Higher-Order and Tuple-Based Massively-Parallel Prefix Sums

Sepideh Maleki
Department of Computer Science
Texas State University
San Marcos, TX, USA
smaleki@txstate.edu

Annie Yang
Department of Computer Science
Texas State University
San Marcos, TX, USA
ayang@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX, USA
burtscher@txstate.edu

Abstract

Prefix sums are an important parallel primitive, especially in massively-parallel programs. This paper discusses two orthogonal generalizations thereof, which we call higher-order and tuple-based prefix sums. Moreover, it describes and evaluates SAM, a GPU-friendly algorithm for computing prefix sums and other scans that directly supports higher orders and tuple values. Its templated CUDA implementation unifies all of these computations in a single 100-statement kernel. SAM is communication-efficient in the sense that it minimizes main-memory accesses. When computing prefix sums of a million or more values, it outperforms Thrust and CUDPP on both a Titan X and a K40 GPU. On the Titan X, SAM reaches memory-copy speeds for large input sizes, which cannot be surpassed. SAM outperforms CUB, the currently fastest conventional prefix sum implementation, by up to a factor of 2.9 on eighth-order prefix sums and by up to a factor of 2.6 on eight-tuple prefix sums.

Categories and Subject Descriptors D.1.3 [**Programming Techniques**]: Concurrent Programming - parallel programming; G.4 [**Mathematical Software**]: Algorithm design and analysis, Efficiency, Parallel and vector implementations

General Terms *Algorithms, Performance, Design*

Keywords Higher-order prefix sums, tuple-based prefix sums, carry propagation, GPU programming

1. Introduction

Prefix sums are a fundamental building block in parallel programming because they allow many seemingly serial computations to be expressed in a way that can easily be parallelized. The benefit of prefix sums is proportional to the amount of parallelism, which is why they are widely used in programs for massively parallel accelerators like GPUs. Moreover, prefix sums are likely to become more important in the future as the amount of hardware parallelism grows.

The prefix sum of a sequence of n values is a new sequence of n values where the value at position i is the sum of all the values in the input sequence up to position i. If the sum includes the input value at position i, it is an inclusive prefix sum. Otherwise, it is an exclusive prefix sum. The following serial code computes the inclusive prefix sum of the values in array A and stores the result back into array A.

```
for (i = 1; i < n; i++) {
  A[i] = A[i] + A[i - 1];
}
```

This code performs O(n) computations on O(n) data. Due to the loop-carried dependency, each iteration can only be executed after the previous iteration has finished, making the code inherently sequential. However, several approaches for computing prefix sums in parallel are known [2][12][14]. They only require O(log n) parallel steps. The total work performed by the most efficient of these approaches is O(n), i.e., the same as the serial algorithm.

It is the existence of these work-efficient parallel implementations that make prefix sums so useful. There are many computations where every result value is data dependent on previous values, yet these computations can be mapped to or expressed in terms of prefix sums and therefore executed in parallel. Examples include radix sort, quicksort, lexical analysis, polynomial evaluation, stream compaction, histograms, and string comparison [2]. The work presented in this paper is motivated by data compression, another domain with algorithms that appear sequential because many of them decompress values based on previously decompressed values.

At a high level, most data-compression algorithms transform a linear sequence of input values, e.g., a file, into a shorter sequence of output values using two main components: a data model and a coder. Roughly speaking, the goal of the model is to accurately predict the next value in the input sequence. The residual, i.e., the difference between each actual value and its predicted value, will be close to

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4261-2/16/06...
http://dx.doi.org/10.1145/2908080.2908089

zero if the model is accurate for the given data. Note that the residual sequence contains as many values as the original sequence but tends to be more compressible. The residuals are then compressed using the coder by mapping them in such a way that frequently encountered values or patterns produce a shorter output than infrequently encountered items. The inverse operations are performed to decompress the data.

A simple yet effective and widely deployed data model is delta encoding, which computes the difference sequence, meaning that it replaces each value with the difference between it and the previous value in the sequence. Delta encoding is, for example, used in image compression and especially in speech compression, where several international standards exist that are based on it, e.g., G.726. Since delta encoding by itself does not compress the sequence of values but makes it easier to compress by other means, it is always used in combination with a coder algorithm.

It is trivial to compute the difference sequence in a single parallel step. In fact, delta encoding is embarrassingly parallel when not done in place, i.e., the result needs to be written into a separate array. However, this is not the case for delta decoding, where the decoded prior value is needed to decode the current value. As it turns out, each decoded value amounts to the sum of all prior differences. Hence, delta decoding is tantamount to computing the prefix sum and can, therefore, be computed in parallel. The following example illustrates delta encoding (computing the difference sequence) and delta decoding (computing the prefix sum).

input values:	`1, 2, 3, 4, 5, 2, 4, 6, 8, 10`
differences:	`1, 1, 1, 1, 1, -3, 2, 2, 2, 2`
prefix sum:	`1, 2, 3, 4, 5, 2, 4, 6, 8, 10`

The standard delta encoder effectively predicts the current value to be the same as the prior value in the sequence and emits the difference. We call this *order one*, which utilizes a constant extrapolation to predict the values. Higher-order predictions are also possible and have the potential to be more accurate. For instance, a second-order approach linearly extrapolates the prediction of the current value from the previous two values, a third-order prediction computes a quadratic extrapolation based on the prior three values, and so on. Unsurprisingly, a simple prefix sum is not sufficient to decode these higher-order encodings (see below).

Moreover, data often appear in tuples. For example, a sequence with tuples of size two might look like this:

$$x_0,\ y_0,\ x_1,\ y_1,\ x_2,\ y_2,\ \ldots,\ x_{n-1},\ y_{n-1}$$

In tuple-based linear sequences, it is frequently the case that the values from the same location within the tuples correlate more with each other than values from different locations. Effective delta encoders take this into account and compute the difference sequence using the value from the same location in the prior tuple rather than the immediately preceding value. For instance, in the above sequence, it would subtract x_{k-1} from x_k and y_{k-1} from y_k, thus avoiding the mixing of x and y values. Hence, for a linear sequence of values representing s-tuples, we need to simultaneously compute s independent prefix sums, where the m^{th} such prefix sum is over the values at positions $m + j \cdot s$ for $j = 0, 1, 2,$ etc. Again, a conventional prefix sum is insufficient to decode such tuple-based encodings.

This paper describes two generalizations of prefix sums to handle these higher-order and tuple-based cases, respectively. Prefix sums have been generalized before to work with arbitrary binary associative operations instead of just with sums. That generalization is called a prefix *scan*. Our generalizations are orthogonal to each other and to scans, i.e., all three of them can be used together, in some combination, or in isolation. For clarity of exposition, we restrict the discussion to inclusive prefix sums and present and evaluate our implementation of higher-order and tuple-based prefix sums separately.

Since the conventional prefix sum is a special case of these generalizations, i.e., order one with a tuple size of one, we started out by designing a new prefix-sum algorithm that is more suitable as a baseline for implementing the higher-order and tuple-based support. Our implementation comprises a single stage and reads and writes each sequence value only once from/to global memory, i.e., it is communication-optimal in main memory. In contrast, some prior algorithms compute partial prefix sums on multiple data chunks in parallel, then perform a prefix sum over the last value of each chunk to compute the carries, and finally add the appropriate carry to every element of each chunk, i.e., they require three stages. This approach is inefficient because it reads and writes all values twice, once in the first stage and once in the third stage.

We named our prefix-sum algorithm SAM, implemented it in CUDA, and successfully ran it on various GPUs, data types, associative operators, and problem sizes ranging from 2^{10} to 2^{30} items, including non-powers-of-two. We selected three widely-used libraries that support prefix sums, CUB [5], CUDPP [6], and Thrust [26], for performance comparisons on Kepler and Maxwell GPUs. When computing prefix sums of 32- or 64-bit integers, our SAM code outperforms all of these libraries on a Titan X for very large problem sizes. On the older K40, SAM outperforms CUDPP and Thrust on medium and large problem sizes, but CUB yields the best performance. Note, however, that CUB employs GPU-architecture-specific code to boost its performance whereas SAM uses a fixed algorithm that is implemented in a single templated CUDA kernel with 100 statements, including the support for higher orders and tuples. For very large inputs, SAM even matches the memory-copy speed on the Titan X, which cannot be exceeded. Moreover, SAM delivers the highest throughputs on the Titan X for higher-order prefix-sum computations. On order eight, it is up to 2.9

times faster. It is also faster than CUB on the K40 but only above about order five. Similarly, SAM outperforms CUB on tuple-based prefix sums when the tuple size is above about five elements. On eight-tuples, SAM is up to a factor of 2.6 faster.

This paper makes the following main contributions.

1) It describes tuple-based prefix sums as well as an implementation thereof that is more efficient for sufficiently large tuples than alternative implementations because its register usage is independent of the tuple size.

