HIGH PERFORMANCE AND SCALABLE RADIX SORTING: A CASE STUDY OF IMPLEMENTING DYNAMIC PARALLELISM FOR GPU COMPUTING

DUANE MERRILL *and* ANDREW GRIMSHAW

Department of Computer Science, University of Virginia
Charlottesville, Virginia 22904, USA

Received January 2011
Revised March 2011
Communicated by Guest Editors

ABSTRACT

The need to rank and order data is pervasive, and many algorithms are fundamentally dependent upon sorting and partitioning operations. Prior to this work, GPU stream processors have been perceived as challenging targets for problems with dynamic and global data-dependences such as sorting. This paper presents: (1) a family of very efficient parallel algorithms for radix sorting; and (2) our *allocation-oriented* algorithmic design strategies that match the strengths of GPU processor architecture to this genre of dynamic parallelism. We demonstrate multiple factors of speedup (up to 3.8x) compared to state-of-the-art GPU sorting. We also reverse the performance differentials observed between GPU and multi/many-core CPU architectures by recent comparisons in the literature, including those with 32-core CPU-based accelerators. Our average sorting rates exceed 1B 32-bit keys/sec on a single GPU microprocessor. Our sorting passes are constructed from a very efficient parallel prefix scan "runtime" that incorporates three design features: (1) *kernel fusion* for locally generating and consuming prefix scan data; (2) *multi-scan* for performing multiple related, concurrent prefix scans (one for each partitioning bin); and (3) *flexible algorithm serialization* for avoiding unnecessary synchronization and communication within algorithmic phases, allowing us to construct a single implementation that scales well across all generations and configurations of programmable NVIDIA GPUs.

Keywords: Parallel sorting, radix sorting, GPU, prefix scan, prefix sum, kernel fusion

1. Introduction

Much effort has been spent investigating fast and efficient algorithmic primitives for GPU stream architectures, and sorting has been no exception [1]. High performance sorting is particularly desirable for GPUs: although they excel at data-independent transformations of large data sets, applications often need to sort or partition the resulting elements. This is particularly true of multi-GPU configurations where data and work must be redistributed among nodes. Because of the large problem sizes typical of GPU applications, inefficient sorting can be a major bottleneck of application performance. In addition, discretionary sorting has the potential to serve as a "bandwidth amplifier" for problems involving pointer-chasing and table lookups by smoothing otherwise incoherent memory accesses.

Although parallel sorting methods are highly concurrent and have small computational granularities[1], sorting on GPUs has been perceived as challenging in comparison to more conventional multi/many-core CPU architectures. Sorting is representative of a class of stream transformations that have irregular and global data-dependences. For this class of dynamic problems, a given output element may depend upon any/every input element and, in the general case, even the number of output items may be variable. The "difficulties" with problems like sorting are typically attributed to the global data dependences involved, an absence of efficient fine-grained hardware synchronization mechanisms, and a lack of writable, coherent caches to smooth over irregular memory access patterns [2,3]. In addition to list-processing problems (sorting, duplicate-removal, etc.), virtually all problems requiring queuing structures fall into this genre (e.g., search-space exploration, graph traversal, etc.). Our work refutes this conventional wisdom that GPU processors are mediocre platforms for sorting.

In this paper, we focus on the problem of sorting large sequences of elements, specifically sequences comprised of hundreds-of-thousands or millions of fixed-length, numeric keys. We primarily discuss two specific variants in this paper: sequences comprised (a) 32-bit integer keys paired with 32-bit satellite values; and (b) 32-bit keys only. We have generalized our solution strategy for numeric keys and structured values of other types and sizes as well.

1.1. *Contributions*

Algorithm design and flexibility. We describe an *allocation-oriented* design philosophy for implementing dynamic parallelism for problems such as sorting. We begin with a framework for cooperative allocation: i.e., each thread of execution can produce zero or more output elements in accordance to its input, and threads must cooperate to determine where these outputs can be written. This framework is derived from parallel prefix scan algorithms that we have designed expressly for the GPU bulk synchronous machine model. We then employ *kernel fusion* and *multi-scan* strategies to tailor this framework for sorting passes that partition elements into specific bins. While prefix scan has long been used as a subroutine for implementing sorting passes on SIMD architectures, we are the first to use it as the foundational "runtime" that drives the entire computation.

A critical aspect of our design approach is what we term *flexible algorithm serialization*. From a software engineering perspective, one of the biggest strengths of the GPU programming model is that it decouples virtual threads and threadblocks from actual hardware threads and cores. From a performance perspective, however, this introduces substantial overhead: when virtual threads scale with problem size, so does unnecessary communication and synchronization through shared memory spaces. An ideal implementation will restrict the amount of concurrency to match the underlying hardware, i.e., just enough to saturate the actual parallel and overlapped processing elements.

---

[1] E.g., comparison operations for comparison-based sorting, or shift and mask operations for radix sorting.

Like many problems, prefix scan can be implemented with a variety of sequential and parallel algorithms. We refer to our prefix scan implementation as a *strategy* [4] because it is a hybrid composition of several different sequential and parallel algorithms. We use different algorithms for composing different phases of computation, where each phase is intended to exploit a different memory space or aspect of computation.

The nature of the GPU machine model imposes severe performance cliffs for applications that mismatch or underutilize the underlying processor. The performance aspects of GPU processor organization (e.g., SIMD widths, core counts, register file and shared-memory sizes, etc.) vary widely among vendors, market entry points, and architectural versions. To accommodate this variety, we design the individual phases of computation to be flexible in terms of the number of steps they implement. This flexibility allows us to match computational granularities to particular GPU processors.

We implement this flexibility using meta-programming techniques, i.e., we use the compiler's preprocessing features to expand and generate code based upon architectural configurations and sorting types. This allows us to write a single implementation that the compiler can unroll into specialized assemblies that are well-tuned for specifically targeted problems and hardware. Our single sorting implementation scales well across all generations of programmable NVIDIA GPUs.

Performance. We demonstrate a radix sorting approach that is substantially faster than previously published techniques for sorting large sequences of fixed-size, numeric keys. We consider the sorting algorithms described by Satish et al. [5,6,7] to be representative of the current state-of-the-art in GPU sorting. In comparison, our work demonstrates multiple factors of speedup of up to 3.8x for all fully-programmable generations of NVIDIA GPUs. For GF100-based GPUs, we achieve sustained sorting rates in excess of 1 billion keys per second.

Recent work has made apples-to-oranges comparisons of large-problem sorting between GPU and many/multi-core CPU processors [8,7,9]. We revisit these same comparisons between NVIDIA and Intel architectures with our implementation and show: G80s to outperform Core-2 quad-cores, G92s to outperform Core i7 quad-cores, and GT200s to outperform 32-core Knight's Ferry processors (KNF-MIC). Comparing the most powerful processors from both genres, the NVIDIA GF100 architecture yields a 1.8x speedup over the results presented for the Intel KNF-MIC[9].

In addition, we describe a novel *early-exit* optimization that significantly improves sorting performance for inputs with banded key diversity. When a sorting pass detects that all keys have the same digit at the same digit-place, the pass for that digit-place is short-circuited, reducing the cost of that pass by 83%. This makes our implementation suitable for even low-degree binning problems where sorting would normally be overkill.

1.2. *Organization of the Paper*

Section 2 briefly motivates GPU sorting applications, overviews the GPU machine and programming models, discusses the corresponding challenges for dynamic parallelism, and reviews the radix sorting method. Section 3 describes our design and performance

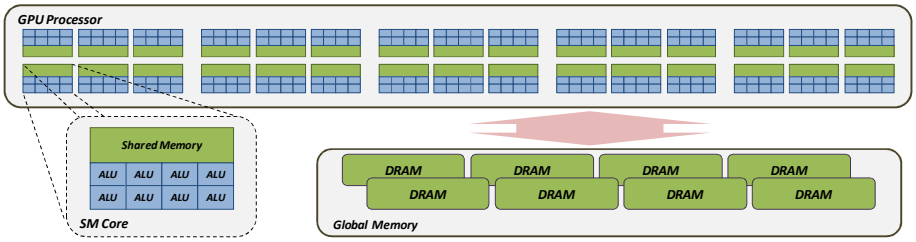

Fig. 1. A typical GPU organization comprised of 30 SM cores, each having 8 SIMD ALUs and a local shared memory space. Globally-visible DRAM memory is off-chip.

model, providing an overview of our parallel scan primitive and the details for how we extend it to provide stable digit-sorting functionality. Section 4 presents our performance evaluations. Section 5 discusses other GPU sorting efforts, further challenges faced by the GPU programming model, and concludes.

2. Background

2.1. *GPU Sorting Applications*

Sorting is germane to many problems in computer science. As an algorithmic primitive, sorting facilitates many problems including binary search, finding the closest pair, determining element uniqueness, finding the k^{th} largest element, and identifying outliers[10,11]. For large-scale problems on distributed memory systems (e.g., graph algorithms for clusters and supercomputers [12]), sorting plays an important role in improving communication efficiency by coalescing messages between nodes.

