Parallel Prefix Sum with SIMD

Wangda Zhang
Columbia University
zwd@cs.columbia.edu

Yanbin Wang
Columbia University
yw3372@columbia.edu

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

ABSTRACT

The prefix sum operation is a useful primitive with a broad range of applications. For database systems, it is a building block of many important operators including join, sort and filter queries. In this paper, we study different methods of computing prefix sums with SIMD instructions and multiple threads. For SIMD, we implement and compare horizontal and vertical computations, as well as a theoretically work-efficient balanced tree version using gather/scatter instructions. With multithreading, the memory bandwidth can become the bottleneck of prefix sum computations. We propose a new method that partitions data into cache-sized smaller partitions to achieve better data locality and reduce bandwidth demands from RAM. We also investigate four different ways of organizing the computation sub-procedures, which have different performance and usability characteristics. In the experiments we find that the most efficient prefix sum computation using our partitioning technique is up to 3x faster than two standard library implementations that already use SIMD and multithreading.

1. INTRODUCTION

Prefix sums are widely used in parallel and distributed database systems as building blocks for important database operators. For example, a common use case is to determine the new offsets of data items during a partitioning step, where prefix sums are computed from a previously constructed histogram or bitmap, and then used as the new index values [6]. Applications of this usage include radix sort on CPUs [30, 25] and GPUs [29, 20], radix hash joins [16, 3], as well as parallel filtering [32, 4]. In OLAP data cubes, prefix sums are precomputed so that they can be used to answer range sum queries at runtime [15, 10, 21]. Data mining algorithms such as K-means can be accelerated using parallel prefix sum computations in a preprocessing step [18]. In data compression using differential encoding, prefix sums are also used to reconstruct the original data, and prefix-sum computations can account for the majority of the running time [22, 23].

The prefix sum operation takes a binary associative operator $\oplus$ and an input array of n elements $[a_0, a_1, \ldots, a_{n-1}]$, and outputs the array containing the sums of prefixes of the input: $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus a_{n-1})]$. This definition is also known as an inclusive scan operation [5]. The output of an exclusive scan (also called pre-scan) would remove the last element from the above output array, and insert an identity value at the beginning of the output. In this paper, we use addition as the binary operator $\oplus$ and compute inclusive scans by default.

The basic algorithm to compute prefix sums simply requires one sequential pass of additions while looping over all elements in the input array and writing out the running totals to the output array. Although this operation seems to be inherently sequential, there are actually parallel algorithms to compute prefix sums. To compute the prefix sums of n elements, Hillis and Steele presented a data parallel algorithm that takes $O(\log n)$ time, assuming there are n processors available [14]. This algorithm performs $O(n \log n)$ additions, doing more work than the sequential version, which only needs $O(n)$ additions. Work-efficient algorithms [19, 5] build a conceptual balanced binary tree to compute the prefix sums in two sweeps over the tree, using $O(n)$ operations. In Section 3, we implement and compare SIMD versions of these data-parallel algorithms, as well as a vertical SIMD algorithm that is also work-efficient.

In a shared-memory environment, we can speed up prefix sum computations using multiple threads on a multicore platform. Prefix sums can be computed locally within each thread, but because of the sequential dependencies, thread t_m has to know the previous sums computed by threads $t_0 \ldots t_{m-1}$ in order to compute the global prefix sum results. Thus, a two-pass algorithm is necessary for multithreaded execution [34]. There are multiple ways to organize the computation subprocedures, depending on (a) whether prefix sums are computed in the first or the second pass, and (b) how the work is partitioned. For example, to balance the work among threads, it is necessary to tune a dilation factor to the local configuration for optimal performance [34]. To understand the best multithreading strategy, we analyze and compare different multithreaded execution strategies (Section 2.1).

More importantly, with many concurrent threads accessing memory, the prefix sum can become a memory bandwidth-bound computation. To fully exploit the potential of the

*This research was supported in part by a gift from Oracle Corporation.

11th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'20), August 31, 2020, Tokyo, Japan.

hardware, we must take care to minimize memory accesses and improve cache behavior. To this end, we propose to partition data into cache-sized partitions so that during a two-pass execution, the second pass can maximize its usage of the cache instead of accessing memory again (Section 2.2). We also develop a low-overhead thread synchronization technique, since partitioning into smaller data fragments can potentially increase the synchronization overhead.

In summary, the main contributions of this work are:

- We study multithreaded prefix sum computations, and propose a novel algorithm that splits data into cache-sized partitions to achieve better locality and reduce memory bandwidth usage.
- We discuss data-parallel algorithms for implementing the prefix sum computation using SIMD instructions, in 3 different versions (horizontal, vertical, and tree).
- We experimentally compare our implementations with external libraries. We also provde a set of recommendations for choosing the right algorithm.

1.1 Related Work

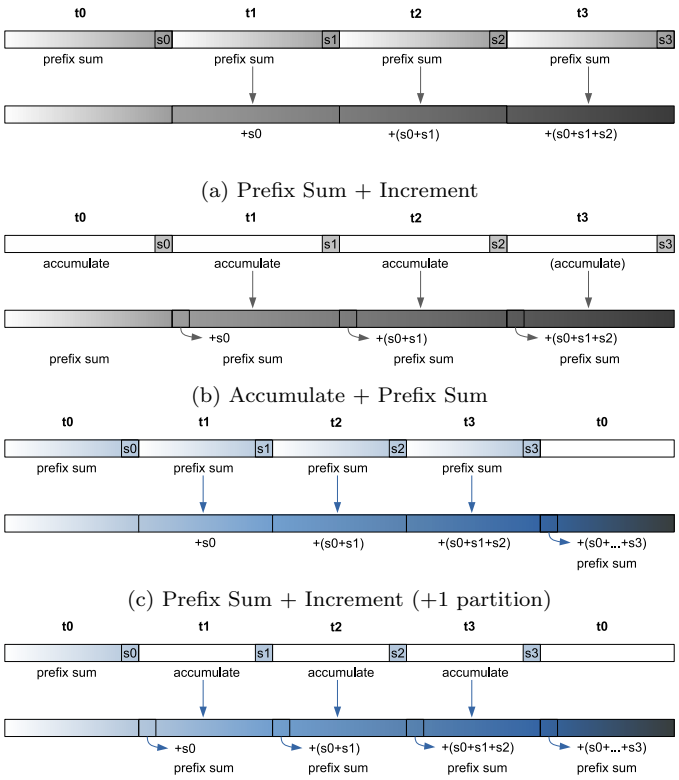
Beyond databases, there are also many other uses of prefix sums in parallel computations and applications, including but not limited to various sorting algorithms (e.g., quicksort, mergesort, radix-sort), list ranking, stream compaction, polynomial evaluation, sparse matrix-vector multiplication, tridiagonal matrix solvers, lexical analysis, fluid simulation, and building data structures (graphs, trees, etc.) in parallel [6, 5, 31].