2) It describes higher-order prefix sums and an implementation thereof that outperforms alternative implementations for sufficiently high orders because its number of main-memory accesses is independent of the order.

3) It presents a latency-hiding technique for propagating carries between dependent persistent thread blocks that only requires a constant amount of auxiliary memory, which is important for higher -order and tuple-based prefix sums.

4) It makes the CUDA implementation of SAM, which unifies all of the above in a single 100-statement kernel, available at http://cs.txstate.edu/~burtscher/research/SAM/.

The rest of this paper is organized as follows. Section 2 describes our SAM algorithm, the two prefix-sum generalizations, and efficient CUDA implementations thereof. Section 3 discusses related work on prefix sums. Section 4 presents the evaluation methodology and the testbed. Section 5 studies and analyzes the performance results. Section 6 concludes with a summary and future work.

2. SAM Algorithm and Implementation

CUDA-capable GPUs expose three levels of parallelism to the programmer. The first level is represented by the *warps*. Warps are tightly coupled sets of 32 contiguous threads that execute in lockstep, that is, a thread is either disabled or executes the same instruction (operating on different data) in the same clock cycle as the other threads in the warp. Threads can exchange data within a warp using the shuffle machine instruction without explicit synchronization.

The second level is represented by the *thread blocks*, each of which can hold up to 1024 threads in current GPUs. The actual number of threads per block is chosen by the programmer. All threads in a block have access to a software-controlled data cache called *shared memory*, which allows them to exchange data at L1-cache speeds. However, such data exchanges typically require synchronization. The GPU provides machine instructions for this purpose that implement thread-block-wide barriers.

The third level is represented by the *grid*, which consists of up to two billion thread blocks on modern GPUs. The programmer has to choose how many thread blocks to launch. Threads from different blocks can only communicate via *global* (i.e., main) memory. There is no grid-wide barrier, so programmers must resort to locks and/or memory-fence operations to implement any needed synchronization. However, there is an implied global barrier at the end of each grid. Communication through global memory is backed by a shared L2 cache. Since a fully occupied GPU processes tens of thousands of active threads, the L2 cache only has enough capacity to hold a few words per thread on average, i.e., less than the combined capacity of the register files.

Due to warp-based execution, all active threads in a warp always access memory at the same time. The GPU's memory subsystem attempts to optimize such bulk loads and stores. For example, if the warp threads simultaneously access words in main memory that lie in the same aligned 128-byte segment, the hardware merges the 32 reads or writes into one *coalesced* memory transaction that is as fast as accessing a single word. Warps whose threads touch multiple 128-byte segments result in correspondingly many memory transactions that are executed serially and are therefore slower.

Whereas all threads in a warp and all warps in a thread block are simultaneously mapped to the GPU hardware, a GPU will only map as many thread blocks at the same time as it has resources for (multiprocessors, registers, shared memory capacity, block contexts, and thread contexts). Whenever a block finishes running, the hardware will start working on the next block until the entire grid has been executed. This makes it difficult to exchange data between blocks because some of the blocks may not be running. To avoid this situation, some CUDA programs query the underlying hardware, only launch as many blocks as can simultaneously be active, and assign multiple work items to each thread [11]. SAM uses this persistent-thread model.

2.1 Standard Prefix Sum Implementation

To keep the discussion concise, we only cover inclusive prefix sums and assume thread blocks with 1024 threads in this section. GPU implementations of prefix sums are typically hierarchical to maximally exploit the above-mentioned three levels of hardware parallelism. In particular, the input array over which to compute the prefix sum is broken up into chunks of 1024 elements to match the threads per block, and each chunk is further subdivided into subchunks of 32 elements to match the threads per warp.

In phase one, each warp computes an independent prefix sum on its subchunk using a series of shuffle instructions. The resulting last element from each warp is recorded in an auxiliary 32-element array in shared memory. In phase two, after synchronizing the warps within each thread block using a barrier instruction, a single warp computes the 32-element prefix sum of the values in the auxiliary array and writes the results back to the auxiliary array. In phase three, after another barrier, all the threads in each block grab the appropriate entry from the auxiliary array and add it to their result from phase one, thus completing the prefix-sum computation of the 1024-value chunk assigned to each block.

Since prefix-sum calculations require only a few registers per thread and the second phase does not exhibit much parallelism, high-performance implementations simultaneously process several values per thread, i.e., the chunk size is a multiple of 1024 input elements, which improves resource utilization. Moreover, this approach increases the amount of parallel work in phase two and better amortizes the barriers.

The algorithm outlined so far only computes prefix sums at the chunk level. To combine these partial solutions into the global solution, the same three-phase approach needs to be applied again at a coarser level of granularity. The last prefix-sum element from each chunk needs to be stored in an auxiliary array, the prefix sum over this array has to be computed, and the resulting carries need to be added to every value in each chunk. Figure 1 visualizes these steps.

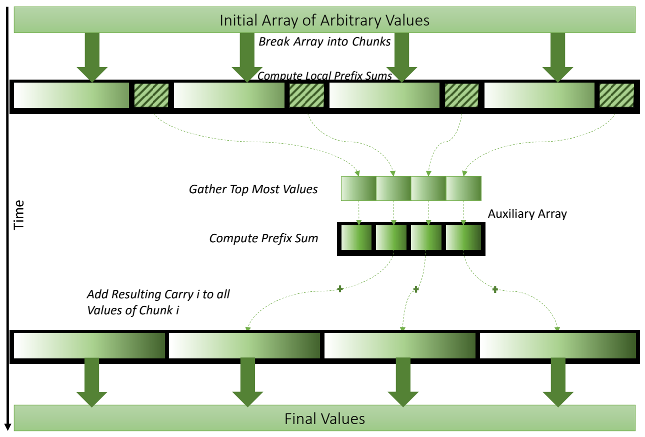
Initial Array of Arbitrary Values
Break Array into Chunks
Compute Local Prefix Sums
Gather Top Most Values
Auxiliary Array
Compute Prefix Sum
Add Resulting Carry i to all Values of Chunk i
Time
Final Values

Figure 1. Hierarchical parallel prefix sum algorithm

Since this calculation is performed across blocks, the auxiliary array has to be stored in global memory. Moreover, due to the lack of an explicit barrier across the entire grid, each phase at this coarser granularity is typically implemented as a separate *kernel* call (grid launch) to exploit the implicit barrier at the end of each grid. Note that the values held in global memory persist between kernel calls, but not the values in the shared memory or in the registers. Hence, the intermediate results need to be written to global memory at the end of each phase and reloaded at the beginning of the next phase. Very large inputs may require a third, even coarser level of granularity to compute the full prefix sum.

The prefix-sum implementations of libraries like Thrust and CUDPP are based on a hierarchical approach similar to the one described above. They launch multiple kernels when computing prefix sums over large inputs and read and write every element twice, once in the first phase and then again in the third phase, yielding a communication-inefficient implementation that accesses global memory twice as often as in the ideal case.

2.2 SAM Base Implementation

Our SAM algorithm employs the approach outlined above for computing intra-block prefix sums, that is, it first computes the prefix sums of each subchunk and then combines them into a block-wide prefix sum. At this point, it is important to note that GPUs can only run a few dozen thread blocks simultaneously. Thus, instead of waiting for all thread blocks to finish before combining their local sums and using them to correct each value, SAM writes the local sum of each thread block to an auxiliary array as soon as it has been computed, executes a memory fence, and then writes a ready flag to a second auxiliary array to indicate the availability of the sum. Next, it reads the local sums of all prior thread blocks, waiting for any that are not yet available, adds them up, and uses the accumulated carry to correct the values in the block before writing the final result to global memory. This approach requires only a single kernel call, i.e., a single phase. More importantly, the values over which the prefix sum is computed are only read from global memory once, processed, and then written back to global memory once.

Since our SAM implementation uses persistent blocks, the same thread block processes every k^{th} chunk where k is the number of persistent thread blocks, which is a hardware dependent constant. This is key because it enables the next chunk within a block to benefit from the prior chunk's accumulated carry value. In particular, the thread block only needs to incrementally update the prior carry by adding in the local sum it just computed plus the k-1 local sums from the intervening chunks (that are processed by the other blocks), as illustrated in Figure 2. Since k is a small constant, 30 and 48 on our GPUs, this technique lowers the cost of computing a carry to O(1) per chunk. Furthermore, it makes it possible to implement the two auxiliary arrays as circular buffers with 3k elements, meaning that the local sums and ready flags require O(1) storage and should therefore remain resident in the shared L2 cache. For performance reasons, SAM allocates a little over 3k elements in the auxiliary arrays to make their size a power of two.

