Single-pass Parallel Prefix Scan with Decoupled Look-back

Duane Merrill
NVIDIA Corporation
dumerrill@nvidia.com

Michael Garland
NVIDIA Corporation
mgarland@nvidia.com

Abstract

We describe a work-efficient, communication-avoiding, single-pass method for the parallel computation of *prefix scan*. When consuming input from memory, our algorithm requires only ~2*n* data movement: *n* inputs are read, *n* outputs are written. Our method embodies a *decoupled look-back* strategy that performs redundant work to dissociate local computation from the latencies of global prefix propagation. Implemented by the CUB library of parallel primitives for GPU architectures, the performance throughput of our parallel prefix scan approaches that of copy operations. Furthermore, the single-pass nature of our method allows it to be adapted for (1) in-place compaction behavior, and (2) *in-situ* global allocation within computations that oversubscribe the processor.

1. Introduction

Parallel *prefix scan* is a fundamental parallel computing primitive. Given a list of input elements and a binary reduction operator, a prefix scan produces a corresponding output list where each output is computed to be the reduction of the elements occurring earlier in the input. A *prefix sum* connotes a prefix scan with the addition operator, i.e., each output number is the sum of the corresponding numbers occurring previously in the input list. An *inclusive* scan indicates that the i^{th} output reduction incorporates the i^{th} input element. An *exclusive* scan indicates the i^{th} input is not incorporated into the i^{th} output reduction. Applications of scan include adder design, linear recurrence and tridiagonal solvers, parallel allocation and queuing, list compaction and partitioning, segmented reduction, etc. For example, an exclusive prefix sum across a list of allocation requirements [8,6,7,5,3,0,9] produces a corresponding list of allocation offsets [0,8,14,21,26,29,29].

In this report, we describe the *decoupled-lookback* method of single-pass parallel prefix scan and its implementation within the open-source CUB library of GPU parallel primitives [21]. For highly parallel architectures, prefix sum is a scalable mechanism for cooperative allocation within dynamic and irregular data structures [4, 20]. Contemporary GPUs are at the leading edge of the current trend of increased parallelism in computer architecture, provisioning tens of thousands of data parallel threads. As such, prefix scan plays an important role in many GPU algorithms.

NVIDIA Technical Report NVR-2016-002, March 2016.
CUB v1.0.1, August 2013

This research was developed, in part, with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

In modern computer systems, the performance and power consumption of prefix scan is typically bound by the cost of data movement: reading inputs and writing results to memory is generally more expensive than computing the reduction operations themselves. Therefore communication avoidance (minimizing last-level data movement) is a practical design objective for parallel prefix scan. The sequential prefix scan algorithm requires only a single pass through the data to accumulate and progressively output the running total. As such, it incurs the optimal 2*n* data movement: *n* reads and *n* writes.

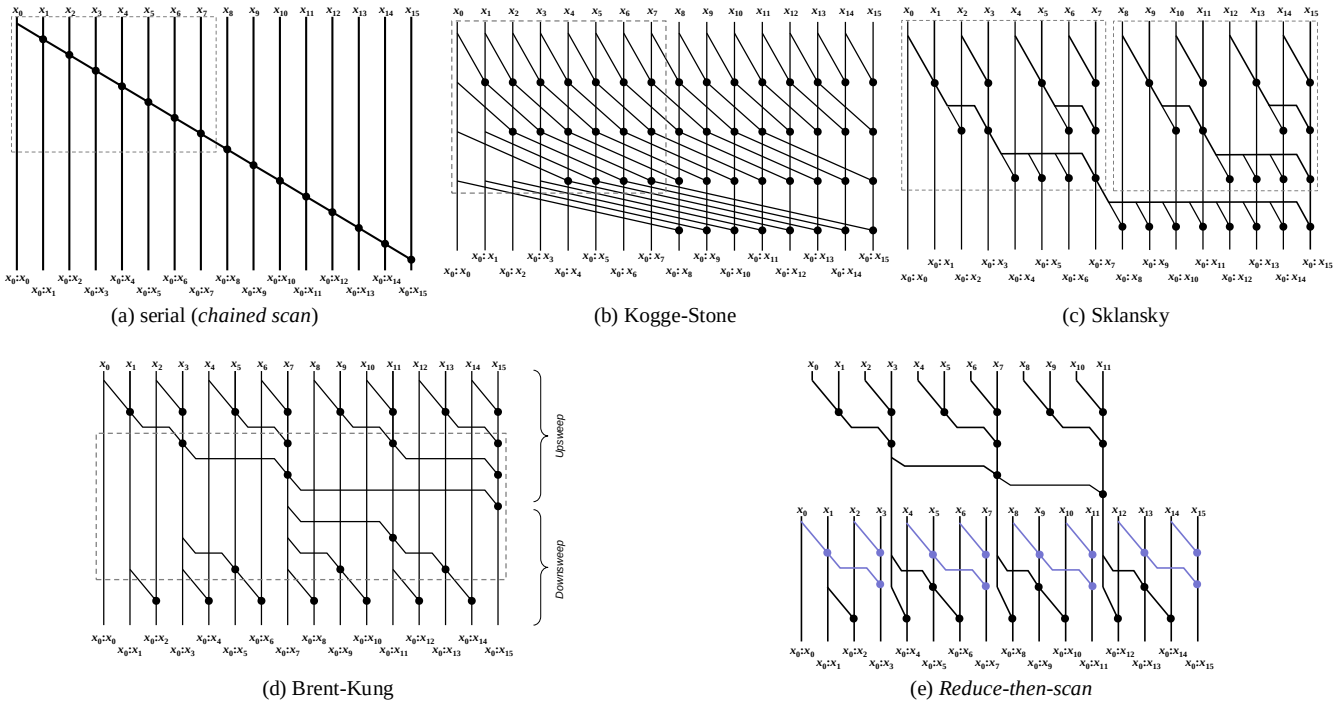
Contemporary GPU scan parallelization strategies such as *reduce-then-scan* are typically memory-bound, but impose ~3*n* global data movement [2, 16, 22]. Furthermore, they perform two full passes over the input, which precludes them from serving as *in-situ* global allocation mechanisms within computations that oversubscribe the processor. Finally, these scan algorithms cannot be modified for in-place compaction behavior (selection, run-length-encoding, duplicate removal, etc.) because the execution order of thread blocks within the output pass is unconstrained. Separate storage is required for the compacted output to prevent race conditions where inputs might otherwise be overwritten before they can be read.

