Parallel Scan for Stream Architectures

Technical Report CS2009-14
Department of Computer Science, University of Virginia.
December 2009

Duane Merrill (dgm4d@virginia.edu) Andrew Grimshaw (grimshaw@virginia.edu)

Abstract

One of the most useful algorithmic primitives for parallel processing is *scan* (also known as prefix scan, prefix sum, prefix reduction, etc.). Because the computational granularity of concurrent scan tasks is so small, the memory bandwidth between physical processing elements and globally-visible storage banks is the limiting hardware resource. Current implementations of parallel scan for GPGPU stream architectures do not maximize memory bandwidth: they either make inefficient use of device memory accesses, are computationally-bound due to high dynamic instruction counts, or both. In this work we present three implementations of parallel scan that address these bandwidth inefficiencies. These new implementations are memory-bound, utilize 100% of achievable memory bandwidth, and only require the use of a constant amount of global device memory for the storage of intermediate results. On our target platform, all three provide a 1.7x performance speedup over the scan primitives provided by the CUDA Parallel Primitives (*CUDPP*) library, exhibiting up to a 64% reduction in dynamic instruction count. Our particular scan implementations are valuable in their own right, but more importantly we have developed a generalized design methodology that should allow us to construct bandwidth-optimal implementations for any stream device having similar machine and programming models.

Contents

1 Introduction

Algorithm designers rely on algorithmic primitives as basic building blocks for solving more complex problems. One of the more useful primitives for list and array processing applications is *scan* (also known as prefix scan, prefix sum, prefix reduction, etc.). Parallel scan can be found in a wide variety of problem domains, e.g., sorting, stream compaction, construction of trees and summed-area tables, etc. [1, 2, 3]. As a fundamental building block, implementations of scan play an important role within the software developer's "toolbox". Quality implementations of scan for parallel architectures are particularly valuable due to the difficulty of constructing concurrent code that is both correct and achieves maximal performance from the underlying hardware.

A salient characteristic of parallel scan is that the computational granularity of concurrent tasks is miniscule, often comprising only a single binary instruction (e.g., addition). This aspect of scan makes it particularly amenable to ultra-fine grained computational environments, e.g., directly within electronic circuits and, more recently, within stream architectures such as general-purpose graphics processing units (GPGPUs). The primary design consequence of such small computational granularity is that the memory bandwidth between physical processing elements and globally-visible memory banks will be the limiting hardware resource: raw memory bandwidth is often orders of magnitude slower than the accompanying computational throughput.

As an example, consider the NVIDIA GeForce GTX-285: thirty thread multiprocessor cores combine to provide $355 \text{x} 10^9$ binary scalar operations per second[1], yet the accelerator only provides a device memory bandwidth of $40 \text{x} 10^9$ words per second (where a word is 4 bytes). This translates into an 8.9x differential between computational and memory throughput. The majority of parallel scan strategies require roughly equivalent amounts of memory accesses and computational tasks, which leads us to the expectation that GPGPU implementations should be able to operate at peak memory throughput, even when considering the additional computations necessary for address calculations, synchronization barriers, and conditional expressions.

Unfortunately we find this not to be true. Current implementations of parallel scan for NVIDIA CUDA GPGPUs do not maximize memory bandwidth: they either make inefficient use of device memory accesses, exhibit high dynamic instruction counts, or both [8, 17, 21]. We argue that common stream programming patterns and idioms are responsible. The stream programming model forces programmers to make design and configuration choices that are largely unrelated to the problem at hand, yet significantly impact the ability for the underlying memory subsystem to move data to and from processor cores. For example, we find a large performance variance (up to 2.1x) amongst what we consider to be reasonable configurations for threadblock size, patterns of loads and stores, and other data-movement concerns.

Additionally, the stream programming paradigm fundamentally encourages programmers to decompose problems in ways that map individual data elements to their own logical threads of execution. This data-parallel pattern leads to streaming computations in which the numbers of threads and, more importantly, their corresponding grouping constructs scale with the problem size. As a result, these implementations make unnecessary use of global device memory in proportion to problem size in order to store the results of intermediate computations.

1.1 Contributions

In this work we present three implementations of parallel scan that address these bandwidth inefficiencies: *merrill_tree*, *merrill_srts*, and *merrill_linear*. These new implementations are memory-bound, utilize 100% of achievable memory bandwidth, and only require the use of a constant amount of global device memory for the storage of intermediate results. Our implementations also perform well on all problem sizes: previous CUDA implementations have been shown to perform poorly on problem sizes that are not multiples of powers-of-twos or impose unnecessary caps on problem sizes. Our

[1] (30 stream multiprocessors) x (8 threads active threads per clock cycle) x (1.48GHz clock)

implementations have no problem size limitations (other than the amount of global device memory on the accelerator) and exhibit a smooth, monotonic performance curve as problem sizes increase. The result is a ~1.7x performance speedup over the scan primitives provided by the CUDA Parallel Primitives (*CUDPP*) library.

The primary consequence of this work is that we have shown the ability to construct parallel scan (and reduction) implementations for GPGPU stream architectures that fully leverage the underlying device memory bandwidth. In the process of doing so, we have made several important contributions.

We have developed new algorithms for implementing parallel prefix circuit strategies in programmable hardware. Our efforts were specifically focused on designing depth-optimal algorithms for performing computation within the SIMD width and work-optimal algorithms for avoiding common architectural hazards in an effort to reduce unexpected work overhead. Interestingly, we could not find any published iterative algorithms for the depth-optimal Sklansky construction, so we present several versions of our own for SIMD use. One of these approaches, in particular, illustrates the use of what we call a *thread-specialization table*, a novel stream programming mechanism for orchestrating divergent intra-warp behavior without paying the penalties associated with divergent instruction flow. In addition, we have developed an algorithm implementing the Brent-Kung construction that is much more effective at avoiding memory bank conflicts than a Blelloch implementation that has been supplemented with padding techniques. Although not all of these algorithms found their way into the three scan implementations that we constructed for our particular hardware, they may prove useful for other stream platforms exposing different architectural details (e.g., relaxed coalescing requirements, thread-specialization tables, etc.).

Our three particular scan implementations are valuable in their own right, but more importantly we have developed a generalized design process that should allow us to construct bandwidth-optimal implementations for any stream device having similar machine and programming models. The process is a top-down approach that begins with a flexible meta-strategy for hierarchical composition and then lets the hardware and architectural details guide the selection of suitable algorithms and configuration parameters towards a platform-specific solution. Decisions within the design process are intended to be of the convenient "drop-in" type, e.g., "*select algorithm X during phase Y*", "*use data-movement configuration Z for memory level M*", etc. The entire process is driven by the single requirement that the resulting implementation make optimal use of the limiting memory resource (i.e., device memory, in the case of GPGPUs). This entails operating at peak memory bandwidth while minimizing the number of accesses made to that memory space.

In regards to parallel scan, our meta-strategy of choice is *reduce-then-scan*. This meta-strategy allows phases of independent work to be composed with a minimal amount of interaction, perfect for the hierarchical nature of stream processor architectures. The work for each phase (i.e., meta-timestep, if you will) can be performed by any algorithm implementing reduction or scan, allowing the composition to leverage different algorithms during different phases of computation that play to the strengths of each.

We employ a philosophy in which we decompose the design process into separate, orthogonal concerns. For example, the meta-strategy calls for functionally-abstract reduce and scan kernels. When constructing them, we first survey the performance landscape of various data-movement "kernel skeletons" for a specific hardware platform. These skeletons are varied in terms of how threads should be scheduled on the streaming multiprocessors and how those threads move data to/from the global device memory. After selecting an optimal configuration for moving data through the stream processor cores, we can turn our attention to solving smaller problem instances of reduce and scan whose sizes have been determined by the data-movement configuration.

1.2 Report Organization

Section 2 presents an overview of stream architectures, the scan problem and its variants, and a brief overview of the models of computation (Circuit Families and PRAM) most commonly used to reason about scan strategies and algorithms. Section 3 reviews strategies for parallel scan (with a particular focus on more recent incarnations for GPGPU architectures) and presents several new algorithms that avoid common architectural hazards. Section 4 describes our design methodology,

specifically how we derived our particular scan implementations. Section 5 provides performance analyses and Section 6 discusses avenues of future work.

2 Background

2.1 Stream Architectures

Modern GPGPU stream architectures are intended to operate in conjunction with a "host" platform containing one or more scalar CPUs and an I/O chipset. They are often considered as co-processors or accelerators because they rely on the host platform to orchestrate data movement and program invocation. The two primary facets of the streaming paradigm are the abstract machine model and the programming paradigm. We provide a general review of both in this subsection, supplemented with specific details of the NVIDIA CUDA platform.

2.1.1 Stream machine model

In the stream machine model, many hardware-scheduled execution contexts, or *threads*, run copies of the same imperative program, or *kernel*. While exhibiting many characteristics of SIMD (single instruction, multiple data) and SPMD (single program, multiple data) architectures, the stream machine model does not quite fit either category.

An architecture that is purely SIMD entails a single instruction stream for the entire set of processing elements. The attractiveness of this genre lies in its data-parallel nature and the more practical implications of a single hardware unit that is responsible for instruction issue. The primary drawback is that the synchronous execution of all processing elements can lead to significant underutilization, particularly during memory stalls and program divergence (when subsets of processors sit idle as both halves of a conditional are executed).

Instead of managing a single instruction stream for all processing elements, it is typical for modern stream processors to implement smaller, fixed-size SIMD groupings of execution contexts (known as *warps* in CUDA parlance). As a SIMD grouping, all threads of execution within a warp share a single instruction stream. A warp itself is executed on a SIMD core, or *stream multiprocessor* (SM) that is comprised of homogeneous processing elements (i.e., ALUs). Distinct warps are not run in lockstep and may diverge, which implies that these stream architectures do not meet the strict SIMD definition.

Although warps constitute independently executing instances of the same program, the stream model does not quite meet the SPMD definition either: memory spaces are primarily shared rather than private. Stream machine models typically expose three levels of explicitly managed storage spaces that vary in terms of visibility and latency: per-thread registers, shared memory that is local to a collection of warps running on a particular thread multiprocessor, and a large off-chip global device memory that is accessible to all threads. A kernel program must explicitly move data from one memory space to another.

Modern GPU processor dies typically contain several tens of stream multiprocessor cores. Each SM contains only enough ALUs to actively execute a single warp[2], yet maintains and schedules amongst the execution contexts of many warps. This approach is analogous to the idea of symmetric multithreading (SMT); the distinction being that that the SM hardware is multiplexing amongst warp contexts instead of individual thread contexts. This translates into tens of warp contexts per core, and tens-of-thousands of thread contexts per GPU die.

This style of SMT enables stream architectures to hide massive amounts of latency by switching amongst warp contexts when architectural, data, and control hazards would normally introduce stalls. The result is a more efficient utilization of fewer physical ALUs. The net effect of this latency-hiding is that maximal instruction throughput occurs when the number of thread contexts is much greater than the number of ALUs on the GPU die, allowing the system to behave as if it had many more physical processors than it does.

[2] Streaming multiprocessors within the NVIDIA Fermi architecture feature enough ALUs to actively run two warps.

The heavy reliance upon SMT techniques for hiding latency has several practical design implications. By implementing fewer ALUs, the memory subsystem can be made more efficient because there are fewer physical destinations that must be interconnected. The latency-tolerant design philosophy implies that expensive techniques for coherent demand-caching, out-of-order execution, branch prediction, speculative execution, and other performance mechanisms employed by traditional scalar processors need not figure prominently. The result is that significant portions of power, space, and transistor budgets are freed up, allowing them to be spent instead on facilities for warp scheduling and maintenance, ALUs, and simple shared memories.

2.1.2 Stream programming paradigm

The stream programming paradigm is fundamentally different from those intended for general purpose CPUs. Although stream processor ISAs present an imperative style of programming, they do not support common idioms such as the program stack or an I/O interrupt model. The lack of a suitable memory space precludes the ability to maintain an arbitrarily-sized program stack for each thread context: "auto-variables" are stored within a thread context's variable-sized register file. As such, stream kernels cannot be programmed to leverage stack-based recursion. The lack of an interrupt model precludes stream kernels from being able to react to external events, such as device I/O notifications or memory signals akin to page-faults. Additionally, the stream instruction and data regions are kept separate in order to avoid coherence issues, preventing stream kernels from self-modifying their own code.

At its heart, the stream programming paradigm involves constructing a single kernel function that is run by many threads of execution. In order to provide logical problem decomposition in a manner that facilitates execution decomposition across the hardware streaming multiprocessors, programming models often expose hierarchical grouping constructs for threads. The CUDA programming framework exposes two levels of grouping: a *threadblock* of individual threads that share a local shared-memory space, and a *grid* of homogeneous threadblocks that encapsulates all of the threads for a given kernel. (The SIMD warp is not a first-class programming construct.) Threads are uniquely identified by the combination of their rank in their threadblock and that threadblock's rank in the grid.

Cooperation amongst threads is based on the bulk-synchronous model: coherence in the memory spaces is achieved through the programmatic use of synchronization barriers. Different barriers exist for the different memory spaces: threadblock synchronization instructions exist for threads within local shared memory, and global memory is guaranteed to be consistent at the boundaries between kernel invocations because the executions of sequentially-invoked kernels are serialized. The idea is that the programmer only has to reason about consistency at certain points during program execution instead of having to consider all possible interleavings.

Similar to SPMD programming models like MPI and PVM, each thread of execution can use its identifier to determine which portion of the problem to operate on, and programmers can write control flow statements to allow threads to further specialize their execution. In CUDA, threadblocks and grids can be enumerated in up to three-dimensions, encouraging programmers to decompose data-parallel problems spatially. A common idiom is to create a thread for each data element in the input set.

In this style of data decomposition, a stream kernel is simply a short, finite function written from the point of view of a single thread. The thread determines its identity, reads its corresponding input elements from global device memory, performs some small computation (possibly involving local cooperation), and writes its result back to global device memory. The host platform orchestrates the global flow data by repeatedly invoking new stream kernel instances, each containing a grid of threads that is initially presented with a consistent view of the results from the previous kernel invocation. The dependence of the stream architecture upon the host platform for kernel invocation (and therefore global device memory consistency) generally implies that all stream kernels must terminate, thus limiting their application to the class of decidable problems.

Like many programming models, the streaming paradigm (and CUDA in particular) struggles to provide abstractions that facilitate elegant, portable solutions while fully leveraging the underlying hardware. Unfortunately many of the architectural details that are abstracted from the programmer have drastic effects upon performance, which is arguably the primary reason

for availing oneself of parallel hardware in the first place. As we discuss further in the following sections, architectural considerations such as SIMD widths, memory bank conflicts, coalescing requirements for global device memory operations, and write-after-read hazards all figure prominently in the design of implementations that maximize the efficiency of the hardware.

2.2 Definition of Scan and Related Primitives

Scan is a higher-order function that takes as input an n-element list $[x_0, \ldots, x_{n-1}]$ and a binary associative combing operator $\oplus$ and produces an equivalently-sized output list $[y_0, \ldots, y_{n-1}]$. There are several variations of scan, all of which share the characteristic that the i^{th} output element is a function of the previous input elements (i.e., it has a prefix dependency). The most common version, *exclusive scan*, is defined as:

$$scan_{exclusive}\left([x_0, \ldots, x_{n-1}], \oplus\right) = [y_0, \ldots, y_{n-1}] \quad \text{where}$$

$$y_i = \bigoplus_{0 \le a \le i-1} x_a \quad \text{when } 0 < i < n$$

$$= id_{\oplus} \quad \text{when } i = 0$$

Because each output element depends exclusively on the previous inputs, there is an issue as to how the first output value, y_0, should be defined. Exclusive scans rely upon the existence of an identity element $id_{\oplus}$ for a given combining operator $\oplus$. An identity element must have the property that $x_a \oplus id_{\oplus} = x_a$, which allows exclusive scan to be well-defined in that $y_0 = id_{\oplus}$. (For example, $id_{+} = 0$ for addition, $id_{*} = 1$ for multiplication, etc.) As an example, consider the following application of an additive, exclusive scan:

$$scan_{exclusive}([8, 6, 7, 5, 3, 0, 9], +) = [0, 8, 14, 21, 26, 29, 29]$$

The prefix dependency is a common characteristic of problems involving ordered lists of elements. This is reflected in that many software libraries contain implementations of several scan variants. *Inclusive scan* is similar to exclusive scan, with the exception that the i^{th} output element is also dependent on the i^{th} input element, i.e., $y_i = \oplus(x_0, \ldots, x_i)$. *List compaction* is a form of additive exclusive scan in which the input elements $\in \{0,1\}$ (e.g., "invalid" and "valid"): the resulting vector can be used to map sparsely arranged data onto a more compact data structure. *Reverse scan* (also known as *backward scan*) processes the input elements with a "postfix dependency", i.e., $y_i = \oplus(x_{i+1}, \ldots, x_{n-1})$. *Segmented scan* is a composition of scan instances: the input is a sequence of list segments, typically delineated by marker flags, each of which is to be scanned separately.

2.3 Models for Parallel Scan

Two of the most popular models of parallel computation are the *parallel random access machine* (PRAM) model, and the *circuit-families* model. Significant contributions to parallel scan research have been made under the umbrellas of both models. In this subsection we describe these contributions and the salient characteristics of each.

2.3.1 The circuit-families model

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). Because of its history with low-level hardware applications, a significant portion of parallel scan research is grounded in the *circuit model* of parallel computation under the heading of "parallel prefix".

The circuit model of computation appears to lend itself well to reasoning about parallel strategies for stream architectures: it provides a clear, concise, and elegant way of describing vast numbers of extremely simple processing elements and the dataflow relationships of the dependencies between intermediate values. A circuit describes, in the general case, a directed acyclic graph of simple stateless processors (e.g., gates) connected by wires that convey processor output values to the input ports of other processors. The stateless processors that do not share an ancestor-descendant relationship can operate in

parallel, and the entire circuit is responsible for transforming one or more input values into one or more output values. A *circuit family* is a potentially infinite list of circuits (C_0, C_1, C_2, ...), where circuit C_n solves the given problem for an input of size n.

Circuit families are usually evaluated by their *size* and *depth* complexities. The size complexity of a circuit family is a measure of the number of processors as a function of n, and often equates to construction cost and power consumption. The depth complexity of a circuit family is a measure of the length of the longest path from an input value to an output value. It is also described as a function of the input size, n, and is a performance indicator for how long the computation will take.

2.3.2 A circuit theory of scan

A scan circuit C_n transforms n input values into n output values. Such a circuit can be thought of as composition of n binary *reduction* circuits, each producing a single output. A reduction, or fold, is a higher-order function that uses a combining operator $\oplus$ on a list of items to aggregate a single result value.

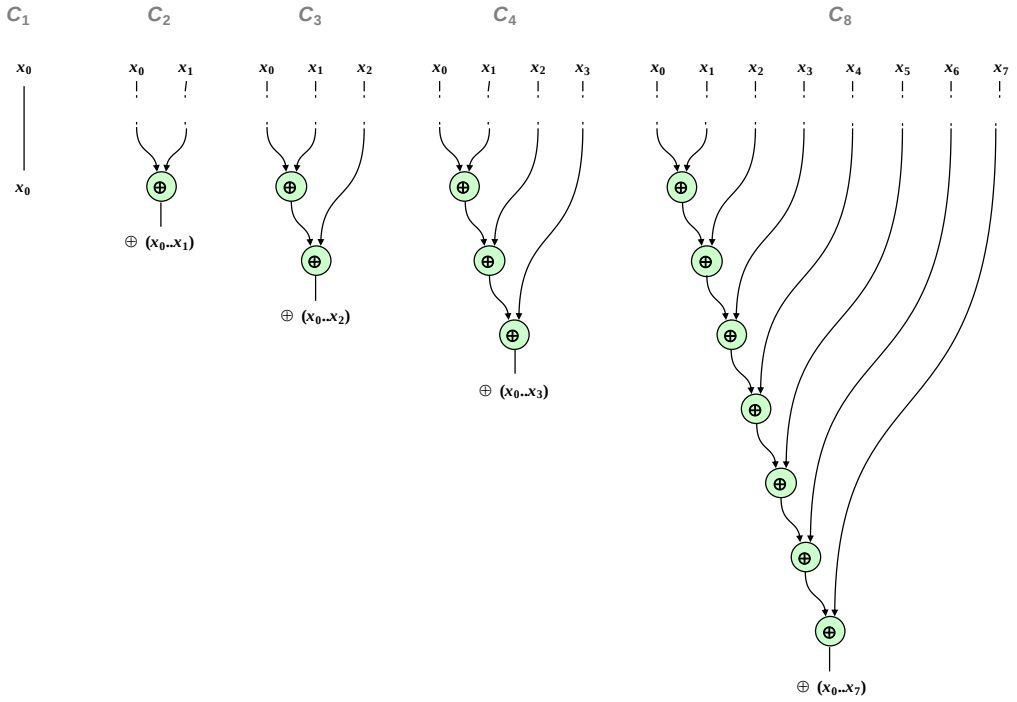


Figure 1. Circuits C_1, C_2, C_3, C_4, and C_8 of a serial reduction circuit family constructed from the binary associative combining operator $\oplus$.

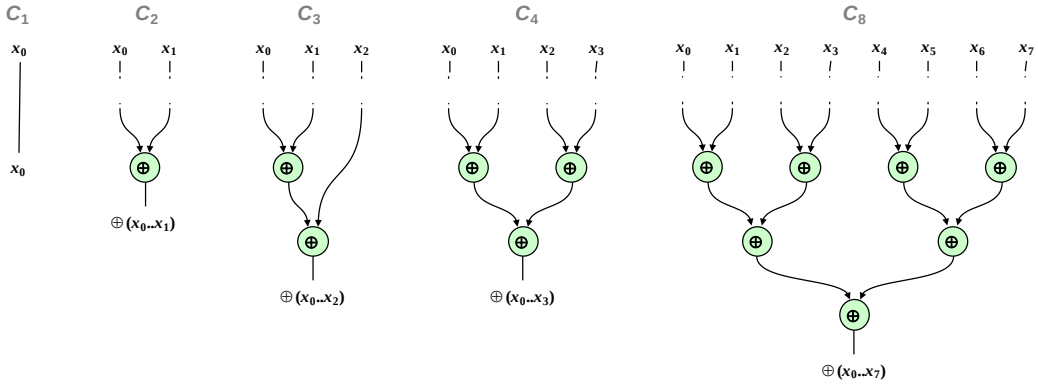


Figure 2. Circuits C_1, C_2, C_3, C_4, and C_8 of a balanced-tree parallel reduction circuit family constructed from the binary associative combining operator $\oplus$.