Algorithms for parallel prefix sums have been studied early in the design of binary adders [19], and analyzed extensively in theoretical studies [8, 7]. More recently, message passing algorithms were proposed for distributed memory platforms [28]. A sequence of approximate prefix sums can be computed faster in $O(\log \log n)$ time [11]. For data structures, Fenwick trees can be used for efficient prefix sum computations with element updates [9]. Succinct indexable dictionaries have also been proposed to represent prefix sums compactly [26].

As an important primitive, the prefix sum operation has been implemented in multiple libraries and platforms. The C++ standard library provides the prefix sum (scan) in its algorithm library, and parallel implementations are provided by GNU Parallel library [33] and Intel Parallel STL [2]. Parallel prefix sums can be implemented efficiently on GPUs with CUDA [12], and for the Message Passing Interface (MPI-Scan) [28]. Although compilers can not typically auto-vectorize a loop of prefix sum computation because of data dependency issues [24], it is now possible to use OpenMP SIMD directives to dictate compilers to vectorize such loops. Since additions over floating point numbers are not associative, the result of parallel prefix sums of floating point values can have a small difference from the sequential computation.

2. THREAD-LEVEL PARALLELISM

We now describe multithreaded algorithms for computing prefix sums in parallel. Note that a prefix-sum implementation can be either in-place, where the prefix sums replace the input data elements, or out-of-place, where the prefix sums are written to a new output array. The following description assumes an in-place algorithm. Out-of-place versions can be implemented similarly and will be discussed in the experimental evaluation.



(a) Prefix Sum + Increment

(b) Accumulate + Prefix Sum

(c) Prefix Sum + Increment (+1 partition)

(d) Accumulate + Prefix Sum (+1 partition)

Figure 1: Multithreaded two-pass algorithms

2.1 Two-Pass Algorithms

To compute prefix sums in parallel, we study four different two-pass algorithms. There are two ways of processing the two passes depending on whether prefix sums are computed in the first or second pass. Figure 1(a) presents one implementation, using 4 threads ($t_0 \dots t_3$) as an example. To enable parallel processing, the data is divided into 4 equal-sized partitions. In the first pass, each thread computes a local prefix sum over its partition of data. After the prefix sum computation, the last element is the total sum of the partition, which will be used in the second pass to compute global prefix sums. These sums can be stored in a temporary buffer array $sums = \{s_0, s_1, s_2, s_3\}$, so it is easier to compute the prefix sums of their own. In the second pass, except for the first thread t_0, each thread t_m increments every element of the local prefix sum by $\sum_{i<m} s_i$, in order to obtain global prefix sums. The prefix sums of the first partition are already the final results, since no increment is needed.

A potential inefficiency of this algorithm is that in the second pass, the first thread $t0$ is idle, so effectively the parallelism is only $(m-1)$ using m threads, which has an impact on performance especially when m is small. To fix this issue, the data can be divided into $(m+1)$ partitions, as shown in Figure 1(c) when $m = 4$. In the first pass, the threads only work on the first 4 partitions, and the prefix sums of the last partition is computed in the second pass. We schedule thread $t0$ to work on the last partition, instead of shifting every thread to the next partition, since this is important for our partitioning technique in Section 2.2.

Figure 1(b) demonstrates a different way of computing

the prefix sums. In the first pass, only the total sum of each partition is accumulated in parallel. Then in the second pass, the global prefix sums can be directly computed in parallel, using $\sum_{i<m} s_i$ as an offset to the input. The benefit of computing only the total sum in the first pass is that there is no memory write as in prefix sum computations. Therefore, this method can potentially require less memory bandwidth when the data size is large. In addition, we do not need the total sum of the last partition, so the last thread $t3$ can be idle in the first pass.

Similarly, Figure 1(d) fixes the idle-thread inefficiency by using one more partition. In the first pass, the prefix sums of $t0$ and the totals sums of $t1 \ldots t3$ are computed in parallel. In the second pass, all threads compute prefix sums with an input offset computed from the *sums* array. Thread $t0$ again works on the last partition. Both partitioning schemes in Figures 1(c) and 1(d) ensure there are no idle threads and all threads work in parallel in either pass.

2.1.1 Load Balancing

For algorithms shown in Figures 1(a) and 1(b), equal partitioning of the data should suffice since each thread does the same work (except for the idle thread). For Figure 1(c), in the second pass, thread $t0$ computes prefix sums while other threads simply do an increment. Although in scalar code, both subprocedures require read, add and write, the increment is easily vectorizable by compilers while the prefix sum cannot be automatically vectorized. (We will also implement explicit SIMD versions of these algorithms.) Similarly, in the first pass of Figure 1(d), thread $t0$ computes a prefix sum while other threads accumulate total sums. These operations may proceed at different rates, and the difference is potentially magnified because prefix sums cannot be autovectorized and need to write back results, while accumulation can be autovectorized and does not write to memory. (The second pass has similar problems when cache-friendly partitioning is used as we shall explain in Section 2.2.)

To compensate for thread $t0$ possibly taking longer, a dilation factor d can be used to reduce the size of the first (or last) partition, in order to balance the work done by every thread [34]. The range of a dilation factor is $d \in [0, 1]$, indicating the ratio of sizes between partition of thread $t0$ to other threads. When $d = 0$, the corresponding partition is nonexistent, so Figure 1(a) is a special case of Figure 1(c) and Figure 1(b) is a special case of Figure 1(d). When $d = 1$, the partitions have equal sizes. In our experiments we find that the dilation factors have to be carefully tuned to achieve best performances, but in practice, standard library implementations typically just use equal-sized partitions by default, which is suboptimal in most cases. In fact, a poorly chosen dilation factor can cause an inefficiency worse than one idle thread, because many threads may become idle as they wait for thread t0 to complete its work.

2.2 Cache-Friendly Partitioning

For big data, one problem with the two-pass algorithms is that after the first pass, most data will have been evicted from the cache, and they have to be accessed again from memory in the second pass. Since the performance difference between cache and memory accesses is quite large, this problem can lead to huge performance impact overall. We therefore propose to partition the entire data in a cache friendly way, so that for each thread, the data it processes reside in cache after the first pass.