Recent literature has demonstrated many applications of GPU sorting. Sorting is a procedural step during the construction of acceleration data-structures, such as octrees [13], KD-trees [14], and bounding volume hierarchies [15]. These structures are often used when modeling physical systems, e.g., molecular dynamics, ray tracing, collision detection, visibility culling, photon mapping, point cloud modeling, particle-based fluid simulation, n-body systems, etc. GPU sorting has found many applications in image rendering, including shadow and transparency modeling[16], Reyes rendering [17], volume rendering via ray-casting [18], particle rendering and animation [19,20], ray tracing [21], and texture compression [22]. GPU sorting has also been demonstrated for parallel hashing [23], database acceleration [24,25], data mining [26], and game engine AI [27].

2.2. *Stream Machine and Programming Models*

The GPU is capable of efficiently executing large quantities of concurrent, ultra-fine-grained tasks. It is often classified as SPMD (single program, multiple data) in that many hardware-scheduled execution contexts, or *threads*, run copies of the same imperative program, or *kernel*.

As depicted in Fig. 1, the typical GPU processor organization entails a collection of cores (*stream multiprocessors*, or SMs), each of which is comprised of homogeneous processing elements (i.e., ALUs). These SM cores employ local SIMD (single instruction, multiple data) techniques in which a single instruction stream is executed by a fixed-size grouping of threads called a *warp*. Similar to symmetric multithreading (SMT) techniques, each SM contains only enough ALUs to actively execute one or two warps, yet maintains and schedules amongst the execution contexts of many warps in order to mask memory latencies and pipeline hazards. This translates into tens of warp contexts per core, and tens-of-thousands of thread contexts per GPU microprocessor.

Language-level constructs for thread-grouping are provided to facilitate logical problem decomposition in a manner that is convenient for mapping blocks of virtual threads onto physical SM cores. A two-level grouping hierarchy is used for programming a single microprocessor. A *threadblock* is a group of threads that will be co-located on the same SM core and share a local memory space. A *grid* is a collection of homogeneous threadblocks, encapsulating all of the virtual threads for a given kernel.

The program running on the host CPU orchestrates a sequential *stream* of global data flow by repeatedly invoking new kernel instances, each of which is initially presented with a consistent view of the results from the previous.

2.3. *Challenges for dynamic parallelism*

Philosophy of problem decomposition. The GPU machine model is designed for *stencil-oriented* problem decompositions. Concurrent tasks are logically defined by the output elements they produce, and kernel programs statically encode input and output locations as a function of thread rank (independent of the computation of other tasks). This approach works well for data-independent problems such as scalar transformations and matrix operations.

This output-oriented focus can pose difficulties for problems with global input dependences. If the dependences within the flow of computation can be regularly defined, some of these problems can be optimally implemented via repeated stencil applications. For example, global reduction can be implemented by executing a logarithmic number of kernels performing local pairwise reduction.

For other globally-dependent problems such as sorting and graph traversal, the straightforward application of stencil kernels results in inferior work complexity. The repeated application of regular swapping stencils is only suitable for implementing sorting networks such as bubble and bitonic sort, which respectively exhibit $O(n^2)$ and $O(n\log_2^2 n)$ work complexity. Stencil-based graph traversal kernels must inspect every vertex at each timestep and exhibit $O(V^2)$ complexity, whereas $O(V+E)$ is optimal. Further still, computations that have no regular dependence structure (e.g., recursive search space exploration) cannot be solved via repeated stencil application at all.

Instead, a decomposition that is oriented towards input items is needed to implement work-optimal solutions for these problems. From the thread perspective, we want to logically associate tasks with specific elements in the input stream. The inputs drive the

execution of these tasks, which may produce zero or more outputs (potentially in different locations). In the context of partitioning-based sorting methods, each thread gathers its key, determines which partition that key belongs, and then must cooperate with other threads to determine where the key should be relocated. For efficient graph traversal and search space exploration, threads must enqueue variable numbers of unvisited items to explore.

By shifting the focus to input items, these problems can all be reduced to the problem of cooperative allocation (*allocation-oriented design*). Unfortunately, traditional mechanisms for mediating allocation contention (e.g., atomic instructions) do not align well with the latency-tolerant, high-throughput philosophy of GPU computing. Because GPU memory accesses to shared-memory spaces have comparatively longer latencies, their serialization leads to significant underutilization and is particularly detrimental to performance. The challenge for these dynamic problems becomes one of designing bulk-synchronous strategies for efficient resource allocation.

Performance pitfalls. GPU architectures are designed for maximum throughput of regularly-structured computation. When the computation becomes dynamic and varied, mismatches with architectural considerations can result in severe performance penalties. Performance is significantly affected by irregular memory access patterns that cannot be coalesced or result in bank conflicts, by control flow divergences between SIMD warp threads that result in thread serialization, and by load imbalances between barrier synchronization points that result in resource underutilization [1].

2.4. *The Radix Sorting Method*

Radix sorting is currently the fastest approach for sorting 32- and 64-bit keys on both CPU and GPU processors [7]. The method relies upon a positional representation for keys, i.e., each key is comprised of an ordered sequence of numeral symbols (i.e., *digits*) specified from least-significant to most-significant. For a given input sequence of keys and a set of rules specifying a total ordering of the symbolic alphabet, the radix sorting method produces a lexicographic ordering of those keys.

The process works by iterating over the digit-places from least-significant to most-significant. For each digit-place, the method performs a stable *distribution sort* of the keys based upon their digit at that digit-place in order to partition the keys into radix r distinct buckets. Given an n-element sequence of k-bit keys, d-bit digits, and $r = 2^d$, a radix sort of these keys will require k/d passes of distribution sorting.

The asymptotic work complexity of the distribution sort is $O(n)$ because each of the n input items needs comparing with only a fixed number of radix digits. With a fixed number of digit-places, the entire radix sorting process is also $O(n)$. When a key is relocated, its global relocation offset is computed as the number of keys with "lower" digits at that digit place plus the number of keys having the same digit, yet occurring earlier in the sequence.

Radix sorting has a variable granularity of computation, i.e., it can trade redundant computation for fewer I/O operations. Increasing d (the number of bits per radix-digit)

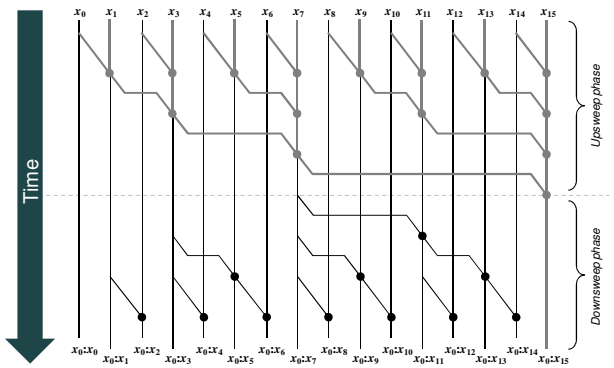

Fig. 2. Flow of computation in the Brent-Kung style for parallel prefix scan, illustrated for n = 16 inputs. Solid lines indicate the flow of data, and circular pads indicate ⊕ operators.

decreases the total number of digit-place passes that need iterating over. However, the number of bins *r* scales exponentially with *d*. This implies that linear decreases in I/O will result in super-linear increases in dynamic instruction counts and local storage requirements. For general processor workloads, we see this phenomenon reflected in the "power law of cache misses", which suggests that the influence of local storage (cache size) upon off-chip memory I/O (cache miss rate) follows a power curve in which squaring the local storage typically only results in halving the memory workload [28].

2.5. *Parallel Radix Sorting*

The fundamental component of the radix sorting method is the distribution sorting pass in which *n* keys are scattered into *r* bins. Because the key distribution is unknown, the sizes and arrangement of these bins in shared memory must be dynamically determined. There are two strategies for constructing bins: (1) using blocks that are allocated online and linked with pointers; and (2) contiguous allocation in which offsets and lengths are computed *a-priori* using a parallel prefix scan algorithm. Most research is focused on the latter: the ability to perform contention-based allocation is non-existent or severely expensive on many parallel architectures (e.g., vector, array, and GPU processors), and traversing linked structures can carry stiff performance penalties.

Prefix scan is a higher-order function that takes as inputs an *n*-element list [x_0, …, x_{n-1}], a binary associative combing operator ⊕, and produces an equivalently-sized output list [y_0, …, y_{n-1}] where each y_i is the ⊕-reduction of the input elements x_0 through x_{i-1}. By using the addition operator, we can use prefix scan to compute a vector of allocation offsets from a vector of length requests.

Parallel implementations of prefix scan typically follow the Brent-Kung family of prefix circuits[29]. As depicted in Fig. 2, the flow of computation exhibits a recursive "hourglass" shape with an *upsweep* phase composed of reduction operators followed by a *downsweep* phase of scan operators. The reduction operators serve to seed their

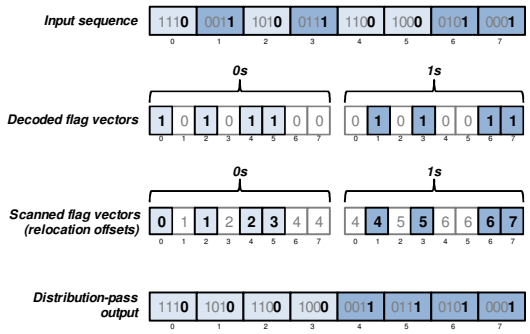

Fig. 3. The traditional split operation: a decoding step combined with prefix scan reveals the scatter offsets required to enact a radix $r = 2$ distribution sort on the first digit-place of an input sequence.

corresponding scan operators with partial reductions (i.e., base offsets). The total number of operators is a linear $O(2n)$.

