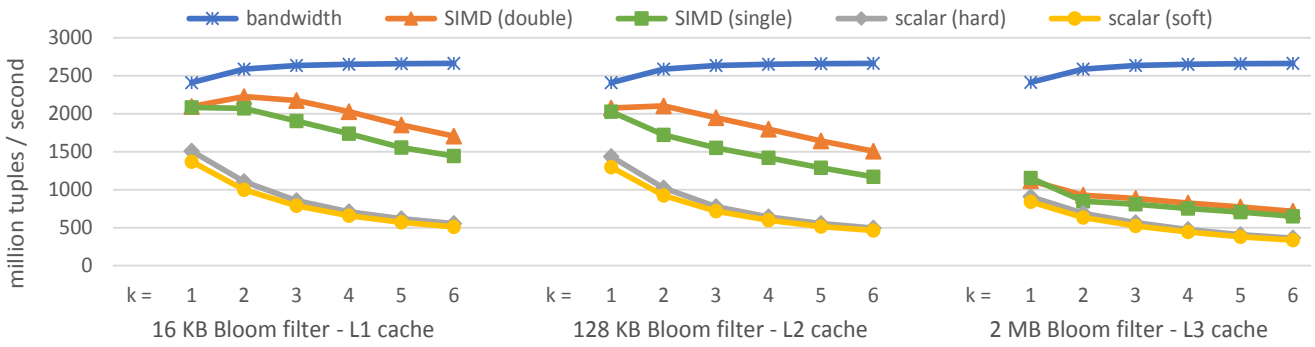

Figure 6: Probing throughput by varying the number of hash functions (and error rate) with cache-resident Bloom filter across the cache levels (32-bit keys, 32-bit payloads, 10 bits per item, 5% of tuples qualify)

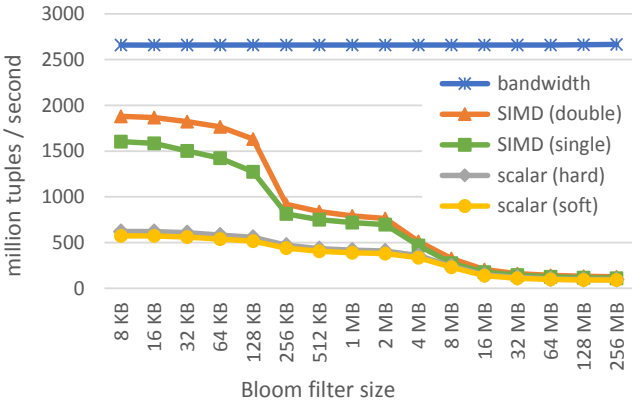

Figure 7: Probing throughput by varying the Bloom filter size (32-bit keys, 32-bit payloads, 10 bits per item, 5 hash functions, 5% of tuples qualify)

when the rate of qualifying tuples is small (e.g., 5% of tuples qualify and 9.52% errors for $k = 1$). The bandwidth bottleneck improves with k because we write fewer false positives.

Figure 7 shows the throughput for Bloom filters of various sizes. The performance first drops when the filter exceeds the capacity of the private caches (L1 and L2). At the L3 cache, performance gradually decreases when the Bloom filter exceeds 2 MB. The filter is accessed read-only during probing, thus, can be shared across threads (and SMT threads) without cache invalidations. The SIMD code is 39% faster compared to scalar code when operating off-cache.

We focus more on cache-resident Bloom filters because probing large Bloom filters can be solved by partitioning the filter [2, 17] into small parts that are cache resident or have the size of a cache line. Adding a partitioning phase incurs known costs [16] and is orthogonal to our approach. An observation that stems from our work is that Bloom filters can be fast when resident in the private caches, so, we can avoid partitioning to filters of one cache line. We also note that when the small size of Bloom filters is unimportant, a large hash table can perform the same task with no false positives and roughly the same cost of one cache line access, given that the load factor is small and collisions are rare.

Figure 8 shows the probing throughput across selectivity rates using an L2 cache-resident Bloom filter. The scalar versions are marginally affected as they are latency-bound. Vectorized Bloom filter probing at a low selectivity runs close to the memory bandwidth and is faster than copying everything to output, as shown by the bandwidth case.

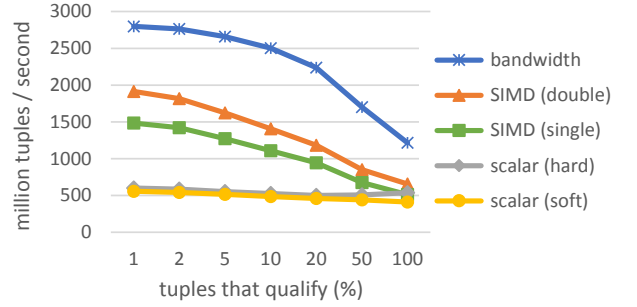

Figure 8: Probing throughput by varying qualifying tuples rate (32-bit keys, 32-bit payloads, 128 KB Bloom filter, 10 bits per item, 5 hash functions)

5. CONCLUSION

We presented vectorized Bloom filters, implemented using the latest SIMD instructions that support non-contiguous loads termed *gathers*. We transform the conditional control flow into data flow without discarding the branching logic of early aborting items that failed the Bloom filter. Our evaluation shows that vectorized Bloom filter probing has 1.4X to 3.3X better performance and the improvement is maximized when the Bloom filter is cache-resident. Our techniques are generic and can be adjusted for optimizing other in-memory database operations that require random accesses. Finally, vectorized code performance will improve every time the SIMD length is increased in future hardware.

6. REFERENCES

[1] P. S. Almeida et al. Scalable Bloom filters. *Information Processing Letters*, 101(6):255–261, Mar. 2007.

[2] K. S. Beyer and S. Rajagopalan. System and method for generating a cache-aware Bloom filter, Oct. 2011. US Patent 8,032,732 B2 (Filed: June 5, 2008).

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[4] B. Chazelle et al. The Bloomier filter: An efficient data structure for static support lookup tables. In *SODA*, pages 30–39, 2004.

[5] J. Chhugani et al. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *VLDB*, pages 1313–1324, 2008.

[6] S. Cohen and Y. Matias. Spectral Bloom filters. In *SIGMOD*, pages 241–252, 2003.

[7] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable Bloom filters. In *SIGMOD*, pages 25–36, 2006.

[8] M. Dietzfelbinger et al. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1), 1997.

[9] L. Fan et al. Summary cache: A scalable wide-area web cache sharing protocol. Technical report, University of Wisconsin-Madison, 1998.

[10] H. Inoue et al. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT*, pages 189–198, 2007.

[11] C. Kim et al. Fast: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[12] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, May 2013.

[13] S. Manegold et al. What happens during a join? dissecting cpu and memory optimization effects. In *VLDB*, pages 339–350, 2000.

[14] A. Pagh et al. An optimal Bloom filter replacement. In *SODA*, pages 823–829, 2005.

[15] O. Polychroniou et al. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN*, 2013.

[16] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, 2014.

[17] F. Putze et al. Cache-, hash-, and space-efficient Bloom filters. *J. Experimental Algorithmics*, 14:4.4–18, Jan. 2010.

[18] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, Aug. 2013.

[19] T. Willhalm et al. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, Aug. 2009.

[20] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.