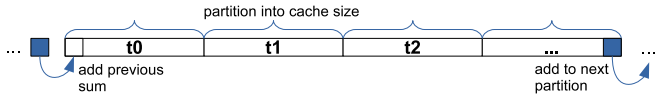


Figure 2: Cache-friendly partitioning

Figure 2 demonstrates this idea, showing that the data are partitioned into cache-sized partitions, and processed by multiple threads in parallel. *Both* passes over the cache resident data are made (in parallel) before proceeding to the next partition. In this way, the second pass reads data from cache rather than from RAM. The entire data is processed sequentially in multiple iterations, but each iteration is computed in parallel by multiple threads. To compute global prefix sums, the first thread $t0$ needs to use the total sum from the previous partition (in the previous iteration) as an offset in the first pass.

If the partitioning uses the methods in Figures 1(a) or 1(b), then the first thread can use the last element of the previous partition as the offset, but this requires the first thread to wait for the second pass of the last partition to complete. A different way is to use the stored *sums* array and compute the sum of all of its elements, including the local sum of last partition, as the offset. This way requires the local sum of the last partition (in the previous iteration) to have been computed, even if it is not necessary to do so using Accumulate in Figure 1(b). However, there are two benefits of doing so: first, the data will reside in cache in the second pass, and second, we can save one synchronization after the second pass as we shall explain shortly. If the partitioning is done as in Figures 1(c) or 1(d) using one more partition, then in the first pass, thread $t0$ can directly access the last element of its previous partition since the same thread $t0$ has processed it.

The effect of partitioning is that during the second pass, accesses to the same data can be served from the cache instead of memory. As we shall demonstrate in the experiments, the size of a partition is better set to roughly half of the size of L2 cache (or L1 cache if SIMD gather/scatter instructions are used in computing local prefix sums). As mentioned earlier, partitioning also changes the dilation factors used in the partitioning scheme in Figures 1(c) and 1(d), because the prefix sum computation in the second pass of the previous iteration will read from cache instead of memory, except for the last partition. For Figure 1(d), we need a second dilation factor for the last partition as well to account for the difference.

2.2.1 Thread Scheduling and Synchronization

To ensure the private cache of a physical core can be accessed during the second pass, we control the thread affinity so that physical threads $t1, t2, \ldots$ continue to process the same data in the second pass. Thread $t0$ works on a different partition, since the first partition does not need a second pass while the last partition does not need the first pass.

We need a barrier synchronization after the first pass, and all threads should wait for it before continuing to the second pass. Without the cache-friendly partitioning, the original two-pass algorithm only needs to synchronize twice: one between the first pass and the second pass, and one to join all the threads after the second pass. With partitioning, more

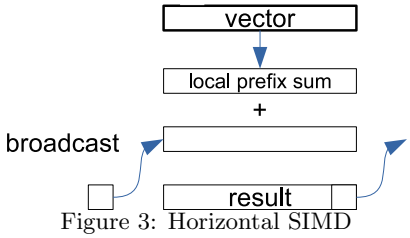


Figure 3: Horizontal SIMD

```
__m512i _mm512_slli_si512(__m512i x, int k) {
  const __m512i ZERO = _mm512_setzero_si512();
  return _mm512_alignr_epi32(x, ZERO, 16 - k);
}
__m512i PrefixSum(__m512i x) {
  x = _mm512_add_epi32(_mm512_slli_si512(x, 1));
  x = _mm512_add_epi32(_mm512_slli_si512(x, 2));
  x = _mm512_add_epi32(_mm512_slli_si512(x, 4));
  x = _mm512_add_epi32(_mm512_slli_si512(x, 8));
  return x; // local prefix sums
}
```

Listing 1: In-register prefix sums with Horizontal SIMD

synchronization points are needed for this multi-iteration process. However, we only need one synchronization in every iteration, instead of two. The synchronization after the second pass is unnecessary, and the second pass of iteration k can be overlapped with the first pass of iteration $(k+1)$. In order to prevent the local sums in the *sums* array of iteration k from being overwritten by sums computed in the first pass of iteration $(k+1)$, we need two *sums* arrays to store the local sums. In iteration k, the sums are stored in $sums_{k\%2}$. Because of the synchronization between the two passes, this double buffering of *sums* is sufficient.

Since we still need one synchronization in every iteration, the overhead of our synchronization mechanism should be as low as possible. As reported by previous studies [13, 27], the latency of different software barrier implementations can differ by orders of magnitude, thus profoundly affecting the overall performance. In our implementation we use a hand-tuned reusable counting barrier, which is implemented as a spinlock with atomic operations. On our experimental platforms with 48 cores, we find its performance is much better than a Pthread barrier and slightly better than the OpenMP barrier. To scale to even more cores, it is potentially beneficial to use even faster barrier implemenations (e.g., tournament barrier [13]). Because of the sequential dependency of prefix sum computations, it is also possible to restrict synchronization to only adjacent threads, rather than using a centralized barrier [35].

3. DATA-LEVEL PARALLELISM

SIMD instructions can be used to speed up the single-thread prefix sum computation. In this paper, we use AVX-512 extensions that have more parallelism than previous extensions. We assume 16 32-bit numbers can be processed in parallel using the 512-bit vectors.

3.1 Horizontal

Figure 3 shows how we can compute prefix sums in a horizontal way using SIMD instructions. The basic primitive is to compute the prefix sum of a vector of 16 elements in register, so that we can loop over the data to compute every 16 elements. The last element of the 16 prefix sums results is then broadcast into another vector, so it can be added to the next 16 elements to compute the global results.

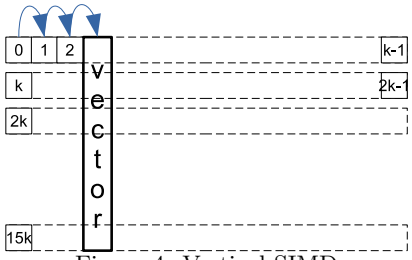


Figure 4: Vertical SIMD

As mentioned in Section 1, [14] presented a data parallel algorithm to compute prefix sums, which can be used to compute the prefix sums in a register. For w elements in a SIMD register, this algorithm uses $\log(w)$ steps. So for 16 four-byte elements, we can compute the prefix sums in-register using 4 shifts and 4 addition instructions. The pseudocode in Listing 1 shows this procedure.