Alternatively, the *chained-scan* GPU parallelization [11, 27] operates in a single pass, but is hindered by serial prefix dependences between adjacent processors that prevent memory I/O from fully saturating [27]. In comparison, our *decoupled-lookback* algorithm elides these serial dependences at the expense of bounded redundant computation. As a result, our prefix scan computations (as well as adaptations for in-place compaction behavior and *in-situ* allocation) are typically capable of saturating memory bandwidth in a single pass.

2. Background

Parallel solutions to scan problems have been investigated for decades. In fact, the earliest research predates the discipline of Computer Science itself: scan circuits are fundamental to the operation of fast adder hardware (e.g., carry-skip adder, carry-select adders, and carry-lookahead adder) [8, 24]. As such, many commonplace scan parallelizations are presented as recursively-defined, acyclic dataflow networks in the *circuit model* [7, 26] of parallel computation. In this model, prefix scan can be thought of as a forest of reduction trees, one for each output. Network size is reduced when reduction trees share intermediate partial sums. For practical use in computer software, scan networks are typically encoded as imperative algorithms in the *PRAM model* [13, 14].

The minimum circuit depth and size for a parallel scan network are $\log_2 n$ steps and n-1 operators, respectively. However, there are no known $O(n)$ work-efficient networks having depth $\log_2 n$. Snir provides a lower-bound regarding *depth+size optimality* (DSO) for parallel prefix networks: for a given network of size s gates and depth d levels, $d + s \geq 2n - 2$ [25]. His research



(a) serial (*chained scan*) (b) Kogge-Stone (c) Sklansky

(d) Brent-Kung (e) *Reduce-then-scan*

Fig. 1. Commonplace scan constructions for $n = 16$.
Dashed boxes illustrate recursive construction.

suggests that as circuit depth is increasingly constrained, DSO networks no longer exist and the size of the networks increases rapidly.

Fig. 1 presents commonplace scan networks relevant to contemporary prefix scan algorithms. Although the serial, or *chained scan*, construction in Fig. 1a has maximal n-1 depth and no concurrent computations, its minimal n-1 size makes it an attractive subcomponent of scan networks designed for oversubscribed processors. Increasing the computational granularity (i.e., items per thread) is a common technique for improving processor utilization by reducing inter-thread communication.

The Kogge-Stone construction [19] in Fig. 1b (and corresponding Hillis-Steele algorithm [17]) is a well-known, minimum-depth network that uses a recursive-doubling approach for aggregating partial reductions. Despite having inefficient $O(n\log_2 n)$ work complexity, its shallow depth and simple shared memory address computations make it an attractive strategy for SIMD architectures (e.g., GPU warps) where inactive processor resources cannot be scavenged[1].

The Sklansky construction [24] in Fig. 1c employs a recursive, *scan-then-fan* approach that also achieves minimum depth $\log_2 n$ at the expense of $O(n\log_2 n)$ work complexity. Compared to Kogge-Stone constructions, these networks exhibit high-radix fan-out when propagating partial prefixes computed by recursive subgraphs. This improved sharing leads to smaller circuit sizes and reduced memory bandwidth overheads.

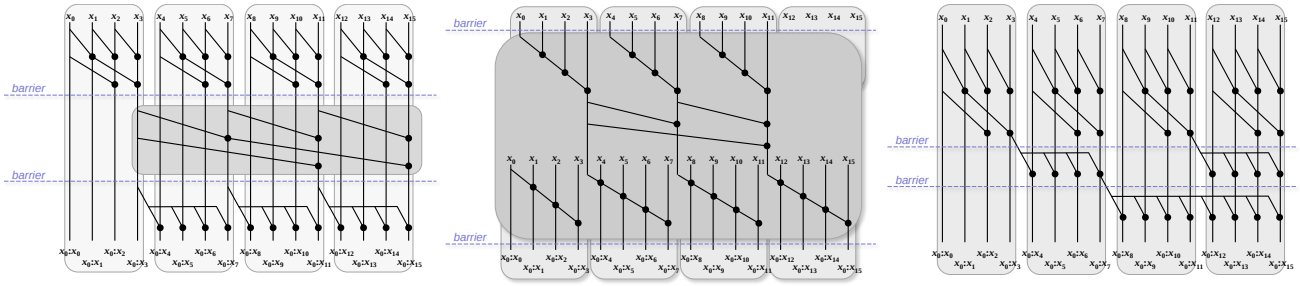
Whereas minimum-depth circuits are fast when the input size is less than or equal to the width of the underlying multiprocessor, minimum-size networks are important for larger problems. Many parallel programming models virtualize the underlying physical processors, causing overall runtime to scale with circuit size instead of depth. Therefore work-efficiency is often a practical design objective for general-purpose prefix scan algorithms.

The Brent-Kung construction [8] in Fig. 1d (and corresponding Blelloch algorithm [4, 5]) is a work-efficient strategy having $2\log_2 n$ depth and $O(n)$ size. Visually, the data flow resembles an "hourglass" shape comprising (1) an *upsweep* accumulation tree having progressively less parallelism, and (2) a *downsweep* propagation tree exhibiting progressively more parallelism. Generalizing the binary operators in the upsweep with radix-b scans and those in the downsweep with radix-b fans, the Brent-Kung strategy exhibits a more pronounced *scan-then-propagate* behavior (as illustrated in Fig. 2a).

For programming models that virtualize an unlimited number of processors, a concern with *scan-then-propagate* data flow is that ~n live values are spilled and filled through last-level memory between upsweep and downsweep phases when the input exceeds on-chip memory. To eliminate the writes, we can simply rematerialize the intermediates during the downsweep phase at the expense of $O(n)$ redundant calculations, as shown in Fig. 1e [9, 12, 22]. This has the effect of converting downsweep behavior from propagation to scan. We refer to this adaptation as the *reduce-then-scan* strategy.

In general, an important property of recursive network design is the ability to mix-and-match different strategies at different levels. Further variation is also possible through operator generalization: whereas these binary operators compute radix-2 scans and fans, network height can be reduced using radix-b subcomponents as building blocks [18]. This flexibility allows for

[1] Kogge-Stone "warpscans" are typical of GPU implementations where (1) SIMD-synchronicity has historically enabled efficient barrier-free communication, and (2) the hardware provisions a "shuffle" crossbar for efficient inter-warp communication.