Figures 1 and 2 above depict two different circuit strategies for binary reduction. Figure 1 employs a serial reduction strategy that exhibits both size and depth complexities equal to n - 1. Computation for this strategy is amenable for mapping onto a uniprocessor as it requires that only one processing element and one intermediate value be "live" at any given time during computation. Figure 2 depicts a parallel reduction strategy based upon balanced binary trees that exhibits a size complexity $s(n) = n - 1$ and an optimal depth complexity $d(n) = \lceil \log_2 n \rceil$. Both figures depict "wire discontinuities" between the set of input values and the logic for a given reduction circuit: the associative nature of $\oplus$ allows them to be "wired in" arbitrarily.

Although the reduction trees for a given scan could be implemented in isolation and executed in parallel, it is much more efficient to compose them together in such a manner so as to calculate and share the same intermediate values, reducing the size of the scan circuit needed. In the serial strategy, it is easy to see that the reduction circuit C_n yields a superposition of all C_1 .. C_n reduction trees when the input value x_k is wired into the operator at depth k: each intermediate value at depth k is the output of the kth reduction.

The design space for parallel prefix circuits, i.e., all possible superpositions of the parallel reduction trees, is quite large. The primary axes within this space are depth, size (both in gates and wiring tracks), and fan-out, all of which affect tradeoffs for speed, power consumption, wiring capacitance, and signal timing.

Perhaps the most important result from the theory of prefix circuits is Snir's proof regarding the size-depth tradeoff for parallel prefix networks [4]: for a given network of size s gates and depth d levels, $d + s \geq 2n - 2$. The amount by which a given prefix network misses the depth-size lower bound of $2n - 2$ is called its *deficiency*. A network with zero deficiency is called *depth-size optimal* (DSO). It is easy to see that the serial prefix network is DSO. For loose depth constraints, a linear tradeoff is observed between the depth and size of DSO networks. However, if the depth constraint is too tight, DSO networks no longer exist and the size of the networks increases rapidly. Much research has been expended towards finding DSO prefix networks of smallest possible depth: the definition of this boundary as a function of problem size is an open problem.

2.3.3 The PRAM model

As scan was popularized as a generic problem-solving primitive, it became clear that there was a strong need for software implementations for programmable architectures, the benefit being that a fixed number of processing elements can be reused to solve arbitrarily-sized problems. The vast majority of parallel algorithms are presented as stored procedures for the *PRAM model* of computation. In the PRAM model, arbitrarily many processors are allowed to perform concurrent operations on cells of a shared memory space, the access times of which are considered to be uniformly constant for that space. This leads to an imperative style of control in which a finite-length procedure can emulate the computations of an entire circuit family: the algorithm enumerates the steps by which the live data values for each circuit level are to be mapped into the cells of specific memory space(s) and the order of operations performed upon them. As a generalized model, it provides no details for how those operations are to be scheduled onto an actual (finite) set of physical processing elements.

In a sequential model of computation, the *time complexity* for an algorithm is a function of the input size that describes the total number of operations, or steps that a machine must make before it halts and returns an answer. In a parallel model of computation, however, multiple operations can be performed in a single step. This necessitates a distinction between the cumulative amount of work being done and the number of timesteps needed to perform it. These separate notions are respectively termed *work complexity* and *step complexity*, and we use them here to evaluate scan strategies, placing a particular emphasis on the practical constants involved.

With an unbounded number of processing elements, algorithmic work complexity and step complexity directly correspond to circuit size and depth complexities. When the amount of concurrent work available exceeds the number of physical processing elements, however, portions of concurrent work must be serialized as the processing elements are reused. As a

result, the observed step complexity diverges from the depth complexity of the corresponding circuit strategy and instead becomes proportional to the cumulative work complexity, i.e., runtime $\propto$ work/processors. For the scan problem, the degree of concurrency will generally dominate the number of parallel processors: virtually every published algorithm has a degree of concurrency that bursts at least as high as 50% of the problem size. (For example, the concurrent workload available for a problem instance containing 64M elements is generally much greater than even the 30K+ hardware thread contexts provided by today's GPGPUs.)

For problem sizes too small to saturate the hardware, however, runtimes will adhere to the circuit depth complexity which, in many cases, is asymptotically less than the PRAM work complexity. The benefit of programmable hardware is that self-adapting hybrid solutions can leverage different parallel strategies with asymptotically different work and step complexities depending on the particular combination of problem size and processing elements at hand.

The PRAM model is particularly convenient for expressing parallel algorithms because computation is described using a simplified pseudo-code that closely resembles the low-level paradigms for how data is accessed and manipulated within the memories of actual computers. PRAM descriptions of algorithms are attractive because they are:

- *Realizable*. Unlike the circuit model, PRAM pseudo-code is expressive enough to describe how data is to be manipulated by abstractions of real computer architectures.

- *Portable*. Its pseudo-code nature makes it generalizable enough for implementation on most platforms with relative ease.

Unfortunately there are corresponding drawbacks for each of these benefits:

- *Semantic obfuscation*. In one sense, PRAM pseudo-code is too detailed: the specific layout by which program state is represented within memory cells can often obscure the general strategy, making it hard for the reader to compare, contrast, and classify different algorithms. It is for this reason that most presentations of PRAM algorithms are accompanied by data-flow illustrations that resemble circuit families.

  As an example of this deficiency, Section 3 describes how the Brent-Kung circuit strategy for parallel scan can be implemented by several distinct PRAM algorithms. They are all isomorphic in the sense that they preserve the same task dependencies, yet are not intuitively so because they employ radically different schemes for manipulating live state within memory.

- *Type-architecture mismatches*. In this sense, PRAM algorithms are often not detailed enough: there are unaddressed aspects of the physical hardware architecture having critical performance and correctness implications. These type-architecture mismatches typically stem from two (potentially interrelated) aspects: (1) how tasks are actually scheduled onto physical processors, and (2) physical memories that violate the constant-access-time assumption.

  As an example of (1), we show in Section 3 that PRAM algorithms implemented for architectures having self-scheduled threads of execution can result in an asymptotically greater work complexity than purported by the algorithm.

  In general, many programmers (and researchers!) make the mistake of putting undue worth into the depth-complexity of a given parallel algorithm. The reality is that depth-complexity is often a useless metric: the heavy reliance upon symmetric multithreading techniques by the underlying hardware leads to an enormous ratio between concurrent tasks and the number of physical processing elements, making work-complexity a far more important metric.

  As an example of (2), we show in Section 3 that even local memory access time can be widely variable, in this case due to the complete serialization of data accesses that arise from memory bank conflicts.

3 Scan Strategies and Algorithms

3.1 Kogge-Stone

3.1.1 Circuit strategy

The Kogge-Stone construction [19] is a well-known, minimum-depth parallel prefix network. It has a small, constant fan-out of two, making it one of the fastest constructions when implemented directly within electronic circuitry. The construction's low fan-out and optimal $\log_2 n$ depth complexity result in a high work complexity: the circuit family requires exactly $n\log_2 n - (n - 1)$ binary operations. The Kogge-Stone strategy is termed *work-inefficient* because of the existence of alternative scan circuits that exhibit linear size.

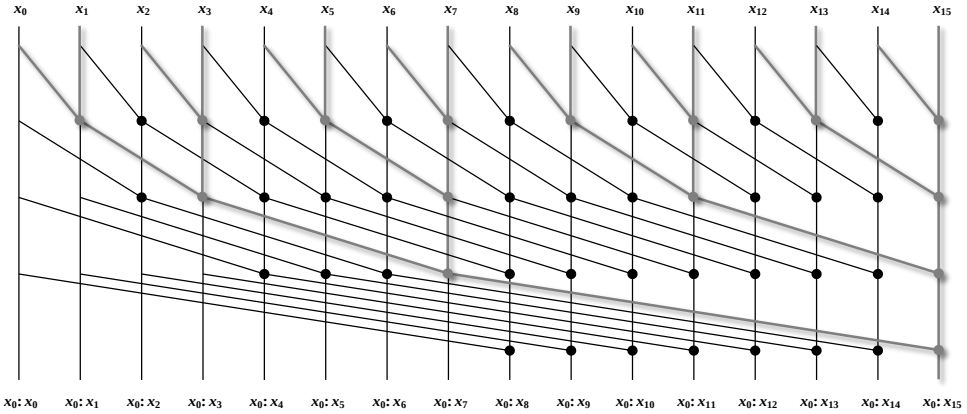


Figure 3. The Kogge-Stone parallel inclusive scan strategy for n=16 elements. The result is computed in four levels using 49 operators. The *spine* is highlighted in gray: it is the largest sub-tree in the network and is used to compute the last output value, $x_0{:}x_{15}$.

The Kogge-Stone strategy works by progressively building partial reductions from consecutive inputs. Figure 3 depicts the flow of computation for a 16-element input list. Each circuit level d produces $2^{(d-1)}$ final output values, each of which is wired into the computation of an output value in every subsequent level. Each level d also produces $n - 2^d$ intermediate values, each of which is wired into two computations of the next level.

3.1.2 Hillis-Steele PRAM algorithm and variants

The strategy is easily implemented for a random-access machine architecture consisting of n virtual processors and a memory m containing n linear memory cells populated with the elements of input list x. Hillis and Steele are popularly credited for their PRAM algorithm for doing so, first presented in the context of the Connection Machine [3]. More recently Horn [5] and Hensley [6] showed how the strategy could be adapted to stream kernel formats for performing stream compaction and summed area table computation, respectively, and the *CUDPP* library of parallel primitives [7,8] incorporates the Hillis-Steele algorithm as part of a hybrid strategy.

```
1.  for d := 1 to log2n do
2.      for k from 2d to n - 1 in parallel do
3.          m[k] := m[k - 2^(d-1)] + m[k];
4.      od
5.  od
```

Listing 1. The Hillis-Steele PRAM algorithm for implementing the Kogge-Stone parallel scan strategy.

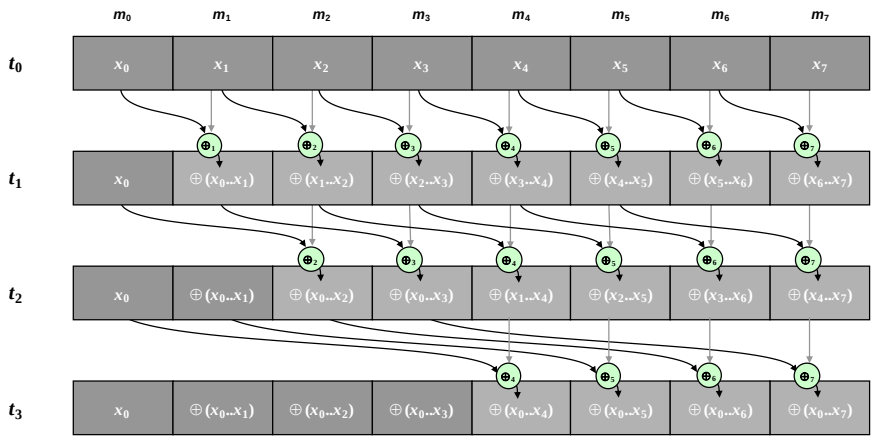


Figure 4. The operation of the Hillis-Steele algorithm on an eight-element input list in which *n* processors are each mapped to an input element. The algorithm runs for three timesteps. Each processor is mapped to a specific data element, which is depicted in light gray when updated during a given timestep.

The Hillis-Steele algorithm, shown in Listing 1, logically associates one processing element $\oplus_i$ per input element x_i. As illustrated in Figure 4, the flow of computation through the scan circuit is orchestrated via synchronous timesteps: timestep t_d encapsulates the operations performed by level d of the scan circuit. For each timestep t_d (where $1 \leq d \leq \log_2 n$), each processor executes the binary associative operator $\oplus$ on its own value and the value offset by $2^{(d-1)}$ elements to the left, storing the result back into the location of its own value.

3.1.3 Data-dependence issues

Unfortunately the Hillis-Steele algorithm can have several issues when implemented for modern stream architectures. Because these architectures are neither completely SIMD nor do they order memory accesses by program counter, the algorithm designer must explicitly address any data-dependence issues.

The algorithm as presented in Listing 1 is rife with write-after-read anti-dependencies. Memory must explicitly be made coherent between timesteps (i.e., after iterations of the outer loop) by inserting synchronization operations. The anti-dependencies in line 4 are of even bigger concern: within a given timestep, one processor must read a particular memory cell before a different processor writes to that same location. "Double-buffering" schemes are typically used for mitigating intra-timestep anti-dependencies, e.g., doubling the amount of shared storage and introducing copy operations, splitting timesteps into sequential read and write sub-phases in which values are synchronously staged into and out of private thread-local registers, etc. The result is that the additional double-buffering instructions tend to dominate the scan operators.

3.1.4 Complexity mismatches

Another issue arises from the machine model's inability to maintain the algorithm's exact work complexity. The mismatch in this case stems from the Hillis-Steele algorithm's strategy of decreasing the number of parallel tasks needed by each timestep.

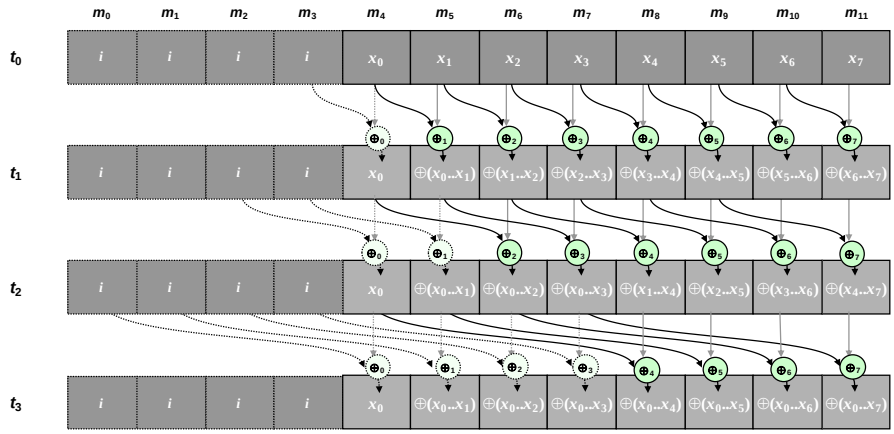
For some data-parallel architectures, a master-scheduler is aware of the specific logical tasks that will be needed to execute a given parallel statement, and it can orchestrate the execution of those tasks in a synchronous fashion. For stream architectures similar to CUDA, task scheduling is decentralized: each thread determines whether or not it will be scheduled to do work. A thread can abstain from a timestep when it determines that it should not to enter into a conditional block. It schedules itself as inactive until the warp's instruction pointer reaches the location of that thread's branch target. In this manner, threads can deactivate themselves, allowing future time-slices to be given to other active threads.

A problem with this behavior is caused by the outer `while` loop that governs the number and ordering of timesteps. This `while` loop is necessary when the problem size is not known before runtime. Each thread must schedule itself to be active between algorithm timesteps in order to evaluate the `while` conditional (as well as a conditional as to whether or not it should participate in that timestep). Therefore each thread is active during each timestep. This type-architecture mismatch results in a slightly worse work complexity of $n\log_2 n$: the $-(n-1)$ factor of the idealized strategy is lost because threads of execution cannot be completely deactivated for the remainder of the calculation.

Because a threadblock cannot solve arbitrarily-large problems, we can often eliminate the outer `while` loop in practice by unrolling it for a fixed number of timesteps. Unfortunately this does not eliminate the problem when shared memory cooperation is required between timesteps. For stream architectures similar to CUDA, coherence of the shared memory space between algorithm timesteps is only guaranteed after the execution of a barrier instruction by *all* threads having access to that shared space. This requires that each thread schedule itself to be active between algorithm timesteps in order to execute a synchronization instruction, if only to immediately decide to deactivate again. The problem remains: each thread is active during each timestep.

3.1.5 SIMD optimizations

The non-linear work complexity and anti-dependence complications make the Kogge-Stone construction unsuitable for computing arbitrarily-sized scan problems on GPGPU stream processors. For architectures with fixed-width SIMD behavior, however, the Kogge-Stone strategy is very efficient when the problem size is smaller than or equal to the SIMD width. There are no anti-dependence hazards or undesired read/write interleavings in this scenario because the processing elements are implicitly in lock-step with each other. In fact, no explicit programmatic synchronization is needed at all.




```
1.  unsigned int idx = threadIdx.x + 4;
2.  m[idx] += m[idx -  1];
3.  m[idx] += m[idx -  2];
4.  m[idx] += m[idx -  4];
```

Figure 5. The operations of an unrolled, divergence-free three-level SIMD Kogge-Stone implementation for an input size $n = 8$. By requiring 50% more storage, threads within the SIMD group can simply offset beyond the input cells into a region populated by identity values.

As a building block for their stream kernels, the *CUDPP* library uses a SIMD Kogge-Stone implementation called "warp scan" to perform 32-element scans (the width of the CUDA SIMD warp). In order to eliminate intra-warp divergence, $n/2$

cells of extra storage preceding the input list are populated with the identity element[3]. This eliminates the need for threads to conditionally determine whether or not work should be performed during a given timestep: the offset logic will simply cause them to index into the identity elements in the event that their offset calculation is out-of-range. The identity element is harmlessly combined into their output value if they index past x_0. Also, there is no extra cost of performing these additional $\oplus$ operations within the warp: the SIMD hardware resources were going to be scheduled for the warp regardless. Figure 5 illustrates the kernel code and operation of the SIMD Kogge-Stone scan for a warp size of eight.

Because of its speed and efficiency, the SIMD Kogge-Stone implementation is a very attractive building block for use within hybrid scan strategies, specifically those that can be recursively constructed. As we discuss further in Section 4, we incorporate the SIMD Kogge-Stone algorithm into our scan and reduce kernels for use when the degree of concurrency within a threadblock drops below the warp size[4].

3.2 Sklansky

3.2.1 Circuit strategy

The Sklansky construction [20] is another well-known, minimum-depth parallel prefix network. It employs a recursive, divide-and-conquer approach that yields depth $\log_2 n$ and size $(n/2)\log_2 n$. Unlike the Kogge-Stone construction, it has a variable amount of fan-out: nodes within the spine have fan-out that is exponential to their depth. Although the strategy is work-inefficient, this improved sharing contributes to a reduced work complexity over Kogge-Stone.

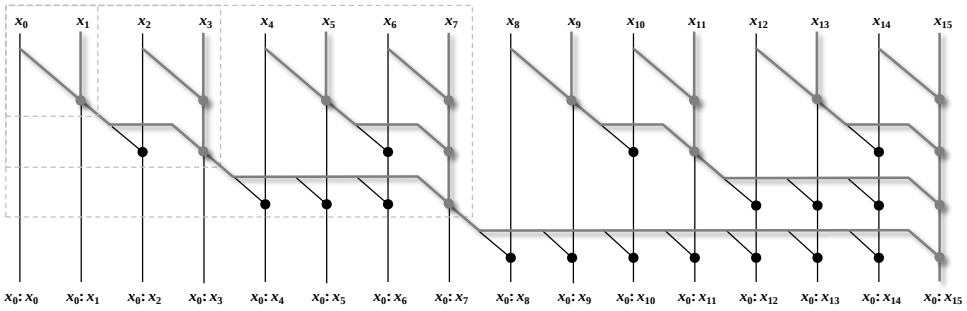


Figure 6. The Sklansky parallel inclusive scan strategy for n=16 elements. The result is computed in four levels using 32 operators. The *spine* is highlighted in gray: it is the largest sub-tree in the network and is used to compute the last output value, $x_0{:}x_{15}$.

Figure 6 depicts the recursive nature of how an n-input Sklansky network can be constructed from two smaller $(n/2)$-input Sklansky networks. The two networks are run in parallel, and the result from the spine of the first network is subsequently reduced into the $n/2$ outputs of the second network in an additional stage. The base-case for this divide-and-conquer strategy is the two-input network consisting of a single binary reduction operation.

A key property of the Sklansky strategy is that each level incorporates exactly $n/2$ operators. This is makes it convenient for mapping onto programmable hardware with a fixed number of processing elements. Compared to the Kogge-Stone construction, it is capable of scanning twice as many inputs given the same number of active processors, making it attractive for fixed-size SIMD application.

[3] Actually, the CUDPP "warp scan" unnecessarily uses $2n$ storage instead of $3n/2$ storage.

[4] Because a 32-element SIMD Kogge-Stone implementation has a 103 deficiency, it is unsuitable for programmable architectures when degree of available concurrency is greater than 32. The *CUDPP* kernels use it in this manner, however; the result is that a substantial number of dynamic instructions are spent reducing identity elements.

3.2.2 Our PRAM algorithm and variants

Recursive algorithms for generating Sklansky circuits abound. For example, programmers can describe Sklansky circuit families using languages like Lava[9], Wired[10], Obsidian[11], etc., and the runtime will compile those descriptions into executable code upon program invocation. While the recursive circuit generators are elegant and concise, they provide little insight into how the compiler will schedule computation onto processing elements and map live variables into memory spaces. Directly implementing Sklansky computation within a recursively-defined stream kernel would prove problematic because the stream machine model does not support a program stack.

PRAM Sklansky Pseudo-code

```
1.  for d := 1 to log2n do
2.      for k from 0 to n/2 in parallel do
3.          block := 2 * (k - (k mod 2^d));
4.          me := block + (k mod 2^d) + 2^d;
5.          spine := block + 2^d - 1;
6.          m[me] := m[me] + m[spine];
7.      od
8.  od
```

Reverse PRAM Sklansky Pseudo-code

```
1.  for d := 1 to log2n do
9.      for k from 0 to n/2 in parallel do
2.          block := 2 * (k - (k mod 2^d));
3.          me := block + (k mod 2^d);
4.          spine := block + 2^d;
5.          m[me] := m[me] + m[spine];
6.      od
7.  od
```

Listing 2. Forward and reverse (inclusive) PRAM algorithms for the Sklansky parallel scan strategy.