Note that unlike previous SIMD extensions (e.g. AVX2), AVX-512 does not provide the `_mm512_slli_si512` instruction to shift the 512-bit vector. Here, we implement the shift using the `valign` instruction. It is also possible to implement this shifting using permutate instructions [2, 17], but more registers have to be used to control the permutation destinations for different shifts, and it appears to be slightly slower (although `valign` and `vperm` use the same number of cycles).

3.2 Vertical

We can also compute the prefix sums vertically by dividing the data into 16 chunks of length $k = n/16$ and then using a vector to accumulate the running sums, as shown in Figure 4. While scanning over the data, we gather the data at indices $(0+j, k+j, 2k+j, \ldots, 15k+j)$ (where $j \in [0,k)$), add to the vector of running sums, and scatter the result back to the original indices position (assuming the indices fit into four bytes). After one pass of scanning over the data, SIMD lane i contains the total sum of elements from $i*k$ to $(i+1)*k-1$. In a similar way to the multithreaded executions (Section 2), we need to update the results in the second pass.

In addition, we can also switch the two passes by first computing just the total sum of each chunk (without the scatter step) in the first pass, and then compute the global prefix sums vertically in the second pass.

No matter which pass prefix sums are computed in, we need two passes for vertical SIMD computation because of data-parallel processing even if it is executed in a single thread. For multithreaded execution, each of the two passes can run in parallel among the threads. Similar to the cache-friendly partitioning for multithreaded execution, we can also partition the data into cache sizes so that the second pass of vertical SIMD implementation can reuse the data from cache (even for single thread).

Different from the horizontal SIMD version, this vertical algorithm is work-efficient, performing $O(n)$ additions (including updating the indices for gather/scatter) without the $O(\log w)$ overhead. Using data-parallel execution, we essentially execute a two-pass sequential algorithm for each of

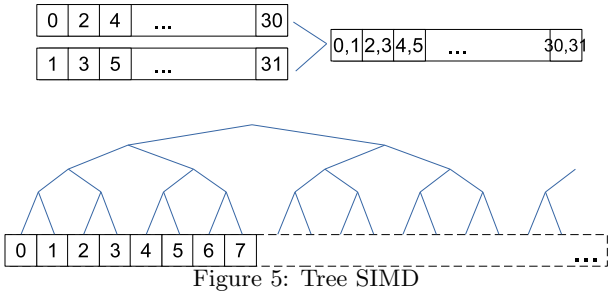


Figure 5: Tree SIMD

the 16 chunks of data. In practice, the $\log w$ extra additions in the horizontal version are computed in register, which is much faster than waiting for memory accesses. Even so, if the gather/scatter instructions are fast, then the vertical algorithm can potentially be faster than the horizontal version.

3.3 Tree

As introduced in Section 1, a work-efficient algorithm using two sweeps over a balanced tree is presented in [6]. As a two-pass algorithm, the first pass performs an up-sweep (reduction) to build the tree, and the second pass performs a down-sweep to construct the prefix sum results. At each level of the tree, the computation can be done in a data parallel way for the two children of every internal tree node. Figure 5 shows the conceptual tree built from the data; for details of the algorithm, refer to the original paper.

To implement this algorithm, we can reuse the `PrefixSum()` procedure in Section 3.1 and mask off the unnecessary lanes in the first pass; the second pass can be done similarly with reversed order of instructions. However, it does not make sense to artificially make the SIMD lanes idle. Instead, we can use a loop of gather/scatter instructions to process the elements in a strided access pattern at every level of the tree. Because the tree is of height $\log n$, the implementation needs multiple passes of gathers and scatters, so although the algorithm is work-efficient in term of addition, it is quite inefficient in memory access. Cache-friendly partitioning helps alleviate this issue for the second pass, but overall this algorithm is not suitable for SIMD implementations.

4. EXPERIMENTS

4.1 Setup

We conducted experiments using the Amazon EC2 service on a dedicated m5.metal instance[1], runing in non-virtualized environments without hypervisors. The instance has two Intel Xeon Platinum 8175M CPUs, based on the Skylake microarchitecture that supports AVX-512. Table 1 describes the platform specifications. The instance provides 300 GB memory, and the measured memory bandwidth is 180 GB/s on two NUMA nodes in total.

We implemented the methods in Sections 3 and 2 using Intel AVX-512 intrinsics and POSIX Threads with atomic operations for synchronization. Our code[2] was written in C++, compiled using Intel compiler 19.1 with `-O3` optimization (scalar code, such as subprocedures Increment and Accumulate, can be autovectorized), and ran on 64-bit Linux operating systems. For our experiemnts, we use 32-bit floating point numbers as the data type, and generate random data so that every thread has 128 MB of data (i.e. 32 million floats) to work with. Algorithms are in-place prefix sum computations, except in Section 4.2.3 where we consider out-of-place algorithms.

[1] `https://aws.amazon.com/intel`

[2] `http://www.cs.columbia.edu/~zwd/prefix-sum`

Table 1: Hardware Specification

Microarchitecture	Skylake
Model	8175M
Sockets	2
NUMA nodes	2
Cores per socket	24
Threads per core	2
Clock frequency	2.5 GHz
L1D cache	32 KB
L2 cache	1 MB
L3 cache (shared)	33 MB

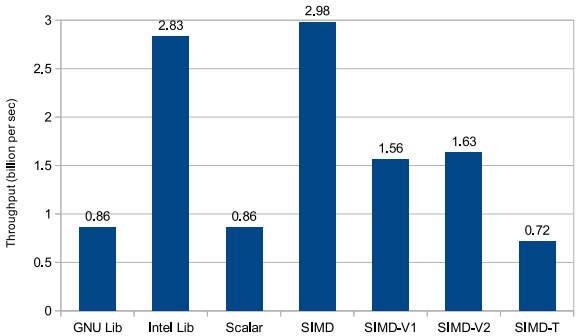


Figure 6: Single-thread throughput

Our handwritten implementations used in the experiments are summarized in Table 2. We compare with the following two external baselines:

- GNU Library [1]. The `libstdc++` parallel mode provides a multithreaded prefix sum implementation using OpenMP. The implementation executes a two-pass scalar computation (Accumulate + Prefix Sum) as discussed in Section 2.
- Intel Library [2]. Intel provides a Parallel STL library, which is an implementation of the C++ standard library algorithms with multithreading and vectorized execution (under the `par_unseq` execution policy). The multithreaded execution is enabled by Intel Threading Building Blocks (TBB), and we can force the Intel compiler to use 512-bit SIMD registers by using the `-qopt-zmm-usage=high` flag (which is set to low on Skylake by default).