(a) A radix-4 Brent-Kung *scan-then-propagate* strategy embedding Kogge-Stone warpscans and propagation fans

(b) A raking radix-4 *reduce-then-scan* strategy embedding serial reductions, serial scans, and a Kogge-Stone warpscan at the root

(c) A radix-4 Sklansky strategy embedding Kogge-Stone warpscans and propagation fans

Fig. 2. GPU hybrid block-scan strategies for $n = 16$ and SIMD warp size 4. Rounded boxes illustrate warp assignments.

hybrid design strategies that efficiently utilize the entire hierarchy of multiprocessor and memory organization.

3. Contemporary GPU scan strategies

Every GPU kernel is executed by a hierarchically-organized collection of threads. The individual threads executing the kernel function are grouped into thread blocks, and these blocks are grouped into kernel grids. The GPU uses thread blocks to efficiently manage the population of running threads on hardware multiprocessors (processor cores). Threads within the same block can cooperate through fast on-chip scratch memory and barrier synchronization.

Prefix scan implementations designed for GPUs are typically constructed from two levels of organization: (1) a global, coarse-grained strategy for computing a device-wide scan using radix-*b* sub-networks for {reduction | scan | propagation}; and (2) a set of local, fine-grained strategies for computing *b*-input {reduction | scan | propagation} within each thread block. We discuss the latter first.

3.1 Block-wide scan strategies

The blocking factor *b* is a function of thread block size and thread grain size, both of which are constrained by the physical resources of the multiprocessor and can be considered tunable constants. In practice, the tiles of block input are typically several thousand items each (e.g., $b = 2048$ items for 128 threads with 16 items per thread). Thus tile size *b* is uncorrelated to global input size *n*, and the asymptotic work-(in)efficiency of any particular block-wide scan construction will not affect that of the global strategy.

Because we desire memory-bound kernels, the primary performance goal of any underlying block-wide strategy is to be efficient enough such that the local computational overhead (inter-thread communication, synchronization, and scan operators) can be absorbed by an oversubscribed memory subsystem. In other words, the computational overhead for block-wide scan must be less than the I/O overhead for the thread block. Improving block-wide scan efficiency beyond this point may reduce power consumption, but will not affect overall performance. For traditional prefix-sum kernels, the block's I/O overhead is dictated by 2*b* data movement: *b* items read and written. For allocation and compaction scenarios, this efficiency constraint can be twice as tight if no output items are actually produced. In CUB, we treat the selection of local scan algorithm as a tuning parameter.

Fig. 2 illustrates several commonplace hybrid strategies for block-wide scan. Their different circuit sizes and depths make each suitable for different-sized data types and scan operators of different complexities. For illustrative purposes, we depict blocks of 16 threads comprised of 4-thread SIMD warps, with one item per thread. Increasing the thread-granularity would entail wrapping the constructs with an outer layer of serial intra-thread reductions and scans. Dotted lines indicate barrier-synchronized communication through shared-memory.

Fig. 2a presents a radix-4 Brent-Kung *scan-then-propagate* strategy that embeds 4-item Kogge-Stone warpscans and propagation fans. At the root of the hourglass, one of the warps is repurposed to warpscan the partial totals from each warp[2]. Only two block-wide barriers are needed if the width of the SIMD warp is greater than the number of warps in the block. This block-wide scan technique was first demonstrated in CUDPP [23] and is one of several block-scan components available in CUB [21]. In this example, the depth is 5 levels and size is 37 operators.

Fig. 2b presents a radix-4 "raking" *reduce-then-scan* strategy in which the entire computation is delegated to a single warp[3]. With 4 items per thread, the delegate warp performs efficient serial upsweep and downsweep networks within the registers of each thread. The root of the hourglass comprises a Kogge-Stone warpscan. Only two block-wide barriers are needed. This type of raking block-wide scan technique was first demonstrated in MatrixScan [12] and is one of several block-scan components available in CUB [21]. Although the network is relatively deep (9 levels), this approach exhibits minimal inter-thread communication and is very efficient (only 29 operators).

Fig. 2c presents a radix-4 Sklansky strategy in which 4-input Kogge-Stone warpscans are coupled with 4-output Sklansky fan propagation. It is also provided as a block-scan component within CUB [21]. Unlike the previous two strategies, the number of block-wide barriers increases with the log of the number of warps. In this example, the depth is 4 levels and size is 36 operators.

3.2 Global scan strategies

Historically, GPU scan implementations have primarily embodied one of three radix-*b* strategies at the global level: *scan-then-propagate*, *reduce-then-scan*, or *chained-scan*.

[2] This can also be considered a variation of the Han-Carlson network construction [15] in which the root of a Brent-Kung construction is simply replaced with a Kogge-Stone network.

[3] "Raking" is a parallel decomposition in which each thread consumes a non-overlapping partition of consecutive items. It is visually reminiscent of the tines of a rake being dragged at an angle along the ground [6].

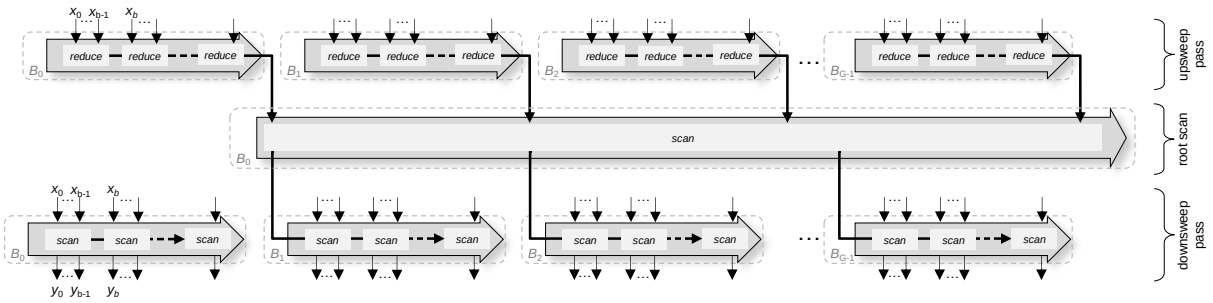


Fig. 3. Three-kernel *reduce-then-scan* parallelization among G thread blocks (~3n global data movement)