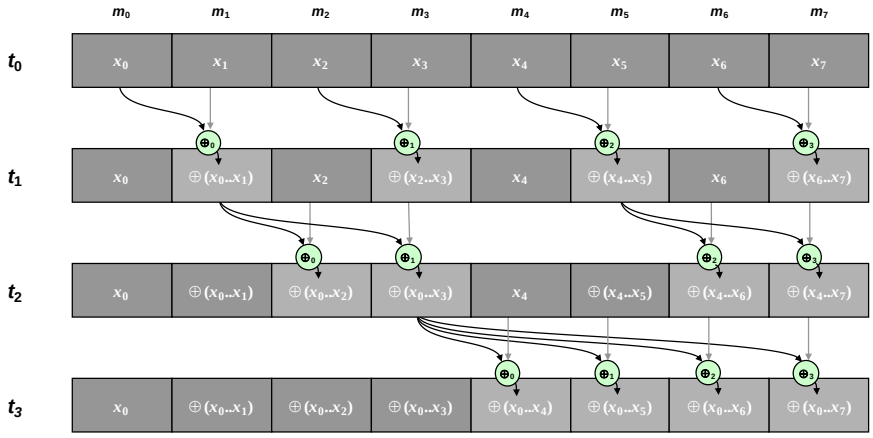


Figure 7. The operation of our forward-scan algorithm for performing an eight-element Sklansky scan. *n*/2 processors are re-mapped onto different elements for each timestep. Memory cells are shaded in light gray when updated during a given timestep. Arrows depicting concurrent register loads are color-coordinated to illustrate potential memory bank conflicts.

Interestingly, we could not find any published iterative algorithms for mapping the Sklansky construction onto a random-access machine, so we present ours here in Listing 2. The algorithm leverages *n*/2 threads of execution and incorporates a memory *m* containing *n* linear memory cells that have been populated with input values. As with the Hillis-Steele algorithm, the flow of computation is orchestrated via synchronous timesteps: timestep t_d encapsulates the operations performed by level *d* of the scan circuit.

Because we allocate half as many processors as input values, each processor must determine which element it needs to update during a given timestep. This decision is aided by the observation that, for a given timestep t_d, memory can be segmented into blocks of 2^{d-1} consecutive memory cells in which the cells of alternating blocks are updated. For example, Figure 7 shows that blocks [m_2..m_3] and [m_6..m_7] are updated during timestep t_2 while blocks [m_0..m_1] and [m_4..m_5] are not. Additionally, the cells within each "active" block all consume the same spine value. The processors updating [m_2..m_3] pull

their spine values from m_1, while the processors updating $[m_6..m_7]$ pull their spine values from m_5. By determining the location of the preceding block, the processors can readily calculate the locations of their *me* and *spine* values.

3.2.3 Shared memory bank conflicts

Many physical memories aggregate individual cells into larger units of sequentially-accessible storage. Concurrent accesses to a shared memory can be made in parallel as long as they are made to words residing in distinct memory banks. On architectures where the number of memory banks and the number of physical processors are not relatively prime (e.g., the NVIDIA CUDA architecture), our algorithm may incur memory bank conflicts when threads in the same SIMD group attempt to access their m_{me} and m_{spine} values in parallel.

For CUDA platforms, this occurs when multiple threads within the same half-warp attempt to access values residing in the same shared memory bank. The architecture currently fields sixteen threads in a half-warp and implements shared memory spaces with sixteen banks.

As an example, our implementation is guaranteed to produce two-way bank conflicts within CUDA, resulting in the cost of two instructions instead of one for each memory access. To visualize this, consider the operation shown in Figure 7 with a half-warp size $H = 4$. The hypothetical architecture would therefore have four shared memory banks, causing m_0 and m_4 to reside in the same memory bank, m_1 and m_5 to reside in the same memory bank, etc. In timestep t_1 threads $\oplus_0$ and $\oplus_2$ simultaneously access memory locations m_1 and m_5 for their *me* values, resulting in a two-way bank conflict. Note that there are no bank conflicts in timestep t_3, however.

More formally, there are two invariant properties of our algorithm regarding CUDA bank conflicts for half-warps of size H. The first is concerned with timesteps after $t_{\log_2 H}$: no bank conflicts are possible because all threads within the half-warp compute consecutive *me* offsets and the same *spine* offset. The second invariant is for timesteps up to and including $t_{\log_2 H}$: all locations concurrently accessed by a half-warp fall within a range δ of each other, where $H < \delta < 2H$, thus resulting in two-way bank conflicts.

3.2.4 Address computation overhead & the thread-specialization table

Our algorithm's biggest shortcoming, however, appears to be the comparatively high cost of address computation. Unfortunately the compiler cannot specialize code for a given stream thread: treating the thread identifier as constant during loop-unrolling and constant-propagation would eliminate the majority of the offset computation instructions, yet would result in different programs for each thread. Comparing the Hillis-Steele algorithm with ours in Listing 2, our algorithm must perform an extra six arithmetic operations[5] in order to derive the two operand offsets.

While this is certainly a severe drawback in the general case, it may be a non-issue for SIMD-sized problems in which work-inefficient scan strategies are desirable. Because the problem sizes are known, the relevant offsets (or absolute addresses, even) can be pre-computed and stored in fast constant memory. An address computation now reduces to loading an offset from a table in constant memory and then performing register-indirect addressing using that value. In a sense, this technique is effecting thread specialization by encoding static program logic into two components: (1) a constant instruction text shared by all threads and (2) a constant table of thread-specific instruction modifiers (which we've termed a *thread-specialization*

[5] Our SIMD Sklansky CUDA kernel code uses bitwise operators to implement our algorithm from Listing 2. The *me* and *block* offset calculations only require six operations because *rightmask* and *leftmask* are propagated as constants during loop unrolling:

```
1.  #pragma unroll
2.  for (i := 1; i < 64; i := i * 2) {
3.      rightmask := i - 1;
4.      leftmask := ¬rightmask;
5.      block := (k & leftmask) << 1;
6.      me := block | rightmask;
7.      spine := block | (k & rightmask) | i;
8.      m[me] := m[me] + m[spine];
9.  }
```

table)[6]. Due to the amount of storage required, this approach works best when the thread diversity and instruction path lengths are kept small, e.g., 32 threads executing a handful of memory operations.

	$thread_0$	$thread_1$	$thread_2$	$thread_3$
0	1	3	4	6
1	0	2	5	7
2	2	3	4	5
3	1	1	6	6
4	4	5	6	7
5	3	3	3	3

```
1.  m[modifier[0][threadIdx.x]] += m[modifier[1][threadIdx.x]];
2.  m[modifier[2][threadIdx.x]] += m[modifier[3][threadIdx.x]];
3.  m[modifier[4][threadIdx.x]] += m[modifier[5][threadIdx.x]];
```

Listing 3. The code text and thread-modifier table for an four-thread, eight-element SIMD Sklansky scan adapted to avoid memory bank conflicts for a half-warp size $H = 4$, unrolled into three levels. The operation of this scan is illustrated in Figure 8.

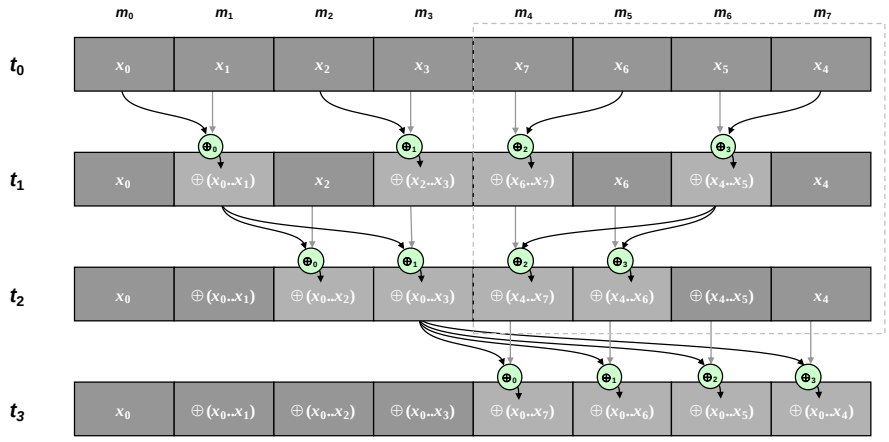


Figure 8. The operation of the table-driven, eight-element SIMD Sklansky scan presented in Listing 3. The input is permuted such that every other block of four input elements are in reverse order. The dashed enclosure highlights memory accesses using offsets derived from the reverse algorithm. Memory cells are shaded in light gray when updated during a given timestep. Arrows depicting concurrent register loads are color-coordinated to illustrate potential memory bank conflicts.

An added benefit of using the thread-specialization table is that we can eliminate the two-way bank conflicts described above. The trick is to use a permuted ordering of the input list x within the memory space m, which can be done when it is loaded. More specifically, we can arrange the elements of x in memory such that every other H input values are listed in reverse order. We can use the address computation logic from the reverse scan algorithm in Listing 3 to pre-compute the load/store offsets for these blocks for the first $\log_2 H$ timesteps. As shown in Figure 8 for half-warp size $H = 4$, this has the result of causing the second half of the half-warp (the second quarter-warp) to use the memory banks not used by the first quarter-warp, yielding zero bank conflicts. In effect, we are orchestrating divergent intra-warp behavior without paying the penalties associated with divergent instruction flow. Computation resumes in the normal, "forward" fashion after timestep $t_{\log_2 H}$.

[6] This specialization technique can be generalized to thread scheduling issues as well. A "synchronization stack" of sorts would remedy the type-architecture mismatches resulting from the thread (de)activation issues described in 3.1.4.

Even though the addressing logic and the operational illustrations for the permuted-input, table-driven implementation look very different than that of the original implementation, it is important to note that the two implementations realize the exact same circuit family depicted in Figure 6. (This can be visually checked by tracing through the computation histories for each of the *n* reduction trees.)

3.2.5 SIMD evaluation

We used CUDA to evaluate our thread-specialization table technique for a 64-element SIMD Sklansky scan, unrolled into six levels. Although the CUDA hardware architecture has a constant cache, we found it unsuitable for use as a thread-specialization table because warp divergence occurs unless all threads access the same value. The fact that each thread in the half-warp would need to access its own memory cell leads to a perfectly degenerate scenario in which all sixteen accesses would be serialized. Instead, we were able to implement the thread-specialization table mechanism using a different architectural "feature". On our CUDA devices, the shared memory local to each streaming multiprocessor is not cleared between kernel executions. This allowed us to run a single "pre-processing kernel" prior to our Sklansky scan kernel in order to stage thread-specific offsets into shared memory. With some trial-and-error, we were able to determine the proper shared-memory configurations that would allow us to place tables into memory where each active threadblock from the scan kernel would find them.

Unfortunately the overall efficiency was not good enough to warrant its use over alternative CUDA SIMD scans. Even without address computations and bank conflicts, the dynamic instruction count was greater than a 64-element SIMD implementation comprised of a five-level Kogge-Stone SIMD scan bookended by a pair of upsweep and downsweep levels. This is due to the fact that NVIDIA GPUs have special-purpose offset registers for relative addressing. A table-driven addressing scheme for CUDA must first move the address out of the thread-specialization table into a general-purpose register, move it into an offset register, and then perform the load/store.

The underwhelming efficiency stems from the fact that a scan operator in our table-driven SIMD Sklansky implementation requires seven instructions versus three for SIMD Kogge-Stone. We need five instructions[7] for loading two operands from shared memory into registers, one for the scan operator itself, and one for the store (which uses the same offset as the second operand). Alternatively, threads in the Kogge-Stone algorithm are logically associated with a particular memory cell and therefore are not obligated to reload that second operator for each timestep. In addition, the simple strides of the SIMD Kogge-Stone implementation are translated into constant-offset loads. The result is that the average SIMD Kogge-Stone operator needs only three instructions: one constant-offset load, one instruction for the scan operator itself, and one to store the accumulated value for use by other threads.

In other architectures without such peculiar memory addressing operations, however, stream kernels incorporating a SIMD Sklansky strategy along with a thread-specialization table should yield fewer steps and a lower dynamic instruction count than a SIMD Kogge-Stone variant.

3.3 Brent-Kung

3.3.1 Circuit strategy

The Brent-Kung construction [18] is a popular "tree-based" strategy having logarithmic depth and linear size. An *n*-input circuit exhibits depth $2\log_2 n - 1$ and size $2n - \log_2 n - 2$. The proportionally-linear size makes the Brent-Kung construction attractive for stored-program scenarios in which the number of input elements exceeds the number of physical processors, e.g., GPGPUs.

[7] We can use a single 32-bit load instruction to simultaneously read two 16-bit offsets from the thread-modifier into a local general-purpose register.

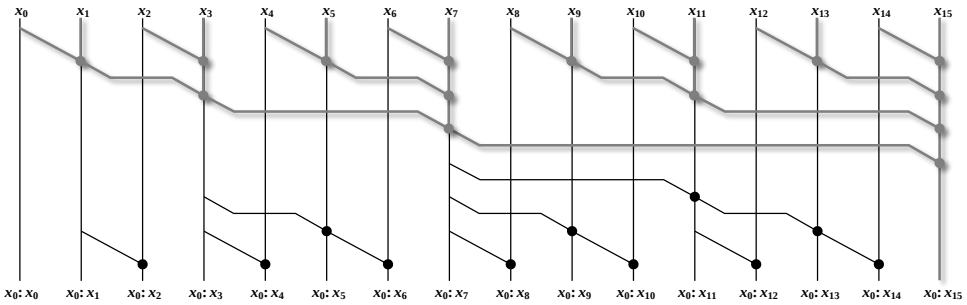


Figure 9. The Brent-Kung parallel inclusive scan strategy for n=16 elements. The result is computed in seven levels using 26 operators. The *spine* is highlighted in gray: it is the largest sub-tree in the network and is used to compute the last output value, x_0:x_{15}. This strategy is often alternatively shown in a manner in which the lower-left "propagation" operations have been slid upwards in time, greedily executing as allowed by their data dependencies.

The Brent-Kung construction can be thought of as existing in a balanced binary-tree communication network. The leaves of network are supplied with the input elements and the interior nodes initially serve to calculate the spine values as partial reductions are accumulated upwards towards root. In the second phase, the accumulated partial reductions are then passed back downwards from the root in a manner that also conveys the aggregates received from the left children down to the right children. The first level of a Brent-Kung circuit requires n/2 operators, the maximum of any level. This allows a machine with p processors to perform a 2p-input scan without the serialization of concurrent tasks.

3.3.2 Blelloch PRAM algorithm and variants

The Brent-Kung strategy is commonly implemented using a PRAM algorithm formalized by Blelloch[2]. More recently, Sengupta et al. showed how concurrent tasks in the Blelloch algorithm can be mapped onto threads of execution for stream architectures [12, 13].

```
1.  for d := 0 to log2n - 2 do
2.      for k from 0 to n - 1 by 2^(d+1) in parallel do
3.          m[k + 2^(d+1) - 1] := m[k + 2^(d+1) - 1] + m[k + 2^d - 1];        // parent node
4.      od
5.  od
6.  m[n - 1] := m[n/2 - 1];
7.  m[n/2 - 1] := id;
8.  for d := log2n - 2 downto 0 do
9.      for k from 0 to n - 1 by 2^(d+1) in parallel do
10.         temp := m[k + 2^d - 1];
11.         m[k + 2^d - 1] := m[k + 2^(d+1) - 1];                            // left child
12.         m[k + 2^(d+1) - 1] := temp + m[k + 2^(d+1) - 1];;                // right child
13.     od
14. od
```

Listing 4. The exclusive-scan Blelloch PRAM algorithm implementing the Brent-Kung strategy. The stride between memory references made during a given timestep is dependent upon the spine level being updated.

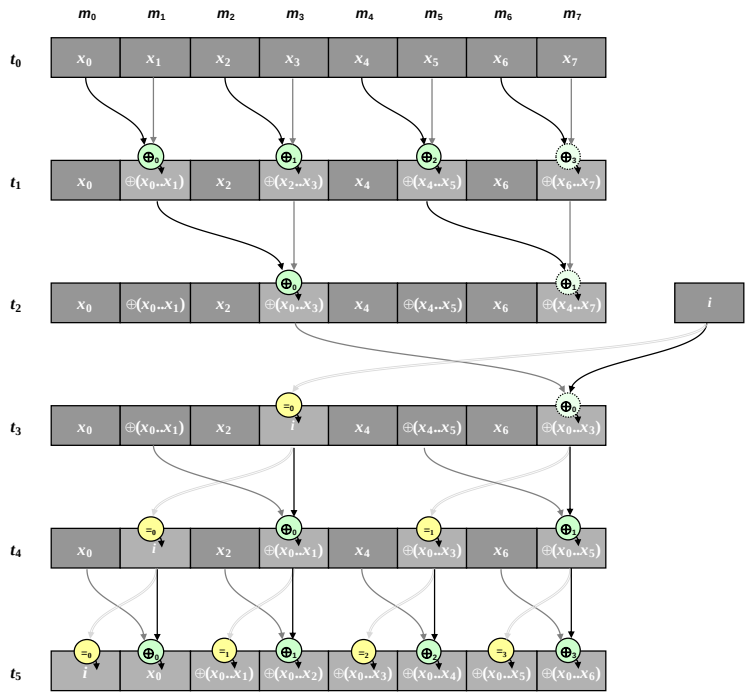


Figure 10. The operation of the exclusive Blelloch scan algorithm on an eight-element input list in which $n/2$ processors are re-mapped onto different elements for each timestep. The algorithm runs for five timesteps. Memory cells are shaded in light gray when updated during a given timestep. Arrows depicting concurrent register loads are color-coordinated to illustrate potential memory bank conflicts.

The Blelloch algorithm presented in Listing 4 above is described as having two separate phases: *upsweep* and *downsweep*. The upsweep phase computes the spine. The original input list begins as the leaf-set for this reduction tree. As the reduction proceeds up the spine, the right siblings for the level's pairings are overwritten with the reductions of those siblings. This process continues upwards until just before the point at which the root node of the reduction tree would be computed: the complete reduction is not needed for exclusive scan. In the downsweep phase, each node in the reduction tree propagates the value of its parent downwards to its two children. The value of the left child is incorporated into the value given to the right child.

3.3.3 Complexity mismatches

Like the Hillis-Steele algorithm, implementations of the Blelloch algorithm for CUDA-like stream architectures suffer from type-architecture mismatches, i.e., complexity behavior that is inconsistent with what one would expect from the PRAM algorithm. As described in Section 3.1.3, the details of the stream machine model can preclude implementations from achieving their expected amortized complexities, particularly for algorithms that require shared memory cooperation between timesteps. As a result, each of the Blelloch algorithm's $n/2$ threads must actively execute a synchronization instruction for all $2\log_2 n$ timesteps. This has the unfortunate effect of ruining the Blelloch algorithm's amortized work complexity, changing it from $\Theta(n)$ to $\Theta(n\log_2 n)$. The mismatch is particularly troublesome for Blelloch scan because it signifies an asymptotic change in complexity[8] (as opposed to a constant-factor change for the Hillis-Steele).

3.3.4 Shared memory bank conflicts

The Blelloch algorithm and the stream machine model also suffer from the classic type-architecture mismatch in which memory access-times are decidedly non-uniform. The culprit in this case is a susceptibility to memory bank conflicts that can lead to perfectly degenerate scenarios in which all accesses to shared memory are serialized. Not only does step-

[8] Tree-based parallel-reduction is another example that can experience an asymptotic change in work complexity.

complexity suffer when this occurs, but work complexity deteriorates as well: work and power are still being expended by SIMD lanes that are ultimately masked off while accesses are executed serially.

The problem occurs on NVIDIA architectures because the stride between parallel threads doubles after every timestep. When the stride is two, the sixteen threads in a half-warp access locations in eight different banks. This incurs two-way conflicts that cause the SIMD access to take twice as long. For the next timestep, the stride is four and the sixteen threads access values in four different banks, incurring four-way conflicts. Each subsequent step with stride s suffers from s-way bank conflicts, takes s-times longer, and ultimately maxes out at $s = 16$ when all accesses are performed serially.

To visualize these bank conflicts, consider the operation of the Blelloch algorithm in Figure 10 for a hypothetical architecture having four shared memory banks and a half-warp size $H = 4$. The arrows representing loads and stores are color-coded: the accesses indicated by like-colored arrows for a given timestep are issued in parallel. Note that every SIMD access performed by the half-warp in timestep t_1 incurs two-way conflicts, and that every access in t_2 incurs perfectly degenerate four-way conflicts.

In their work regarding the adaptation of the Blelloch algorithm for GPGPUs, Harris et al. have addressed the problem of bank conflicts by inserting padding cells into the shared memory array [13]. The amount of extra padding required to eliminate all bank conflicts for the current CUDA architecture follows a geometric progression with factor 1/16: one cell of padding every 16 elements + one cell every 16^2 elements + one cell every 16^3 elements + ... and so on. In the limit, this approach requires $O(n/15)$ padding cells.

When using shared memory, the amount of padding that must be accounted for when indexing a particular word is variable and the costs of performing the additional instructions needed for offset calculation can be substantial. In fact, our evaluations of this technique revealed that more cycles were being spent performing address calculations by the conflict-free and conflict-avoidant variants than were absorbed by bank conflicts during conflict-ridden operation. Analyses of hardware counters revealed that eliminating all bank conflicts incurred a 52% increase in dynamic instruction count, and eliminating a majority (97% of all bank conflicts) incurred a 42% increase. Both approaches resulted in slowdown when run on our GTX-285.

In short, the Blelloch algorithm suffers from a catch-22: in unmodified form it suffers from bank-conflict slowdown, and modified forms incur too much overhead. As such, it has fallen into disfavor by the GPGPU community, with more recent research focusing on alternative strategies [7,17].

3.3.5 Our two-way conflict PRAM algorithm

Although the Blelloch algorithm has turned out to be largely impractical, the Brent-Kung strategy should not be abandoned. In this subsection and the next, we present two new PRAM algorithms in the Brent-Kung style that leverage alternative tree encodings in order to avoid these shortcomings. It is important to note that the operation of the Brent-Kung strategy is fundamentally structured around a balanced binary tree: the spine. As such, any technique for encoding binary trees into a flat array of memory cells can be used as the basis for a unique PRAM scan algorithm that implements this circuit strategy.