4.2 Results

4.2.1 Single-Thread Performance

Figure 6 presents the single-thread throughput results. The (horizontal) SIMD implementation (Section 3.1) using AVX-512 performs the best, processing nearly three billion floats per second. The single-thread Intel library implementation with vectorized execution is slightly slower. Our scalar implementation has the same performance as the scalar GNU library implementation, which is 3.5x slower than SIMD.

Table 2: Algorithm Descriptions

Scalar	Single-thread one-pass scalar implementation
SIMD	Single-thread one-pass horizontal SIMD (Section 3.1). Also used for multithread implementations
SIMD-V1/V2	Single-thread two-pass vertical SIMD (Section 3.2), computing prefix sums in Pass 1 / Pass 2
SIMD-T	Single-thread two-pass tree SIMD (Section 3.3)
Scalar1, SIMD1	Multithread two-pass algorithm, computing prefix sums in Pass 1 (Figure 1(c))
Scalar2, SIMD2	Multithread two-pass algorithm, computing prefix sums in Pass 2 (Figure 1(d))
Scalar*-P, SIMD*-P	Multithread two-pass algorithm with cache-friendly partitioning (Section 2.2)

Using cache-friendly partitioning (with the best partition sizes), the vertical SIMD implementations (Section 3.2) are still about 2x slower than horizontal SIMD. Computing prefix sums in the second pass is slightly better. The Tree implementation (Section 3.3) is slowest even with partitioning because of its non-sequential memory accesses.

4.2.2 Multithread Performance

We then increase the number of threads used to the maximum number of (hyper-)threads supported on the platform. In addition to external baselines, we compare our implementations with and without partitioning. The partition sizes and dilation factors used are the best ones as we shall discuss in Section 4.2.4. Since the vertical SIMD and tree implementations are slow, we omit them in multithreaded experiments.

Figure 7(b) presents the multithreaded throughput results using SIMD. The partitioned SIMD implementations are consistently the best methods across all numbers of threads. SIMD1-P has a throughput of 20.3 billion per second at 48 threads. At this point, it already saturates the memory bandwidth, so its performance does not improve with more than 48 threads used. The highest throughput of SIMD2-P is 20.5 billion per second with 80 threads. Without partitioning, the SIMD1 stablizes at a throughput of 12.3 billion per second. Because the second pass has to access the data from memory (instead of cache) again without partitioning, it is 1.7x slower. SIMD2 has a throughput of 16.7 billion per second, 1.3x faster than SIMD1 because of less memory access in the first pass (no writes).

Figure 7(a) presents the scalar results. Interestingly, although the partitioned scalar implementation Scalar1-P is slow with less than 32 threads, it eventually becomes faster than the non-partitioning SIMD method and also reaches the limit of memory bandwidth at 48 threads. This result shows that the prefix sum converts from a CPU-bound computation to a memory-bound computation as more and more threads are used. In a compute-bound situation (e.g., less than 32 threads), SIMD can improve performance, while in a memory-bandwidth-bound situation, it is more important to optimize for cache locality and reduce memory access. For Scalar2-P, partitioning does not appear beneficial. We think one possible explanation would be that the compiler (or hardware prefetcher) generates nontemporal prefetches (or loads) in the first accumulation pass, meaning that the data is not resident in the cache for the second pass. The non-partitioning scalar implementations are slower than SIMD, but with more threads used, they also become bandwidth-bound and reach the same performance as non-partitioning SIMD.

We also find that library implementations are slower than our handwritten implementations. The Intel library implementation is faster than the GNU library with less than 16 threads, since it uses SIMD implementations, but it appears to not scale well. Even comparing with our non-partitioning implementations, both GNU and Intel implementations have a lower throughput at the maximum system capacity.

4.2.3 Out-of-Place Performance

We now investigate out-of-place computations, where the output goes to a separate array from the input. Figure 8 shows that Scalar2 and SIMD2 perform well; partitioned versions of those algorithms giving about the same throughput, except for scalar code with few threads where the partitioned algorithm performs better. The performance of the GNU library improves for out-of-place computations, but it still performs worse than the Scalar2/Scalar2-P algorithms. While there is little change in the performance of the Scalar1/Scalar1-P and SIMD1/SIMD1-P algorithms when shifting from in-place to out-of-place computations, it appears that the performance of Scalar2/Scalar2-P and SIMD2/SIMD2-P improves.

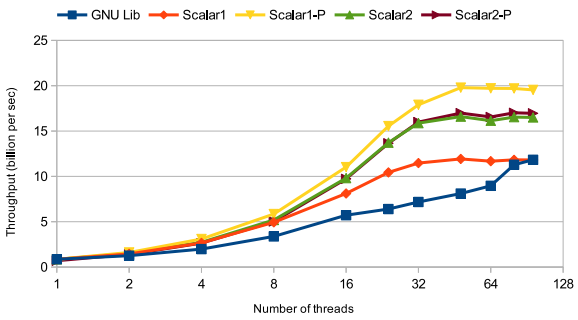
One source of potential improvement is that the out-of-place method is reading from/writing to different memory regions, which may be on different memory banks that can operate in parallel. The in-place method will always be reading/writing to a single memory bank at a time. There are therefore more opportunities for the system to balance the throughput from different memory banks, and achieve higher bandwidth utilization. Support for this hypothesis comes from Figure 9 where we run the algorithms on a single node with a single memory bank. The SIMD versions of SIMD1-P, SIMD2 and SIMD2-P now all perform similarly.

Nevertheless, the scalar performance in Figure 9 shows that there is still a small performance edge for Scalar2/Scalar2-P in the out-of-place algorithms on a single node. We are not certain of the explanation for this effect, but suspect that write-combining buffers may be helping the performance, since the second phase in Scalar2/Scalar2-P does sequential blind writes to the output array.