The up to k-1 local sums (there are fewer for the first few chunks) and the corresponding up to k-1 ready flags are read in parallel using coalesced load instructions. To further minimize memory accesses, only non-ready flags are polled until they are ready. Polling of multiple non-ready flags also happens in parallel and using coalesced accesses. Once the local sums are available, they are read in and summed up in a prefix-sum-like computation in log(k-1) steps.

The polling delays the processing of later chunks until the local sums of the earlier chunks are available. This has the effect of staggering the computation of the chunks. For example, block b computing chunk c may have to wait for block $b-1$ to compute chunk $c-1$. However, block $b-1$ will later compute chunk $c+k-1$ and block b will compute chunk $c+k$, making it unlikely that it will have to wait again since it is already running a little behind block $b-1$. As illustrated in Figure 2, this results is a pipeline-like processing of the chunks, which is quite stable because the control flow and memory-access patterns of prefix-sum computations are not data dependent, i.e., these computations take the same amount of time per chunk irrespective of the actual values over which the prefix sum is computed.

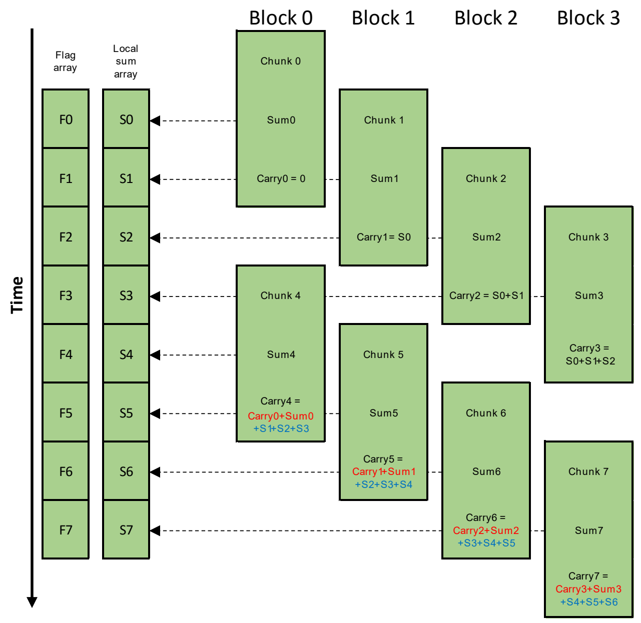

Figure 2. Pipelined processing of chunks in SAM and constant-time carry computation in persistent thread blocks

This carry-propagation scheme trades off redundant computation for improved latency hiding, which is beneficial. The straight-forward way of computing the carry in each thread block and propagating it directly to the next block implies a read-modify-write dependence chain through all the thread blocks. In contrast, our approach employs a write-followed-by-independent-reads pattern, which has a significantly lower latency on modern GPUs, as the result section demonstrates. We believe this technique may also be used to speed up other producer-consumer GPU codes.

Note that CUB employs an even more sophisticated carry propagation scheme [18]. However, we chose the technique described in this section because it requires only O(1) auxiliary memory. This is important for supporting higher orders and tuple values (see below), which require multiple such arrays, one per order and one per tuple element.

In summary, SAM incorporates the following important performance enhancements. 1) It minimizes main-memory accesses through its single-pass approach. 2) It employs persistent threads to help accelerate the carry computations and to minimize the auxiliary array sizes. 3) The "pipelining" minimizes waiting due to inter-block communication, i.e., SAM is able to hide much of the latency of exchanging the sum information. 4) It concurrently processes multiple values per thread to maximally utilize the available register and shared memory space. This increases the chunk size, which reduces the total number of local sums that have to be communicated between thread blocks.

The SAM algorithm performs better on current high-end GPUs than the conventional three-phase approach. Additionally, it is a more suitable baseline upon which to implement higher-order and tuple-based prefix sums, as the following subsections illustrate.

2.3 Tuple-Based Prefix Sums

Computing a tuple-based prefix sum can be accomplished by first reordering the elements, i.e., grouping them by location within the tuple, then performing multiple smaller prefix sums, and finally undoing the reordering. This approach is shown in the following example with two-element tuples.

$$x_0,\ y_0,\ x_1,\ y_1,\ \ldots,\ x_{n-1},\ y_{n-1}$$

reordering

$$x_0,\ x_1,\ \ldots,\ x_{n-1}\ \mid\ y_0,,\ y_1,\ \ldots,\ y_{n-1}$$

computing independent prefix sums

$$\sum_0^0 x_i, \sum_0^1 x_i,\ \ldots, \sum_0^{n-1} x_i \ \mid\ \sum_0^0 y_i, \sum_0^1 y_i,\ \ldots, \sum_0^{n-1} y_i$$

undoing the reordering

$$\sum_0^0 x_i, \sum_0^0 y_i, \sum_0^1 x_i, \sum_0^1 y_i, \ldots, \sum_0^{n-1} x_i, \sum_0^{n-1} y_i$$

This technique works with all prefix-sum codes. However, since the two reordering steps require extra memory accesses, it is slow. For codes that support templated input elements such as CUB, it is possible to compute tuple-based prefix sums directly by defining a tuple data type as well as the plus operator for it, thus avoiding the need for any reordering. However, even this approach results in poor performance for large enough tuple sizes that cause the threads to run out of registers for allocating the tuple elements. This is why SAM goes a step further. It computes tuple-based prefix sums without reordering and without the need for special data types or overloaded operators. It reads the input elements linearly in a fully coalesced manner, directly computes the tuple-based prefix sum, and stores the result back to global memory using fully coalesced writes.

Internally, it accomplishes this by using a stride s when summing up the values, where s is the size of the tuples. For $s = 1$, this amounts to the conventional prefix-sum algorithm. The biggest hurdle in implementing this technique is handling sizes that are not powers of two. Since the number of threads per warp and the number of threads in a block in SAM are powers of two, any non-power-of-two size complicates the local-sum propagation for the following reasons. First, the i^{th} thread in a block does not necessarily process a value that belongs to the same location within a tuple as the value processed by the i^{th} thread in a different block. Second, after a thread block has processed a chunk of data and moves on to processing the next chunk of data, the next value assigned to thread i belongs to a different tuple location than the corresponding value from the prior chunk. To correctly handle these situations, SAM employs a total of s sum arrays, all of which are implemented as circular buffers. Since a thread block always computes all of its sums together, a single ready-flag array is sufficient to indicate whether the s sums are available. Modulo operations are employed to determine which sum each thread needs to use.

2.4 Higher-Order Prefix Sums

No single-step solution exists that we are aware of for directly computing higher-order prefix sums. Until such a solution is found, we have to resort to an iterative approach.

Recall that we need higher-order prefix sums to decode difference sequences above order one. There are closed-form solutions for generating higher-order difference sequences in a single step and in parallel. For example, the second-order difference sequence can be directly computed by subtracting the previous element twice and adding the second previous element to each element in the sequence.

$$\texttt{out}_k = \texttt{in}_k - 2\cdot\texttt{in}_{k-1} + \texttt{in}_{k-2}$$

The following numeric example illustrates this computation. "Missing" values are assumed to be zero.

```
   input values: 1, 2, 3, 4, 5, 2, 4, 6, 8, 10
2nd-order diff: 1, 0, 0, 0, 0, -4, 5, 0, 0, 0
```

Alternatively, higher-order difference sequences can also be computed by repeatedly applying 1st-order (the "normal") differencing, where $\texttt{out}_k = \texttt{in}_k - \texttt{in}_{k-1}$. For instance, the 2nd-order difference sequence is the normal difference sequence of the normal difference sequence, as shown here.

```
 input values: 1, 2, 3, 4, 5, 2, 4, 6, 8, 10
  differences: 1, 1, 1, 1, 1, -3, 2, 2, 2, 2
diff of diffs: 1, 0, 0, 0, 0, -4, 5, 0, 0, 0
```

Note that the second sequence, the difference of the differences, is the same as the directly computed second-order difference sequence above. In general, the q^{th}-order difference sequence is identical to the sequence obtained when applying first-order differencing q times in a row. Since a conventional prefix sum is the inverse operation of first-order differencing, it follows that iteratively computing q prefix sums will decode a q^{th}-order difference sequence.

Utilizing this iterative approach in conjunction with the conventional three-phase prefix-sum algorithm results in $2q$ global memory reads and $2q$ global memory writes for each input element, making it quite inefficient. The baseline SAM implementation only reads each value once, computes the results, and writes each value once. This makes it possible to merely iterate the computation stage q times, which does not increase the number of global memory accesses. Thus, SAM is well suited for higher-order prefix sums because it retains its main-memory communication-optimality for higher-order prefix sums. Hence, SAM's performance advantage over other implementations increases with higher orders.

As is the case for tuple-based prefix sums, the higher-order support in SAM uses additional auxiliary sum arrays, one per order. Moreover, the ready "flags" no longer hold Boolean values but a count that is incremented in each iteration and thus indicates which iterations' local sums have already been computed. Employing counts instead of Booleans means that only one count array is needed, regardless of the order. It is, in fact, also possible to implement higher-order prefix sums with just one sum array. However, this approach would incur additional dependences and latencies as each block would have to wait until the later blocks have read the sum information before it can safely be overwritten with the sum produced by the next iteration.