An important characteristic of Brent-Kung is that the partial reductions serving as the "right child" nodes of the binary spine tree are no longer needed as computation sweeps upwards. This allows their memory locations to be overwritten with new values, as is done by the Blelloch algorithm. Our new algorithm is designed around the observation that there is no necessity for overwriting these values: we use $n - 2$ extra memory cells for the storage of intermediate results. By doing so, we can maintain a constant memory access stride = 2 between threads, and do so for all timesteps. This results in exactly two-way memory bank conflicts for CUDA-like architectures, a minimal-degree of conflict that is achieved without any complicated addressing techniques.

```
1.  a := m;
2.  for d := log2n downto 2 do
3.      for k from 0 to 2^(d-1) − 1 in parallel do
4.          a[2^d + k] := a[2k] + a[2k + 1];          // parent node
5.      od
6.      a := &a[2^d];
7.  od
8.  a[1] := a[0];
9.  a[0] := id;
10. for d := 2 to log2n do
11.     a := &a[-2^d];
12.     for k from 0 to 2^(d-1) − 1 in parallel do
13.         a[2k + 1] := a[2k] + a[2^d + k];          // right child
14.         a[2k] := a[2^d + k];                      // left child
15.     od
16.     a := b;
17. od
```

Listing 5. Our PRAM algorithm for implementing the Brent-Kung parallel scan strategy. Memory references during each timestep are always made using a stride of two.

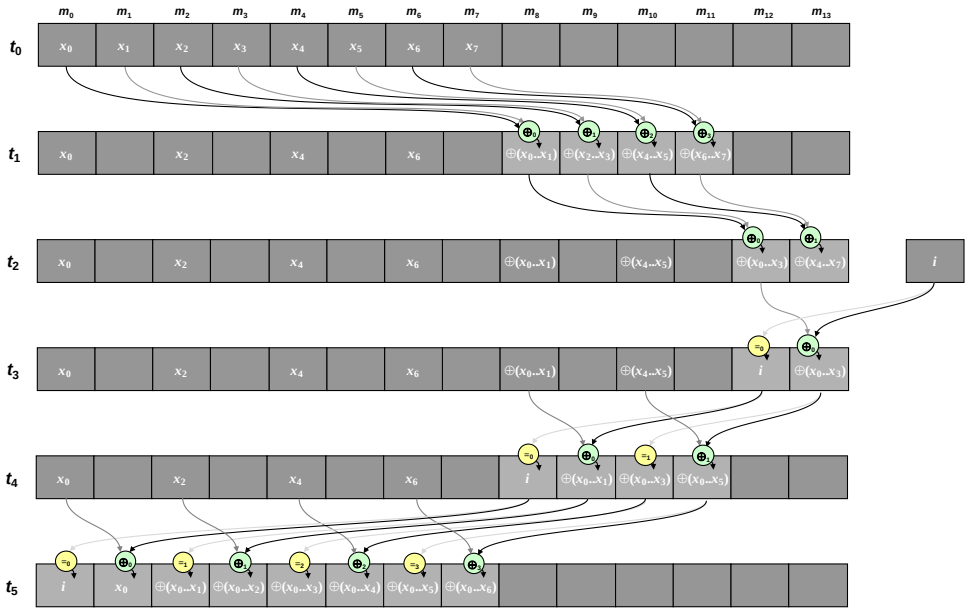


Figure 11. The operation of our algorithm on an eight-element input list in which $n/2$ processors are re-mapped onto different elements for each timestep. The algorithm runs for five timesteps and requires n-2 additional memory cells for the storage of intermediate computations. Memory cells are shaded in light gray when updated during a given timestep. The algorithm exhibits two-way bank conflicts, regardless of half-warp size and the number of memory banks.

The exclusive scan version of our algorithm is presented in Listing 5 above. As a Brent-Kung isomorph, it can be described as having separate upsweep and downsweep phases. Each timestep during the upsweep entails the calculation and storage of spine level `d-1` within a contiguous segment of memory cells beginning at offset `a[2^d]`. The algorithm requires $n - 2$ additional memory cells to accommodate the $n - 2$ interior nodes of the spine. There are `k` threads active during each timestep, and the memory access stride between them can be kept constant because the tree's values are encoded consecutively in level-order.

Each task in the downsweep phase propagates its value downwards to its children, the value of the left child being incorporated into the value of the right. We observe that a temporary value is not needed by our implementation for downsweep propagation because the location of the incoming parent value is not being reused for the left child.

Our algorithm offers a good compromise between bank conflict slowdown and addressing complexity, something that is not possible with the Blelloch algorithm. As with conflict-avoidant Blelloch variants, we incur an extra $O(n)$ amount of space. The cost of a two-way bank conflict is minimal: the break-even overhead for a conflict-avoidant approach would be a single instruction, an unlikely feat. When an upper-bound is placed on problem size (as is the case for intra-threadblock computation), the addressing offsets for our algorithm can be statically unrolled into constants. This is something that is not possible when addressing offsets are a function of thread rank.

3.3.6 Permuted variant

As an interesting alternative, we have developed a Brent-Kung algorithm is free from bank conflicts and requires zero additional storage space. The critical design characteristic of this PRAM algorithm is that it does not encode tree values in a strictly consecutive level-order fashion. This detail is most conspicuous in that the input and output element lists are provided using a permuted ordering.

```
1.  a := m;
2.  for d := log2n - 1 downto 1 do
3.      for k from 0 to 2^d - 1 in parallel do
4.          a[2^d + k] := a[2^d + k] + a[k];                    // parent node
5.      od
6.      a := &a[2^d]
7.  od
8.  a[1] := a[0];
9.  a[0] := id;
10. for d := 1 to log2n - 1 do
11.     a := &a[-2^d];
12.     for k from 0 to 2^d - 1 in parallel do
13.         temp := a[k];
14.         a[k] := a[2^d + k];                                 // left child
15.         a[2^d + k] := a[2^d + k] + a[k];                    // right child
16.     od
17.     a := b;
18. od
```

Listing 6. Our permuted-ordering PRAM algorithm for implementing the Brent-Kung parallel scan strategy. Memory references during each timestep are always made using a stride of one.

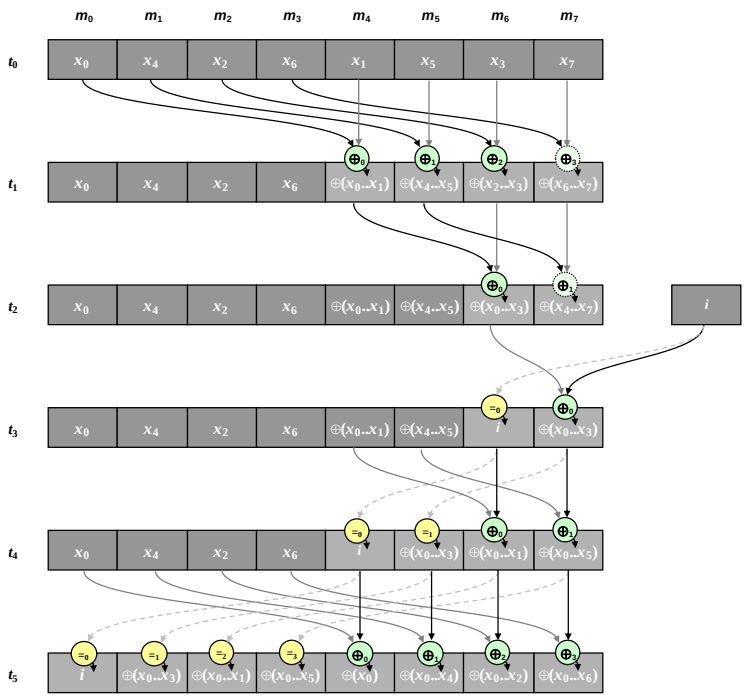


Figure 12. The operation of our permuted-scan algorithm on an eight-element input list in which *n*/2 processors are re-mapped onto different elements for each timestep. The algorithm runs for five timesteps. Memory cells are shaded in light gray when updated during a given timestep. The algorithm is conflict-free and padding-free.

The exclusive-scan version of this permuted-scan algorithm is presented in Listing 6 above. This algorithm is similar to the two-way conflict algorithm from Section 3.3.5 in that each timestep during the upsweep entails the calculation and storage of spine level `d-1` within a contiguous segment of memory cells beginning at offset `a[2^d]`. The encoding of the spine is similar to the Harris et al. algorithm for tree-based binary reduction for GPGPUs [14]: the "left child" and "right child" values reduced by a given thread are offset by the number of active threads at that timestep. The memory access patterns used by our spine reduction are a mirror image of theirs: levels of partial reductions are written at the end of the shared memory array instead of the beginning. The critical operation of this upsweep algorithm depends upon a specific permutation of the input elements such that only the locations of the "right children" in the isomorphic Brent-Kung spine are overwritten.

The downsweep phase propagates partial reductions back down the spine, following the same encoding of left/right child locations as used by the upsweep reduction. As with the Blelloch algorithm, a temporary variable is needed to remember the previous value of the left child before overwriting it with the value propagated down from the parent.

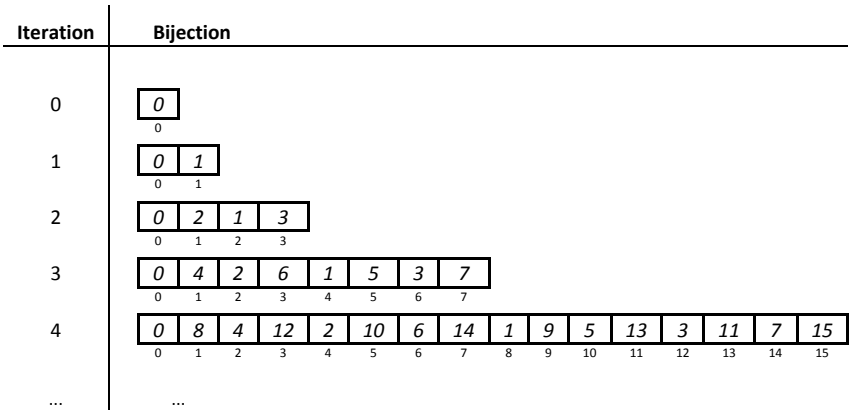


Figure 13. The recursive process for generating an *n*-element bijection of (element rank, shared memory index) relations for use in permute-scan. Each iteration begins by doubling the codomain targets the previous iteration. The set of relations is then supplemented by a copy of itself in which the codomain targets have been incremented by one. The base case is the relation (0,0).

The primary drawback of this approach is that the algorithm needs to be wrapped by a pair of data permutation steps before it can be used with input lists whose elements are stored in consecutive order. Unfortunately we are unaware of an $O(1)$ function for performing such a mapping of element ranks to memory indices. To our knowledge, the computation of this bijection for arbitrary input sizes seems to be $O(\log_2 n)$, as illustrated in Figure 13.

For fixed-size scans, the bijection mapping can be pre-computed, e.g., using the thread-specialization table technique described in Section 3.2.4. We evaluated the permute-scan algorithm as such for problem instances small enough to be cooperatively processed within a single CUDA threadblock. The approach was very efficient in terms of dynamic instruction count and conflict-free use of shared memory. Unfortunately the input and output mappings caused the majority of memory transactions to device memory to become uncoalesced for our particular GPU architecture, resulting in significant slowdown and bandwidth underutilization.

The permuted-scan algorithm still has value on these architectures, however, for scan problems that have relaxed ordering constraints. Applications such as array compaction for resource allocation do not require the input/output mapping steps described above, allowing this algorithm to be used very efficiently.

3.4 Meta-scan

By their nature, GPGPU stream architectures expose a hierarchical memory model in which not all memories are addressable by all threads of computation. The PRAM machine model, on the other hand, implies a universally reachable memory space that is kept coherent between program steps. Attempting to directly adapt a data-parallel PRAM algorithm for these stream architectures would be impractical: it would be terribly inefficient to store all intermediate results back to global memory, invoking a new kernel for each program step.

Instead, the programmer is forced to consider hierarchical decompositions that involve limited cooperation. In the bulk-synchronous programming model, this implies doing as much independent work as possible at the thread and threadblock levels before synchronizing intermediate results. All cooperation implies additional overhead, yet cooperation between threadblocks (i.e., between cores) is the most expensive: reads and writes must be made to slower, off-chip memory spaces and global synchronization inserts bubbles of inactivity into the memory pipelines.

Prior efforts in the area of GPGPU scan have leveraged one of two different "meta strategies" for problems too large to be cooperatively solved by a single threadblock: we term them *scan-then-propagate* and *reduce-then-scan*. We review both approaches in this subsection and then present our own variation, *two-level streaming reduce-then-scan*.

3.4.1 Scan-then-propagate

As described in the previous subsection, the Brent-Kung scan construction entails a base-two tree of parallel computation: an operator in the upsweep phase effects a two-element scan, and an operator in the down-sweep phase distributes one live intermediate value into another. Ralf Hinze formalizes this approach for arbitrary bases [15]: "base circuits" can be used to provide *b*-element scans during the upsweep, followed by *b*-way distributions in the downsweep phase. We refer to this generalization as the *scan-then-propagate* meta-strategy.

In their scan implementations, Sengupta et al. use the scan-then-propagate approach for decomposing problems requiring multiple threadblocks [7,8,13]. Their implementations begin by recursing through $\log_b n$ levels of scan kernels, where *b* is the number of values that can be processed by a single threadblock. The first level contains n/b threadblocks, and each subsequent level comprises a factor of *b* fewer threadblocks than the previous. Each scan threadblock reads *b* inputs and writes *b* intermediate results back to global device memory. The downsweep phase unwinds the recursion: each propagation threadblock reloads the *b* intermediate results from the corresponding upsweep scan block, aggregates its incoming value from the preceding level into each, and writes the updated *b* values back out to global memory.

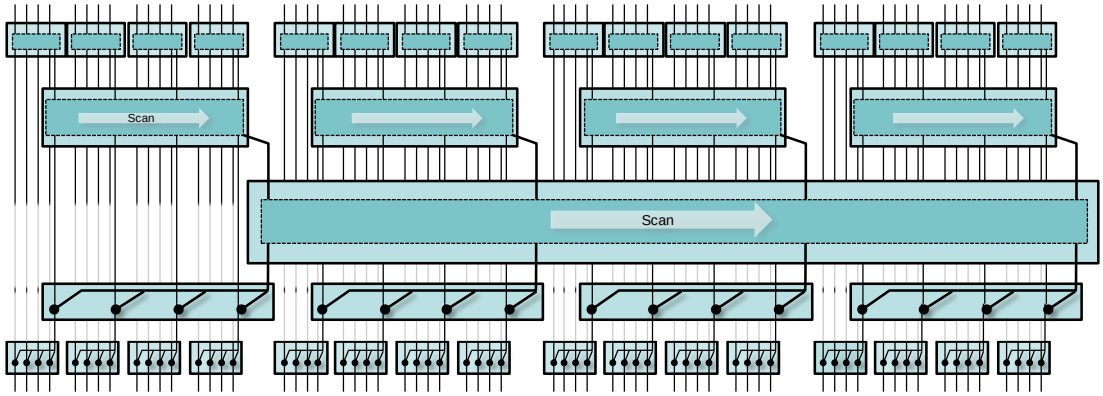


Figure 14. Operation of a *scan-then-propagate* meta-strategy in which each threadblock processes *b*=4 values. The 64-element input requires a three-level computational tree of threadblocks. Scan "upsweep" threadblocks are opaquely shown in dark green; propagation "downsweep" threadblocks are explicitly wired.

Figure 14 illustrates the operation of a scan-then-propagate implementation for a problem requiring three levels of recursion. For a complete *b*-ary tree having *n* exterior nodes, the number of interior nodes is given as $(n - 1)/(b - 1)$. Considering both upsweep and downsweep phases, an *n*-element scan problem will therefore require $O(2(n - 1)/(b - 1) - 1)$ threadblocks. Because each scan and propagate threadblock performs $O(2b)$ memory accesses, we observe that the entire computational tree requires $O(4b(n - 1)/(b - 1) - 2b)$ global memory accesses.

3.4.2 Reduce-then-scan

The *reduce-then-scan* meta-strategy is similar in that it also entails $\log_b n$ levels of stream kernels, but instead executes reduction kernels during the upsweep phase followed by scan kernels during the downsweep phase. The upsweep phase therefore computes the spine of the *b*-ary computational tree; each reduction threadblock reads *b* inputs, aggregates them, and writes a single intermediate result back to global device memory. The intermediate values computed during the reduction kernels are not saved and must be recomputed later. The downsweep phase unwinds the recursion: each scan threadblock reloads the *b* partial reductions used as inputs to the corresponding upsweep reduction block, performs a scan of them using the incoming value from the preceding level as a seed for the first element, and then writes the updated *b* values back out to global memory. The technique was first popularized for two-levels on the Cray Y-MP by Chatterjee et al. [16], and more recently generalized in this fully-recursive style for GPGPU use by Dotsenko et al. [17].

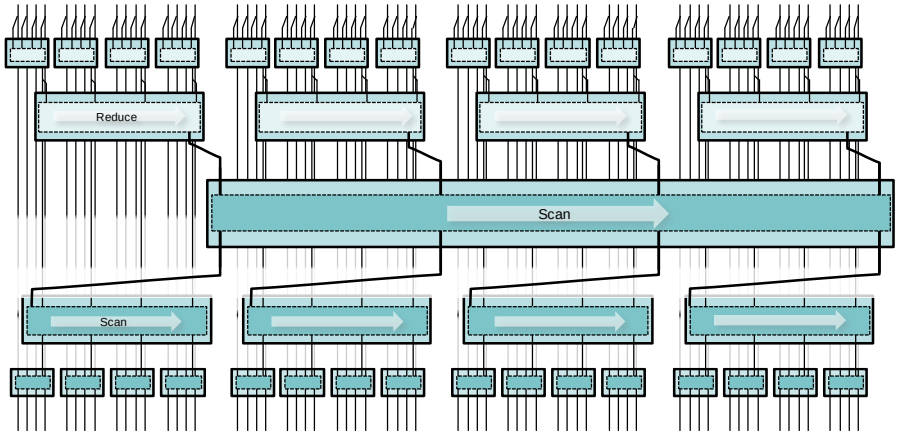


Figure 15. Operation of *reduce-then-scan* meta-strategy in which each threadblock processes b=4 values. The 64-element input requires a three-level computational tree of threadblocks. Reduction "upsweep" threadblocks are shown in light green, scan "downsweep" threadblocks are shown in dark green.

Figure 15 illustrates the operation of a reduce-then-scan implementation for a problem requiring three levels of recursion. As with the scan-then-propagate strategy, an n-element scan problem will require $O(2(n\text{-}1)/(b\text{-}1) - 1)$ threadblocks. Each reduction threadblock performs $O(b)$ memory accesses while each scan threadblock performs $O(2b)$ memory accesses. Therefore we observe that the entire computation requires only $O(3b(n - 1)/(b - 1) - b)$ global memory accesses. At the expense of performing some redundant calculations during the downsweep phase, the reduce-then-scan strategy moves 25% fewer bytes through global memory than scan-then-add.

3.4.3 Two-level streaming reduce-then-scan

Our GPGPU scan implementations restrict themselves to a two-level tree of reduce-then-scan threadblocks. In independent work on parallel-compact, Billeter et al. have proposed a similar two-level meta-strategy [21]. Instead of allocating a unique thread for every input element, we deviate from the data-parallel programming paradigm and instead simply dispatch a fixed number C of threadblocks in which threads are "re-used". We choose C large enough to saturate all SMs. A one-level upsweep reduction is performed producing a second-level (i.e., top-level) scan problem with C inputs, which is small enough to scan with a single threadblock.

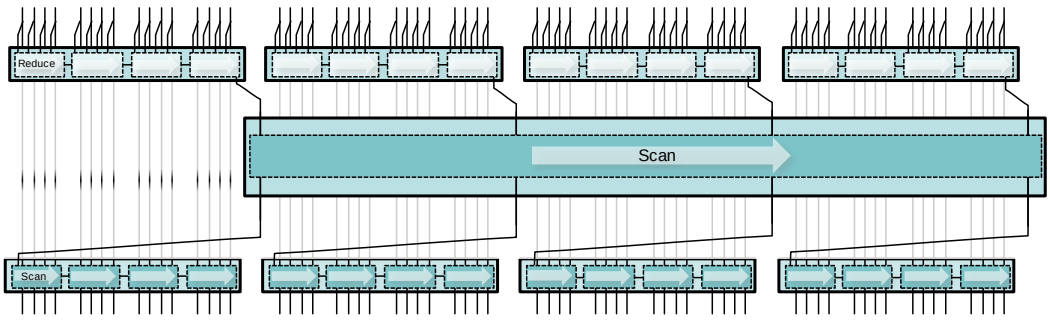


Figure 16. Operation of a two-level *reduce-then-scan* meta-strategy for a 64-element input. This illustration depicts a constant C=4 number of bottom-level threadblocks in which each threadblock serially processes cycles of b=4 values. Reduction "upsweep" threadblocks are shown in light green, scan "downsweep" threadblocks are shown in dark green.

Figure 16 above illustrates the operation of a two-level streaming reduce-then-scan implementation for the same problem depicted for the previous two meta-strategies. The reduction threadblocks re-use threads in order to serially accumulate batches of b values, similar to the reduction strategy described by Harris et al. [14].

The downsweep phase begins with a single threadblock performing the top-level scan of C partial reductions. Once completed, C threadblocks are dispatched to perform the independent bottom-level scans, seeded with the appropriate aggregate from the top-level scan. Because the local shared memory is too small for a downsweep threadblock to scan all n/C values in parallel, it instead streams smaller scans in serial fashion. Cycles of b-sized blocks are read in, scanned in parallel, and written out, all the while serially currying the aggregate into the next b-sized parallel scan. The result is that an entire n-element computation requires only $O(3n + 3C)$ global memory accesses.

Compared to the two recursively-defined meta-strategies, the advantages of our two-level streaming meta-scan are threefold. First, our strategy requires asymptotically fewer kernel launches: a constant three versus $O(\log_b n)$. Second, it also requires asymptotically fewer global memory accesses for intermediate values: a constant $3C$ versus $O(n)$. Finally, it allows us to scan an input problem in-place with only a constant amount of additional storage.

3.4.4 On the handling of arbitrarily-sized problems

For the two fully-recursive meta-strategies, special consideration must be given for problem sizes that are not powers of b. The general technique for accommodating these scenarios is for the last threadblock of each level d to process the last l_d "leftover" values at that level, where $l_d = n_d \bmod b$. These threadblocks are not as efficient because they cannot perform an unguarded block of b loads/stores: they must implement some form of bounds-checking to avoid making illegal memory references.