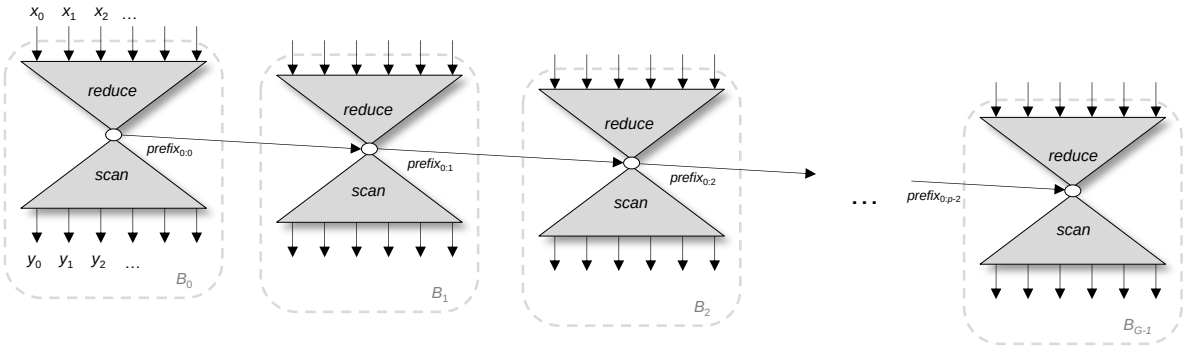


Fig. 4. Single-pass *chained-scan* prefix scan among G thread blocks (~2n global data movement)

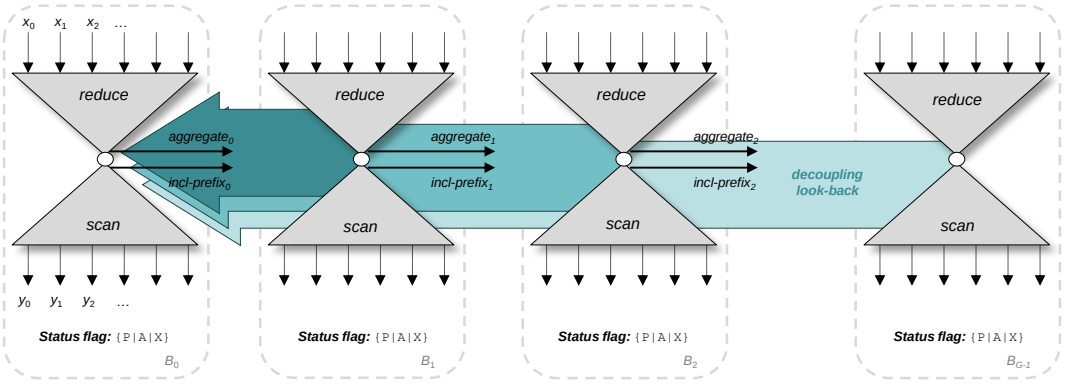


Fig. 5. Single-pass *adaptive look-back* prefix scan among G thread blocks (~2n global data movement)

Scan-then-propagate. The global scan implementations within CUDPP [10, 23] and Thrust [3] are examples of high-radix Brent-Kung data flow, recursively dispatching kernels of block-sized scan networks followed by kernels of block-sized fan propagation. Discounting the negligible I/O of inner levels, they incur ~4n global data movement, with the outermost kernels reading and writing ~n items each.

Reduce-then-scan. The global scan implementations within MatrixScan [12], B40C [1], MGPU [2], and by Ha and Han [16] are examples of *reduce-then-scan* dataflow, dispatching kernels of block-sized reduction networks followed by kernels of block-sized scan networks. MatrixScan does this recursively, whereas the other implementations employ a raking strategy in which the upsweep and downsweep thread blocks process multiple input tiles each, necessitating only a single root scan kernel.

As illustrated in Fig. 3, the input is partitioned evenly among G thread blocks, where G is the number of blocks that can be actively resident on the processor (and is uncorrelated to n). In the first kernel, each thread block reduces the tiles of its partition in an iterative, serial fashion. Then the small list of G block-aggregates is itself scanned. In the third kernel, each thread block iteratively computes a prefix scan across the tiles of its partition, seeded with the appropriate block-prefix computed by the scan of block-aggregates. By switching the behavior of the first upsweep thread block from reduction to scan, Ha and Han are able to elide the last block of the upsweep kernel and the first block of the downsweep kernel. The global data movement is ~3n (~2n items read, ~n items written).

Chained-scan. As an alternative, the *chained-scan* parallelization [27] is a single-pass approach in which thread blocks are each assigned a tile of input, and a serial dependence chain exists between thread blocks. Each thread block will wait on the inclusive prefix of its predecessor to become available. The global data movement is ~2n (n items read, n items written).

As illustrated in Fig. 4, the local block-wide scans are typically implemented using a parallel *reduce-then-scan* strategy. This allows the thread blocks to perform local upsweep and downsweep work in parallel, each core only having to wait at its phase-transition for the required prefix to be produced by its predecessor. The running prefix is then aggregated into the block-wide downsweep.

Chained-scan's performance is limited by the latency of signal propagation between thread blocks. For example, suppose a NVIDIA Tesla C2050 GPU processor with 144GB/sec memory bandwidth, processor cores operating at 1.15GHz, and a latency of 600 clock cycles to pass a message from one core to another through the memory hierarchy. This signaling latency limits the system to a scan throughput of 1.9M partitions/sec. If each core is assigned a partition of 256 inputs (32-bits each), the maximum-achievable scan throughput will be limited to 490M inputs/sec. This is well short of the theoretical memory bandwidth limitation of limit of 18B inputs/sec for simply reading and writing the data.

When limited by partition-signaling, one solution is to increase the partition size. (In the previous example, a partition size of 9000 inputs per thread block would be required to saturate memory bandwidth.) However, it may be impractical to increase the partition size arbitrarily. Many processor designs have limited storage capacity per core. For example, GPU processors provide virtualized cores whose performance is optimized when the working set fits completely within on-chip register file and shared memory resources. Our method, however, is insensitive to partition size and is therefore amenable to diverse architectural configurations.

4. Decoupled Lookback

4.1 Operation

Our method is a generalization of the *chained-scan* approach with dramatically reduced prefix propagation latencies. The idea is to decouple the singular dependence of each processor on its immediate predecessor at the expense of progressively redundant computation. Whereas the *chained-scan* approach has a fixed "look-back" of one partition, our method allows processors to inspect the status of predecessors that are increasingly further away. This is illustrated in Fig. 5.