2.5 Algorithmic Complexity of Carry Propagation

The SAM algorithm incorporates a work-efficient parallel prefix sum computation. This is straightforward to see as the amount of computation per chunk is constant and the algorithm processes $O(n)$ chunks.

However, the amount of work for processing the carries depends on hardware parameters, making it worthwhile to study the algorithmic complexity of SAM's carry-propagation approach in more detail. Since hardware parameters are fixed, this work is constant for a given GPU, but it can vary between different types and generations of GPUs.

The complexity of processing the carries is $O(af \cdot n)$, where af is an architectural factor that captures the average amount of carry-propagation work per input element. To calculate af, we first need to determine c, the total number of carries. As mentioned earlier, each chunk of data requires k carries, where k is the number of executing thread blocks. Assuming that each chunk contains e elements, there will be n over e chunks. Thus, we obtain $c = k \cdot n / e$.

The number of concurrently executing thread blocks (k) is, in turn, determined by m, the number of streaming multiprocessors (SMs) in the GPU, and b, the number of thread blocks needed per SM to fully occupy the GPU. So, $k = m \cdot b$.

The number of elements per chunk (e) is determined by how many input elements a thread block can simultaneously process. This depends on t, the number of threads per thread block, and r, the number of registers available to each thread. Hence, $e = t \cdot O(r)$. We use $O(r)$ rather than r because some registers are needed for performing computations and are not available for holding input elements.

By substitution, we get $c = k \cdot n / e = m \cdot b \cdot n / (t \cdot O(r))$. In words, this means the total number of carries that SAM has to process is proportional to the number of SMs in the GPU, the number of thread blocks running on each SM, and the number of elements in the input; it is inversely proportional to the number of threads per thread block and the number of registers per thread. Since *af* is proportional to the amount of work per input item for processing the carries, i.e., $af \approx c / n$, we find that $af = O(m \cdot b / (t \cdot r))$.

To gain insight into how the architectural factor *af* might change in the future, we studied the four existing generations of compute-capable GPUs from NVIDIA. The four generations, from oldest to youngest, are called Tesla, Fermi, Kepler, and Maxwell. For each generation, we selected the best-performing single-chip compute GPU. Since there are no Maxwell-based compute GPUs yet, we use the best-performing graphics GPU in this case. The pertinent hardware parameters for each GPU, along with the corresponding architectural factor, are listed in Table 1.

Table 1. Hardware parameters of the best-performing single-chip NVIDIA GPUs from different generations (m = number of SMs, b = minimum number of thread blocks per SM to fully occupy GPU, t = threads per block, r = number of registers available per thread, af = resulting architectural factor scaled by a thousand for better readability)

GPU	generation	*m*	*b*	*t*	*r*	*af* * 1000
C1060	Tesla	30	2	512	16	7.32
M2090	Fermi	16	2	768	21.3	1.96
K40	Kepler	15	2	1024	32	0.92
Titan X	Maxwell	24	2	1024	32	1.46

The trend over time is clearly towards more registers per thread and more threads per thread block. The minimum number of thread blocks per SM to reach maximum occupancy has remained constant so far. The number of SMs per GPU has been decreasing except for the latest generation, where it increased again. Overall, the architectural factor has been dropping rapidly over the first three generations of GPUs but increased again in the current generation. Note, however, that the Titan X is not a compute GPU.

We conclude that it is unclear how the architectural factor will develop in the future. If it decreases, the relative performance of SAM's carry-propagation scheme over approaches that propagate a single carry from chunk to chunk should increase in the future. If it increases, SAM's carry-propagation scheme may have to be replaced by a more efficient $O(n)$ scheme (cf. Section 5.4), which would trivially make the entire SAM algorithm $O(n)$, regardless of the GPU used.

It should be noted, however, that the $O(af \cdot n)$ complexity analysis in this section still omits some constant factors and therefore does not fully capture the actual runtime. For example, the computation-over-memory-access-speed ratio is also important as SAM's carry-propagation scheme trades off extra computation for reduced memory latency.

3. Related Work

Scan operations were first proposed by Iverson as part of the APL programming language [16]. Blelloch is one of the primary researchers to have worked extensively on prefix sums. He describes scans as an important primitive in parallel computing and how they can be used to simplify the description of algorithms [1]. He investigated the effect of including certain scan operations as "unit-time" primitives in P-RAM models. The study concludes that scans improve the asymptotic running time of many algorithms by an $O(\log n)$ factor, greatly simplify the description of many algorithms, and are significantly easier to implement than alternative techniques. He also defines the all-prefix-sums operation (which we simply call "prefix sum"), shows how to implement it in P-RAM, and illustrates various applications thereof, including sorting, lexical analysis, string comparison, polynomial evaluation, and stream compaction [2].

A number of studies about parallel prefix sums and their applications have been published. For example, Blelloch developed many efficient algorithms that are based on scans such as radix sort [1]. Ladner and Fischer introduced a general method for deriving efficient parallel solutions to the fixed-length problems solved by a finite-state transducer, which they simulate using prefix scans [17].

The earliest GPU implementations of scans were written using pixel shaders for "non-uniform stream compaction" [15] and "summed-area table generation" [13]. Sengupta et al. developed the first CUDA program of a segmented scan by extending some of their earlier work [24]. Dotsenko et al. [7] ported to CUDA the algorithm written by Chatterjee et al. for a Cray Y-MP supercomputer [3].

Sengupta et al. were the first to publish a work-efficient $O(n)$ GPU implementation of a scan [25]. Together with other co-authors, he then introduced a block-level parallel scan algorithm for GPUs that simplifies earlier methods considerably [22]. Later, they produced some of the most efficient scan and segmented scan implementations for GPUs by tailoring their algorithm to the natural granularity of the machine and minimizing synchronization [23]. Greß et al. introduced an alternative scan implementation for steam compaction that also exhibits $O(n)$ work complexity [9][10].

To the best of our knowledge, no prior study on tuple-based prefix sums or their implementation exists. Whereas the higher-order prefix sums discussed in this paper have also not been published before, they are a form of a linear recursive filter [8]. Several optimized GPU versions of linear recursive filters, which represent a generalized prefix scan, exist [4][21]. However, these implementations are based on the communication-inefficient three-phase approach.

3.1 Recent Prefix-Sum Implementations for GPUs

There have been quite a few studies of parallel prefix scans in recent years, and there exists a number of libraries that support this primitive. Merry studied several of these libraries and compared their performance [19]. Of the libraries he investigated, he found CUB to provide the most efficient implementation of prefix sums on GPUs (see below).

The CUDA Data Parallel Primitives (CUDPP) library implements the classic three-phase approach described in Section 2.1 [6]. As such, it performs 4*n* global memory accesses when processing an input with *n* elements. Similarly, the Thrust library, which is part of the CUDA Toolkit, employs a two-pass scan-then-propagate technique that also requires 4*n* data movement [26]. As a consequence, both of these libraries do not perform well on large inputs.

MGPU is more efficient and only performs 3*n* global memory accesses [20]. It achieves this because the first pass of its two-pass reduce-then-scan strategy is read-only.

StreamScan implements a matrix-based intra-block scan approach that is communication efficient and only requires 2*n* data movement [27]. It runs in a single computation phase and, therefore, does not need any global barriers and only a single kernel invocation. Its implementation is optimized to fully utilize the available register space. Moreover, the authors designed an auto-tuning framework to optimize the implementation for both AMD and NVIDIA GPUs, which they can do as StreamScan is written in OpenCL. SAM adopts all of these ideas, including the auto-tuner, which runs when SAM is installed and determines the optimal number of input elements to allocate to each thread for different ranges of problem sizes. However, SAM is implemented in CUDA and uses a different algorithm.

As mentioned above, the CUB library from NVIDIA currently provides the fastest GPU implementation of prefix sums. It also incorporates a work-efficient, single-pass method with 2*n* data movement [18]. It performs very well across different GPU types because it comprises multiple algorithms. In particular, it employs different kernel specializations, grain sizes, local scan algorithms, and strategies for rearranging data between threads for each GPU architecture. Moreover, parts of it are implemented in PTX assembly. As a consequence, CUB's code base is much larger, more complex, and less portable than SAM's single 100-statement kernel that is written entirely in CUDA. It should be noted, though, that SAM uses auto-tuning, which CUB does not require. However, this auto-tuning makes it likely that SAM will perform well on future GPUs whereas CUB may need to be updated with appropriate specializations.