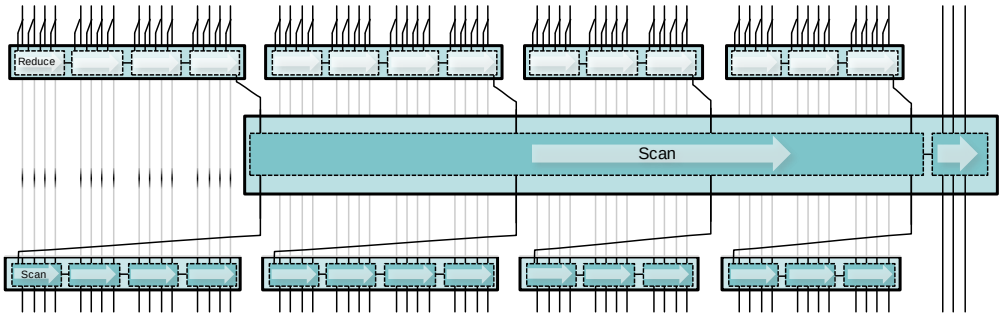


Figure 17. Operation of a two-level *reduce-then-scan* meta-strategy for a 59-element input. This illustration depicts a constant C=4 number of bottom-level threadblocks in which each threadblock serially processes cycles of b=4 values. Half of the bottom-level kernels run four cycles, the remaining run three. The final three elements are scanned by the top-level kernel. Reduction "upsweep" threadblocks are shown in light green, scan "downsweep" threadblocks are shown in dark green.

Our two-level strategy, on the other hand, affords us an alternative for handling leftover values. Instead of special-casing the last threadblock in the upsweep/downsweep levels, we simply scan these leftover values within the top-level threadblock, seeding them with the partial reduction accumulated from the bottom-level. In addition, our strategy must accommodate problem sizes where n is not a multiple of C. In general, the bottom-level must cumulatively process $\lfloor n/b \rfloor$ cycles of b elements. Each threadblock executes a minimum of $\lfloor n/(Cb) \rfloor$ cycles, with the first $\lfloor n/b \rfloor \bmod C$ threadblocks executing an additional cycle. As an example, Figure 17 above depicts the operation of both extra cycles and leftover elements for a 59-element input problem.

4 A Rigorous Approach for Scan Algorithm Design

The key to achieving maximal scan performance is constructing solutions that optimally utilize the full device memory bandwidth afforded by the underlying stream hardware. There are three primary design considerations that influence the ability for a scan implementation to do so: memory usage, bandwidth tuning, and computational overhead. Each of these can be treated as orthogonal concerns, addressed in separate design stages. This section illustrates our process by walking though the designs of our three new scan implementations: *merrill_tree*, *merrill_srts*, and *merrill_linear*.

4.1 Stage 1: Meta-strategy Selection

Our first consideration is memory usage: the number of accesses to device memory that must be made for a given problem size and the distribution of these access requests amongst concurrent threadblocks. This is an algorithm-selection problem, specifically at the meta-scan level. We can functionally abstract away the nuances of threadblock design, treating them instead as opaque building blocks that provide specific types of functionality. The problem is then to determine what types of threadblocks to use and how grids of them should be composed serially in a way that minimizes the amount of device storage needed for input, output, and intermediate results.

The design space at this level is influenced by the nature of device memory coherence: all threadblocks must terminate before device memory can be made coherent. This equates to a load balancing problem in which threadblocks must be assured roughly equivalent amounts of work. Threadblocks are not only programmatically homogenous in accordance with the machine model, but they need to be homogenously loaded as well.

Surveying the meta-scan approaches outlined in Section 3.4, we find that all three exhibit good load balancing characteristics. Our *two-level reduce-then-scan* strategy, however, is the thriftiest in terms of data that must be pushed through the device memory subsystem. As the clear choice, we use it as the basis for all three of our implementations. This strategy is parameterized by b and C: the cooperative cycle-size and the number of concurrent threadblocks to launch, respectively. The appropriate values for these parameters will be influenced by the specific hardware, and we detail their selection in the next two stages.

4.2 Stage 2: Data Movement Kernel Skeletons

In this design stage, we will determine the fastest way to move data in (and out) of a threadblock, resulting in a set of kernel skeletons. We will then use these bandwidth-tuned skeletons in the next stage to flesh out the necessary scan and reduction logic.

In the previous design stage, we determined that we would be using the *two-level reduce-then-scan* meta-strategy. The two kernels involved are *reduce* and *scan*, but we are still not concerned with their internal mechanics. Our reduce and scan requirements demonstrate two general patterns of data movement: *in-only* and *in-out*. A reduction threadblock will read in $O(n)$ inputs and write out $O(1)$ outputs. A scan threadblock will read in $O(n)$ inputs and write out $O(n)$ outputs. These patterns are common amongst a wide variety of kernel algorithms: summation, minimum-finding, voting, etc., exhibit the one-way *in-only* pattern; parallel scan, Poisson stencils, matrix multiplication, etc., exhibit the two-way *in-out* pattern. For any given stream processor, we can use an auto-tuning process to select the best *in-only* and *in-out* skeletons for that platform: empty code that simply moves data in (and out) of a threadblock. The value of this stage exists beyond our immediate problem of parallel scan: these skeletons can be reused for a wide variety of kernel algorithms.

The *in-out* skeleton is about as functionally simple as a stream kernel can get: a thread simply loads data from one memory location and stores it to another. The *in-only* skeleton is even simpler. When a person is tasked with actually implementing one, however, they quickly find that they are confronted with a bevy of design and configuration choices that are largely unrelated to the problem at hand. How many threads should be allocated for each threadblock? Should memory transactions be made in terms of single elements (i.e., 64-byte memory transactions comprised of four-byte accesses from a half-warp), or vector-types (e.g., 128-byte memory transactions comprised of two-component, eight-byte accesses)? Should a read be immediately followed by a write, or should we make two reads followed by two writes? Or four? Should we use

synchronization instructions to prevent threadblocks from overlapping memory operations (i.e., disallowing some warps run ahead of others into the next reads/writes)?

We selected four orthogonal dimensions in which to perform a parameter space study, choosing a small subset of reasonable values from each domain to test:

- Threads per block: {128, 256, 512}
- Memory transaction size: {64B, 128B}
- Number of reads-per-cycle: {one, two, four}
- Overlapped I/O: {yes, no}

This yields a search space of 36 configurations, and we test this search space for both *in-out* and *in-only* patterns. Each configuration dictates its own cooperative cycle-size parameter *b* such that:

$$b = \frac{\text{(threads per block)(transaction size)(reads-per-cycle)}}{\text{(16 threads-per-halfwarp)(4 bytes per element)}}$$

4.2.1 Skeletons for variable-sized grids

Although it is our ultimate destination, we can't begin by surveying skeletons for fixed-sized grids. This is because the threadblock occupancy for a given kernel implementation is likely to influence the best value *C* for that kernel, and occupancy will depend upon the specific resource requirements of that kernel. For example, different kernel implementations may require different amounts of shared memory which will lead to different threadblock occupancies for the SM cores. We therefore survey data movement configurations for the next best alternative: variable-sized grids. We will tune for *C* in the next stage.

For this experiment, our test kernels only perform one cycle per threadblock and the grid size scales as needed with the problem size. The configurations were evaluated using the experimental setup and problem suite described in Section 5.1. The problem sizes themselves are rounded down to the nearest multiple of *b* for each configuration: the threadblocks performing the brunt of the work within our meta-strategy are not concerned with the guarded movements of leftover elements.

The results are provided in Appendix sections 7.1.1 (*in-out*) and 7.1.2 (*in-only*). For the *in-out* pattern, we also plot the intrinsic CUDA API method `cudaMemcpy(<src>, <dest>, <size>, cudaMemcpyDeviceToDevice)` as a reference point. As expected, we see that throughputs improve dramatically until memory resources become saturated, at which point they generally plateau into steady-state. In order to quantitatively compare these different configurations, we approximate this steady-state value for each configuration by averaging over all input sizes greater than 32M.

We draw several interesting conclusions from these results. The first is that there is a significant variance in performance between configurations: throughput differentials of up to 1.3x for *in-out* and up to 2.1x for *in-only*. The variance is substantial, especially given that we considered all of these to be reasonable configuration choices.

Secondly, we note that not all plots are monotonically increasing. This implies that there are some curious nuances in the way that threadblocks and warps are scheduled: another memory type-architecture mismatch that exemplifies unexpected scaling behavior. It would be unfortunate for a carefully load-balanced implementation to exhibit such variable performance[9].

A third observation is that the memory subsystem in our GTX-285 is unable to deliver as much memory bandwidth for the *in-out* pattern as it does for the *in-only* pattern. The fastest *in-only* configurations are able to average a steady-state bandwidth of 152.9 GiBytes/sec (peaking at 154.4 GiBytes/sec), which is very close to the 159 x10^9 bytes/sec maximum

[9] We hypothesize that CUDPP scan exhibits monotonic throughput for two reasons: (*i*) its kernels implement a configuration pattern that is monotonically increasing (128-thread, 128B transaction, quad-load, non-overlapped I/O), and (*ii*) it is not clear that it is completely memory-bound (see Section 5.5).

published in the technical specifications for the GPU. The best *in-out* configurations were able to average 136.1 GiBytes/sec (peaking at 140.2 GiBytes/sec). A notable result is that these *in-out* configurations out-perform the intrinsic `cudaMemcpy` method (132.1 average GiBytes/sec).

4.2.2 Expectations for fixed-size grids

Without any particular kernel implementation in mind, we surveyed the data movement configurations for fixed size grids of $C = 150$ threadblocks. We considered 150 threadblocks a reasonable evaluation point: all threadblocks can be actively scheduled across the thirty SM cores provided by our GTX-280. This results in a threadblock occupancy ratio of 5/8 for each core, providing up to 3.2KB of shared memory and 3,276 registers (e.g., 25 registers per thread over 128 threads) per threadblock.

As described in Section 3.4.4, each threadblock for a given data movement configuration executes a minimum of $\lfloor n/(Cb) \rfloor$ cycles where the cycle size b is determined by the specifics of the that particular configuration. The first $\lfloor n/b \rfloor \bmod C$ threadblocks execute an additional cycle.

The results of this second parameter space study provided in Appendix sections 7.1.3 (*in-out*) and 7.1.4 (*in-only*). We discovered several interesting consequences of fixing the grid size. On one hand, the skeleton throughputs are now monotonically increasing. On average, they are plateauing at better and more consistent steady-state values. This bodes much better for our scan and reduce kernels. On the other hand, the plots have developed periodic discontinuities: input sizes leading to dramatically deteriorated performance. As we describe later on, we will be forced to develop compensation heuristics that can avoid these problem spots by nudging to different values of C.

In addition to threadblock scheduling, this study is fundamentally different from the previous variable-sized-grid survey in that each threadblock now performs multiple data movement cycles. The chaining of in-out cycles further begs for explanation as to why overlapped versus non-overlapped I/O configurations behave differently. We find it very curious that the inclusion of synchronization barriers between memory transactions for a given threadblock can have such drastic effects. It seems reasonable that coercing concurrent warps towards the same temporal access behavior could possibly influence the memory subsystem. However, independent threadblocks running on the same SM can't be corralled in this fashion: their overlapped execution quite likely results in an arbitrary mix of concurrent transactions types. Without knowing the implementation details of the memory subsystem, it's difficult to develop a rationale for this overlapped/non-overlapped behavior

In order to investigate further, we briefly experimented by adding artificial computational loads to some of the better-performing *in-out* skeleton configurations. We did this by adding an empty for-loop between read and write sections, controlled by a volatile loop counter. We observed that as computational load was increased, the <128 thread, 128B transaction, double load, non-overlapped> configuration lost its advantage over the overlapped version. Not only does this slowdown occur before the bandwidth-cap on computational overhead, it also seems to be more in response to path length (i.e., computational latency) instead of overall work: slowdown was manifested as loop iterations increased, regardless of whether all warps participated or just one. At this point in our investigation, we can only conclude that some I/O skeletons have a smaller tolerance for computational latency than others, and this tolerance is unrelated to the normal bandwidth cap on dynamic instructions.

4.3 Stage 3: Kernel Logic

In perhaps the most interesting design stage, we construct matching reduce and scan kernels from the most promising data movement skeletons. The fundamental requirement of this stage is that an implementation must not deteriorate the throughput afforded by the data movement skeleton chosen for it. In order to do this, kernels must focus on minimal work and avoiding architectural hazards (e.g., divergence, bank conflicts, etc.).

We know the cutoff for exactly how much kernel work can be done for a given input size before the kernel is no longer memory-bound. As discussed in the introduction, the GTX-285 exhibits an 8.94x differential between computational and

memory throughputs (in terms of logical thread instructions and four-byte elements). This implies that there an overhead cap of 8.94 instructions-per-element for *in-only* reduction kernels, above which it is impossible to achieve maximal utilization of device memory bandwidth. Doubling this equates to an overhead cap of 17.9 instructions-per-element for *in-out* kernels. However, the accelerator only appears to provide a physically-achievable maximum of 138 GiB/sec for the *in-out* pattern, resulting in a more realistic overhead cap of 20.6 instructions-per-element for scan kernels.

4.3.1 merrill_tree Scan

The design problem at the threadblock level is one of algorithm selection and composition. We continue with the top-down theme of beginning with a malleable strategy and letting the architectural details guide our decisions towards a platform-specific solution. By composing the *merrill_scan* kernel as a flexible hierarchy of reduce-then-scan strategies, we can leverage different algorithms in different ways that play to the strengths of each.

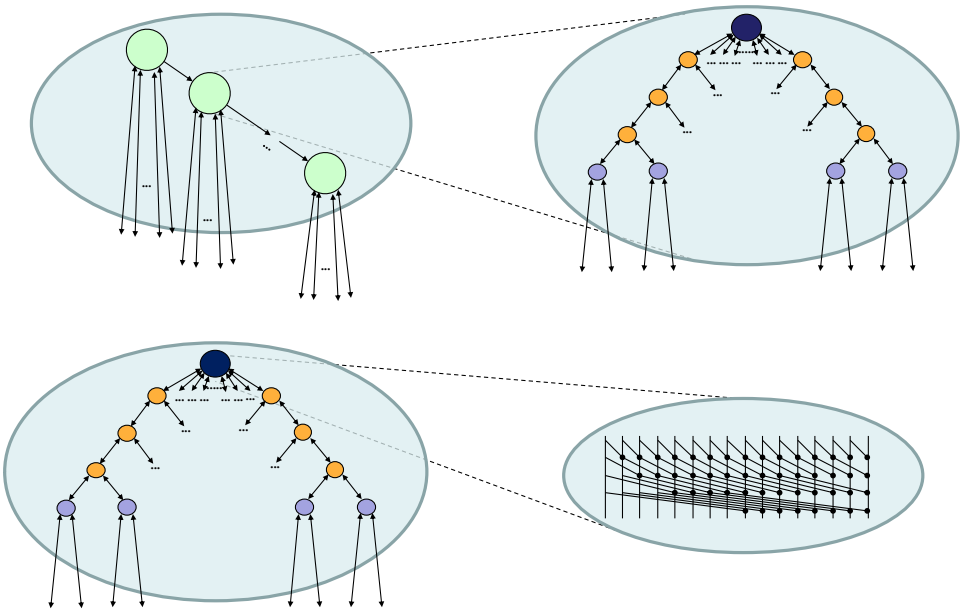


Figure 18. The decomposition of the generalized *merrill_tree* scan kernel. From the top down, each threadblock processes an independent sublist of elements, progressing serially in cycles of cooperative work (light green). Each cycle implements a three-phase reduce-then-scan. The bottom phase serially scans vector-component input items from registers into shared memory (purple). The second phase is a tree scan of our design (orange), and the top-phase is a SIMD Kogge-Stone scan (dark blue).

The reduce-then-scan strategy decomposes nicely into three phases of upsweep/downsweep operation: (1) *independent processing* in registers, (2) *inter-warp cooperation*, and (3) *intra-warp cooperation*. These three phases are illustrated in Figure 18 using purple, orange, and dark-blue colored tasks, respectively.

The bottom phase transitions the problem loaded into registers by the data movement skeleton into a smaller version that will fit into shared memory. This phase is necessary when the number of elements *b* loaded by the skeleton is greater than can be accommodated by a threadblock's shared memory allocation. For example, each thread in a double-load, 128-byte transaction skeleton will have four elements to contribute per cycle. Under the maximum SM thread occupancy for the GT200 architecture (1,024 threads), direct storage for every element would require more physical shared memory than is available[10]. If each thread first reduces its two-component vector into a single intermediate value, those intermediate values can then be placed into shared memory without problem. Serial reduction/scan in registers is a good fit at this stage because the work complexity is theoretically and practically minimal, an important consideration when all threads are active.

[10] Although each SM has 16KB of physical shared memory, not all of it is available for use. Portions of it are allocated for system use, e.g., for ancillary data such as kernel input parameters.

The NVIDIA product documentation specifies that memory accesses are only efficient when they can be *coalesced*, i.e. the addresses referenced by a SIMD half-warp must fall within a contiguous memory segment. If a thread performs multiple loads for a given cycle, the coalescing mandate entails that those elements will be offset by a stride that is a multiple of the half-warp size (and often a multiple of the number of threads in the threadblock). The *independent processing* phase can be leveraged extensively within reduction kernels because there are no prefix dependencies between elements from different loads. Threads within scan kernels, on the other hand, cannot independently reduce elements from different loads. In this case, independent processing is limited to the reduction/propagation of adjacent elements from vector loads.

The middle phase is *inter-warp cooperation*. As problem size constricts during upsweep, so does the number of active warps. The exchange of intermediate results between threads requires the use of shared memory barriers. This phase can be implemented with any minimal-work, parallel strategy for reduction and scan. In order for the kernel to be adaptable to data movement skeletons of different cycle sizes, the middle phase should be flexible in terms of the number of steps it can perform. Our designs are named after this middle phase.

The top phase of the intra-threadblock reduce-then-scan strategy is *intra-warp cooperation*. Algorithmic work-efficiency is not as important when the active problem size has become smaller than the SIMD width. SIMD resources are used regardless at this point, so work efficiency is directly tied to step-efficiency.

Because our design is flexible with respect to cycle size *b*, we greedily select the best column of *in-out* configurations tabled in Appendix sections 7.1.1 and 7.1.3 as a starting point. Due to their superior throughputs, we begin our efforts using the <128-thread, 128B transaction, single-load> configurations. This entails a scan kernel cycle of $b = 256$ elements, allowing us to flesh out the three phases of reduce-then-scan as follows:

- *Bottom Phase*: A one-step, 128-wide serial reduce-then-scan in registers, placing the resultant partial reduction in shared memory.
- *Middle Phase*: Two steps of our tree algorithm for the Brent-Kung strategy in shared memory (64-wide, and 32-wide). Requires 768 bytes of shared memory per threadblock.
- *Top Phase*: Five steps of synchronization-free SIMD Kogge-Stone "warp-scan" in shared memory (32-wide). Requires 192 bytes of shared memory per threadblock.

Implementing this for a variable-sized grid of threadblocks, we observed that the dynamic instruction count for our scan kernel was too large: we averaged 22.8 thread instructions per input element, making it impossible for us to obtain our goal of 136+ GiBytes/sec.

We can perform fewer instructions by switching to a larger cycle size. This results in fewer cycles and a smaller proportion of time spent in the less-efficient middle and top phases. We fall back to the double-load versions of the <128-thread, 128B transaction> configurations. This entails a scan kernel cycle of $b = 512$ elements, and we compose our phases as follows:

- *Bottom Phase*: For each of two load steps: a one-step, 128-wide serial reduce-then-scan in registers, placing the resultant partial reduction in shared memory.
- *Middle Phase*: Three steps of our tree algorithm for the Brent-Kung strategy in shared memory (128-wide, 64-wide, and 32-wide). Requires 1,792 bytes of shared memory per threadblock.
- *Top Phase*: Five steps of synchronization-free SIMD Kogge-Stone "warp-scan" in shared memory (32-wide). Requires 192 bytes of shared memory per threadblock.

Implementing this scan kernel for a variable-sized grid of threadblocks, we determined that the dynamic instruction count averaged 18.9 thread instructions per input element, making it possible for it to be completely memory bound. (The computational overhead will ultimately decrease when we switch to a fixed-size grid.) Unfortunately we discovered that the computational latency was apparently greater than the threshold point at which non-overlapped I/O performance begins to degrade, forcing us to switch to the overlapped I/O version of this configuration.

As part of the *merrill_tree* implementation, we need an upsweep reduction kernel to complement the downsweep scan kernel. Our reduction kernel design is composed of the same three phases as the scan kernel, yet without the corresponding downsweep. Reviewing the throughput results from Section 4.2.1, we chose the <128-thread, 64B transaction, quad-load, overlapped I/O> configuration, implementing our phases as follows:

- *Bottom Phase*: For each of four load steps: a one-step, 128-wide serial reduction into an accumulator register. Repeat for all $n/(Cb)$ cycles.
- *Middle Phase*: Three steps of the upsweep reduction portion of our permuted tree algorithm (Section 3.3.6) in shared memory (128-wide, 64-wide, and 32-wide). Requires 1,792 bytes of shared memory per threadblock.
- *Top Phase*: Five steps of synchronization-free SIMD Kogge-Stone "warp-scan" in shared memory (32-wide)[11]. This phase requires no additional shared memory, as the scan can be performed in place.

The next design step is to determine an appropriate value for *C*, our fixed grid-size. We do this by evaluating *merrill_tree* throughput as a function of grid-size, plotting results for five different problem sizes so as to get reasonable representation across the spectrum of problem-sizes.