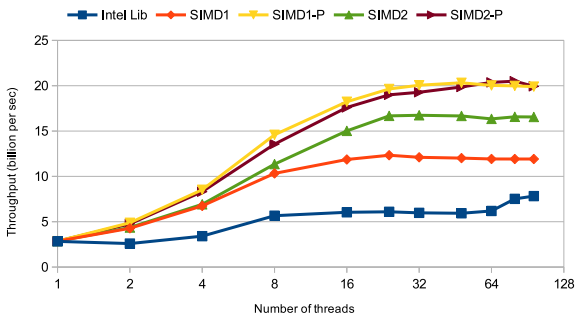
4.2.4 Effect of Partition Sizes

In Figure 10(a), we tune the partition sizes for our partitioned scalar and SIMD implementations with 48 threads. The x-axis is in log scale. By trying out possible partition sizes covering the cache sizes at different levels, we find that with 48 (and fewer) threads, the best partition size is 128K floats per thread, which takes 512 KB, i.e. half the size of L2 caches. With 96 threads, the best choice of partition size is 64K floats per thread, as a result of two hyperthreads sharing the L2 cache. Large partition sizes make caching ineffective, while small partition sizes can increase the synchronization overhead.

Partitioning also helps with the vertical and tree implementations, which need two passes even with a single thread. In Figure 10(b), we find that by partitioning the input data

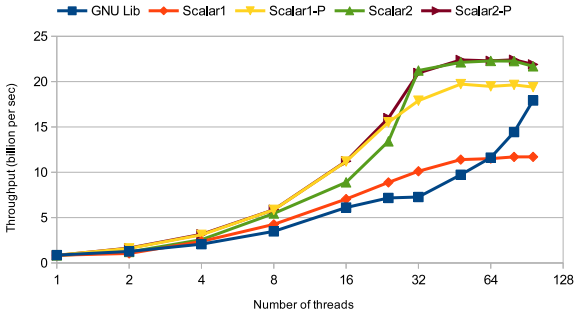


(a) Scalar

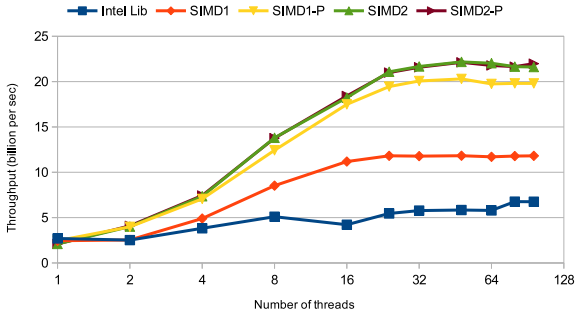


(b) SIMD

Figure 7: Multithread throughput

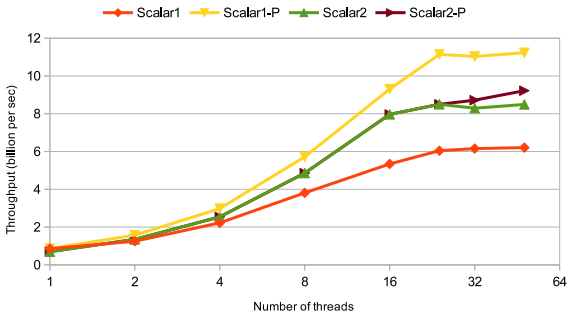


(a) Scalar

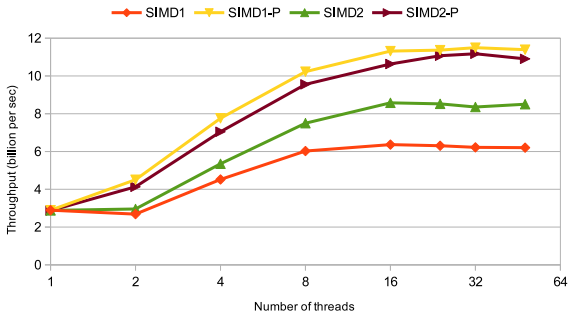


(b) SIMD

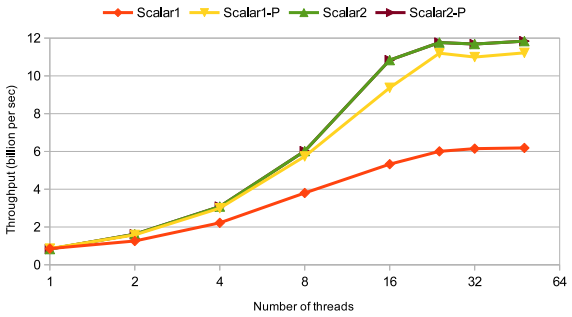
Figure 8: Multithread throughput (out-of-place)



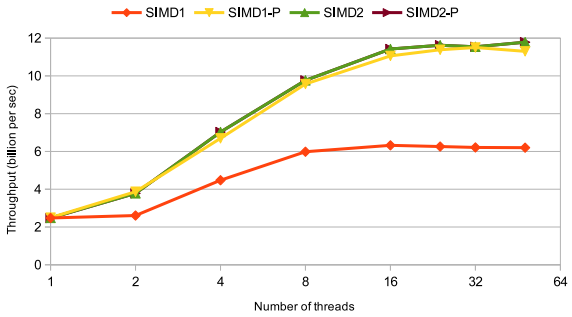
(a) Scalar (in-place)



(b) SIMD (in-place)

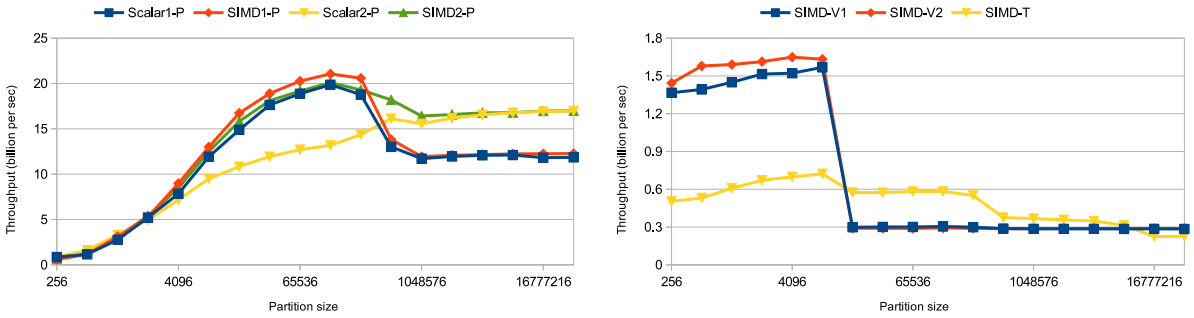


(c) Scalar (out-of-place)



(d) SIMD (out-of-place)

Figure 9: Throughput on a single node



(a) Multithreaded implementations with partitioning (b) Single-thread implementations with partitioning

Figure 10: Effect of partition sizes

into partitions of 8K floats (L1 cache size), we can increase the throughput of SIMD-V1 and SIMD-V2 from 0.3 to about 1.6 billion per second, and the throughput of SIMD-T from 0.2 to 0.7 billion per second. This result shows that partitions fitting into L1 cache is best for the performance of gather and scatter instructions. Even with this enhancement, the throughput is still much lower than other algorithms.