As each partition is processed, its status is updated with the partition-wide aggregate. Each aggregate is simply the reduction of items within partition, and can be computed independently from other partitions. Because the aggregates are readily available, each processor is generally free to peruse the aggregates recorded for preceding partitions, progressively accumulating them until complete exclusive prefix is known (the running total across all prior partitions). The partition's status is then updated with the inclusive prefix, which is computed from the exclusive prefix and the partition-wide aggregate. The exclusive prefix is then used to begin the partition's downsweep scan phase. Furthermore, an opportunistic encounter with a predecessor's inclusive prefix permits early-termination of the look-back phase.

More specifically, each parallel processor within our method operates as follows:

1. ***Initialize the partition descriptor***. Each processor's partition is allocated a status descriptor having the following fields:
   - *aggregate*. Used to record the partition-wide aggregate as computed by the upsweep phase of partition-scan.
   - *inclusive_prefix*. Used to record the partition's inclusive prefix as computed by reducing *aggregate* with the accumulated look-back from preceding partitions.
   - *status_flag*. The flag describes the partition's current status as one of the following:

     A – aggregate available. Indicates the *aggregate* field has been recorded for the associated partition.

     P – prefix available. Indicates the *inclusive_prefix* field has been recorded for the associated partition.

     X – invalid. Indicates no information about the partition is available to other processors. All descriptors are initialized with status ***X***.
2. ***Synchronize.*** All processors synchronize to ensure a consistent view of initialized partition descriptors.
3. ***Compute and record the partition-wide aggregate***. Each processor computes and records its partition-wide *aggregate* to the corresponding partition descriptor. It then executes a memory fence and updates the descriptor's *status_flag* to ***A***. Furthermore, the processor owning the first partition copies *aggregate* to the *inclusive_prefix* field, updates *status_flag* to *P*, and skips to Step 6 below.
4. ***Determine the partition's exclusive prefix using decoupled look-back***. Each processor initializes and maintains a running *exclusive_prefix* as it progressively inspects the descriptors of increasingly antecedent partitions, beginning with the immediately preceding partition. For each predecessor, the processor will conditionally perform the following based upon the predecessor's *status_flag*:

   X **--** Block (or continue polling) until the *status_flag* is not *X*.

   A **--** The predecessor's *aggregate* field is added to *exclusive_prefix* and the processor continues on to inspect the preceding tile.

   P **--** The predecessor's *inclusive_prefix* field is added to *exclusive_prefix* and the look-back phase is terminated.
5. ***Compute and record the partition-wide inclusive prefixes.*** Each processor adds *exclusive_prefix* to *aggregate* and records the result to the descriptor's *inclusive_prefix* field. It then executes a memory fence and updates the descriptor's *status_flag* to ***P***.
6. ***Perform a partition-wide scan seeded with the partition's exclusive prefix.*** Each processor completes the scan of its partition, incorporating *exclusive_prefix* into every output value.

The computation of each processor can proceed independently and in parallel with other processors throughout steps 1, 3, 5, and 6. In Step 4 (decoupled look-back), each processor must wait on its predecessor(s) to finish Step 3 (record partition-wide reduction).

4.2 Properties

Our method has the following properties:

- ***Safety***. The algorithm will run to completion if the system guarantees forward-progress for all processors. Forward progress ensures that no processor will wait indefinitely in Step 4: every predecessor is free to record its *aggregate* in

Partition ID	0	1	2	3	4	5	6	7
Flag	*P*	*A*	*A*	*P*	*A*	*P*	*X*	*A*
Aggregate	2	2	2	2	2	2	-	2
Inclusive Prefix	2	4	-	8	-	12	-	-

Fig. 6. "Snapshot" of execution state during a small prefix sum problem computed by eight processors over eight intput partitions.

Step 3. (And aggregates are sufficient to compute an exclusive prefix.)

- ***Minimal waiting***. Blocking will be minimal for systems that provide fair or nearly-fair scheduling. Fairness ensures that all processors will have recorded their *aggregates* in roughly the same amount of time. This includes the *inclusive_prefix* of the first block. As a result, processors are generally to freely accumulate all of their predecessor aggregates with minimal blocking or waiting.
- ***Constant-bound look-back***. The amount of look-back is constant given a finite number processing elements. (All physically-realizable computer systems have a finite number of processors.) A constant amount of look-back ensures an optimal overall work-complexity of $O(n)$. This property is true regardless whether processors are assigned:
  - **A single partition of size *n*/*p***. There are only a finite number of preceding partitions for each processor to inspect
  - **Multiple partitions of constant size**. The assignment of n/p partitions is striped across the processors and partitions are processed one at a time to completion, i.e., $processor_0$ scans $partition_0$, $partition_P$, $partition_{2P}$, $partition_{3P}$, etc. Each partition may inspect at most p predecessors before it reaches the *prefix* recorded for a partition previously processed by the same processor.
- ***Accelerated signal propagation***. Under the *chained-scan* strategy, the act of sharing an *inclusive_prefix* is only capable able of releasing the processor's immediate decedent from blocking look-back. Under our strategy, the recording of a partition's *inclusive_prefix* is capable of releasing *all* decedent processors.
- ***Support for non-commutative scan operators***. Our method only applies the reduction operator to consecutive inputs (or consecutive partial reductions).

4.3 An Example

Suppose a small prefix sum problem computed by eight processors over eight partitions where, for illustrative purposes, the sum of each partition equals 2. The snapshot of execution state in Fig. 6 illustrates many of the relative processor schedules and short-circuiting opportunities that may manifest under our method. The state of each processor is as follows:

- $processor_0$ is finished. It has computed and recorded its *aggregate*, copied it to the *inclusive_prefix* field, and set its flag to ***P***.
- $processor_1$ has computed and recorded its *aggregate*, set its flag to ***A***, computed and recorded its *inclusive_prefix*, but has not yet updated its flag to ***P***. Its *exclusive_prefix* was obtained from its immediate predecessor's *inclusive_prefix.*
- $processor_2$ has computed and recorded its *aggregate*, set its flag to ***A***, but has not started inspecting predecessors.
- $processor_3$ is finished. It has computed and recorded its *aggregate* , set its flag to ***A***, computed and recorded its *inclusive_prefix*, and set its flag to ***P***. It used a look-back window of three predecessors to determine its *exclusive_prefix.*
- $processor_4$ has computed and recorded its *aggregate*, set its flag to ***A***, but has not started inspecting predecessors.
- $processor_5$ is finished. It has computed and recorded its *aggregate* , set its flag to ***A***, computed and recorded its *inclusive_prefix*, and set its flag to ***P***. It used a look-back window of two predecessors to determine its *exclusive_prefix.*
- $processor_6$ has not yet started.
- $processor_7$ has computed and recorded its *aggregate*, set its flag to ***A***, and is waiting on its immediate predecessor to valid.