In its simplest form, a distribution sort can be implemented using a binary *split* primitive [30] comprised of two prefix scans over two n-element binary encoded flag vectors: the first initialized with 1s for keys whose digit was 0, the second to 1s for keys whose digit was 1. The two scan operations are dependent: the scan of the 1s vector can be seeded with the number of zeros from the 0s scan. After the scans, the i^{th} element in the appropriate flag vector will indicate the relocation offset for the i^{th} key. An alternative is to perform one large scan over the concatenation of the two vectors, as shown in Fig. 3.

As described in Table 1, a naïve GPGPU distribution sort implementation can be constructed by simply invoking a parallel prefix scan primitive between separate *decoding* and *scatter* kernels. The decoding kernel would be used to create a concatenated flag vector of rn elements in global memory. After scanning, the scatter kernel would redistribute the keys (and values) according to the scan results. This approach suffers from an excessive memory workload that scales with $2^d/d$. As a consequence of this dependency, the overall memory workload will be minimized when the number of radix digit bits $d = 1$. This provides little flexibility for tuning the sorting granularity to minimize and overlap the workloads for I/O and computation.

As an alternative, practical sorting implementations have used a histogram-based strategy [31,32]. For typical parallel machines, the number of parallel processors $p << n$. This makes it natural to distribute the input sequence amongst processors. Using local resources, each processor can compute an r-element histogram of digit-counts. By only sharing these histograms, the global storage requirements are significantly reduced. A single prefix scan of these histograms provides each processor with the base digit-offsets for its block of keys. These offsets can then be applied to the local key rankings within the block to distribute the keys.

Prior GPU radix sort implementations use this approach, treating each threadblock as a logical processor operating over a fixed-size block of b keys[33,34,5]. The procedure

Table 1. Procedures for distribution sorting, described as sequences of kernel launches. The I/O models of memory workloads are specified in terms of d-bit radix digits, radix $r = 2^d$, local block size of b keys, and an n-element input sequence of k-bit keys.

Naïve GPU distribution sort		
Kernel	***Read I/O Workload***	***Write I/O Workload***
1. Decode keys and compute flag vectors	n keys	nr counts
2. Flag scan: upsweep reduction	nr counts	(insignificant)
3. Flag scan: top-level scan	(insignificant)	(insignificant)
4. Flag scan: downsweep scan	nr counts + (insignificant)	nr offsets
5. Scatter keys to appropriate bin	nr offsets + n keys	n keys
	Total I/O for all k/d passes:	**(k/d) $(5n(2^d) + 3n)$**
GPU histogram-based distribution sort [5,7]		
Kernel	***Read I/O Workload***	***Write I/O Workload***
1. Locally sort blocks at current digit-place into digit-segments	n keys	n keys
2. Compute block histograms of digit counts	n keys	nr/b counts
3. Histogram scan: upsweep reduction	nr/b counts	(insignificant)
4. Histogram scan: top-level scan	(insignificant)	(insignificant)
5. Histogram scan: downsweep scan	nr/b counts + (insignificant)	nr/b offsets
6. Scatter sorted digit-segments of keys to appropriate bin	nr/b offsets + n keys	n keys
	Total I/O for all k/d passes:	**(k/d) $(5n(2^d)/b + 7n)$**
Our GPU allocation-oriented distribution sort		
Kernel	***Read I/O Workload***	***Write I/O Workload***
1. Allocation scan: upsweep reduction (locally decode and reduce flag counts)	n keys	(insignificant)
2. Allocation scan: top-level scan of flag counts	(insignificant)	(insignificant)
3. Allocation scan: downsweep scan (locally decode and scan flag counts, scatter keys)	n keys + (insignificant)	n keys
	Total I/O for all k/d passes:	**(k/d) $(3n)$**

of Satish et al. is representative of this approach, and is reviewed in Table 1. Because of the decision to keep b constant, the number of threadblock "processors" grows with problem size and the overall memory workload still scales exponentially with d, although significantly reduced by common block-sizes of 128-1024 keys. This elicits a global minimum in which there exists an optimal d to produce a minimal memory overhead for a given block size. For example, when block size $b = 512$ and key size $k = 32$, a radix digit size $d = 8$ provides minimal memory overhead. As a point of comparison, their implementation imposes an explicit I/O workload of 56.6 words per key (where words and keys are 32-bits, $d = 4$ bits, and $b = 1024$ keys). As we describe in the next section, our approach only moves 24 words per key for similar problems.

3. Our Radix Sorting Strategy

Our radix sorting strategy strives to obtain maximal overall system utilization for a given target architecture. Our design strategy consists of:

(i) Designing kernel stages that reduce the aggregate memory workload
(ii) Implementing distribution sorting kernels with a flexible radix sorting granularity *d*.
(iii) Specializing local algorithms (e.g., tile sizes for blocking keys) to maximize SM resource usage.
(iv) Using an analytical performance model to select a granularity *d* that minimizes the aggregate workload over all digit passes

3.1. *Reducing Aggregate Memory Workload*

We can reduce the aggregate memory workload for a distribution sorting pass using two techniques: (1) *kernel fusion* and (2) *threadblock serialization*.

Kernel Fusion. Kernel fusion is a particularly effective technique for reducing aggregate memory workload by co-locating sequential steps in the stream pipeline within a single kernel. When data-parallel steps can be combined, the intermediate results can be passed from one step to the next in local registers or shared memory instead of through global, off-chip memory. As detailed in Table 1, our approach collapses the five kernels from the naïve approach into three.

We implement kernel fusion by inserting our own digit-decoding and key-scattering logic directly into the kernels for prefix scan. The flag values obtained by a thread when decoding a given key can be passed directly via registers to upsweep or downsweep scan logic. Similarly, the ranking results computed by downsweep scan can be locally conveyed to scattering logic for relocating keys and values. Additionally, the keys themselves need not be re-read from global memory for scattering; they were obtained earlier by the downsweep decoding logic within the same kernel closure.

The overall amount of memory traffic is dramatically reduced because we obviate the need to move flags through global device memory. The elimination of the corresponding load/store instructions also increases the computational efficiency, further allowing our algorithm to exploit those resources. Instead of simply serving as procedural subroutine, prefix scan becomes "runtime" that drives the entire distribution sorting pass.

Threadblock Serialization. As illustrated in Fig. 4, a fully-recursive Brent-Kung scan computation has a height of $O(2\log_b n)$ kernel launches when each threadblock is assigned to a fixed-size block of *b* elements. This is currently the prevailing strategy for GPU prefix scan [35,36].

Our distribution sorting kernels, however, are derived from prior work regarding efficient, *two-level reduce-then-scan* GPU prefix scan primitives [37]. As illustrated in Fig. 5, we compose scan using only three kernels regardless of problem size: an upsweep reduction, a top-level scan, and a downsweep scan. Instead of allocating a unique thread for every input key, the bottom-level kernels dispatch a fixed-size grid of *C* threadblocks

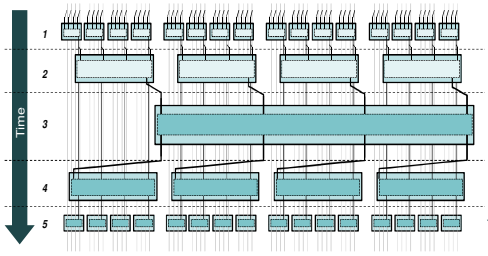

Fig. 4. A typical recursive threadblock decomposition for prefix scan. In this example with block size b=4 keys, the scan requires five kernel launches: two upsweep reduction kernels (light blue), and three downsweep scan kernels (dark blue).

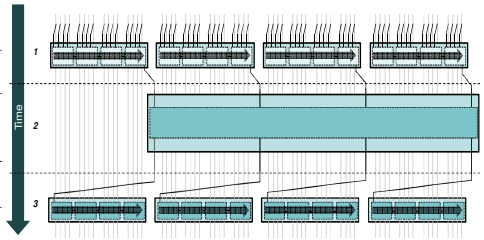

Fig. 5. A fixed, two-level threadblock decomposition for prefix scan requires only three kernel launches, regardless of input size. In this example, we reuse C=4 reduction threadblocks (light blue) and scan threadblocks (dark blue) are to process multiple tiles of b=4 keys

in which threads are re-used to process the input in successive "tiles" of b keys each. Partial reductions are accumulated in thread-private registers as tiles are processed serially.

There are several important benefits to restricting the amount of parallel work:

- There is typically not enough work required of the interior kernels of the "hourglass" to saturate the processor during their execution. As the least-efficient phase of computation, we want to dispose of this work as quickly as possible. In the recursive approach, the total number of interior blocks scales with $O(n/b)$ and requires $O(\log_b n)$ kernel launches. Alternatively, our approach requires only a single kernel launch to dispatch of a small, constant $O(rC)$ amount of interior work.
- Currying partial reductions by leaving them in registers is free. This elides $O(n/b)$ global memory reads and writes at a savings of 2-4 instructions per round-trip obviated (offset calculations, load, store).
- Computations common to each block of keys can be hoisted, computed once, and reused. This includes shared memory initialization, the calculation of common offsets for shared and global memory, and predicates (particularly those tested against thread rank).