4.2.5 *Effect of Dilation Factors*

Figure 11 demonstrates the effect of dilation factors. We compare equal partitioning (no dilation) with best dilation factors found in our tuning process. In most situtations shown, it is clear that tuning the dilation factor is required to achieve good performance.

The improvement of using one more partition (in every iteration) is largest when the cache-friendly partitioning is not used. Improving thread utilization can indeed improve the performance, especially with a smaller number of threads when the performance is not memory-bound. With cache-friendly partitioning, the difference is fairly small, especially when SIMD is used.

As we have demonstrated that the default dilation $d = 1$ as used in standard library implementations is suboptimal, users should tune this parameter for better performance. However, it is not very convenient to tune this parameter because although it is the ratio of two different subprocedures, the real performance depends on multiple factors in reality. For example, Figure 12 shows the throughput results with varying dilation factors for the Scalar1 implementation. Using different number of threads, the best dilation factor changes from 0.2 to 0.8, reflecting a changing balance of CPU and memory latencies as memory bandwidth becomes saturated.

From the above results, we observe that if we want to use one more partition (Figures 1(c) and 1(d)), then the dilation factors must be tuned and sometimes it is hard to find a fixed best factor across all configurations. On the other hand, since we have almost the same high performance using the cache-friendly partitioning strategy, without using the extra partition in every iteration, it is more robust to use the partitioning schemes in Figures 1(a) and 1(b).

4.2.6 *Effect of High-Bandwidth Memory*

To study the effect of memory bandwidth, we also conducted experiments on a Xeon Phi machine based on the Knights Landing (KNL) architecture. Although it is not a mainstream machine for database query processing, it has a multi-channel RAM (MCDRAM) providing much higher bandwidth than regular RAM (the load bandwidth is 295 GB per second and the store bandwidth is 220 GB per second). Our experimental machine has 64 physical cores and 16 GB MCDRAM. Figure 13 shows the results using 64 threads. It is clear that the cache-friendly partitioning (into L2-cache sized units) helps with every algorithm in regular memory (LBM). In the high bandwidth memory (HBM), we observe a higher throughput for all the methods. However, the partitioning does not improve performance at all, and it is better not to partition with a finer granularity (i.e., the optimal partition size per thread is simply 1/64 of the data size).

To explain these results, observe that there is an overhead to partitioning, including extra synchronization. When data is in RAM, the payoff (reduced memory traffic) is important because the system is memory bound, and so the overhead is worth paying. When the data is in high-bandwidth memory, the system never becomes memory-bound. With bandwidth as an abundant resource, it does not pay to reduce bandwidth needs by doing extra work.

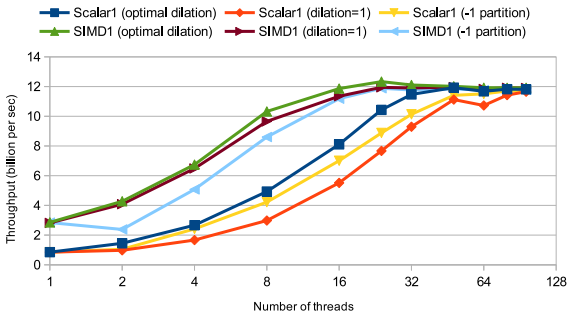
5. CHOOSING THE RIGHT ALGORITHM

We now summarize our results, and make recommendations for the efficient implementation of prefix sum algorithms/libraries.

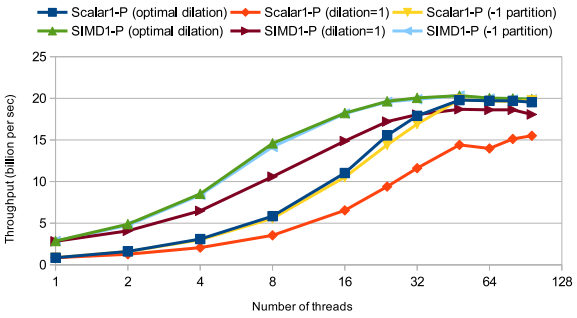
Observation 1. It is hard to get configuration parameters like the dilation factor right. Dilation factors depend on a ratio of subalgorithm speeds that depends on many factors and is hard to calibrate in advance. The performance gains from not wasting one thread are small, particularly if many threads are available, and the risks of suboptimal performance when the wrong dilation factor is used are high. We should also remark that the partitioning approach also needs a configuration parameter to determine the partition sizes. However, the best partition size appears to be easily determined from the L2 cache size, which can be obtained from the system at runtime.

Observation 2. It is worthwhile to pursue bandwidth optimizations like partitioning if bandwidth a bottleneck. Experiments on a machine with high-bandwidth memory show that partitioning helps when the data is in slow RAM, but hurts performance when data is in fast RAM.

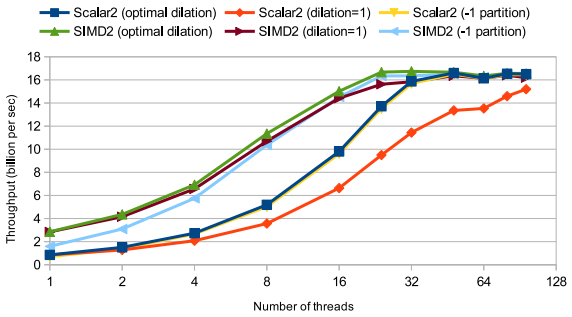
Observation 3. Over all experiments, the partitioning variant SIMD2-P has the most robust performance in all



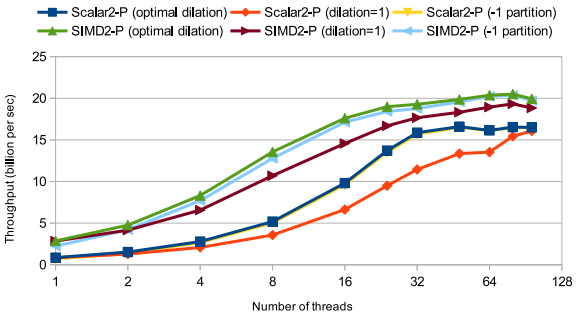
(a) Prefix sum + Increment, no partitioning