CUB uses a variable look-back strategy for propagating the carries to hide the communication latency. This decoupled chaining, which performs redundant carry computations similar to SAM's approach, includes an opportunistic short-circuit in the event that the full carry is already available. In other words, SAM actively propagates the carries through each thread block whereas CUB laggardly pulls the running carry along in global memory. While CUB's approach may result in fewer redundant computations, it adds a timing dependency. As a consequence, SAM computes a deterministic result on a given GPU and input even for pseudo-associative operators such as floating-point addition whereas the precise order in which the prefix-sum operators are applied by CUB may vary from one run to the next.

4. Experimental Methodology

In addition to SAM 1.1, we evaluate the following three widely-used implementations of prefix sums. The first is from the Thrust library, which is included in the CUDA Toolkit 7.5 [26]. The second is from the CUDPP library 2.2 [6]. The third is from the CUB library 1.5.1 [5].

We evaluate the four codes on prefix sums over 32-bit and 64-bit integer sequences with between one thousand and one billion elements. We only measure the kernel runtime, from which we compute the throughput (items processed per second). We repeated each experiment three times and report the average performance.

We present results for two GPUs. The first is a GeForce GTX Titan X, which is based on the Maxwell architecture. The second is a Tesla K40c, which is based on the Kepler architecture. The Titan X has 3072 processing elements distributed over 24 multiprocessors that can hold the contexts of 49,152 threads. Each multiprocessor has 96 kB of shared memory and 48 kB of L1/texture cache. The 24 multiprocessors share a 2 MB L2 cache as well as 12 GB of global memory with a theoretical peak bandwidth of 336 GB/s. We use the default clock frequencies of 1.1 GHz for the processing elements and 3.5 GHz for the GDDR5 memory. The K40 has 2880 processing elements distributed over 15 multiprocessors that can hold the contexts of 30,720 threads. Each multiprocessor has 48 kB of texture memory as well as 64 kB of cache that is split between the shared memory and the L1 data cache. The 15 multiprocessors share a 1.5 MB L2 cache as well as 12 GB of global memory with a peak bandwidth of 288 GB/s. We disabled ECC protection of the main memory and use the default clock frequencies of 745 MHz for the processing elements and 3 GHz for the GDDR5 memory. Both GPUs are plugged into 16x PCIe 3.0 slots in the same system, which has dual 10-core Xeon E5-2687W v3 CPUs running at 3.1 GHz. The host memory size is 128 GB and the operating system is CentOS 6.7.

We compiled all codes with nvcc 7.5. Even though the Titan X supports compute capability 5.2, we used the "-O3 -arch=sm_35" compiler flags for both GPUs as this resulted in slightly faster code on the Titan X. This is not the case for CUB, so we used "-O3 -arch=sm_52" instead.

5. Results and Analysis

We first investigate SAM's performance on conventional prefix sums, which is tantamount to a tuple size of one and an order of one. Then we separately evaluate its performance for orders above one and tuple sizes above one. Note, however, that SAM also fully supports higher-order prefix sums and scans with tuple sizes above one.

5.1 Conventional Prefix Sums

This subsection investigates the throughput of normal prefix-sum computations on inputs with power-of-two sizes between 2^{10} and 2^{30} as well as with power-of-ten sizes between 10^3 and 10^9. Powers of two are frequently used in practice. We include the powers of ten to show how well the various codes perform on some other sizes. Note that none of the tested codes support input sizes above 4 GB, i.e., 2^{30} items for 32-bit integers and 2^{29} items for 64-bit longs.

Figure 3 shows the throughput in billions of 32-bit words processed per second on the Titan X for Thrust, CUDPP, CUB, and our implementation of SAM.

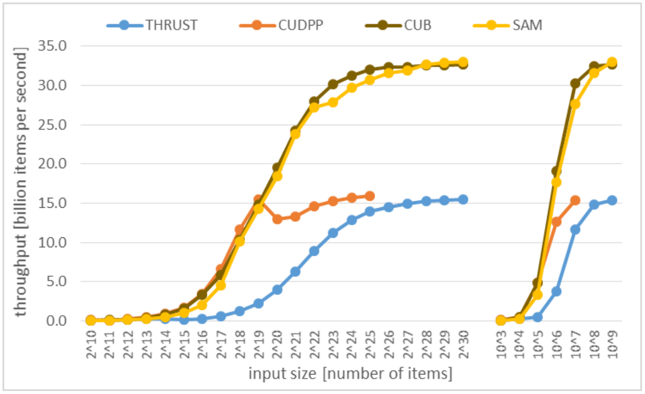

Figure 3. Prefix-sum throughput of 32-bit integers for different problem sizes on the Titan X

SAM and CUB outperform the other two approaches on medium and large inputs because of their communication optimality, i.e., the $2n$ versus $4n$ main-memory accesses. Hence, for problem sizes above about 2^{22}, they provide about twice the throughput of Thrust and CUDPP. Note that CUDPP does not support problem sizes above 2^{25}. Thrust performs better than SAM on inputs of up to 2^{12} items, CUDPP on inputs of up to 2^{19} items, and CUB on inputs of up to 2^{27} items. These codes are expected to perform better than SAM because SAM includes support for higher orders and tuple values, which the other implementations do not. The algorithm used by CUB is superior to SAM on small and medium inputs, but CUB includes assembly instructions.

SAM delivers throughputs of up to 33 billion integers processed per second, which is higher than the maximum throughput of the other codes. We surmise that this good performance is a consequence of SAM's constant amount of auxiliary memory. While SAM accesses its auxiliary memory O(n) times just like the other algorithms do, using O(1) sized circular buffers results in better locality and thus more cache hits. As a result, the benefit of SAM's approach increases for larger inputs compared to other implementations that require a linear amount of auxiliary memory. Since every four-byte item is read and written once, this amounts to 264 GB/s of global memory throughput, which is 78.6% of the theoretical peak bandwidth of the Titan X's main memory. Interestingly, simply copying the input array to the output array using cudaMemcpy, i.e., without performing any computation, delivers the same throughput. This demonstrates that SAM is truly communication optimal (as well as fully memory bound) for large inputs. It is also worth noting that 264 GB/s is several times higher than the theoretical peak memory bandwidth of current CPU systems, meaning that GPUs can compute prefix sums over large inputs faster than multicore CPUs.

Figure 4 shows the throughput in billions of 64-bit words processed per second on the Titan X for Thrust, CUDPP, CUB, and SAM.

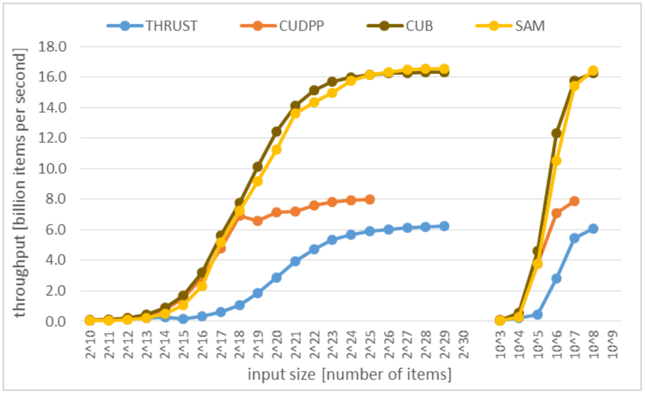

Figure 4. Prefix-sum throughput of 64-bit integers for different problem sizes on the Titan X

The performance behavior on 64-bit longs shown in Figure 4 and the resulting findings are nearly identical to those of the 32-bit integers from Figure 3. However, the absolute throughputs in items per second are about half as high. This is expected since twice as many bytes need to be accessed and processed per word. In fact, SAM again matches the cudaMemcpy throughput for the largest inputs.

Figure 5 shows the throughput in billions of 32-bit words processed per second on the K40 for Thrust, CUDPP, CUB, and SAM.

The K40 throughputs are substantially lower for all four algorithms than those of the newer Titan X. SAM is faster than Thrust and CUDPP on medium and large inputs on the K40. However, CUB exceeds SAM's performance by about 50% on the large inputs. Relatively speaking, the SAM implementation appears less suited for this older GPU than the other three algorithms. We believe this is at least in part due to SAM's carry-propagation scheme, which trades extra computation for improved memory latency hiding. Since the K40's memory is clocked 4.0 times faster than its processing elements (PEs) but the Titan X's memory is only clocked 3.2 times faster than its PEs, trading off extra computation is more advantageous on the Titan X.