3.2. *Flexible Computational Granularity*

Prefix scan is a very memory-bound operation [37], affording us a "bubble of opportunity" to fuse in sorting logic and ratchet up the computational granularity with virtually no overhead. In this subsection, we illustrate this bubble and describe how we implement a flexible radix sorting granularity d to leverage it.

3.2.1. *Bubbles of free computation*

One of the attractive facets of GPU latency-hiding is that the hardware easily manages the overlap of I/O and computation, freeing the programmer from the burden of

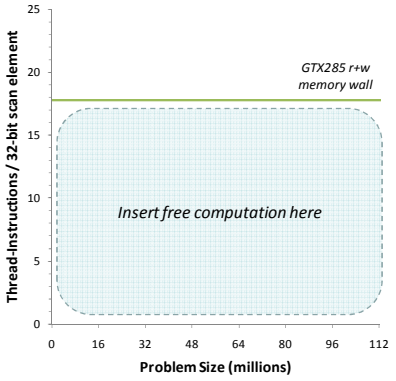

Fig. 6. At an ideal 354 x10^9 thread-instructions/s and 159 x10^9 bytes/s, the GTX285 can overlap 17.8 instructions with every two words of memory traffic.

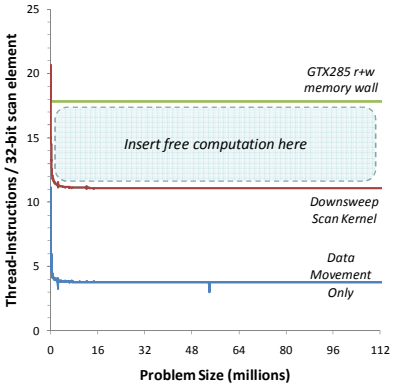

Fig. 7. Free cycles within a downsweep scan kernel that moves two words while executing 4 data movement and 8 local scan instructions for each input element [37].

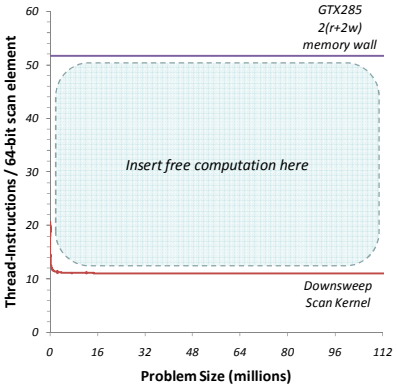

Fig. 8. Free cycles within a downsweep sorting scan/scatter kernel that reads two words, writes two words, and has a write partial-coalescing penalty of two words.

implementing complicated pre-fetching and software pipelining techniques. The over-threaded nature of GPU architecture provides predictable and usable cycles for extra computation within I/O-bound kernels.

Fig. 6 depicts the bubble of free-computation below the GTX-285 *memory wall*, i.e., the ideal 17.8 thread-cycles that can be executed per 32-bit word copied in and out by the memory subsystem. For the same memory workload, Fig. 7 shows the bubble for a prefix scan downsweep scan kernel after accounting for data movement and scan instructions [37].

As shown in Fig. 8, the ideal bubble is tripled for a downsweep scan kernel with a memory workload sized to distribute key-value pairs. It must read a pair of two words,

write a pair of two words, and pay a partial coalescing penalty of two words. (As we discuss in Section 4, key-scattering produces up to twice as much memory traffic due to partial coalescing. Additionally, the bubble is even larger in practice due to the slightly lower achievable bandwidth rates.)

The result is a rather large window that not only allows us to construct distribution sorting *inside* of prefix scan, but to be more flexible with the granularity of sorting computation as well.

3.2.2. *Multi-scan*

We have generalized our prefix scan implementation for *multi-scan*, i.e., to compute multiple, dependent scans concurrently in a single pass. This allows us to efficiently compute the prefix sums of radix $r > 1$ flag vectors without imposing any significant additional workload upon the memory subsystem. Our three multi-scan kernels listed in Table 1 operate as follows:

(i) ***Upsweep reduction***. For a multi-scan distribution sorting pass, the upsweep reduction kernel reduces n inputs into rC partial reductions. In our implementation, the reduction threads employ a *loop-raking* strategy [38] in which each thread accumulates flags from consecutive tiles, similar to Harris et al. [39]. For each tile, a thread gathers its key, decodes the digit at the current digit-place, and increments the appropriate flag (kept in private registers). After processing their last tile, the threads within each threadblock cooperatively reduce these private flags into r partial reductions, which are then written out to global device memory in preparation for the top-level scan.

(ii) ***Top-level scan***. The single-threadblock, top-level scan serves to scan the partial reduction contributions from each of the C bottom-level threadblocks. Continuing our theme of multiple, concurrent scans, we have generalized it to scan a concatenation of rC partial reductions.

(iii) ***Downsweep scan/scatter***. In the downsweep scan/scatter kernel, threadblocks perform independent scans of their tile sequence, seeded with the partial sums computed by the top-level scan. For each tile, threads re-read their keys, re-decode them into local digit flags, and then scan these flags using the local multi-scan strategy described in the next subsection. The result is a set of r prefix sums for each key that are used to scatter the keys to their appropriate bins. This scatter logic is also responsible for loading and similarly redistributing any paired satellite values. The r aggregate counts for each digit are serially curried into the next b-sized tile.

As shown in Table 1, only a constant number of memory accesses are used for the storage of intermediate results, and there are no longer any coefficients that are exponential in terms of the number of radix digit bits d. This implies that memory workload will monotonically decrease with increasing d, positioning our strategy to take

advantage of any additional computational power that may allow us to increase d in the future. Our strategy can operate with a radix digit size $d \leq 4$ bits on current NVIDIA GPUs before exponentially-growing demands on local storage prevent us from saturating the device. With $d = 4$ and $k = 32$-bit keys-only sorting, our algorithm requires the memory subsystem to explicitly process only 24 words per key.

3.3. *Specializing Local Multi-scan Algorithms*

In this subsection, we describe our downsweep scan/scatter kernel in more detail to illustrate the different phases of local scan and the degrees of freedom by which we can parameterize it to best fit the resources provided by the underlying hardware.

3.3.1. *Downsweep kernel operation*

Fig. 9 illustrates this computation from the point of a single threadblock processing a particular tile of input values.

(i) ***Key decoding.*** Threads within the decoding logic collectively read b keys, decode them according to the current digit-place, and create the private-register equivalent of r flag vectors of b elements each.

(ii) ***Local multi-scan.*** The scan logic is replicated r-times, ultimately producing r vectors of b prefix sums each: one for each of the r possible digits. It is implemented as a flexible hierarchy of reduce-then-scan strategies composed of three phases of upsweep/downsweep operation:

   (a) *Thread-independent* processing in registers, shown in blue. This phase serves to transition the problem from the tile size b into a smaller version that will fit into shared memory and back again. This provides flexibility in terms of making maximal use of the available register file and for facilitating different memory transaction sizes (e.g., 1/2/4-element vector load/stores), all without impacting the size of the shared-memory allocation.

   (b) *Inter-warp cooperation*, shown in orange. In this phase, the threads within a single warp independently reduce/scan though the partial reductions placed in shared memory by the other warps. This serial raking process transitions the problem size into one that can be cooperatively processed by a single warp and back again, and is similar to the scan techniques described by Dotsenko et al. [36]. This phase provides flexibility in terms of facilitating shared memory allocations of different sizes, supporting alternative SIMD warp sizes, and accommodating arbitrary numbers of warps per threadblock. For example, we double the GT200 tile size for the newer GF100 architecture because of the increased amount of shared memory per SM core