(b) Prefix sum + Increment



(c) Accumulate + Prefix Sum, no partitioning



(d) Accumulate + Prefix Sum

Figure 11: Effect of dilation factors

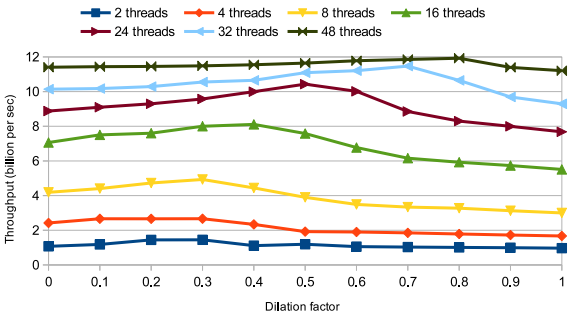


Figure 12: Varying dilation factors with different number of threads

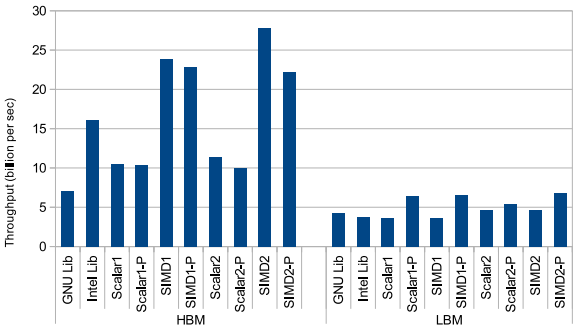


Figure 13: Throughput on Knights Landing

conditions, while SIMD1-P performed slightly better for in-place computations. Scalar code often reached the same plateau as SIMD code with many threads, but the SIMD code was significantly faster at lower thread counts where bandwidth was not saturated.

Observation 4. There are some subtle interactions between in-place/out-of-place choices and algorithm structure, particularly for scalar code. There are also compiler-driven effects, where simpler loop structures in scalar code can be automatically vectorized.

Observation 5. Tree-based algorithms are not competitive because of poor memory locality. An alternative vertical SIMD method seems reasonable in theory, but does not perform well because on current machines, gather/scatter instructions are relatively slow.

6. CONCLUSIONS

In this paper we have implemented and compared different ways of computing prefix sums using SIMD and multithreading. We find that efficient SIMD implementations are better able to exploit the sequential access pattern, even if it is not work-efficient in theory. Using AVX-512, we observe more than 3x speedup over scalar code in a single-threaded execution. In a parallel environment, the memory bandwidth quickly becomes the bottleneck and SIMD adds even more pressure on memory access. An efficient multithreaded implementation needs to be cache-friendly, with minimal overhead of thread synchronization. In the experiments we find that the most efficient prefix sum computation using our partitioning technique with better caching is up to 3x faster than standard library implementations that already use SIMD and multithreading.

7. REFERENCES

[1] Libstdc++ parallel mode. https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html.

[2] Parallel stl. https://github.com/oneapi-src/oneDPL.

[3] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment*, 10(5):517–528, 2017.

[4] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the conference on high performance graphics 2009*, pages 159–166, 2009.

[5] G. E. Blelloch. *Vector models for data-parallel computing*, volume 2. MIT press Cambridge, 1990.

[6] G. E. Blelloch. Prefix sums and their applications. *Synthesis of Parallel Algorithms*, pages 35–60, 1993.

[7] S. Chaudhuri and J. Radhakrishnan. The complexity of parallel prefix problems on small domains. In *Proceedings., 33rd Annual Symposium on Foundations of Computer Science*, pages 638–647. IEEE, 1992.

[8] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and computation*, 81(3):334–352, 1989.

[9] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and experience*, 24(3):327–336, 1994.

[10] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Relative prefix sums: An efficient approach for querying dynamic olap data cubes. In *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*, pages 328–335. IEEE, 1999.

[11] T. Goldberg and U. Zwick. Optimal deterministic approximate parallel prefix sums and their applications. In *Proceedings Third Israel Symposium on the Theory of Computing and Systems*, pages 220–228. IEEE, 1995.

[12] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[13] C. Hetland, G. Tziantzioulis, B. Suchy, M. Leonard, J. Han, J. Albers, N. Hardavellas, and P. Dinda. Paths to fast barrier synchronization on the node. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 109–120, 2019.

[14] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[15] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. *ACM SIGMOD Record*, 26(2):73–88, 1997.

[16] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[17] M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell. Extending openmp* with vector constructs for modern multicore simd architectures. In *International Workshop on OpenMP*, pages 59–72. Springer, 2012.

[18] K. J. Kohlhoff, V. S. Pande, and R. B. Altman. K-means for parallel architectures using all-prefix-sum sorting and updating steps. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1602–1612, 2012.

[19] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

[20] N. Leischner, V. Osipov, and P. Sanders. Gpu sample sort. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010.

[21] D. Lemire. Wavelet-based relative prefix sum methods for range sum queries in data cubes. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 6. IBM Press, 2002.

[22] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[23] D. Lemire, L. Boytsov, and N. Kurz. Simd compression and the intersection of sorted integers. *Software: Practice and Experience*, 46(6):723–749, 2016.

[24] S. Maleki, Y. Gao, M. J. Garzar, T. Wong, D. A. Padua, et al. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382. IEEE, 2011.

[25] 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*, pages 1493–1508, 2015.

[26] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4):43–es, 2007.

[27] A. Rodchenko, A. Nisbet, A. Pop, and M. Luján. Effective barrier synchronization on intel xeon phi coprocessor. In *European Conference on Parallel Processing*, pages 588–600. Springer, 2015.

[28] P. Sanders and J. L. Träff. Parallel prefix (scan) algorithms for mpi. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 49–57. Springer, 2006.

[29] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[30] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 351–362, 2010.

[31] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. 2007.

[32] S. Sengupta, A. Lefohn, and J. D. Owens. A work-efficient step-efficient prefix sum algorithm. 2006.

[33] J. Singler and B. Konsik. The gnu libstdc++ parallel mode: software engineering considerations. In

Proceedings of the 1st international workshop on Multicore software engineering, pages 15–22, 2008.

[34] J. Singler, P. Sanders, and F. Putze. Mcstl: The multi-core standard template library. In *European Conference on Parallel Processing*, pages 682–694. Springer, 2007.

[35] S. Yan, G. Long, and Y. Zhang. Streamscan: fast scan algorithms for gpus without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 229–238, 2013.