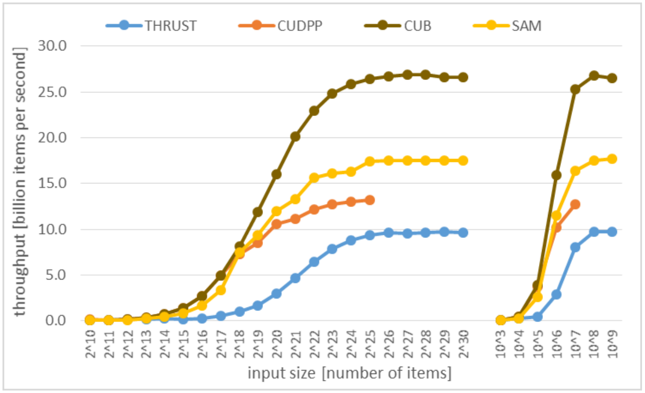

Figure 5. Prefix-sum throughput of 32-bit integers for different problem sizes on the K40

Figure 6 shows the throughput in billions of 64-bit words processed per second on the K40 for the four algorithms. Again, the general trends for the 64-bit data are similar to those of the 32-bit data, including SAM's drop in relative performance. The absolute throughputs are about a factor of two lower due to the aforementioned reasons.

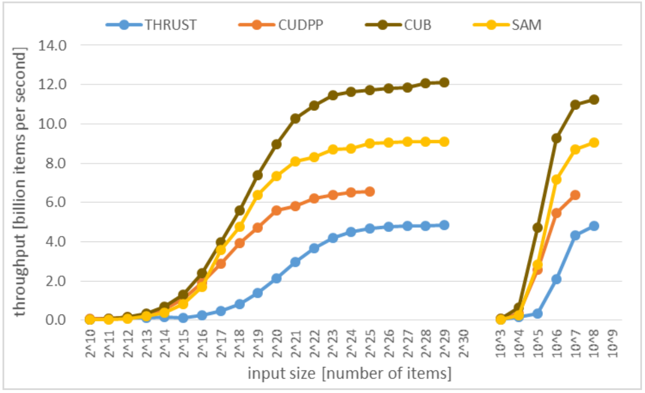

Figure 6. Prefix-sum throughput of 64-bit integers for different problem sizes on the K40

For the most part, the performance behavior as a function of the problem size is similar for all tested prefix-sum implementations, both word sizes, and the two GPUs. In particular, the throughput is low for small problem sizes, increases rapidly for medium problem sizes, and is high and remains fairly stable for large problem sizes. Since our GPUs require over 30,000 and 49,000 threads, respectively, to fully occupy the hardware, the throughput is low for problem sizes that are too small to even allocate a single input element to each thread. Above that threshold, the performance increases quickly with increasing problem size as more and more input elements can be assigned to each thread, which results in better utilization of the hardware, most notably the 32 registers per thread. Above problem sizes of about 2^{23}, the hardware utilization starts to saturate and the throughput plateaus.

In summary, SAM performs well across different word sizes and GPU architectures. On medium and large inputs, it outperforms Thrust and CUDPP by a significant margin. In a few cases, it even beats CUB. This is particularly surprising given that we designed SAM as a baseline algorithm to simplify the implementation of higher-order and tuple-based prefix sums, whose performance we evaluate next.

5.2 Higher-Order Prefix Sums

This subsection investigates the throughput of higher-order prefix-sum computations. Since CUB is the currently fastest GPU implementation of conventional prefix sums, we only show results for CUB and SAM. As described in Section 2.5, SAM implements higher-order prefix sums by internally iterating over the computation code while only reading the input from and writing the output to main memory once. In contrast, using CUB necessitates iterating over all of its code to compute higher-order prefix sums.

Figure 7 shows the throughputs in billions of 32-bit words processed per second on the Titan X for orders two, five, and eight. The single digit after the algorithm name in the legend indicates the order. The conventional prefix sum has order one. We focus on small orders as larger orders are less likely to be used in practice. For improved readability, we do not include results for orders 3, 4, 6, and 7 in the figure. However, we verified that those results follow the same trends.

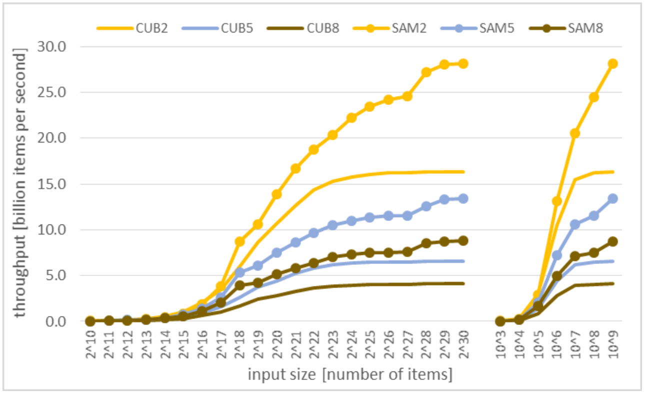

Figure 7. Higher-order prefix-sum throughput of 32-bit integers for different problem sizes on the Titan X

In general, the throughputs decrease as the order increases due to the growing number of iterations. Since SAM iterates over only part of its code, it is superior to CUB. The performance advantage increases with higher orders because SAM always executes the same number of global-memory accesses whereas CUB accesses main memory more often for higher orders. For example, with 2^{27} items, SAM outperforms CUB by 52% on order two, 78% on order five, and

87% on order eight. On some small input sizes with order eight, SAM is almost three times faster than CUB.

Figure 8 shows the higher-order throughputs in billions of 64-bit words processed per second on the Titan X. Again, the single digit after the algorithm name indicates the order.

Expectedly, the throughputs on 64-bit values are about half of the throughputs with 32-bit values. The relative behavior and trends are nearly identical to those discussed above for the 32-bit words. Even the speedup factors of SAM over CUB are very similar.

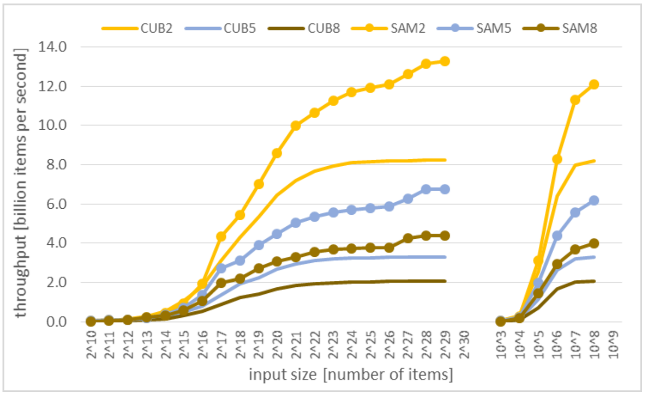

Figure 8. Higher-order prefix-sum throughput of 64-bit integers for different problem sizes on the Titan X

Figure 9 shows the throughputs in billions of 32-bit words processed per second on the K40 for orders two, five, and eight. The throughputs are substantially lower on this older GPU than on the Titan X. Moreover, CUB clearly outperforms SAM on order two, outperforms it a little on order five, and is tied on order eight. The reason for SAM's poorer performance relative to CUB on the K40 is that the baseline performance of CUB is much higher than that of SAM (cf. Figure 5). Hence, it takes more iterations, i.e., higher orders, before SAM starts to exceed CUB's performance.

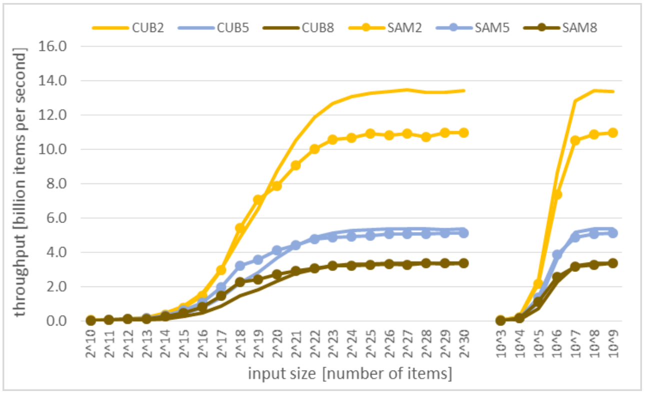

Figure 9. Higher-order prefix-sum throughput of 32-bit integers for different problem sizes on the K40

Figure 10 shows the higher-order throughputs in billions of 64-bit words processed per second on the K40. Again, the throughputs are much lower for 64-bit values than they are for 32-bit values. However, SAM's relative performance over CUB is higher because the baseline performance advantage of CUB over SAM is a little smaller for 64-bit values than for 32-bit values (cf. Figure 6). As a consequence, on order eight, SAM is already faster than CUB.