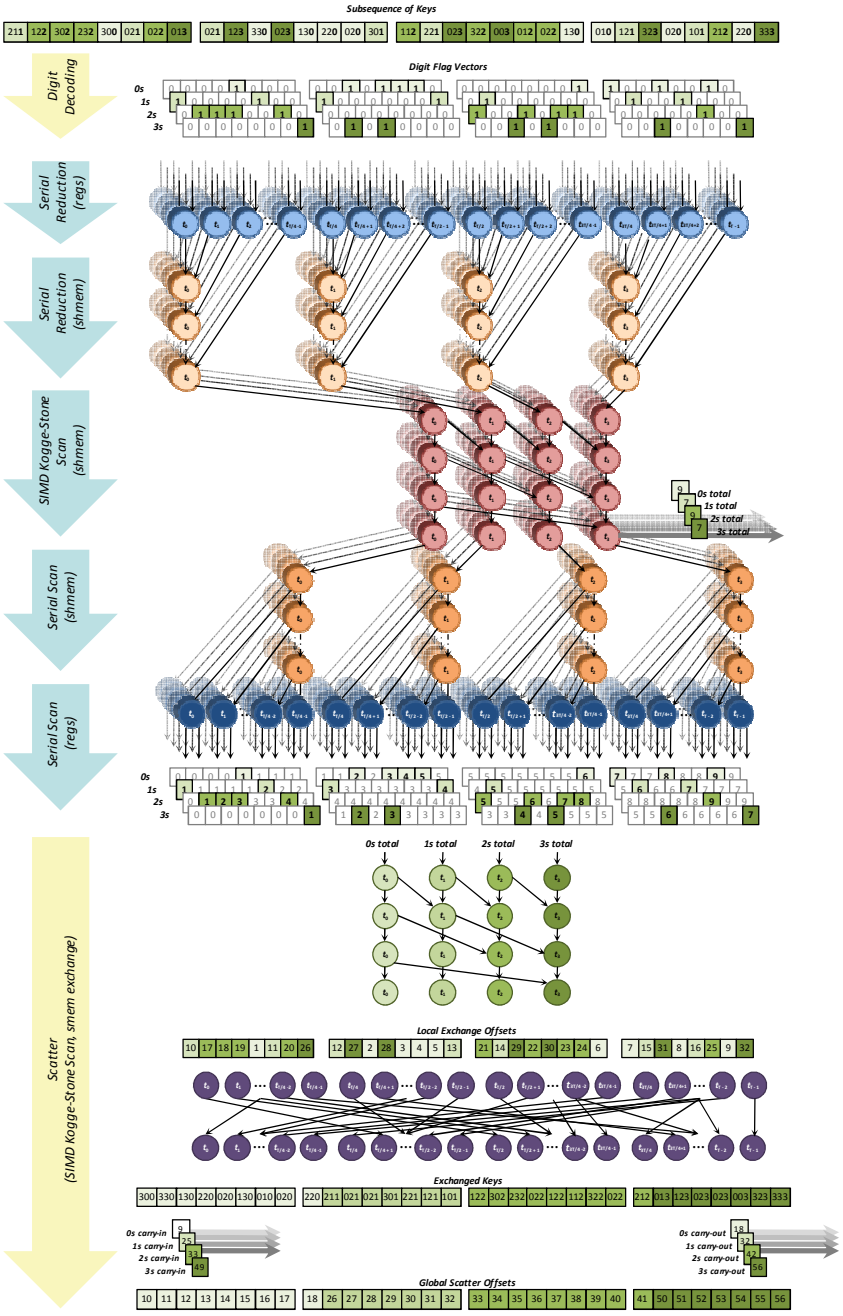

Fig. 9. The operation of a downsweep multi-scan CTA that incorporates fused binning and scatter logic. Computation and time flow downward for an input block size of $b = 32$ keys, a radix $r = 4$ digits, and a warp-size $w = 4$ threads. The scan stages are labeled in light blue, the fused stages in yellow. Circles indicate the assignment of a given thread t_i to a binary addition task. Flag vector encoding is not shown. The blue thread-independent processing phase is shown to perform vector-2 loads/stores.

(c) *Intra-warp cooperation*, shown in red. For a given warp-size of w threads, the intra-warp phase implements $\log_2 w$ steps of a Kogge-Stone scan [40] in a synchronization-free SIMD fashion as per [35]. The r running digit totals from the previous tile are carried into this SIMD warpscan, incorporated into the prefix sums of the current tile's elements, and new running totals are carried out for the next tile, all in local shared memory.

(iii) ***Key scattering.*** The scatter operation is provided with the tile of keys, their local ranks/prefix sums, the tile's digit totals, and the incoming running digit totals. Although each scatter thread could use this information to distribute the same keys that it obtained during decoding, doing so would result in poor write coherence. Instead we implement a key exchange. We use the local ranks to scatter keys into a pool of local shared memory, repurposing the raking storage. Then consecutive threads can acquire consecutive keys and scatter them to global device memory with a minimal number of memory transactions. We compute a SIMD prefix sum of the local digit totals in order to determine the locations of each of the r segments of newly-coherent keys within this pool.

Although our two-phase scatter procedure is fairly expensive in terms of dynamic instruction overhead and arbitrary bank conflicts, it is much more efficient than the sorting phase implemented by [5]. Their sorting phase performs d iterations of binary-split, exchanging keys (and values) d times within shared memory, whereas our approach only exchanges them once.

3.3.2. *Occupancy concerns*

Radix sorting exhibits a fundamental tradeoff with regard to tile size b versus SM core occupancy. Both the tile size and the number of radix digits r will determine the local storage requirements (e.g., register and shared memory allocations) for a given threadblock. Increasing the tile size has two effects: it increases the write coherence for scattering keys; and it comparatively lowers the relative overheads from the work-inefficient portions of our local scan. However, too large a tile size will prevent the SM cores from being occupied by enough threadblocks to cover shared-memory latencies when only raking warps are active. This performance cliff occurs at different tile sizes for different architectures, and is dependent upon register and shared memory availability and pipeline depths.

3.3.3. *Optimizations*

Our implementation incorporates two important optimizations for improving the efficiency and utility of the radix sorting method.

Composite, bitwise-parallel scan. In order to increase the computational efficiency of our implementation, we employ a method for encoding multiple binary-valued flag

vectors into a single, composite representation [41]. By using the otherwise unused high-order bits of the flag words and the bitwise parallelism of addition, our *composite scan* technique allows us to compute several logical scan tasks while only incurring the cost of a single parallel scan.

For example, by breaking a tile of keys into *subtiles* of 256-element multi-scans, the scan logic can encode up to four digit flags within a single 32-bit word, with one byte used for each logical scan. The bit-wise parallelism of 32-bit addition allows us to effectively process four radix digits with a single composite scan. To process the sixteen logical arrays of partial flag sums when $d = 4$ bits, we therefore only need local storage for 4 scan vectors. For GT200, the minimum occupancy requirements and local storage allocations allow us to unroll two subtiles per tile. For GF100 processors, we unroll four subtiles per tile.

Early exit. In many sorting scenarios, the input keys reflect a *banded* distribution. For example, the upper bits are often all zero in many integer sorting problems. Similarly the sign and exponent bits may be homogenous for floating point problems. If this is known *a priori*, the sorting passes for these corresponding digit-places can be explicitly skipped. Unfortunately this knowledge may not be available for a given sorting problem or there may be abstraction layers that prevent application-level code from being aware that a radix-based sorting method is being used.

To provide the benefits of fewer passes for banded keys in a completely transparent fashion, we implement a novel, *early-exit* decision check at the beginning of the downsweep kernel in each distribution sorting pass. By inspecting the partition offsets output by the top-level scan kernel, the downsweep threadblocks can determine if all keys have the same bits at the current digit place. If there are no partition offsets within the range $[1, n-1]$, the downsweep kernel is able to terminate early, leaving the keys in place.

Some passes cannot be short-circuited, however. When sorting signed or floating point keys, the first and last passes must be executed in full, if just to perform the "bit-twiddling" [5] that is necessary for these types to be radix-sorted.

3.4. *Analytical Model*

The computational workload for distribution sorting passes can be decomposed into two portions: work that scales directly with the size of the input (i.e., moving and decoding keys); and work that scales proportionally with the size of the input multiplied by the number of radix digits (i.e., the r concurrent scans). Because the computational overhead of the downsweep scan kernel dominates that of the upsweep reduction kernel, we base our granularity decisions upon modeling the former.

We model downsweep kernel work in terms of the following tasks: (1) data-movement to/from global memory; (2) digit inspection and encoding of flag vectors in shared memory; (3) shared-memory scanning; (4) decoding local rank from shared memory; (5) and locally exchanging keys and values prior to scatter. For a given key-value pair, each task incurs a fixed cost α in terms of thread-instructions. The flag-

encoding and scanning operations will also incur a per-pair cost of β instructions per composite scan.

$$instrs_{\text{scankernel}}(n,r) = n\left(\alpha_{\text{mem}} + \alpha_{\text{encflags}} + \alpha_{\text{scan}} + \alpha_{\text{decflags}} + \alpha_{\text{exchange}} + \frac{r}{4}\left(\beta_{\text{encflags}} + \beta_{\text{scan}}\right)\right) \quad (1)$$

We can use instrumentation to determine these coefficients for a given architecture / processor family. For example, $instrs_{\text{scankernel}}(n,r) = n\ (51.4 + 1.0r)$ for sorting pairs of 32-bit keys and values on the NVIDIA GT200 architecture. The instruction costs per pair are (not including warp-serializations):

$\alpha_{\text{mem}} = 6.3$ $\quad$ $\alpha_{\text{scan}} = 10.7$ $\quad$ $\alpha_{\text{exchange}} = 14.7$
$\alpha_{\text{encflags}} = 5.5$ $\quad$ $\alpha_{\text{decflags}} = 13.9$

The instruction costs per pair per composite scan are:

$\beta_{\text{encflags}} = 2.6$ $\quad$ $\beta_{\text{scan}} = 1.4$

Minimizing this parameterized function for GT200, the radix sorting granularity with the lowest computational overhead is $d = 4$ bits ($r = 16$ radix digits). Parameterizations for G80, G92, and GF100 architectures yield the same granularity.

Although $d = 4$ is minimal for the GTX-285, the model predicts that the downsweep kernel will still be compute-bound: the overhead per pair is 67.4 instructions, which exceeds the memory wall of 52 instructions illustrated in Fig. 8. For this specific processor, the performance will strictly be a function of compute workload. However, as increases in computational throughput continue to outpace those for memory, it is likely that the bubble of memory-boundedness will eventually provide us with room to increase d past the minimum computational workload without penalty in order to reduce overall passes.

4. Evaluation

This section presents the performance of our allocation-oriented radix sorting strategy. We present our own performance measurements for Satish et al. (CUDPP v1.1) as a reference for comparison, and contrast our sorting performance with the x86-based many-core sorting results from [8,7,9].