4.4 Adaptations and Optimizations

This section describes various adaptations and optimizations that we have applied to our basic method.

Virtual processors. Many programming models such as CUDA employ an abstraction of virtual processors that are dynamically scheduled on the hardware's physical processors[4]. Each processor is given a numeric rank by which it can index its corresponding input partition. However, the runtime that schedules virtual processors may run them arbitrary order and without preemption. As a result, our original method is susceptible to deadlock in Step 4: the machine may be occupied with a set of virtual processors that will wait indefinitely for the results of predecessors that cannot be scheduled until the active set retires. This can be remedied by providing each virtual processor with an identifier that guarantees every preceding partition has been actively scheduled. For example, each virtual processor can obtain such an identifier upon activation by atomically-incrementing a global counter.

Fence-free descriptor updates. The descriptor updates in Steps 3 and 5 each require three memory operations: (1) an update to *aggregate* or *inclusive_prefix*; (2) a memory fence; and (3) an update to *status_flag.* The memory fences preserve a valid, consistent view of descriptors across all processors. Otherwise the compiler or the memory subsystem may reorder the updates to these fields (e.g., by first writing ***A*** to *status_flag* and then updating *aggregate*). Such orderings would invite a small window of time between write operations in which peer processors would be susceptible to a view of invalid state.

Memory fences can incur a performance penalty, however. This occurs when the fence implementation prevents write

[4] By oversubscribing physical hardware with an abundance of virtual processors, parallel programs can be made portable across computers having different processor configurations. Furthermore, "over-partitioning" is often a useful technique for mitigating transient load imbalances between partition workloads.

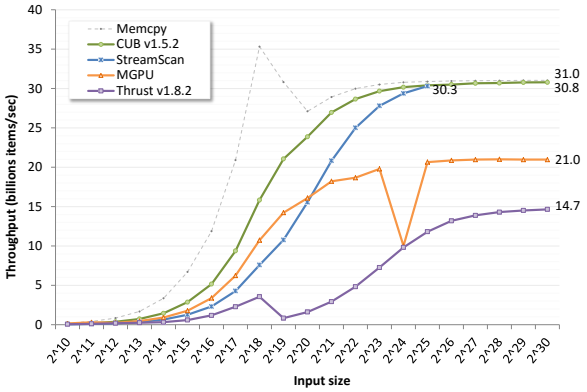


Fig. 7. 32-bit device-wide prefix sum throughput (NVIDIA Tesla M40, ECC off)

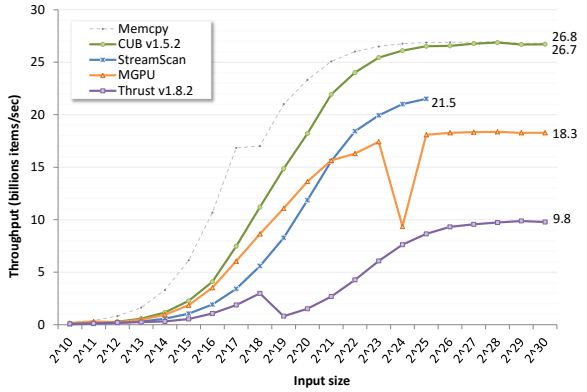


Fig. 8. 32-bit device-wide prefix sum throughput (NVIDIA Tesla K40, ECC off)

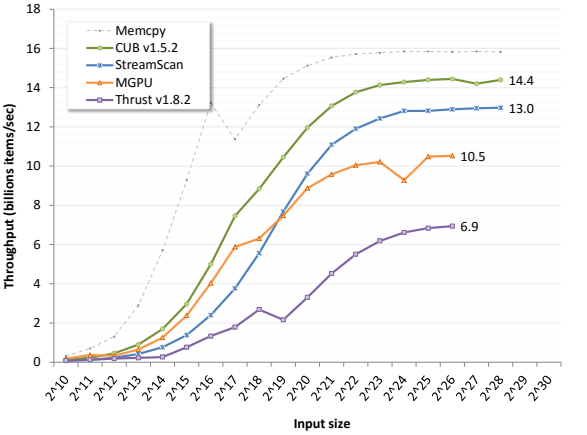


Fig. 9. 32-bit device-wide prefix sum throughput (NVIDIA Tesla C2050, ECC off)

operations from being pipelined or overlapped, i.e., the first write must complete before the latter can be made. This can result in unwanted signaling latency.

The fence can be eliminated if the status flag and the corresponding value being updated can be combined into a single architectural word. Consider prefix sum over 32-bit integers. If the architecture supports 64-bit loads and stores, a single 64-bit write of {***A***, 2} is sufficient to guarantee all peer processors will have a consistent view of partition status and the corresponding aggregate value of 2. Furthermore, these scenarios obviate the need for partition descriptors to maintain separate fields for aggregate and inclusive prefix; a simple *value* field will suffice.

Parallelized look-back. The latency of variable look-back can be further reduced by inspecting predecessors in parallel. A parallelization of Step 4 can be quite advantageous for processors having a SIMD style of parallelism. Compared to a single thread, the incremental memory and computational workload from having an entire SIMD group of threads participate is often negligible. The procedure of Step 4 is modified so that a set of *t* threads can simultaneously inspect a *window* of *t* preceding partitions, with each thread being assigned it to monitor its own predecessor:

- Threads poll (or block) until their respective predecessor is no longer flagged ***X*** (invalid).
- Once all threads have observed valid predecessors, the thread-set will conditionally perform one of the following based upon their status flags:
  - ***All predecessors have status A***. Each thread reads its predecessor's *aggregate*. A local reduction of these aggregates is computed and added to the running *exclusive_prefix*. The entire window is slid back *p* partitions and threads return the previous step to inspect the preceding block of predecessors.
  - ***At least one predecessor has status P.*** Each thread reads the corresponding *aggregate* or *inclusive_prefix* from its predecessor's partition descriptor. A local segmented-reduction of these values is computed in which each thread is flagged as being a segment-head if its predecessor has status ***P***. The last segment's total is added to the running *exclusive_prefix* and the look-back phase is terminated.