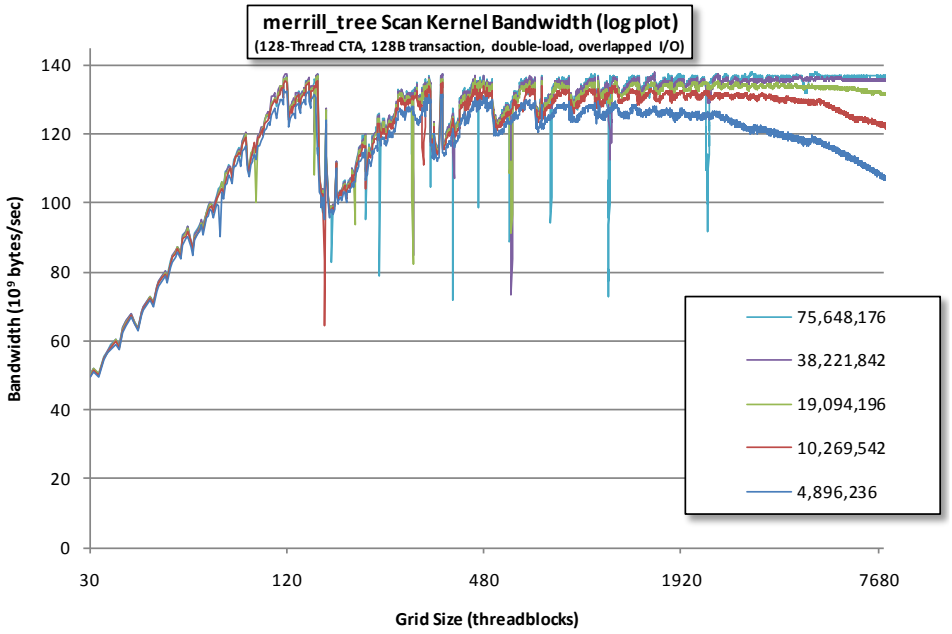


Figure 19. Log-plot of scan kernel bandwidth for *merrill_tree* as a function of grid-size for five problem instances: 4896236, 10269542, 19094196, 38221842, and 75648176 elements.

The results of this experiment are depicted in Figure 19. Throughput grows quickly as the SMs fill up with active threadblocks. The system experiences a small drop in throughput after every period of thirty threadblocks, likely an artifact of uneven loading across the thirty SMs. There is a much larger drop every 180 threadblocks (the maximum occupancy for thirty SMs given our shared memory usage): leftover threadblocks must wait to be scheduled until spots begin to free up, yet there are not enough of these leftover blocks to keep the SMs filled after the last full batch has completed. We observed distinct peaks for each problem (118, 150, 150, 1589, and 3144 threadblocks, respectively), after which throughput decreases steadily. This is indicative of the increasing overhead placed on the threadblock dispatcher as the number of threadblocks grows, and this falloff is more pronounced for smaller problem sizes. We chose a value of $C = 150$ for our implementation: it

[11] We use a scan algorithm here to perform a reduction task: the reduction tree for 32 leaves cannot be computed more efficiently than as performed by the SIMD scan.

provides peak or near-peak throughput for our sample set of problems, there are no leftover blocks to be scheduled after the initial dispatch, and they can be divided up evenly amongst the SMs.

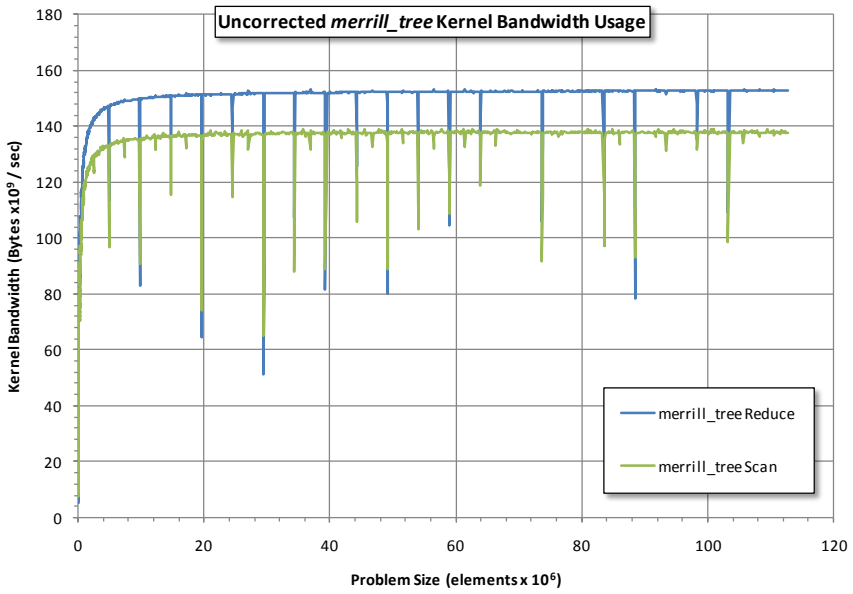


Figure 20. Uncorrected problem throughput for *merrill_tree* as a function of problem size (grid-size $C = 150$).

Now that we have fixed the grid-size, we must deal with any degenerate behavior that may manifest itself as a result. Figure 20 depicts our kernel bandwidths as a function of problem size. We notice several modes of deterioration, the shortest having a period of 2,457,600 elements, or $32Cb$. In order to eliminate this behavior, we modify our kernel dispatch logic in the host to iteratively nudge C to smaller values until it determines that the problem size is sufficiently distant from a multiple of $32Cb$.

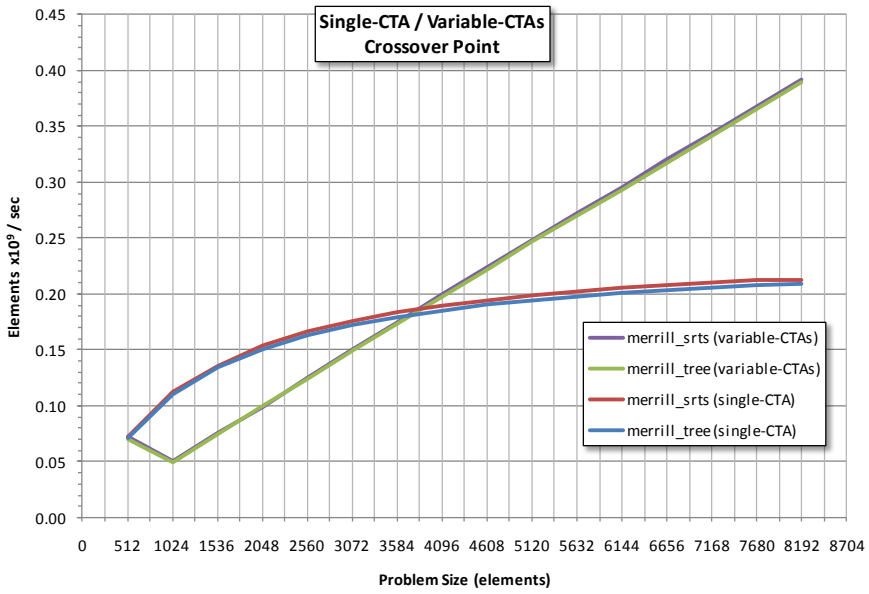


Figure 21. The crossover point between performing a single-threadblock scan versus a two-level meta-scan. Problem throughput is plotted as a function of problem size.

The final task is to determine the crossover point at which the throughput afforded by spreading computation over multiple SM cores outweighs the cost of performing a two-level meta-scan. Figure 21 compares single-threadblock throughput (in which the number of cycles scales with problem size) versus variable-sized-grid throughput (in which the number of bottom-level threadblocks scales with problem size, each threadblock performing one cycle). We observe that the optimal transition to a two-level scan at occurs at 4,096 elements for both *merrill_tree* and *merrill_srts* implementations, with *merrill_srts* outperforming *merrill_tree* in the single-threadblock scenario by 2.1%. We present a full evaluation of the final *merrill_tree* implementation in Section 5.

4.3.2 merrill_srts Scan

Although the *merrill_tree* implementation is able to perform scan optimally with respect to its data-movement skeleton, we had to dismiss several slightly faster skeletons in order to accommodate its computational overhead. Fortunately we can make a drop-in replacement of the tree scan algorithm in the middle phase with an array of concurrently executing serial reductions/scans. As Dotsenko et al. point out, tree scan algorithms have an inherently higher work overhead than serial reduce-then-scan algorithms because of the necessary activation conditionals and barrier instructions [17].

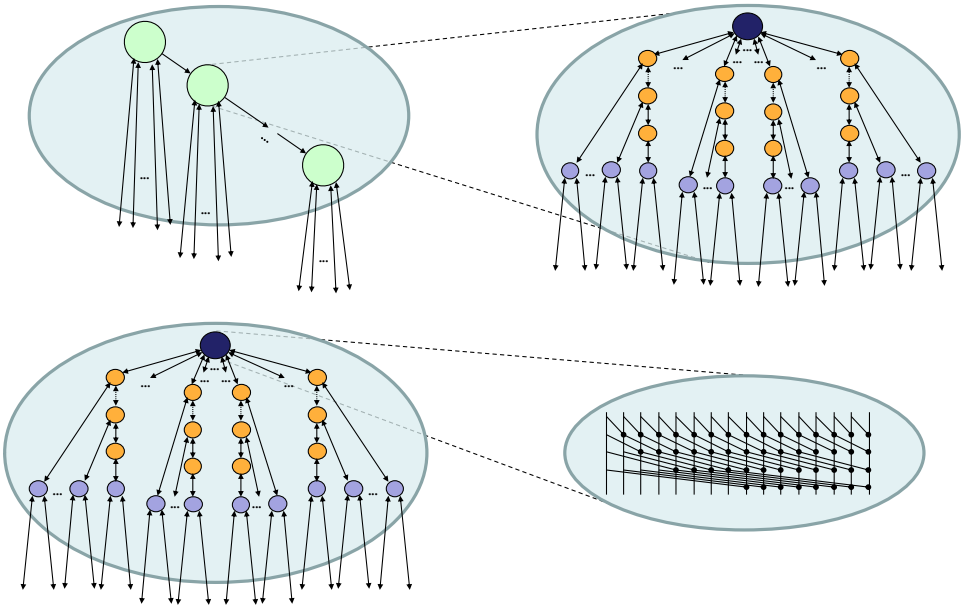


Figure 22. The decomposition of the generalized *merrill_srts* scan kernel. From the top down, each threadblock processes an independent sublist of elements, progressing serially in cycles of cooperative work (light green). Each cycle implements a three-phase reduce-then-scan. The bottom phase serially scans vector-component input items from registers into shared memory (purple). The second phase is a concurrent set of serial reduce-then-scans, and the top-phase is a SIMD Kogge-Stone scan (dark blue).

Figure 22 above depicts the decomposition of the generalized *merrill_srts* design. The serial operation of the middle phase is similar to the bottom phase, except values are iterated over in rows of shared memory instead of registers. The purpose of this middle phase is to transition the problem placed into shared memory by the bottom phase into one matching the SIMD width of the top phase. Therefore the number of rows in the array of serial scans will be equal to the SIMD width. The length of each row will be dependent on the differential between the cycle size *b* and the fixed SIMD width.

As with the previous design, we begin our efforts using the <128-thread, 128B transaction, single-load> configurations. This entails a scan kernel cycle of $b = 256$ elements, allowing us to flesh out the three phases of reduce-then-scan as follows:

- *Bottom Phase*: A one-step, 128-wide serial reduce-then-scan in registers.
- *Middle Phase*: Four steps of 32-wide serial reduce-then-scan in shared memory. Requires 2,176 bytes of shared memory per threadblock. (Rows must be padded up to 17 elements to avoid bank conflicts.)
- *Top Phase*: Five steps of synchronization-free SIMD Kogge-Stone "warp-scan" in shared memory (32-wide). Requires 192 bytes of shared memory per threadblock.

Implementing this scan kernel for a variable-sized grid of threadblocks, we determined that the dynamic instruction count averaged 13.6 thread instructions per input element, making it possible for it to be completely memory bound. Unfortunately we discovered we were unable to obtain our target bandwidth of 136+ GiBytes/sec: the computational latencies for both overlapped and non-overlapped configurations were greater than the threshold at which point I/O performance begins to degrade. We fall back to the more tolerant double-load <128-thread, 128B transaction, non-overlapped I/O> configuration. This entails a scan kernel cycle of $b = 512$ elements, and we compose our phases as follows:

- *Bottom Phase*: For each of two load steps: a one-step, 128-wide serial reduce-then-scan in registers.
- *Middle Phase*: Eight steps of 32-wide serial reduce-then-scan in shared memory. Requires 2,176 bytes of shared memory per threadblock. (Rows must be padded up to 17 elements to avoid bank conflicts.)
- *Top Phase*: Five steps of synchronization-free SIMD Kogge-Stone "warp-scan" in shared memory (32-wide). Requires 192 bytes of shared memory per threadblock.

Because the *merrill_tree* and *merrill_srts* scan kernels are functionally equivalent, we can reuse the same *merrill_tree* reduction kernel to compliment *merrill_srts* scan.

The next design step is to determine an appropriate value for *C*, our fixed grid-size. We do this by evaluating *merrill_srts* throughput as a function of grid-size, plotting results for five different problem sizes so as to get reasonable representation across the problem-size domain.

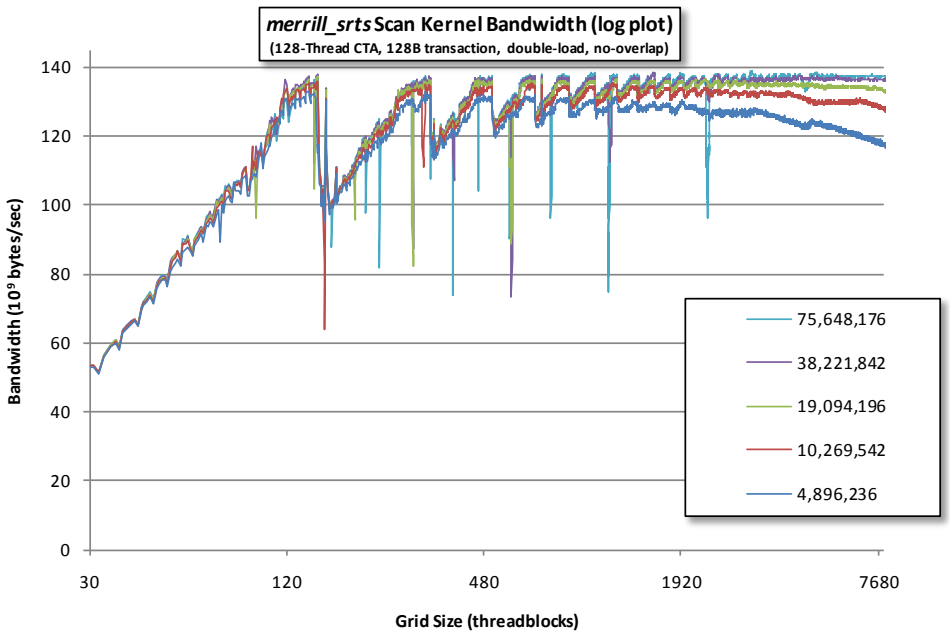


Figure 23. Log-plot of scan kernel bandwidth for *merrill_srts* as a function of grid-size for five problem instances: 4896236, 10269542, 19094196, 38221842, and 75648176 elements.

The results of this experiment are depicted in Figure 23, and are very similar to those from the grid-size evaluation of *merrill_tree*. Throughput grows quickly as the SMs fill up with active threadblocks, reaches a peak, and falls off as grid-size increases (more so for smaller problems). There are periodic dips in performance: small drop-offs having a period of thirty threadblocks and larger drop-offs every 180 threadblocks (the maximum occupancy for thirty SMs given our shared memory usage). The peak throughputs for our sample problems occurred at 140, 150, 150, 1589, and 3146 threadblocks, respectively. As with *merrill_tree*, we chose a value of $C = 150$ for our implementation: it provides peak or near-peak throughput for our sample set of problems, there are no leftover blocks to be scheduled after the initial dispatch, and they can be divided up evenly amongst the SMs.

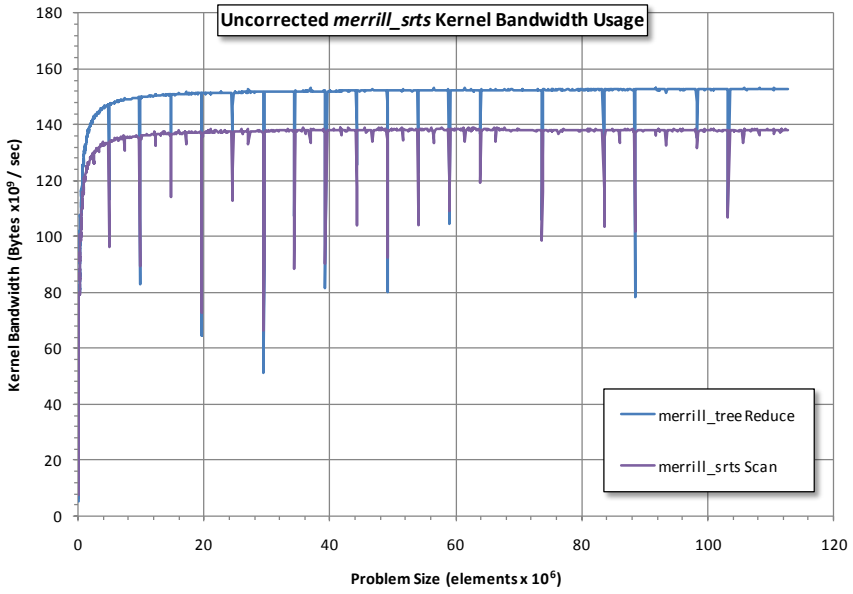


Figure 24. Uncorrected problem throughput for *merrill_srts* as a function of problem size (grid-size $C = 150$).

Now that we have fixed the grid-size, we must deal with any degenerate behavior that may manifest itself as a result. Figure 24 depicts our kernel bandwidths as a function of problem size. We notice the same periodic modes of deterioration as with *merrill_tree*, the shortest having a period of 2,457,600 elements, or $32Cb$. In order to eliminate this behavior, we modify our kernel dispatch logic in the host to iteratively nudge C to smaller values until it determines that the problem size is sufficiently distant from a multiple of $32Cb$.

4.3.3 merrill_linear Scan

In the previous two designs, each threadblock has been assigned a single input list of elements scan/reduce. A given work assignment entailed some variable number of cooperative cycles, each of which was processed using an intra-threadblock hierarchy of reduce-then-scan strategies. For each cycle of b elements, this upsweep/downsweep process resulted in $O(2b)$ work.

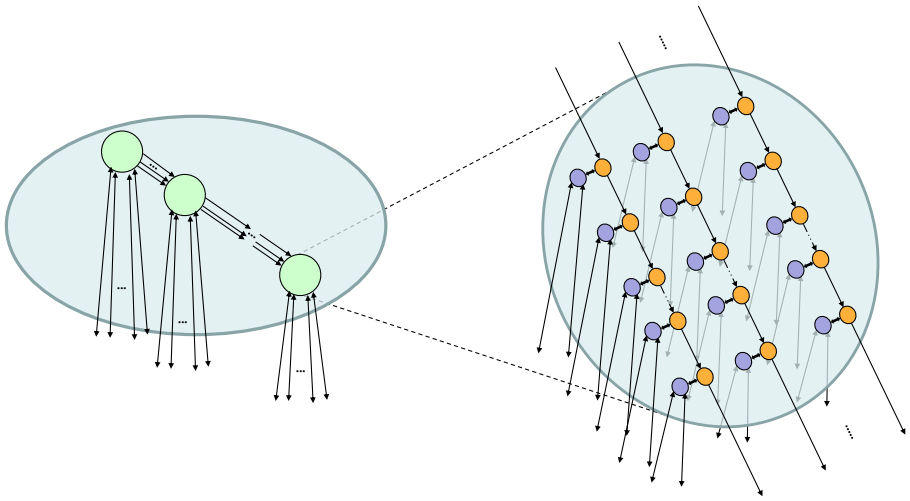


Figure 25. The decomposition of the generalized *merrill_srts* scan kernel. From the top down, each threadblock processes *S* independent sublists, progressing serially in cycles of work (light green). Each cycle implements a two-level reduce-then-scan, with two input items being reduced from registers into shared memory (purple) where they can be scanned serially (orange).

We take a different approach with *merrill_linear* in order to perform only *O*(*b*) work per cycle. The generalized design is similar to *merrill_srts*, but with the elimination of the top-phase. Without a top phase to compose partial reductions, *merrill_linear* kernels are effectively treating each threadblock assignment as *S* independent input sublists for scanning/reducing. A downsweep propagation phase is therefore not needed because the work assigned to each middle-phase lane is for a different, independent input sublist: there is no parallel work that needs composing. This results in a kernel design in which a large number of threads participate in data movement while a much smaller subset of these threads perform the bulk of the reduction work.

An interesting aspect of this design is how it affects the larger meta-strategy. Because these scan threadblocks process sets of independent input sublists, *merrill_linear* is not capable of a one-level scan. It requires a top-level single-list scan to compose these sublists. We chose the *merrill_srts* scan kernel to fulfill this responsibility as it is the faster of our two scan kernels in terms of single-block problem sizes. Because each bottom-level *merrill_linear* reduction threadblock will provide *S* partial reduction values to the top-level, the top-level threadblock must scan *CS* values instead of just *C*. Given the performance characteristics of single-block *merrill_srts*, we will want to keep *CS* under 4,096 elements, the inflection point at which it becomes more efficient to use multiple threadblocks.

Like all of our work, the design process for the *merrill_linear* implementation proceeds in a top-down fashion from observations made regarding the memory subsystem. The need for coalescing loads and stores plays an interesting role in the *merrill_linear* design. In our previous designs, consecutive half-warps would make transactions to consecutive memory blocks. The design for *merrill_linear* forces a different pattern: we will have to implement a transaction stride so that different half-warps can concurrently stream through different input sublists.

In order to see how this would affect throughput, we experimented with several of the higher-performing data skeletons by implementing a large stride between half-warps. Contrary what the documentation would indicate, we discovered that performance was deteriorated. We determined that throughput could only be maintained if strides were made between *whole* warps.

For our architecture, these transaction-striding requirements imply each input sublist will have 32 thread-reads made to it during each cooperative cycle. Therefore we need to have a 32-value shared memory row for each input sublist into which data-movement threads can place their values for subsequent processing by a scanning thread. Because the GT200

architecture only provides 16KB of shared memory per SM, we are forced into a tradeoff between higher values of *S* (more independent rows per threadblock) versus threadblock occupancy. Ideally we would like to have *S* be a multiple 32, our SIMD width. Unfortunately the size of the requisite shared memory array leads to a threadblock occupancy of 3/8 on this architecture. We choose a middle ground: *S* = 16 rows. With one cell of padding per row, each scan threadblock requires 2,112 bytes of shared memory, resulting in an SM threadblock occupancy of up to 6/8.