Configuration and methodology. Our primary test environment consisted of a Linux platform with an NVIDIA GTX-285 GPU running the CUDA 3.2 compiler and driver framework. Our analyses are derived from measurements taken over a suite of ~3,000 randomly-sized input sequences, each initialized with 32-bit keys and values sampled from a uniformly random distribution. We primarily evaluate key-value pair sorting, but also report results for keys-only sorting. Our measurements for elapsed time, dynamic

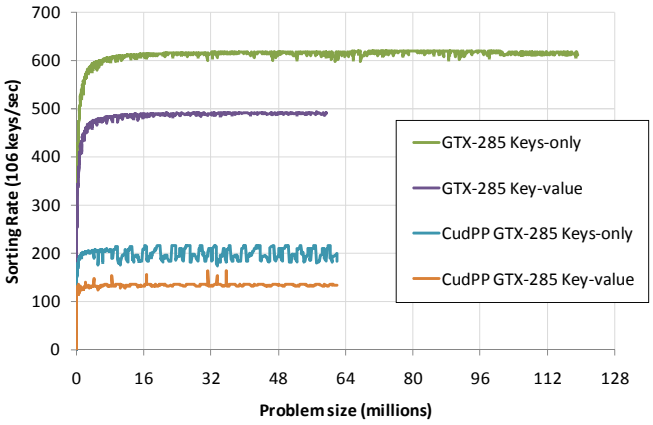

Fig. 10. Keys-only and key-value pair radix sorting rates

Table 2. Saturated 32-bit sorting rates for input sequences larger than 16M elements

Device	***Keys-only Sorting Rate*** *(10^6 32-bit keys/s)*		***Key-Value Sorting Rate*** *(10^6 32-bit pairs/s)*	
Name	*CUDPP Radix*	***Our Radix (speedup)***	*CUDPP Radix*	***Our Radix (speedup)***
NVIDIA GTX 480	349	**1005 (2.9x)**	257	**775 (3.0x)**
Tesla C2050	270	**742 (2.7x)**	200	**581 (2.9x)**
NVIDIA GTX 285	199	**615 (2.8x)**	134	**490 (3.7x)**
NVIDIA GTX 280	184	**534 (2.6x)**	117	**449 (3.8x)**
NVIDIA Tesla C1060	176	**524 (2.7x)**	111	**333 (3.0x)**
NVIDIA 9800 GTX+	111	**265 (2.0x)**	82	**189 (2.0x)**
NVIDIA 8800 GT	83	**171 (2.1x)**	63	**129 (2.1x)**
NVIDIA Quadro FX5600	66	**147 (2.2x)**	55	**110 (2.0x)**
Intel 32-core Knight's Ferry MIC [9]	560			
Intel quad-core i7 [7]	240			
Intel Q9550 quad-core [8]	138			

instruction count, warp serializations, memory transactions, etc., are taken directly from GPU hardware performance counters. Our analyses are reflective of *in situ* sorting problems: they preclude the driver overhead and the overheads of staging data to/from the accelerator, allowing us to directly contrast the individual and cumulative performance of the stream kernels involved.

Overall sorting rates. Fig. 10 plots the measured radix sorting rates exhibited by our d = 4 bits implementation and the CUDPP primitive. We observe that the radix sorting performances plateau into steady-state as the GPU's resources become saturated. In addition to exhibiting 3.7x and 2.8x speedups over the CUDPP implementation on the same device, our key-value and key-only implementations provide smoother, more

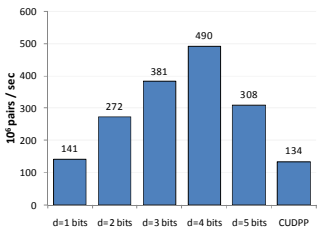

Fig. 11. Saturated sorting rates (32-bit key+value, GTX285)

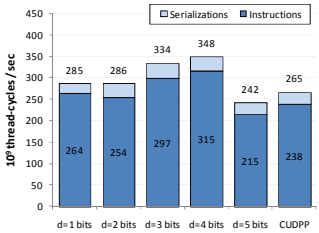

Fig. 12. Average utilized computational throughput (32-bit key+value, GTX-285).

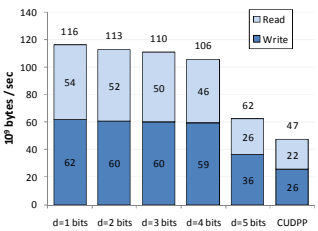

Fig. 13. Average utilized memory bandwidth (32-bit key+value, GTX-285).

consistent performance across the sampled problem sizes. Table 2 presents speedups over CUDPP of 2.0x - 3.8x for all generations of NVIDIA GPU processors.

Recent publications for this genre of sorting problems have set a precedent of comparing the sorting performances of the "best available" many-core GPU and CPU microarchitectures [8,7,9]. The saturated sorting rates on these devices for input sequences of 16M+ keys are denoted in Table 2. Using our method, all of the NVIDIA GPUs outperform the Intel Xeon Core2 Quad Q9550 CPU and, perhaps more strikingly, our sorting rates for previous-generation GT200 GPUs exhibit performance on par with or better than the unreleased Intel 32-core Knights Ferry.

Fig. 11 illustrates the effects of increasing the number of radix digit bits d (and thus decreasing the number of distribution sorting passes) for $1 \leq d \leq 5$ bits. We observe that throughput improves as d increases for $d < 5$. When $d \geq 5$, two issues conspire to impair performance, both related to the exponential growth of the $r = 2^d$ radix digits that need scanning. The first is that the cumulative computational workload is no longer decreasing with reduced passes. As predicted by the model, the two bottom-level kernels are compute-bound under this load. Continuing to increase the overall computational workload will only result in progressively larger slowdowns. The second issue is that the increased local storage requirements (i.e., registers and shared memory) prevent SM saturation: the occupancy per GPU core is reduced from 640 to 256 threads.

Resource utilization. Fig. 12 and Fig. 13 show the respective computational and I/O throughputs realized by our variants and the CUDPP implementation. The GTX-285 provides realistic maximums of 354 $x10^9$ thread-cycles/s and 136-149 $x10^9$ bytes/s, respectively. Our 3-bit and 4-bit variants achieve more than 94% of this computational throughput. The 1-bit, 2-bit, and CUDPP implementations have a mixture of compute-bound and memory-bound kernels, resulting in lower overall averages for both workloads. The 5-bit variant illustrates the effects of under-occupied SM cores: its kernels are compute-bound, yet it only utilizes 68% of the available computational throughput.

Computational workloads. Our five variants in Fig. 14 corroborate our model of computational overhead versus digit size: workload decreases with the number of passes until the cost of scanning radix digits becomes dominant at $d = 5$. This overhead is

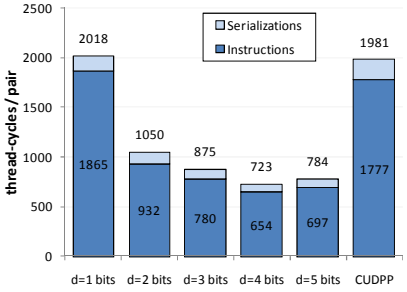

Fig. 14. Aggregate computational overhead (32-bit key+value, GTX-285).

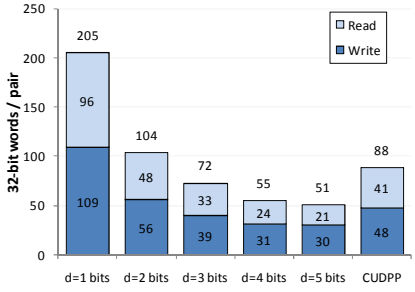

Fig. 15. Aggregate memory overhead (32-bit key+value, GTX-285).

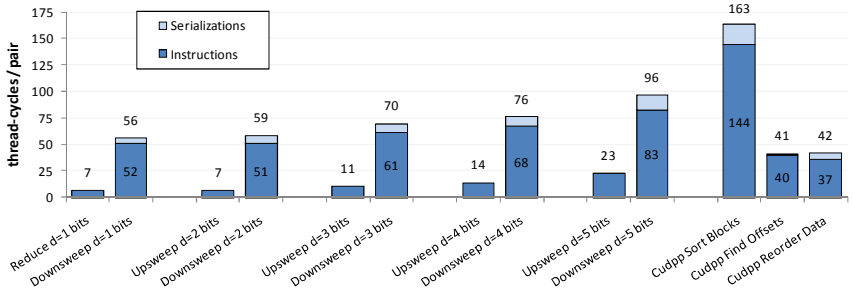

Fig. 16. Computational overhead per distribution sorting pass (32-bit key+value, GTX-285).