In-place compaction behavior. This one-pass scan strategy is also amenable for implementing parallel algorithms that exhibit compaction behavior (i.e., sequence transformations where only some of the data items are retained). Examples include *select-if*, *reduce-value-by-key*, *run-length-encode*, etc. Underlying these methods is a prefix sum over an array of binary flags, where a given flag is set if the corresponding data item is to be kept within the compacted output. For each data item, the prefix sum of the preceding flags equals the scatter offset for which that data item is to be written within the compacted output.

A beneficial consequence of using our single-pass design is that these compaction operations can operate in-place, i.e., without requiring a separate storage for the compacted output. Because of the signaling, each processor is guaranteed that all preceding processors have at least read their inputs. This allows a given processor to write its outputs to their compacted locations without risking overwriting a predecessor's inputs before it has had an opportunity to read them itself. For traditional three-pass versions of these algorithms, the parallel processors in downsweep stage operate independently of each other, and therefore have no guarantee regarding the safety of overwriting the inputs of preceding processors. To our knowledge, we are the first to present in-place solutions for these algorithms for the GPU machine model.

5. Evaluation

5.1 Comparison with contemporary GPU implementations

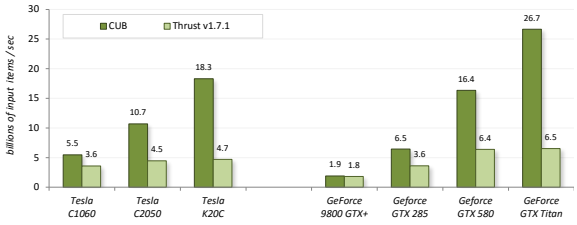
In this subsection, we evaluate 32-bit device-wide prefix sum performance as a function of problem size for the following GPU-based scan implementations:

	StreamScan	*MGPU*	*Thrust*
Saturated M40	1.00x	1.37x	2.08x
Saturated K40	1.23x	1.46x	2.73x
Saturated C2050	1.12x	1.47x	2.10x
H-mean saturated	1.11x	1.43x	2.27x

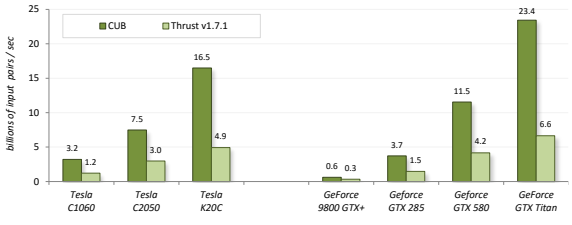
Fig. 10. CUB speedup for large inputs

	StreamScan	*MGPU*	*Thrust*
H-mean M40	1.62x	1.35x	2.99x
H-mean K40	1.67x	1.18x	2.87x
H-mean C2050	1.54x	1.20x	2.73x
H-mean all	1.60x	1.19x	2.80x

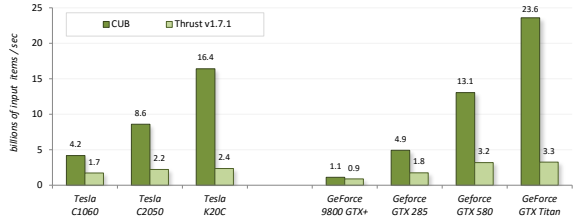
Fig. 11. Average CUB speedup



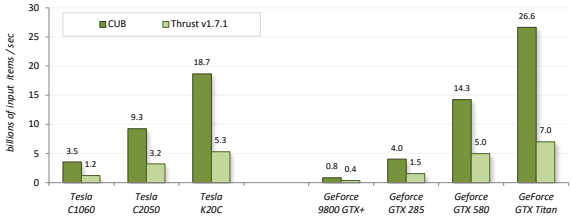
(a) *select_if*()
int32 data w/ 50% uniform-random selection



(b) *reduce_by_key*()
{int32,fp32} pairs w/ average segment length 500



(c) *partition_if*()
int32 data w/ 50% uniform-random selection



(d) *run_length_encode*()
int32 data w/ average segment length 500

Fig. 12. Performance of compaction-like algorithms across 32M inputs

- ***CUB*** [21]: Our single-pass *decoupled-lookback* parallelization with ~2*n* data movement (including the adaptations and optimizations described in the previous sections).
- ***StreamScan*** [27]: A single-pass *chained-scan* parallelization with ~2*n* data movement. StreamScan is a 32-bit implementation (OpenCL), which precludes very large problem sizes. Furthermore, it is auto-tuned per problem size.
- ***MGPU*** [2]: A three-kernel *reduce-then-scan* parallelization with ~3*n* data movement.
- ***Thrust*** [3]: A recursive *scan-then-propagate* parallelization with ~4*n* data movement.

We also measure the throughput performance of CUDA's global `memcpy` operation. Copy serves as an ideal performance ceiling for prefix scan because it shares the same minimum I/O workload, is completely data-parallel, and has no computational overhead.

We conducted our evaluation using the three most recent generations of NVIDIA Tesla GPU processors (all with ECC disabled): Maxwell-based M40 (Fig. 7), Kepler-based K40 (Fig. 8), and Fermi-based C2050 (Fig. 9). Our CUB performance meets or exceeds that of the other implementations for all architectures and problem sizes. For Kepler and Maxwell platforms, CUB throughput is able to match the performance ceiling of `memcpy` for large problems, and cannot be improved upon nontrivially.

Despite extensive per-input auto-tuning, StreamScan performance is hindered by the latencies of serial prefix propagation. This is manifest in two ways: (1) the roofline saturation of StreamScan throughput occurs at relatively higher problem sizes on all architectures, and (2) StreamScan is unable to match `memcpy` throughput on Fermi and Kepler architectures where on-chip resources (register file and shared memory) preclude blocking factors large enough to cover roundtrip L2 cache latency.

Furthermore, these results largely match our performance speedup expectations. If were to assume memory-bound operation for all implementations, we would expect speedups 1x, 1.5x, and 2x versus StreamScan, MGPU, and Thrust, respectively. In practice, for very large problems (capable of saturating the processor), we achieve harmonic mean speedups of 1.1x, 1.4x, and 2.3x, respectively. Fig. 10 further enumerates saturated CUB speedup per architecture, and Fig. 11 enumerates harmonic-mean CUB speedup for all problem sizes.