Our shared memory space for cooperation is comprised of 512 elements: 16 rows of 32 elements each. Our options for data movement skeletons are therefore: 128-thread blocks with quad-loads, 256-thread blocks with double-loads, or 512-thread blocks with single-loads. After reviewing the throughput results presented in Section 4.2, we further narrow our selection to the pair of <128-thread, 128B transaction, quad-load> configurations. This gives our implementation a cycle size *b* = 1,024 elements. The two-component transaction type implies that the threadblock will actually be implementing a two-phase reduce-then-scan strategy. The two-component vectors are reduced into single values before being pushed up into shared memory. After those values are serially scanned, the results are pushed down into a two-element scan before being written back out to device memory.

Implementing this scan kernel for a variable-sized grid of threadblocks, we determined that the dynamic instruction count averaged 13.2 thread instructions per input element, making it possible for it to be completely memory bound. (The computational overhead will ultimately decrease when we switch to a fixed-size grid.) Unfortunately we discovered that the computational latency was greater than the threshold at which point non-overlapped I/O performance begins to degrade, forcing us to switch to the overlapped I/O version of this configuration.

Reviewing the throughput results from Section 4.2.1, we chose the <128-thread, 64B transaction, quad-load, overlapped I/O> for our companion reduction kernel. The design composition for the *merrill_linear* reduction kernel is nearly identical to that of the *merrill_tree* reduction kernel. Instead of having consecutive warps stride through consecutive memory blocks, the stride between warps follows that of the scan kernel: *n*/(*CS*) elements.

The next design step is to determine an appropriate value for *C*, our fixed grid-size. We do this by evaluating *merrill_linear* throughput as a function of grid-size, plotting results for five different representative problem sizes.

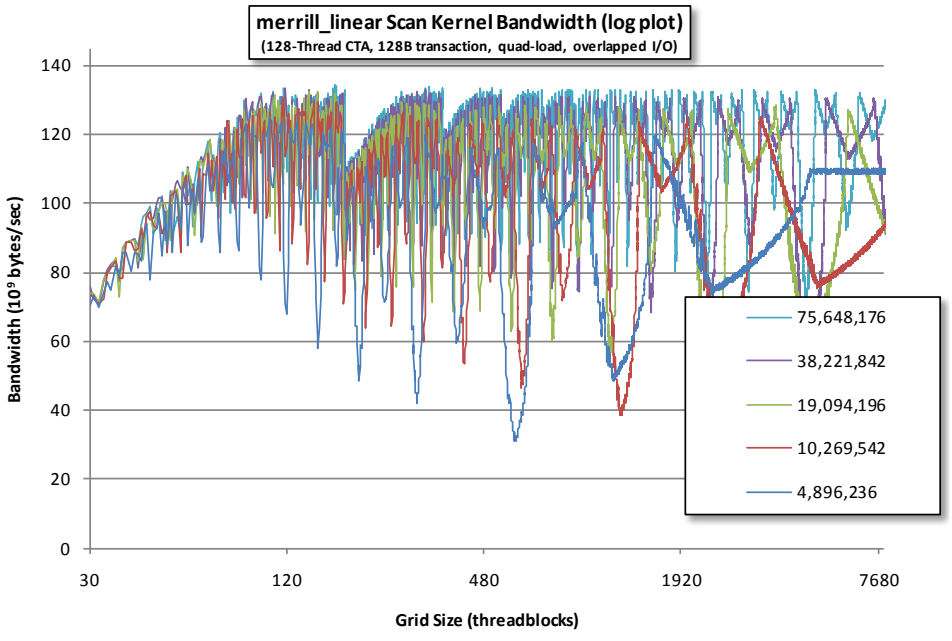


Figure 26. Log-plot of scan kernel bandwidth for *merrill_linear* as a function of grid-size for five problem instances: 4896236, 10269542, 19094196, 38221842, and 75648176 elements.

The results of this experiment are depicted in Figure 26. Throughput grows quickly as the SMs fill up with active threadblocks. The system experiences a periodic drop in throughput every 180 threadblocks (the maximum occupancy for thirty SMs given our shared memory usage): leftover threadblocks must wait to be scheduled until spots begin to free up, yet there are not enough of these leftover blocks to keep the SMs filled after the last full batch has completed. We observe that peak throughput occurs in the range of 106-169 threadblocks for these problem instances, after which bandwidth steadily decreases. This is indicative of the increasing overhead placed on the threadblock dispatcher as the number of threadblocks grows, and is more pronounced for smaller problem sizes. We chose a value of $C = 150$ for our implementation: it falls nicely in the middle of the range above, there are no leftover blocks to be scheduled after the initial dispatch, and they can be divided up evenly amongst the SMs.

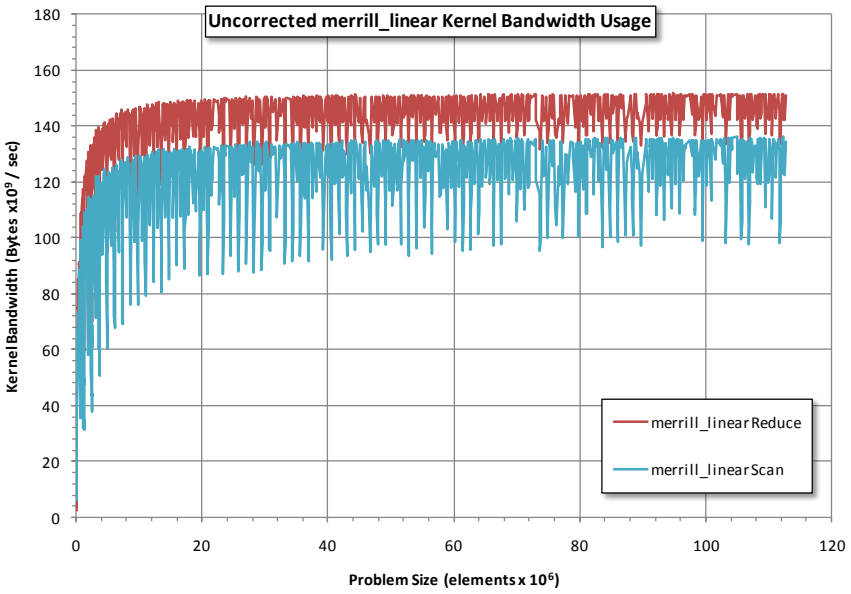


Figure 27. Uncorrected problem throughput for *merrill_linear* as a function of problem size (grid-size C = 150).

The final task is to deal with any periodic deterioration that manifests itself as a result of fixing the grid-size. Figure 27 depicts our kernel bandwidths as a function of problem size. We notice several periodic modes of deterioration, the shortest having a period of 614,400 elements, or $4Cb$. In order to eliminate this behavior, we modify our kernel dispatch logic in the host to iteratively nudge C to smaller values until it determines that the problem size is sufficiently distant from a multiple of $4Cb$. We present a full evaluation of the final *merrill_linear* implementation in Section 5.

4.4 Current Implementations of GPGPU Parallel Scan

The three most relevant implementations of parallel prefix for GPGPU stream architectures are the open-source *CUDPP* implementation by Sengupta et al. (NVIDIA and UC Davis) [8], the *MatrixScan* implementation developed by Dotsenko et al. (Microsoft Reasearch) [17], and the parallel compaction implementation by Billeter et al. (Chalmers University) [21].

4.4.1 CUDPP

CUDPP implements a scan-then-propagate meta-strategy for integrating work from different threadblocks. The size of the threadblock tree scales linearly with problem size: the host launches $\log_{1024} n$ levels of scan kernels (upsweep) followed by $\log_{1024} n$ levels of propagation kernels (downsweep).

The 128-thread *CUDPP* scan kernel has a cooperative cycle of 1,024 elements, each thread contributing eight elements obtained from two 4-component vector loads. There is no reuse of threads amongst cycles: each threadblock executes exactly one cycle.

The cooperative cycle is itself implemented as a three-phase scan-then-propagate meta-strategy. The bottom phase entails serial computation in registers to shrink the problem so that it will fit into shared memory. For each of the two loads, the implementation performs a three-step, 128-wide serial scan-then-propagate in registers, resulting in two groups of 128 partial reductions each, one group per load. For each of the two groups of partial reductions, the middle phase concurrently executes four 32-wide, five-step SIMD Kogge-Stone warp-scans in shared memory. The top phase then performs a single 8-wide, three-step SIMD Kogge-Stone warp-scan in shared memory to compose the partial reductions from the middle-phase.

The *CUDPP* implementation is inefficient in that the scan-then-combine composition of threadblocks means that the entire primitive must move 33% more data through device memory than a reduce-then-scan composition. Compared to a two-level tree of threadblocks, the $\log_{1024}n$-level tree imposes $O(n)$ unnecessary memory transactions for intermediate calculations, which is exacerbated by the output of $O(b)$ intermediate values per threadblock instead of $O(1)$. Given limited device memory, this storage overhead imposes limitations on problem size. In addition, the use of work-inefficient SIMD Kogge-Stone "warp-scans" for processing phases other than the top-phase of a cooperative threadblock cycle results in a much higher dynamic instruction count than necessary. Finally, the data movement configuration is not tunable for optimal throughput for a given CUDA device.

4.4.2 MatrixScan

MatrixScan implements a reduce-then-scan meta-strategy for integrating work from different threadblocks. The size of the threadblock tree scales linearly with problem size: the host launches $\log_{1024}n$ levels of reduce kernels (upsweep) followed by $\log_{1024}n$ levels of scan kernels (downsweep).

The 256-thread *MatrixScan* scan kernel has a cooperative cycle of 1,024 elements, each thread contributing four elements obtained from four separate loads directly into shared memory. This leads to a threadblock occupancy of only 3/8 per SM core for current CUDA architectures. There is no reuse of threads amongst cycles: each threadblock executes exactly one cycle.

The cooperative cycle is itself composed of a three-phase reduce-then-scan meta-strategy. All three phases implement serial reduce-then-scan strategies in shared memory: a 32-wide, 31-step bottom phase; an eight-wide, three-step middle phase; and a one-wide, seven-step top phase.

By employing a reduce-then-scan meta-strategy, the *MatrixScan* implementation is more efficient than *CUDPP* in terms of device memory accesses. Its inefficiencies stem from the fact that it performs more than one phase of serial reduce-then-scan per storage location. Within the hierarchy of threadblocks, doing so results in $O(n)$ unnecessary transactions for storing intermediate calculations within device memory. Within the cooperative work cycle, doing so results unused SIMD resources. Finally, there is no procedure for tuning the *MatrixScan* data movement configuration to provide optimal throughput for a given CUDA device.

4.4.3 Chalmers Prefix Sum

In their work on stream compaction for GPGPUs, Billeter et al. briefly describe a prefix-sum variant of their stream compaction implementation. They implement a two-level reduce-then-scan meta-strategy for integrating work from $C = 120$ bottom-level threadblocks. Their 128-thread scan kernel has a cooperative cycle of 256 elements, each thread contributing two elements obtained from a 2-component vector load directly into shared memory. The cooperative cycle is comprised of an eight-step Kogge-Stone strategy implemented using the Hillis-Steele algorithm.

By employing a work-inefficient strategy for problems larger than the SIMD width, their scan kernel exhibits unnecessarily high dynamic instruction counts in the same manner as *CUDPP*. Judging from their single evaluation point (32M element

scan on using a GTX-280 averaging 3.7ms), we hypothesize that their scan kernel is indeed computationally bound[12]. As with the *CUDPP* and *MatrixScan* implementations, there is no procedure for tuning their data-movement configuration to provide optimal throughput for a given CUDA device.

5 Performance Evaluation

We evaluated our three new scan implementations *merrill_tree*, *merrill_srts*, and *merrill_linear* along with the *CUDPP* v1.1 implementation as a reference point. Our primary comparison points are overall program throughput, individual kernel bandwidth, overall computational overhead, and individual kernel computational overhead.

5.1 Test Configuration

Our test platform consisted of a high-end PC running Ubuntu Linux (version 8.04.3) with an Intel CPU (Intel i7 2.66 GHz quad-core) and an NVIDIA GTX-285 GPU (GT200 architecture). The GTX-285 implements a 1.48 GHz shader clock, 159 GiB/s (10^9 bytes) device memory bandwidth, and 1 GB device memory. For reference, we used versions v2.2 of the CUDA toolkit and v185.18.14 of the CUDA device driver.

Our analyses are primarily derived from performance measurements taken over a variety of problem sizes. Unless otherwise noted, all measurements are in regard to a suite of 2,022 input problems comprised of:

- 1,000 problems with sizes sampled uniformly from the range $[2^5, 2^{26.75}]$, randomly initialized
- 1,000 problems with sizes sampled log-normally (base-2) from the range $[2^5, 2^{26.75}]$, randomly initialized
- 22 problems with sizes comprising the powers-of-two between 2^5 and 2^{26}, randomly initialized

The data points presented are the averages of measurements taken from 200 iterations of each problem instance. The measurements themselves (e.g., elapsed time, dynamic instruction count, memory transactions, etc.) are taken from hardware performance counters located within the GPU itself. Thus our analyses preclude driver overhead and the overheads of staging data onto and off of the accelerator, allowing us to directly contrast the individual and cumulative performance of the stream kernels involved.

5.2 Overall Throughput

We begin with a presentation of cumulative GPU performance, i.e., the speed by which these implementations are able to scan their input problems. For each problem iteration, we used the profiler to record the elapsed GPU time for all kernel invocations (e.g., reduce, scan, etc.) made during that execution. These kernel execution times are summed to reflect the total time spent within the GPU for that iteration, and then averaged over all iterations for that problem instance.

[12] For a brief comparison point, we evaluated our own *merrill_srts* implementation on a spare GTX-280, averaging 3.16 ms for that problem size, a speedup of 1.17x. We determined our implementation to be bandwidth-bound: our scan kernel averaged 10.9 instructions per element, and the *in-out* bandwidth cap for the GTX-280 (141.7 peak GiBytes/sec, 1.3GHz shader clock) is 17.6 instructions per element.

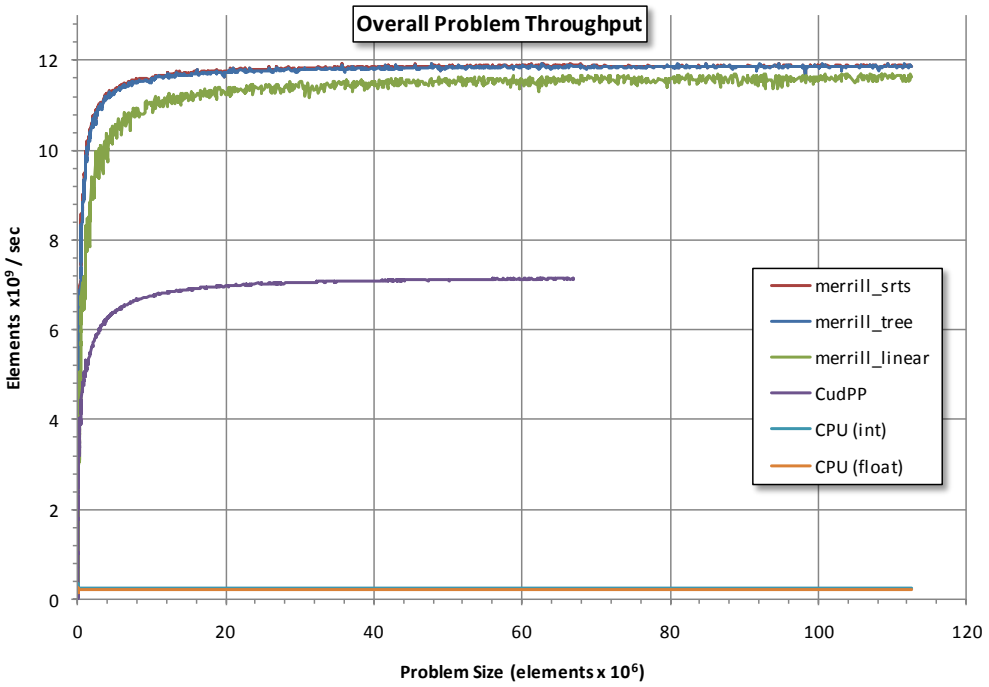


Figure 28. Problem throughput as a function of problem size, broken down by implementation.

Figure 28 above presents this timing information indirectly in terms of problem throughput. Instead of plotting elapsed GPU time as a function of problem size, we find it useful to compare the inverted derivatives, i.e., elements-scanned-per-second as a function of problem size. This affords us a better visualization of performance trends and comparative speedup. We note that neither *CUDPP* scan nor *merrill_linear* are plotted for the entire range. The plot for *CUDPP* scan performance terminates early at 64M elements, an artifact of that implementation's reliance upon variably-sized grids and the threadblock scheduling limitations of the hardware. The plot for *merrill_linear* begins at 4,096 elements because it is only applicable for problem sizes large enough to warrant a two-level scan.

We see that the performance for all four GPU-based implementations improves exponentially until the GPU's resources become saturated, at which point it plateaus into steady-state. The steady-state throughputs (averaging over all problem sizes greater than 32M) are as follows:

Table 1: Overall Throughput (elements x10^9 / second)

merrill_tree	*merrill_srts*	*merrill_linear*	*CUDPP*	*CPU (integer)*	*CPU (float)*
11.848	11.871	11.559	7.113	0.243	0.198

Our *merrill_tree* and *merrill_srts* scans exhibit nearly identical performance characteristics, with *merrill_srts* being 0.2% faster on average due to its scan kernel's slightly faster data-movement configuration. Both exhibit 1.7x, 49x, and 60x speedups over *CUDPP* scan, serial CPU integer scan, and serial CPU float scan, respectively. The *merrill_linear* implementation operates slightly slower (2.4%) compared to our other two, on average. We see that the grid-size tuning described in Section 4 has been effective in removing the periodically degenerate performances for all three of our scan implementations.

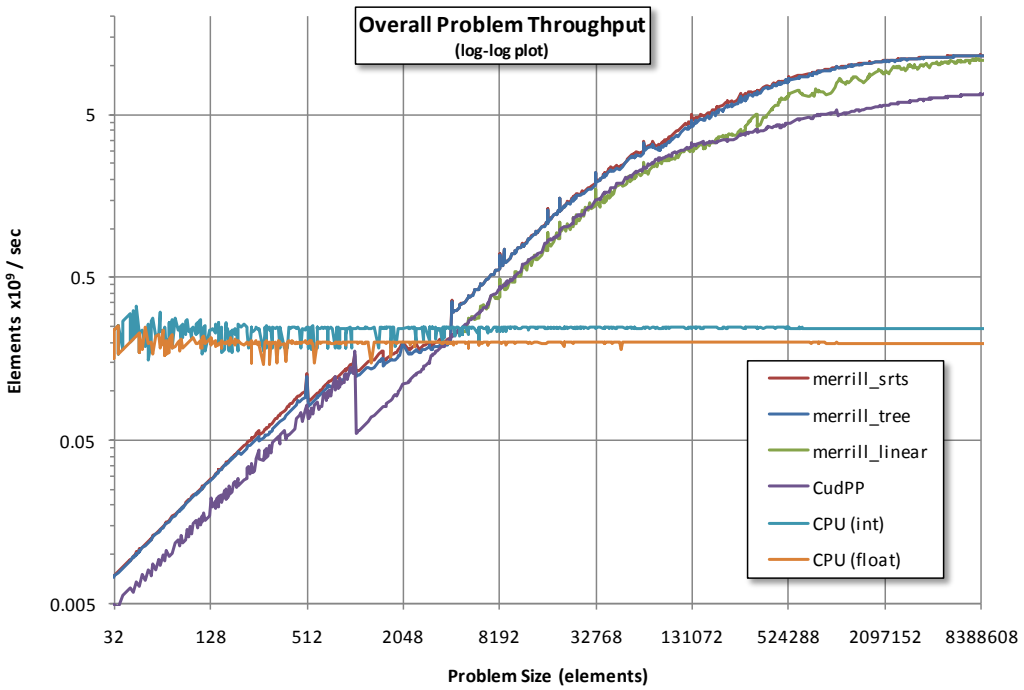


Figure 29. Log-log plots of problem throughput as a function of problem size, broken down by implementation.

The log-log plots of Figure 29 above highlight several interesting aspects of these scans. For our test platform, the crossover point at which it becomes better to use the GPU over the CPU is the same for all four scan implementations: approximately 4k elements. We see that the performance of the *CUDPP* implementation deteriorates briefly but dramatically as problem-size crests 1,024 elements, the point at which it transitions from a single-level scan into a two-level scan. Our implementations avoid this regression by making the transition to a two-level scan at 4,096 elements instead, as per our performance tuning in Section 4. The throughputs for our scans plateau briefly as the resources within the single SM saturate, and then jump back into exponential growth as the implementations shift gears and begin to leverage the resources of the remaining SMs. However, our *merrill_tree* and *merrill_srts* scan implementations do observe small periodic performance dips every 512 elements, most noticeable for small problem sizes. This is due to the incremental overhead of looping another 512-element cooperative scan cycle.

5.3 Kernel Bandwidth

We also used the profiler to record the numbers and sizes of the device memory transactions made by all bottom-level kernel invocations. We don't consider the top and mid-level kernel invocations for analysis because their lifetimes aren't sufficient for saturating the accelerator. We take the average number of bytes moved by each kernel type for a given problem size and multiply by ten: the GT200 architecture only provides memory performance counters for one of the SM clusters. (A GT200 cluster contains three SMs, and there exist ten such clusters.) Dividing by the average GPU time for that kernel type yields the average bandwidth utilized by that kernel type for the given problem size.