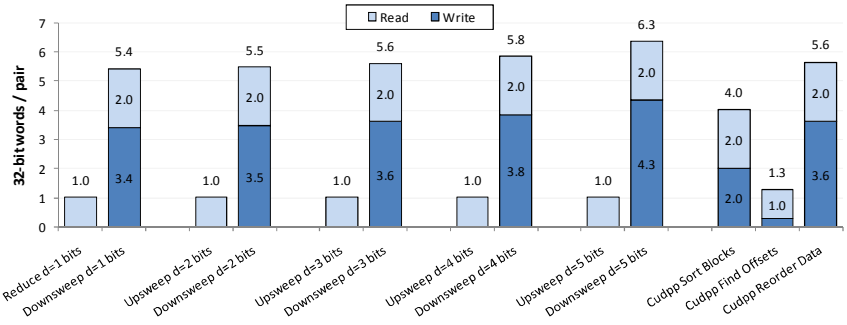

Fig. 17. Memory overhead per distribution sorting pass (32-bit key+value, GTX-285).

inclusive of the number of thread-cycles consumed by scalar instructions as well as the number of stall cycles incurred by the warp-serializations that primarily result from the random exchanges of keys and values in shared memory. The 723 thread-cycles executed per input pair by our 4-bit implementation may seem substantial, yet efficiency is 2.7x that of the CUDPP implementation.

Fig. 16 presents the computational overheads of the individual kernel invocations that comprise a distribution sorting pass. For $d > 2$, we observe that the workload deltas

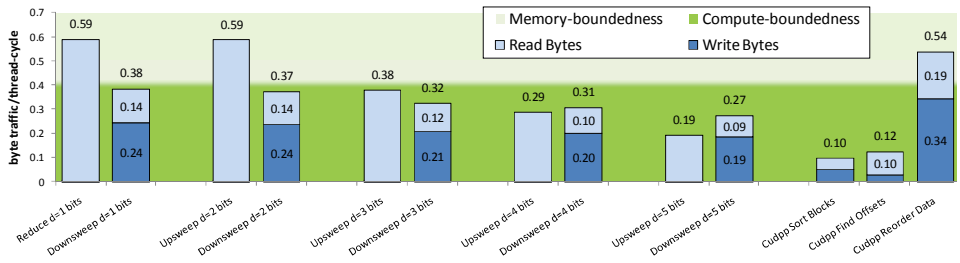

Fig. 18. Memory vs. compute workload ratios for individual stream kernels. The two-tone backdrop demarcates the memory wall for the GTX-285 (i.e., 0.45 bytes/cycle), illustrating the degree to which kernels are memory-bound or compute-bound.

between scan kernels double as d is incremented, scaling with r as expected and validating our model of instruction overhead. Our 1-bit and 2-bit variants do not follow this parameterized model: the optimizing compiler produces different code for them because flag vector encoding requires only one composite scan.

Memory workloads. The overall memory workloads for these implementations are shown in Fig. 15. We confirm that our memory overhead monotonically decreases with increasing d. As predicted by our I/O model, the memory workload of the CUDPP implementation (4-bit digits) is 1.6x that of our 4-bit variant.

Fig. 17 illustrates the memory overheads for a single distribution sorting pass, broken down by kernel and type. Although our scatter instructions logically write two 32-bit words per pair, we observe that the hardware issues separate transactions when threads within the same half-warp scatter keys to different memory segments. Extra memory bandwidth is used when this occurs: the memory subsystem rounds up to a full-sized transaction (e.g., 32B / 64B / 128B), even though only a portion of it may contain actual data.

On the GT200 architecture, this extra traffic scales with the number of partitions r, starting at 70% overhead when $r = 2$. For $r = 16$, this overhead exceeds our explicit I/O model by 28%, whereas the CUDPP implementation incurs only a 22% cumulative increase.

GPU processors are less efficient when consecutive kernels oscillate between being memory-bound and being compute-bound. Fig. 18 illustrates the corresponding workload ratios for each of the stream kernels relative to the GTX-285 memory wall. We see that for $d > 2$, our distribution sorting streams only contain compute-bound kernels. The CUDPP implementation contains a mixture of memory-bound and extremely compute-bound kernels, resulting in an overall underutilization of hardware

Key diversity. The distribution of key-bits can have an impact on the overall sorting performance. The traditional expectation for radix sorting is that a perfectly uniform distribution of key bits will yield worst-case performance because it produces the highest number of fragmented scatter transactions. As the bits become less uniform and certain digits become more likely than others, the expected number of memory segments touched

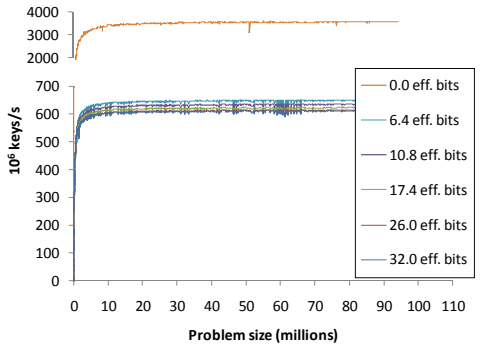

Fig. 19. Sorting performance with varying key entropy (32-bit keys-only, GTX-285)

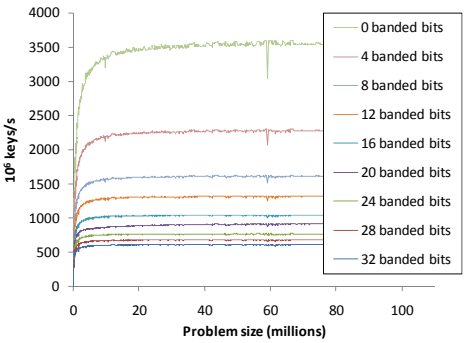

Fig. 20. Sorting performance for key distributions with banded ranges (32-bit keys-only, GTX-285).

per SIMD-write decreases. The increased efficiency typically results in marginal performance increases.

Fig. 19 confirms that our implementation's performance improves as keys become less random. We evaluate this by using a technique for generating key sequences with different "entropy" levels[42]. The idea is to generate keys that trend towards having more 0s than 1s bits by repeatedly bitwise-ANDing together uniformly-random bits. For each bitwise-AND, the number of 1s bits per key decreases. An infinite number of bitwise-AND iterations results in a completely uniform key-distribution of all 0s. Our evaluation of entropy reduction reveals a range of improvement of up to 11% for keys-only sorting, and the early-exit optimization is activated at zero-effective bits.

Interestingly enough, our performance improvement is not a result of lower memory workload. These kernels are compute-bound; the speedup is instead gained from the elimination of warp-serialization hazards that stem from bank conflicts incurred during key exchange.

Banding is perhaps a more likely form of reduced key diversity. In these scenarios, keys are differentiated only by narrow bands of key bits. Fig. 20 illustrates the effectiveness of our early-exit optimization. Without specifying any explicit information to the sorting implementation, we evaluate sorting performance on key distributions whose keys differ only by variously-sized "banded" bit fields.

For GTX-285 keys-only sorting, we observe that the cost of a distribution pass is reduced by 83% when short-circuited. The result is a potential boosting of sorting rates by up to 5.8x. As an example, consider an array of 32-bit integers containing the same nominal information as a similarly-sized array of 8-bit characters. Our implementation will discover this banding information transparently and sort these 32-bit keys at a rate of 1.6 billion keys/s on the GTX-285, a speedup of 2.6x.

5. Discussion

5.1. *Comparison with Other GPU Sorting Methods*

Radix sorting methods make certain positional and symbolic assumptions regarding the bitwise representations of keys. A comparison-based sorting method is required for a set of ordering rules in which these assumptions do not hold. A variety of comparison-based, top-down partitioning and bottom-up merging strategies have been implemented for the GPU, including quicksort [43,24], most-significant-digit radix sort[34], sample-sort [44,45], and merge sort [5]. The number of recursive iterations for these methods is logarithmic in the size of the input sequence, typically with the first or last 8-10 iterations being replaced by a small local sort within each threadblock.

There are several contributing factors that have historically given radix sorting methods an advantage over their comparison-based counterparts. First, comparison-based sorting methods must have work-complexity $O(n\log_2 n)$ [10], making them less efficient per key as problem size grows. Second, for problem sizes large enough to saturate the device (e.g., several hundred-thousand or more keys), a radix digit size $d \geq 4$ will result in fewer digit passes than recursive iterations needed by the comparison-based methods performing binary partitioning. Third, the amount of global intermediate state needed by these methods for a given level in the tree of computation is proportional to the width of that level, as opposed to a small constant amount for our radix sort strategy. Finally, parallel radix sorting methods guarantee near-perfect load-balancing amongst GPU cores, an issue of concern for comparison-based methods involving pivot selection.

5.2. *The Multi-scan Abstraction*

Multi-scan is related to, but different from the *segmented scan* problem [46]. The segmented-scan problem connotes a single vector comprised of multiple segments, each segment an independent scan problem. For radix sorting purposes, there no segments *per-se*. We cannot afford to actually construct such a vector of r segments in global memory. Instead, we generate and consume local portions of our flag-vector scan problems in parallel. In addition, these scans are not completely independent: their partial reductions are concatenated for the top-level scan, resulting in a total ordering of partition offsets.

Multi-scan can be considered as a continuation of the histogram-scanning technique: histogram-scanning stops after the top-level scan, whereas multi-scan continues the parallel downward sweep, making it a suitable abstraction for multi-partition and multi-queue allocation problems.

5.3. *Code Specialization*

The optimal sorting and unrolling granularities may be different for different GPU processor architectures as well as for different data types (e.g., different tile sizes and sequential unrolling for sorting 8-bit characters versus 64-bit doubles). We rely on the

C++ compiler for template expansion, constant propagation, annotated static loop unrolling, and preprocessor macro expansion in order to produce an executable assembly that is well-tuned for the specifically targeted hardware *and* problem. These meta-programming techniques, i.e., "programming the compiler", are necessary for achieving good performance on an architecture where the overheads of runtime decision-making are substantially magnified by parallelism. The benefit of meta-programming is that we can author a single high-level source that is capable of high performance sorting across many architectures and data types.