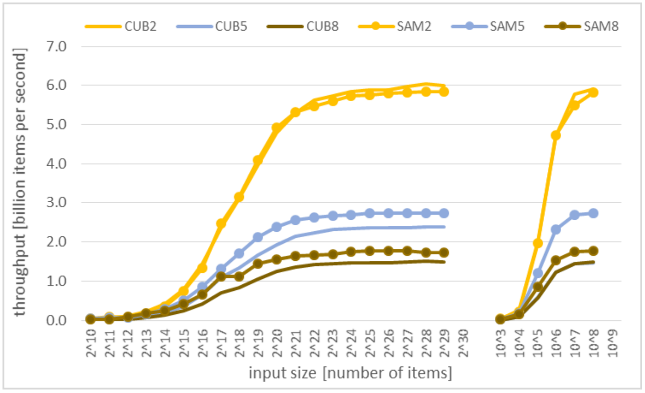

Figure 10. Higher-order prefix-sum throughput of 64-bit integers for different problem sizes on the K40

In summary, SAM delivers the highest throughputs on the Titan X for higher-order prefix-sum computations. It is also faster than CUB on the K40 but only above a certain order. Due to its communication-efficient design, SAM's performance advantage increases with higher orders over CUB.

5.3 Tuple-Based Prefix Sums

This subsection investigates the throughput of tuple-based prefix-sum computations. Since the input size needs to be an integer multiple of the tuple size, some of the inputs are actually a few elements shorter than indicated in the figures. Again, we only show results for CUB and SAM in this subsection. SAM implements tuple-based prefix sums by internally computing multiple strided prefix sums. To perform tuple-based prefix-sum computations using CUB, we declared a tuple data type and a corresponding plus operator.

Figure 11 shows the throughputs in billions of 32-bit words processed per second on the Titan X for tuples holding two, five, and eight words. The single digit after the algorithm name in the legend indicates the tuple size. The conventional prefix sum has one word per tuple. We focus on small tuple sizes as they are more frequently used in practice. For improved readability, we do not include results for 3, 4, 6, and 7 words per tuple. However, we verified that these results are in line with the results shown.

Both CUB's and SAM's performance decreases with larger tuple sizes. In case of SAM, this is primarily due to the larger number of carries that need to be maintained.

However, SAM's throughput decreases more slowly with increasing tuple size, which is why, on the large inputs, it is 17% slower than CUB on two-tuples but 20% faster on five-tuples and 34% faster on eight-tuples. In other words, SAM's direct approach of computing tuple-based prefix sums becomes more beneficial with larger tuple sizes. There are two main reasons for this. First, larger tuple sizes increase the register pressure in CUB as entire tuples are assigned to each thread. SAM's approach allocates the same number of values per thread independent of the tuple size. Second, larger tuple sizes result in progressively less coalesced memory accesses in CUB as multiple consecutive values are accessed by the same thread. In contrast, SAM always accesses consecutive values by contiguous threads, i.e., in a fully coalesced fashion, regardless of the tuple size.

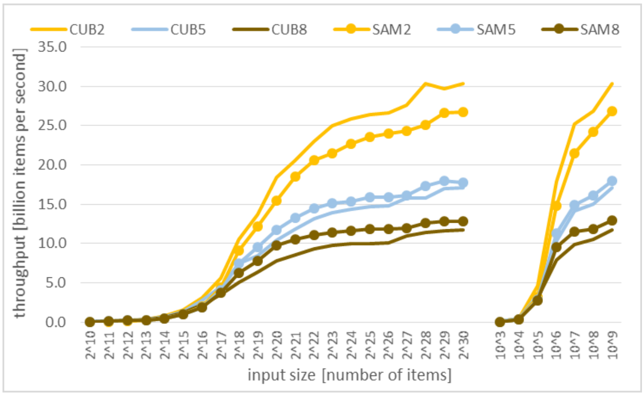

Figure 11. Tuple-based prefix-sum throughput of 32-bit integers for different problem sizes on the Titan X

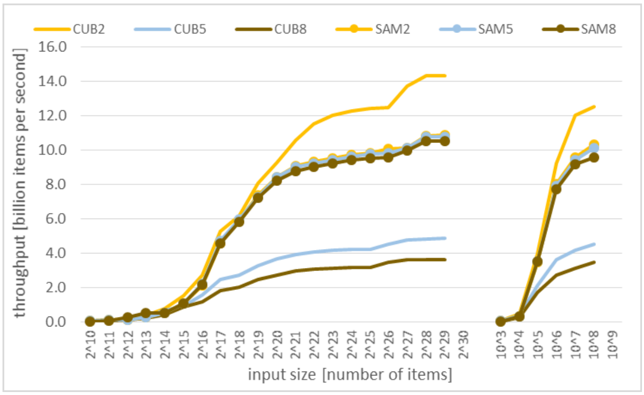

Figure 12. Tuple-based prefix-sum throughput of 64-bit integers for different problem sizes on the Titan X

Figure 12 shows the tuple-based throughputs in billions of 64-bit words processed per second on the Titan X. Expectedly, the performance is again roughly half of the performance with 32-bit values. Interestingly, the throughput on the 64-bit words is nearly the same for two, five, and eight words per tuple. We are not sure why this happens for 64-bit inputs on the Titan X but not on the K40 or for 32-bit inputs, but we consistently obtain these results. Nevertheless, SAM is again slower than CUB on two-tuples but faster on five-tuples and much faster on eight-tuples for the large inputs.

Figure 13 shows the throughputs in billions of 32-bit words processed per second on the K40 for two-, five-, and eight-tuples. Unsurprisingly, the throughputs are significantly lower than those on the Titan X. Moreover, the relative performance of SAM over CUB is again noticeably smaller on the K40 since the baseline performance of CUB is so much higher on this GPU. As a consequence, CUB is faster on both two- and five-tuples. Nevertheless, SAM still outperforms the CUB-based code on the eight-tuples.

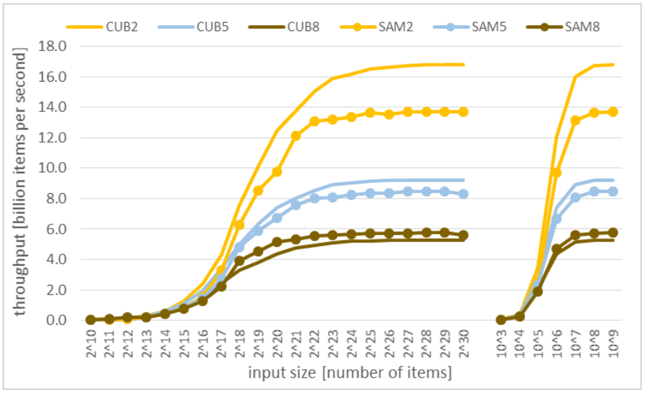

Figure 13. Tuple-based prefix-sum throughput of 32-bit integers for different problem sizes on the K40

Figure 14 shows the tuple-based throughputs in billions of 64-bit words processed per second on the K40, which are about half that of the 32-bit values. Moreover, the benefit of SAM over CUB is a little higher, and SAM now outperforms CUB already on the five-tuples.

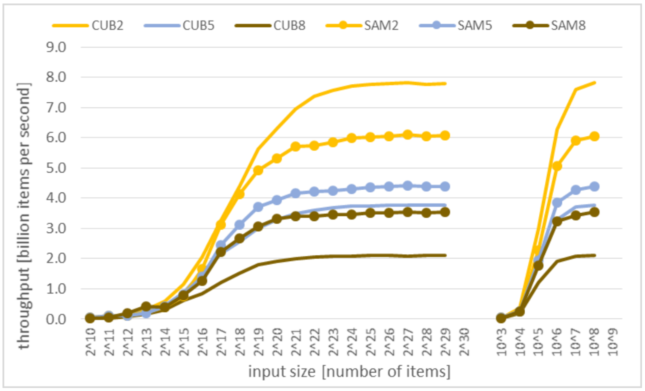

Figure 14. Tuple-based prefix-sum throughput of 64-bit integers for different problem sizes on the K40

In summary, SAM's performance is generally lower for larger tuples as more carries need to be maintained. However, its algorithm results in smaller throughput decreases for larger tuple sizes than that of CUB and therefore eventually exceeds CUB's performance, which happens around five elements per tuple.

5.4 Carry Propagation

This subsection investigates SAM's throughput with two different carry-propagation schemes on conventional prefix sums. For brevity, we only show results for 32-bit values.

As discussed in Section 2.2, SAM's approach trades off extra computations for better latency hiding. To study the effectiveness of this approach, we also evaluate SAM with a simple carry chain, which we refer to as "chained". In the chained implementation, every thread block writes the total carry to the auxiliary array rather than just its local sum. This increases the latency until the carry becomes available but means the next block only needs to read one value instead of *k*-1 values that have to be added up. Note that the chained approach has O(*n*) complexity.

Figure 15 shows the throughputs in billions of 32-bit words processed per second on the Titan X for the chained and SAM's carry-propagation scheme. Figure 16 shows the throughputs in billions of 32-bit words processed per second on the K40 for the same two carry-propagation schemes.