5.2 Adaptation for compaction behavior

Using CUB collective primitives for data movement, we have applied this single-pass scan strategy to construct very fast, performance-portable implementations of various compaction algorithms:

- *select-if*: applies a binary selection functor to selectively copy items from input to output
- *partition-if*: applies a binary selection functor to split copy items from input into separate partitions within the output

- *reduce-by-key*: Reduces segments of values, where segments are demarcated by corresponding runs of identical keys
- *run-length-encode*. Computes a compressed representation of a sequence of input elements such that each maximal "run" of consecutive same-valued data items is encoded as a single item along with a count of the elements in that run

These algorithms all make use of prefix sum as a method to determine where output items should be placed. The scatter offset for a given thread is the count of preceding items to be written by lower-ranked threads. Not only is the Thrust scan algorithm less efficient, but the process of item-selection must be either (a) be memoized in off-chip temporary storage, which consumes extra bandwidth; or (b) run twice, once during upsweep and again during downsweep. In comparison, item-selection can be fused with our prefix scan strategy without incurring additional I/O or redundant selection computation.

Fig. 12 illustrates the throughput of our CUB-based primitives versus the functionally-equivalent versions implemented by Thrust. Our implementations of these operations are 4.1x, 7.1x, 3.5x, and 3.8x faster, respectively.

6. Conclusion

Our method is a novel generalization of the chained-scan approach with dramatically reduced prefix propagation latencies. The principal idea is for processors to progressively inspect the status of predecessors that are increasingly further away. We demonstrate that, unlike prior single-pass algorithms, our method is capable of fully saturating DRAM bandwidth by overlapping the propagation of prefix dependences with a small amount of redundant computation. We have also shown this strategy to be amenable for implementing parallel algorithms that exhibit in-place compaction behavior, i.e., it does not require separate storage for the compacted output.

Another important distinction between our method and prior work is nondeterministic execution scheduling. Whereas contemporary scan parallelizations are constructed from static data flow networks (i.e., the order of operations is fixed for a given problem setting), our method allows parallel processors to perform redundant work as necessary to avoid delays from serial dependences. An interesting implication is that scan results are not necessarily deterministic for *pseudo*-associative operators. For example, the prefix sum across a given floating point dataset may vary from run to run because the number and order of scan operators applied by CUB may vary from one run to the next.

Our CUDA-based implementation is freely available within the open-source CUB library of GPU parallel primitives (v1.0.1 and later) [21].

References

[1] Back40 Computing: Fast and efficient software primitives for GPU computing: *http://code.google.com/p/back40computing/*. Accessed: 2011-08-25.

[2] Baxter, S. 2013. Modern GPU. NVIDIA Research. *http://nvlabs.github.io/moderngpu/*.

[3] Bell, N. and Hoberock, J. Thrust. *http://thrust.github.io/*. Accessed: 2011-08-25.

[4] Blelloch, G.E. 1990. *Prefix Sums and Their Applications*. Synthesis of Parallel Algorithms.

[5] Blelloch, G.E. 1989. Scans as primitive parallel operations. *IEEE Transactions on Computers*. 38, 11 (Nov. 1989), 1526–1538.

[6] Blelloch, G.E. et al. Solving linear recurrences with loop raking. 416–424.

[7] Borodin, A. 1977. On Relating Time and Space to Size and Depth. *SIAM Journal on Computing*. 6, 4 (1977), 733–744.

[8] Brent, R.P. and Kung, H.T. 1982. A Regular Layout for Parallel Adders. *Computers, IEEE Transactions on*. C-31, 3 (Mar. 1982), 260–264.

[9] Chatterjee, S. et al. 1990. Scan primitives for vector computers. *Proceedings of the 1990 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 1990), 666–675.

[10] cudpp - CUDA Data Parallel Primitives Library - Google Project Hosting: *http://code.google.com/p/cudpp/*. Accessed: 2011-07-12.

[11] CUSPARSE: *https://developer.nvidia.com/cusparse*. Accessed: 2013-06-05.

[12] Dotsenko, Y. et al. 2008. Fast scan algorithms on graphics processors. *Proceedings of the 22nd annual international conference on Supercomputing* (New York, NY, USA, 2008), 205–213.

[13] Fortune, S. and Wyllie, J. 1978. Parallelism in random access machines. *Proceedings of the tenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1978), 114–118.

[14] Goldschlager, L.M. 1982. A universal interconnection pattern for parallel computers. *J. ACM*. 29, 4 (Oct. 1982), 1073–1086.

[15] Han, T. and Carlson, D.A. 1987. Fast area-efficient VLSI adders. *Computer Arithmetic (ARITH), 1987 IEEE 8th Symposium on* (May 1987), 49–56.

[16] Ha, S.-W. and Han, T.-D. 2013. A Scalable Work-Efficient and Depth-Optimal Parallel Scan for the GPGPU Environment. *IEEE Transactions on Parallel and Distributed Systems*. 24, 12 (2013), 2324–2333.

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

[18] Hinze, R. 2004. An Algebra of Scans. *Mathematics of Program Construction*. D. Kozen, ed. Springer Berlin / Heidelberg. 186–210.

[19] Kogge, P.M. and Stone, H.S. 1973. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*. C-22, 8 (Aug. 1973), 786–793.

[20] Merrill, D. 2011. *Allocation-oriented Algorithm Design with Application to GPU Computing*. University of Virginia.

[21] Merrill, D. 2013. CUB. 1.0.1. A library of warp-wide, block-wide, and device-wide GPU parallel primitives. NVIDIA Research. *http://nvlabs.github.io/cub/*. Accessed: 2013-08-08.

[22] Merrill, D. and Grimshaw, A. 2009. *Parallel Scan for Stream Architectures*. Technical Report #CS2009-14. Department of Computer Science, University of Virginia.

[23] Sengupta, S. et al. 2008. *Efficient parallel scan algorithms for GPUs*. Technical Report #NVR-2008-003. NVIDIA.

[24] Sklansky, J. 1960. Conditional-Sum Addition Logic. *IEEE Transactions on Electronic Computers*. EC-9, 2 (Jun. 1960), 226–231.

[25] Snir, M. 1986. Depth-size trade-offs for parallel prefix computation. *Journal of Algorithms*. 7, 2 (Jun. 1986), 185–201.

[26] Valiant, L.G. 1976. Universal circuits (Preliminary Report). *Proceedings of the eighth annual ACM symposium on Theory of computing* (New York, NY, USA, 1976), 196–203.

[27] Yan, S. et al. 2013. StreamScan: fast scan algorithms for GPUs without global barrier synchronization. *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2013), 229–238.