There are several drawbacks to meta-programming under the current model of compilation, all related to the fact that this extra programmer-supplied detail (e.g., the relationships between unrolling steps and tile sizes) is lost after the high-level source is compiled down into an intermediate representation. First, authors of library routines (like our radix sorting implementation) cannot possibly hope to distribute them in the form of precompiled binary libraries. The number of specializations that would arise from simply compiling the cross-product of the built-in numeric data-types with today's existing architectures would result in untenable library bloating. Instead, library providers must distribute high-level sources that can be #included, similar to the C++ standard template library. This places an unwanted burden on library authors who may not want to share the details of their high-level source code or contend with lengthier compile times.

Secondly, the meta-programming approach is subject to performance regression by omission. Although code specialization by data type can be driven by well-defined sets of traits (e.g., representation size, signed/unsigned, etc.), specialization by architecture requires up-to-date processor configuration details at compile-time. For example, running an assembly of our radix sorting code on a GF100 processor that was unrolled for the GT200 architecture will result in a 2x slowdown. The configuration parameters within our source require regular maintenance as new architectures are released at the current frequency of ~18 months. A better solution might entail a mechanism for the programming model to retain meta-programming directives within the intermediate representation, allowing just-in-time compilation by the loader/driver to specialize and unroll executable code for the specific target processor.

5.4. *Conclusions*

We have presented efficient radix-sorting strategies for ordering large sequences of fixed-length keys (and values) on GPU stream processors. Our performance results demonstrate multiple factors of speedup over existing GPU implementations, and we believe our implementations to be the fastest available for any fully-programmable microarchitecture.

These results motivate a style of flexible algorithm design for GPU stream architectures that can maximally exploit the memory and computational resources, yet easily be adapted for a diversity of underlying hardware configurations. Our allocation-oriented framework provides us substantial flexibility with respect to radix sorting

granularity, well-positioning our approach to take advantage of increasing computational throughputs that outpace improvements in memory bandwidth.

Our results also serve to demonstrate the suitability of GPU computing for dynamic problems that had been previously considered ill-fitting of the architectural genre. As a case study on allocation-oriented problem decomposition, we provide our sorting to the public as part of the Thrust productivity library for GPU computing [47].

6. References

[1] J D Owens et al., "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879-899, May 2008.

[2] Victor W Lee et al., "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th annual international symposium on Computer architecture*, Saint-Malo, France, 2010, pp. 451--460.

[3] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure, "On the limits of GPU acceleration," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism (HotPar'10)*, Berkeley, CA, 2010, pp. 13--13.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addisson-Wesley, 1995.

[5] Nadathur Satish, Mark Harris, and Michael Garland, "Designing efficient sorting algorithms for manycore GPUs," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1-10.

[6] GPGPU.org. [Online]. http://gpgpu.org/developer/cudpp

[7] Nadathur Satish et al., "Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort," , 2010, pp. 351--362.

[8] Jatin Chhugani et al., "Efficient implementation of sorting on multi-core SIMD CPU architecture," *Proc. VLDB Endow.*, pp. 1313-1324, 2008.

[9] Nadathur Satish et al., "Fast Sort on CPUs, GPUs and Intel MIC Architectures," techreport 2010.

[10] Donald Knuth, *The Art of Computer Programming*. Reading, MA, USA: Addison-Wesley, 1973, vol. III: Sorting and Searching.

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to Algorithms*, 2nd ed.: McGraw-Hill, 2001.

[12] Guojing Cong, George Almasi, and Vijay Saraswat, "Fast PGAS Implementation of Distributed Graph Algorithms," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, 2010, pp. 1--11.

[13] Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo, "Data-Parallel Octrees for Surface Reconstruction," *EEE Transactions on Visualization & Computer Graphics*, p. to appear, 2010.

[14] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo, "Real-time KD-tree construction on graphics hardware," in *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, Singapore, 2008, pp. 1-11.

[15] J Pantaleoni and D Luebke, "HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry," in *Proceedings of the Conference on High Performance Graphics (HPG '10)*, Saarbrucken, Germany, 2010, pp. 87--95.

[16] Erik Sintorn and Ulf Assarsson, "Real-time approximate sorting for self shadowing and transparency in hair rendering," in *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, Redwood City, California, 2008, pp. 157--162.

[17] Kun Zhou et al., "RenderAnts: interactive Reyes rendering on GPUs," in *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, Yokohama, Japan, 2009, pp. 1--11.

[18] Bernhard Kainz et al., "Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore GPUs," in *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, Yokohama, Japan, 2009, pp. 1--9.

[19] Peter Kipfer, Mark Segal, and Rüdiger Westermann, "UberFlow: a GPU-based particle engine," in *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Grenoble, France, 2004, pp. 115--122.

[20] Jonathan M Cohen, Sarah Tariq, and Simon Green, "Interactive fluid-particle simulation using translating Eulerian grids," in *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, Washington, D.C., 2010, pp. 15--22.

[21] Charles Loop and Kirill Garanzha, "Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing," in *Eurographics*, 2010.

[22] Ignacio Castaño. (2007, February) High Quality DXT Compression Using CUDA. [Online]. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/dxtc/doc/cuda_dxtc.pdf

[23] Dan A Alcantara et al., "Real-time parallel hashing on the GPU," in *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, Yokohama, Japan, 2009, pp. 1--9.

[24] Bingsheng He et al., "Relational joins on graphics processors," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Vancouver, Canada, 2008, pp. 511--524.

[25] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, Chicago, IL, 2006, pp. 325--336.

[26] Naga Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha, "Fast and approximate stream mining of quantiles and frequencies using graphics processors," in *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, Baltimore, MD, 2005, pp. 611--622.

[27] Jeremy Shopf, Joshua Barczak, Christopher Oat, and Natalya Tatarchuk, "March of the Froblins: simulation and rendering massive crowds of intelligent and detailed creatures on GPU," in *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, Los Angeles, CA, 2008, pp. 52--101.

[28] A Hartstein, V Srinivasan, T R Puzak, and P G Emma, "Cache miss behavior: is it √2?," in *CF '06: Proceedings of the 3rd conference on Computing frontiers*, Ischia, Italy, 2006, pp. 313-320.

[29] R P Brent and H T Kung, "A Regular Layout for Parallel Adders," *IEEE Trans. Comput.*, vol. 31, no. 3, pp. 260--264, March 1982.

[30] Siddhartha Chatterjee, Guy Blelloch, and Marco Zagha, "Scan primitives for vector computers," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, New York, New York, 1990, pp. 666-675.

[31] Andrea C Dusseau, David E Culler, Klaus E Schauser, and Richard P Martin, "Fast Parallel Sorting Under LogP: Experience with the CM-5," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, pp. 791-805, 1996.

[32] Marco Zagha and Guy Blelloch, "Radix sort for vector multiprocessors," in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Albuquerque, NM, 1991, pp. 712--721.

[33] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens, "Scan Primitives for GPU Computing," in *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, San Diego, CA, 2007, pp. 97--106.

[34] Bingsheng He, Naga K Govindaraju, Qiong Luo, and Burton Smith, "Efficient gather and scatter operations on graphics processors," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, NV, 2007, pp. 1-12.

[35] Shubhabrata Sengupta, Mark Harris, and Michael Garland, "Efficient Parallel Scan Algorithms for GPUs," NVIDIA, Technical Report NVR-2008-003, 2008.

[36] Yuri Dotsenko, Naga K Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli, "Fast scan algorithms on graphics processors," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, Island of Kos, Greece, 2008, pp. 205-213.

[37] Duane Merrill and Andrew Grimshaw, "Parallel Scan for Stream Architectures," University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14, 2009.

[38] Guy E Blelloch, Siddhartha Chatterjee, and Marco Zagha, "Solving Linear Recurrences with Loop Raking," in *Proceedings of the 6th International Parallel Processing Symposium*, 1992, pp. 416-424.

[39] Mark Harris. (2007) Optimizing parallel reduction in CUDA. [Online]. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[40] Peter M Kogge and Harold S Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. Comput.*, vol. 22, pp. 786-793, 1973.

[41] Duane Merrill and Andrew Grimshaw, "Revisiting Sorting for GPGPU Stream Architectures," University of Virginia, Charlottesville, VA, Technical Report CS2010-03, 2010.

[42] K Thearling and S Smith, "An improved supercomputer sorting benchmark," in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing (SC '92)*, Minneapolis, Minnesota, 1992, pp. 14--19.

[43] Daniel Cederman and Philippas Tsigas, "GPU-Quicksort: A practical Quicksort algorithm for graphics processors," *J. Exp. Algorithmics*, vol. 14, pp. 1.4--1.24, 2009.

[44] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders, GPU sample sort, 2009.

[45] Frank Dehne and Hamidreza Zaboli, Deterministic Sample Sort For GPUs, 2010.

[46] Guy Blelloch, "Prefix Sums and Their Applications," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-90-190, 1990.

[47] Thrust. [Online]. http://code.google.com/p/thrust/