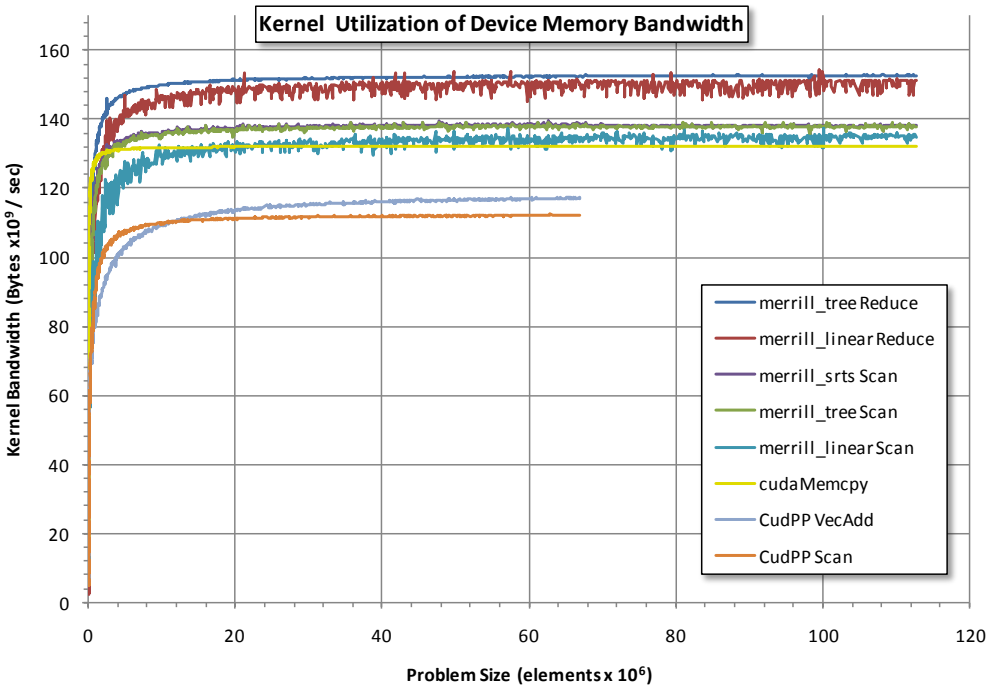


Figure 30. Device memory bandwidth consumed as a function of problem size, broken down by kernel type.

Figure 30 above presents the utilization of device memory bandwidth for each of the four GPU scan implementations, broken down by kernel type. The performance of the CUDA API method `cudaMemcpy(<src>, <dest>, <size>, cudaMemcpyDeviceToDevice)` is included for reference. The same *merrill_tree* reduction kernel accompanies both *merrill_tree* and *merrill_srts* implementations, whereas the *merrill_linear* implementation operates its own type of reduction kernel. The *in-out* pattern of data movement through device memory is exhibited by our three scan kernels, the *CUDPP* scan and propagate kernels, and the intrinsic `cudaMemcpy` operation: they all move the same amount of data. The two reduction kernels both exhibit the *in-only* pattern of device data movement and can be compared against each other. The kernel bandwidths improve exponentially until the GPU's memory subsystem becomes saturated, at which point they plateau into steady-state. The steady-state throughputs (averaging over all problem sizes greater than 32M) are as follows:

Table 2: Kernel Bandwidth (GiB / second)

	merrill_tree Reduce	*merrill_linear Reduce*	*merrill_tree Scan*	*merrill_srts Scan*	*merrill_linear Scan*	*CUDPP Scan*	*CUDPP VecAdd*	*CudaMemcpy*
Avg	152.44	149.87	137.79	138.17	134.27	111.99	116.53	132.10
Max	153.13	154.50	139.28	139.34	137.17	112.59	117.40	132.12

As with cumulative performance, the *merrill_tree* and *merrill_srts* scan kernels exhibit nearly identical performance characteristics, the difference being the slightly faster non-overlapped I/O data-movement configuration used by *merrill_srts*. All three of our scan kernels meet or exceed the bandwidth expectations set by their 150-threadblock skeleton kernel configurations (Appendix section 7.1.3) and exhibit better bandwidth utilization than the intrinsic CUDA `cudaMemcpy` function (up to 1.04x). In comparison with *CUDPP* scan kernel, our three scan kernel implementations demonstrate speedup of up to 1.23x.

We observe that the *merrill_tree* and *merrill_linear* reduction kernels perform within 4.1% and 5.7% of the theoretical maximum memory bandwidth of the accelerator, respectively. These two reduction kernels also exceed the bandwidth expectations set by their 150-threadblock skeleton kernel configurations (Appendix section 7.1.4). Although they share the same data-movement configuration, the *merrill_linear* reduction kernel is less able to overcome the periodic deficiencies induced by fixing the grid size, making it 1.7% slower than the *merrill_tree* reduction kernel on average.

Individually, the superior bandwidth utilization of our kernels is the result of our skeleton-driven approach towards kernel design. Cumulatively, the *CUDPP* propagate kernel (VecAdd) plays a significant role in the overall speedup of our implementations: as the companion kernel for *CUDPP* scan, it only utilizes 0.77x the bandwidth of our companion reduction kernels and must move twice the amount of data.

5.4 Overall Computational Overhead

We used the profiler to record the number of dynamic instructions executed by each implementation per problem instance, both cumulative and broken down by kernel type. In order to get the average number of logical thread instructions executed per problem size, we first take the average number of instructions reported for that problem size and multiply by 32. (The GT200 instruction counter only tracks SIMD warp issuances.) We then multiply again by 30: only one of the SMs is equipped with an instruction counter.

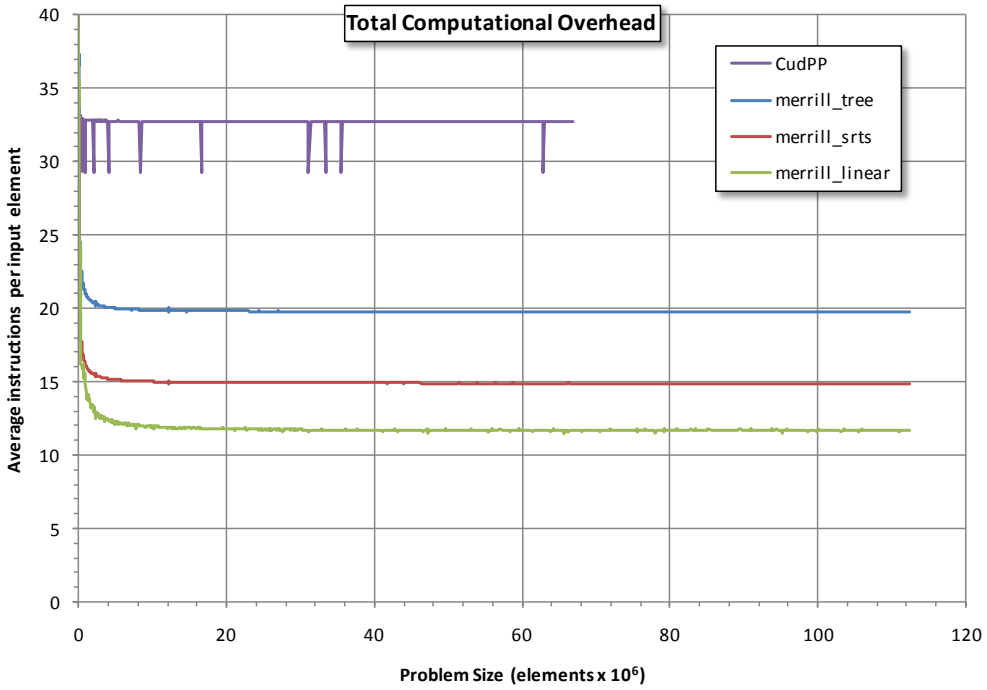


Figure 31. Computation as a function of problem size, broken down by implementation.

Similar to our treatment of overall throughput, we present dynamic instruction counts in terms of a derivative: instructions-per-element as a function of problem size. In this fashion, Figure 31 above contrasts the computational overheads incurred by each scan implementation. The range for these plots begins at 128k elements, the point at which all SMs are maximally occupied and the instruction counter profiling data becomes reliable.

We observe that the average number of instructions-per-element decreases quickly and then attains a steady-state. We notice that the *CUDPP* implementation not only achieves steady-state slightly earlier than our scans, but also exhibits periodic dips when encountering problem instances that are multiples of 1,024 elements, indicating its ability to optimize away

computation for certain problem sizes. The steady-state computational overheads (averaging over all problem sizes greater than 32M) are as follows:

Table 3: Cumulative Computational Overhead (instructions / input element)

merrill_tree	*merrill_srts*	*merrill_linear*	*CUDPP*
19.80	14.93	11.75	32.72

As we expected, our efforts are much more efficient than the *CUDPP* implementation. The *merrill_tree*, *merrill_srts*, and *merrill_linear* implementations execute 40%, 54%, and 64% fewer instructions than *CUDPP*, respectively.

We note that under more favorable circumstances, the computational overhead exhibited by the *merrill_linear* implementation could be reduced by ~8%. The *merrill_linear* scan kernel only uses only 16 of 32 SIMD lanes for serial scanning due to a conflict between coalescing requirements and limited shared memory resources. A target architecture without this conflict would allow us to double the SIMD utilization and halve the scan length per cycle. These serial scan instructions presently account for 25% of the scan kernel's dynamic instruction mix and 16% of the entire implementation's instruction mix.

5.5 Scan Kernel Computational Overhead

By determining the computational overheads of the individual scan kernels, we can evaluate how much leeway the SMT logic has in terms of scheduling warps so that device memory bandwidth can be fully utilized. The GTX-285 exhibits an 8.94x differential between computational and memory throughputs (in terms of logical thread instructions and four-byte elements) which equates to an overhead cap of 17.9 instructions-per-element for *in-out* kernels, above which it is impossible to achieve maximal utilization of device memory bandwidth.

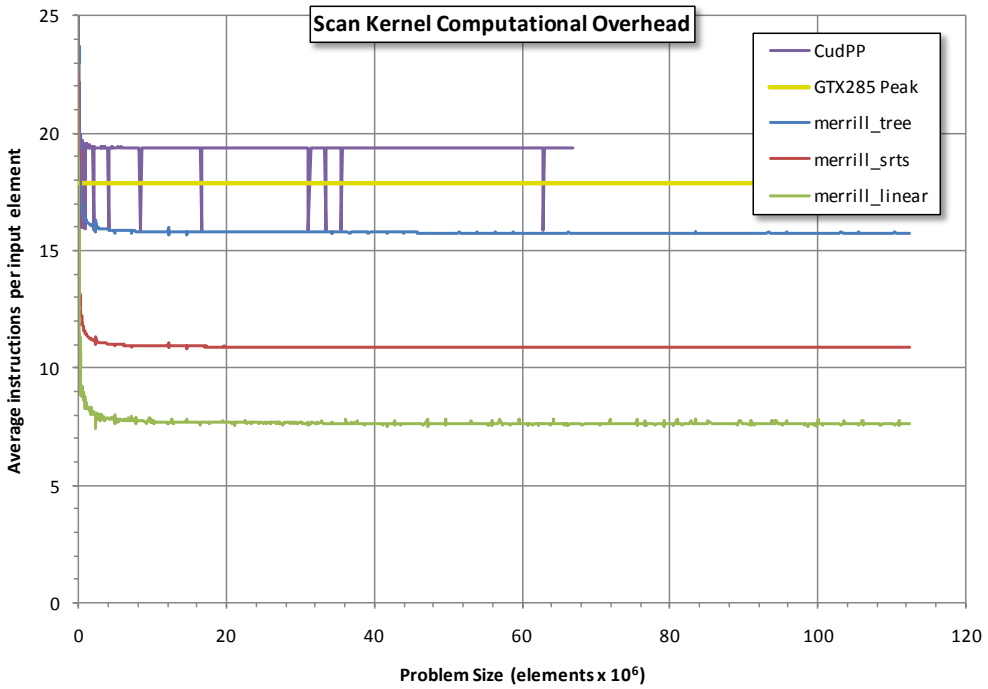


Figure 32. Computation as a function of problem size, broken down by scan kernel type.

Figure 32 presents the computational overheads of the individual bottom-level scan kernels. As with our evaluation of kernel bandwidth, we didn't consider the top and mid-level kernel invocations for analysis. The steady-state computational overheads (averaging over all problem sizes greater than 32M) are presented in the table below:

Table 4: Scan Kernel Computational Overhead (instructions / input element)

merrill_tree Scan	*merrill_srts Scan*	*merril_linear Scan*	*CUDPP Scan*
15.77	10.90	7.66	19.32

When only considering scan kernels, the comparative efficiency of our scan designs is still significant, but not as dramatic. The *merrill_tree*, *merrill_srts*, and *merrill_linear* scan kernels execute 18%, 44%, and 60% fewer instructions than the *CUDPP* scan kernel, respectively.

On paper, the *CUDPP* scan kernel instruction overhead exceeds the cap of 17.9 instructions-per-element, implying that no amount of data-movement tuning will allow it to consume the GTX-285's theoretical maximum of 159 GiB/sec of device memory bandwidth. However, the accelerator appears to provide a physically achievable maximum of 138 GiB/sec for *in-out* kernels, resulting in a more realistic overhead cap of 20.6 instructions-per-element. This would give our particular hardware an extremely small margin of error (~6%) in which to optimally schedule a hypothetical bandwidth-tuned *CUDPP* scan computation, making it unlikely that it could be brought up to speed with our implementations on our test platform.

6 Future Work

The primary product of this work is a trio of parallel scan (and reduction) implementations for the NVIDIA GTX-285 GPGPU that that fully leverage the underlying device memory bandwidth. In the process of doing so, we developed a design process that should allow us to do the same for any GPGPU product that exposes the same abstract machine and programming model. In order to further validate our methodologies, it would be useful to employ this process for other CUDA architectures such as the older G80 platform or the newer Fermi architecture.

In addition, it would be interesting to apply our design process for other problems beyond parallel scan. For example, having a clear view of the performance landscape for different data-movement skeletons would provide an advantage when designing any type of stream kernel. Another avenue of future work would be to investigate whether the strategy of fixed-size grids with reusable work cycles would be applicable for other problems that have traditionally been implemented for GPGPUs using hierarchical groupings of arbitrary depth, such as FFT.

At this juncture, we have not yet supplemented our scan implementations with variants for segmented-scan, reverse-scan, compaction, radix sort, and other related primitives. We feel that this process will be a straightforward one, yielding implementations for those problems that are just as efficient as the ones we have developed for scan. After doing so, it will be interesting to package these solutions into a binary-compatible replacement for *CUDPP*, allowing us to evaluate our improved primitives for real applications.

As a subset of this work, we have developed new algorithms for implementing several popular parallel prefix circuit strategies in programmable hardware. For future work, it would be interesting to investigate potential algorithms for other types of zero-deficiency prefix networks, perhaps inventing new zero-deficient strategies designed for minimal work (instead of the ever-popular minimal depth).

7 Appendix

7.1 Data-movement Skeletons

Here we present the results from the bandwidth tuning procedures described in Section 4.2.1, obtained using the experimental setup and input suite described in Section 5.1.

7.1.1 In-out pattern, variable grid-sizes (GTX-285)

The table below presents the bandwidth usage for our 36 *in-out* configuration instances, averaged over problem instances large enough to completely saturate the memory subsystem. As a point of reference, the equivalent CUDA API method `cudaMemcpy(<src>, <dest>, <size>, cudaMemcpyDeviceToDevice)` exhibited 132.098 x10^9 bytes / second.

Table 5: Average bandwidth for problem sizes greater than 32M (bytes x10^9 / second)

Load Pattern	128-Thread CTA (vec-1)	128-Thread CTA (vec-2)	256-Thread CTA (vec-1)	256-Thread CTA (vec-2)	512-Thread CTA (vec-1)	512-Thread CTA (vec-2)
Single	126.600	136.030	128.108	136.556	106.271	123.406
Single (no-overlap)	126.428	136.112	127.986	136.809	105.912	125.318
Double	120.958	125.429	125.278	120.932	121.493	115.036
Double (no-overlap)	127.094	135.106	126.828	127.411	110.789	122.072
Quad	115.525	110.137	119.550	113.175	107.856	105.384
Quad (no-overlap)	126.163	118.961	121.221	119.561	116.907	116.604

7.1.2 In-only pattern, variable grid-sizes

The table below presents the bandwidth usage for our 36 *in-only* configuration instances, averaged over problem instances large enough to completely saturate the memory subsystem. As a point of reference, the technical specifications for the GTX-285 indicate a theoretical maximum bandwidth of 159 x10^9 bytes / second.

Table 6: Average bandwidth for problem sizes greater than 32M (bytes x10^9 / second)

Load Pattern	128-Thread CTA (vec-1)	128-Thread CTA (vec-2)	256-Thread CTA (vec-1)	256-Thread CTA (vec-2)	512-Thread CTA (vec-1)	512-Thread CTA (vec-2)
Single	142.790	113.964	121.847	110.416	85.427	100.949
Single (no-overlap)	130.881	113.759	103.812	110.155	74.664	98.822
Double	147.596	113.996	136.319	111.502	103.521	112.079
Double (no-overlap)	138.701	113.895	113.433	112.405	88.942	110.484
Quad	152.814	112.441	152.936	110.487	147.710	111.463
Quad (no-overlap)	141.643	114.491	123.402	114.409	110.091	113.083

7.1.3 In-out pattern, 150-threadblock grid

The table below presents the bandwidth usage for our 36 *in-out* configuration instances, each employing a fixed-size grid of 150 threadblocks. The results are averaged over problem instances large enough to completely saturate the memory subsystem. As a point of reference, the equivalent CUDA API method `cudaMemcpy(<src>, <dest>, <size>, cudaMemcpyDeviceToDevice)` exhibited 132.098 x10^9 bytes / second.

Table 7: Average bandwidth for problem sizes greater than 32M (bytes x10^9 / second)

Load Pattern	128-Thread CTA (vec-1)	128-Thread CTA (vec-2)	256-Thread CTA (vec-1)	256-Thread CTA (vec-2)	512-Thread CTA (vec-1)	512-Thread CTA (vec-2)
Single	117.293	138.241	117.678	131.172	120.860	130.007

Single (no-overlap)	113.639	138.248	115.501	128.725	120.910	130.454
Double	128.998	136.735	118.783	127.318	117.106	123.635
Double (no-overlap)	130.932	138.017	117.278	129.910	118.192	130.285
Quad	126.879	122.964	110.897	115.003	115.447	108.958
Quad (no-overlap)	122.616	136.273	117.100	125.312	113.660	126.945

7.1.4 In-only pattern, 150-threadblock grid

The table below presents the bandwidth usage for our 36 *in-only* configuration instances, each employing a fixed-size grid of 150 threadblocks. The results are averaged over problem instances large enough to completely saturate the memory subsystem. As a point of reference, the technical specifications for the GTX-285 indicate a theoretical maximum bandwidth of 159 $x10^9$ bytes / second.

Table 8: Average bandwidth for problem sizes greater than 32M (bytes $x10^9$ / second)

Load Pattern	128-Thread CTA (vec-1)	128-Thread CTA (vec-2)	256-Thread CTA (vec-1)	256-Thread CTA (vec-2)	512-Thread CTA (vec-1)	512-Thread CTA (vec-2)
Single	141.314	117.050	127.562	116.718	144.744	116.272
Single (no-overlap)	132.421	116.497	113.175	111.415	126.062	114.223
Double	149.012	117.038	143.888	114.241	148.732	113.749
Double (no-overlap)	136.760	116.555	116.821	112.289	131.542	115.249
Quad	149.703	113.706	115.405	106.996	131.572	105.830
Quad (no-overlap)	138.343	116.391	117.594	112.476	133.381	115.029

8 References

[1] Blelloch, G. E. 1989. Scans as Primitive Parallel Operations. IEEE Trans. Comput. 38, 11 (Nov. 1989), 1526-1538.

[2] Blelloch, G. E. 1990. Prefix sums and their applications. Tech. Rep. CMU-CS-90-190.

[3] Hillis, W. D. and Steele, G. L. 1986. Data parallel algorithms. Commun. ACM 29, 12 (Dec. 1986), 1170-1183.

[4] Snir, M. 1986. Depth-size trade-offs for parallel prefix computation. J. Algor. 7, 2 (June), 185--201.

[5] Horn, D. 2005. Stream reduction operations for GPGPU applications. In GPU Gems 2, M. Pharr, Ed. Addison Wesley, Reading, MA, Chap. 36, 573--589.

[6] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and its applications. In Eurographics 2005, 2005.

[7] S. Sengupta, M. Harris, and M. Garland. 2008. Efficient Parallel Scan Algorithms for GPUs. NVIDIA Tech. Rep. NVR-2008-003.

[8] CUDPP: CUDA data parallel primitives library, 2009. http://www.gpgpu.org/developer/CUDPP/.

[9] Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. 1998. Lava: hardware design in Haskell. In Proceedings of the Third ACM SIGPLAN international Conference on Functional Programming (Baltimore, Maryland, United States, September 26 - 29, 1998). ICFP '98. ACM, New York, NY, 174-184.

[10] E. Axelsson, K. Claessen, and M. Sheeran. Wired: Wire-aware circuit design. In Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME), volume 3725 of Lecture Notes in Computer Science. Springer Verlag, October 2005.

[11] J. Svensson, M. Sheeran, and K. Claessen, 2008. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In Proceedings of 20th International Symposium on the Implementation and Application of Functional Languages.

[12] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. 2007. Scan primitives for GPU computing. In Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (San Diego, California, August 04 - 05, 2007). D. Fellner and S. Spencer, Eds. SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. Eurographics Association, Aire-la-Ville, Switzerland, 97-106.

[13] Harris M., Sengupta S., Owens J. D.: Parallel prefix sum (scan) with CUDA. In GPU Gems 3, Nguyen H., (Ed.). Addison Wesley, Aug. 2007.

[14] M. Harris, Optimizing parallel reduction in CUDA (2007) URL: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[15] R. Hinze. An algebra of scans. In Mathematics of Program Construction, Proceedings, volume 3125 of LNCS, pages 186-210. Springer-Verlag, 2004.

[16] Chatterjee, S., Blelloch, G. E., and Zagha, M. 1990. Scan primitives for vector computers. In Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (New York, New York, United States). Conference on High Performance Networking and Computing. IEEE Computer Society Press, Los Alamitos, CA, 666-675.

[17] Dotsenko, Y., Govindaraju, N. K., Sloan, P., Boyd, C., and Manferdelli, J. 2008. Fast scan algorithms on graphics processors. In Proceedings of the 22nd Annual international Conference on Supercomputing (Island of Kos, Greece, June 07 - 12, 2008). ICS '08. ACM, New York, NY, 205-213.

[18] Brent, R. P. and Kung, H. T. 1982. A Regular Layout for Parallel Adders. IEEE Trans. Comput. 31, 3 (Mar. 1982), 260-264.

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

[20] Sklansky, J. 1960. Conditional sum addition logic. IRE Trans. Electron. Comput. 9, 2 (June), 226-231.

[21] Billeter, M., Olsson, O., Assarson, U. Efficient Stream Compaction on Wide SIMD Many-Core Architectures. High Performance Graphics 2009, August, 2009