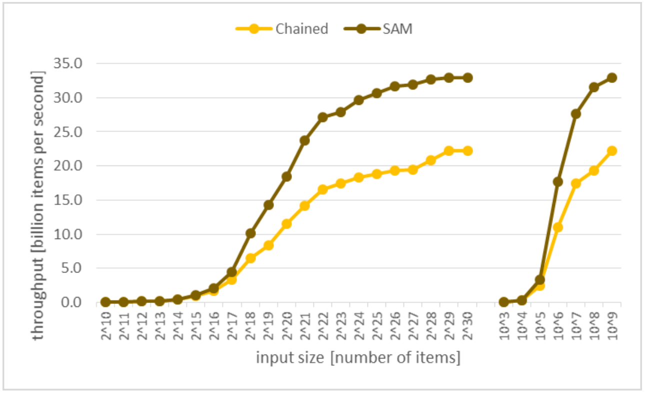

Figure 15. Prefix-sum throughput of 32-bit integers for two carry-propagation schemes on the Titan X

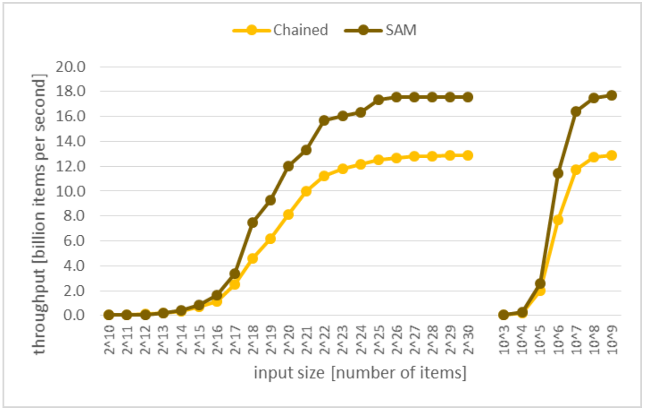

Figure 16. Prefix-sum throughput of 32-bit integers for two carry-propagation schemes on the K40

SAM's write-followed-by-independent-reads technique clearly outperforms the basic read-modify-write chained approach. On large inputs, it is up to 64% faster on the Titan X and up to 39% faster on the K40. SAM's approach helps more on the Titan X than on the K40 because the Titan X is better at trading off extra computations for reduced latency. This is due to architectural differences such as its higher computation-to-memory-speed ratio. In summary, trading off extra computations for improved latency hiding is beneficial and crucial to the overall performance of SAM on both of the tested GPUs.

6. Summary and Future Work

Prefix sums are an essential parallel-programming primitive. Motivated by an important data decompression algorithm, we discuss two generalizations of prefix sums, one to tuple values and the other to higher orders. Moreover, we present and evaluate SAM, a new massively-parallel algorithm for computing prefix sums and scans that natively supports higher orders and tuple values as well as using both at the same time. All of these generalized prefix sums are implemented in a single, compact, templated CUDA kernel. SAM minimizes main-memory accesses, i.e., it reads and writes every element just once irrespective of the problem size, even when computing higher-order prefix sums. Additionally, it maintains full coalescing of the memory accesses and a fixed register usage even for tuple-based prefix sums. As a consequence, SAM's relative performance increases with increasing orders and tuple sizes in comparison to other approaches. SAM employs a carry-propagation scheme that trades off redundant computations for improved latency hiding and that only requires O(1) auxiliary memory. On large inputs, SAM's prefix-sum throughput on a Titan X matches that of cudaMemcpy, which cannot be exceeded.

We successfully tested SAM on different GPUs, data types, binary associative operators, tuple sizes, orders, and problem sizes ranging from 10^3 to 10^9. Performance comparisons on Maxwell- and Kepler-based GPUs show that our code outperforms the prefix-sum implementations of popular libraries like CUDPP and Thrust on large inputs. On the generalized prefix sums, SAM even outperforms CUB by up to 2.9 times on orders and tuple sizes of eight. The open-source CUDA implementation of SAM is publicly available at http://cs.txstate.edu/~burtscher/research/SAM/.

There are several avenues for future work. For example, we could remove the higher-order or tuple-value support from the code to see if that improves the performance, we could expand our study to include more GPUs, or we could study and present measurements for the combined case of higher-order tuple-based prefix sums. It might also be interesting to measure the energy consumption to determine whether the improved performance also results in improved energy efficiency. Moreover, we could evaluate SAM with other associative operators (i.e., scans instead of prefix sums), which we have already done with built-in primitives like max and xor but not described in this paper. Looking further into the future, it may make sense to add support for problem sizes above 4 GB as well as support for parallel kernel execution and virtualization environments where not all SMs of a GPU are always available.

We hope that our work will help others accelerate their codes and maybe even enable the parallelization of algorithms with complex data dependencies that have not yet been successfully parallelized. Furthermore, we believe that our technique of communicating data between dependent persistent thread blocks is also applicable to other codes.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and especially Duane Merrill for his insights and help in getting the most performance out of CUB. The authors' research is supported by the U.S. National Science Foundation under grants 1217231, 1406304, and 1438963, a REP grant from Texas State University, and hardware donations from NVIDIA Corporation.

References

[1] G.E. Blelloch. "Scans as Primitive Parallel Operations." IEEE Transactions on Computers, C-38(ll):1526-1538, 1989.

[2] G.E. Blelloch. "Prefix Sums and Their Applications." In John H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann, 1990.

[3] S. Chatterjee, G.E. Blelloch, and M. Zagha. "Scan primitives for vector computers." Proceedings of the 1990 Conference on Supercomputing, pp. 666–675, 1990.

[4] G. Chaurasia, J.R. Kelley, S. Paris, G. Drettakis, and F. Durand. "Compiling High Performance Recursive Filters." Proceedings of the 7th Conference on High-Performance Graphics, pp 85–94, 2015.

[5] CUB: https://github.com/NVlabs/cub

[6] CUDPP: https://github.com/cudpp

[7] Y. Dotsenko, N.K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. "Fast scan algorithms on graphics processors." Proceedings of the 22nd Annual Int. Conference on Supercomputing, pp. 205–213, 2008.

[8] G. Gautam and S. Rajopadhye. "Simplifying Reductions." Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 30–41, 2006.

[9] A. Greß, M. Guthe, and R. Klein. "GPU-based Collision Detection for Deformable Parameterized Surfaces." Computer Graphics Forum 25, 2006.

[10] A. Greß and G. Zachmann. "GPUABiSort: Optimal Parallel Sorting on Stream Architectures." Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, 2006.

[11] K. Gupta, J.A. Stuart, and J.D. Owens. "A Study of Persistent Threads Style GPU Programming for GPGPU Workloads." Proceedings of Innovative Parallel Computing, 2012.

[12] M. Harris, S. Sengupta, and J.D. Owens, "Parallel prefix sum (scan) with CUDA." GPU Gems 3, 2007.

[13] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. "Fast summed-area table generation and its applications." Computer Graphics Forum, 24(3):547–555, 2005.

[14] W.D. Hillis and G.L. Steele Jr. "Data Parallel Algorithms." Communications of the ACM: 29(12), pp. 1170–1183. 1986.

[15] D. Horn. "Stream reduction operations for GPGPU applications." In M. Pharr (Ed.), GPU Gems 2, chapter 36, pp. 573–589. Addison Wesley, 2005.

[16] K.E. Iverson. "A Programming Language." Wiley, 1962.

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

[18] D. Merrill and M. Garland. "Single-pass Parallel Prefix Scan with Decoupled Look-back." NVIDIA Technical Report NVR-2016-002, NVIDIA Corporation. 2016.

[19] B. Merry. "A performance comparison of sort and scan libraries for GPUs." World Scientific Publishing Company, 2014.

[20] MGPU: http://nvlabs.github.io/moderngpu/

[21] D. Nehab, A. Maximo, R. Lima, and H. Hoppe. "GPU-efficient Recursive Filtering and Summed-area Tables." ACM Transactions on Graphics (SIGGRAPH Asia), 30:6, 2011.

[22] S. Sengupta, M. Harris, and M. Garland. "Efficient parallel scan algorithms for GPUs." In NVIDIA, Santa Clara, CA, 2008 - gpucomputing.net.

[23] S. Sengupta, M. Harris, M. Garland, and J.D. Owens. "Efficient Parallel Scan Algorithms for many-core GPUs". In J. Kurzak, D.A. Bader, and J. Dongarra (Eds.), Scientific Computing with Multicore and Accelerators, Chapman & Hall/CRC Computational Science, chapter 19, pp. 413–442, 2011.

[24] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. "Scan primitives for GPU computing." Graphics Hardware 2007, pp. 97–106, 2007.

[25] S. Sengupta, A.E. Lefohn, and J.D. Owens. "A Work-Efficient Step-Efficient Prefix Sum Algorithm." Proceedings of the Workshop on Edge Computing Using New Commodity Architectures, pp. D-26–27, 2006.

[26] Thrust: https://developer.nvidia.com/thrust

[27] S. Yan, G. Long, and Y. Zhang. "StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchronization." Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 229–238, 2013.