# Revisiting Sorting for GPGPU Stream Architectures[1]

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

## Abstract

This report presents efficient strategies for sorting large sequences of fixed-length keys (and values) using GPGPU stream processors. Our radix sorting methods demonstrate sorting rates of 482 million key-value pairs per second, and 550 million keys per second (32-bit). Compared to the state-of-the-art, our implementations exhibit speedup of at least 2x for all fully-programmable generations of NVIDIA GPUs, and up to 3.7x for current GT200-based models. For this domain of sorting problems, we believe our sorting primitive to be the fastest available for any fully-programmable microarchitecture.

We obtain our sorting performance by using a parallel scan stream primitive that has been generalized in two ways: (1) with local interfaces for producer/consumer operations (*visiting logic*), and (2) with interfaces for performing multiple related, concurrent prefix scans (*multi-scan*). These abstractions allow us to improve the overall utilization of memory and computational resources while maintaining the flexibility of a reusable component. We require 38% fewer bytes to be moved through the global memory subsystem and a 64% reduction in the number of thread-cycles needed for computation.

As part of this work, we demonstrate a method for encoding multiple compaction problems into a single, composite parallel scan. This technique provides our local sorting strategies with a 2.5x speedup over bitonic sorting networks for small problem instances, i.e., sequences that can be entirely sorted within the shared memory local to a single GPU core.

## 1 Introduction

The transformation of the fixed-function graphics processing unit into a fully-programmable, high-bandwidth coprocessor (GPGPU) has introduced a wealth of performance opportunities for many data-parallel problems. As a new and disruptive genre of microarchitecture, it will be important to establish efficient computational primitives for the corresponding programming model. Computational primitives promote software flexibility via abstraction and reuse, and much effort has been spent investigating efficient primitives for GPGPU stream architectures. Parallel sorting has been no exception: the need to rank and order data is pervasive, and the study of sorting problems predates the discipline of Computer Science itself [**1**].

As reusable components, performance concerns of speed and efficiency are top priorities for parallel sorting routines. Sorting techniques that involve partitioning or merging strategies are particularly amenable for GPGPU architectures: they are highly parallelizable and the computational granularity of concurrent tasks is miniscule. This report is concerned with the problem of sorting large sequences of elements, specifically sequences comprised of hundreds-of-thousands or millions of fixed-length, numerical keys. We consider two varieties of this problem genre: sequences comprised (a) 32-bit keys paired with 32-bit satellite values; and (b) 32-bit keys only. The solution strategies we describe here, however, can be generalized for keys and values of other sizes.

As an algorithmic primitive, sorting facilitates many problems including binary searches, finding the closest pair, determining element uniqueness, finding the *k*th largest element, and identifying outliers [**1**,**2**]. Sorting routines are germane to many GPU rendering applications, including shadow and transparency modeling [**3**], Reyes rendering [**4**], volume rendering via ray-casting [**5**], particle rendering and animation [**6**,**7**], ray tracing [**8**], and texture compression [**9**]. Sorting serves as a procedural step within the construction of KD-trees [**10**] and bounding volume hierarchies [**11**,**12**], both of which are useful data structures for ray tracing, collision detection, visibility culling, photon mapping, point cloud modeling, particle-based fluid simulation, etc. GPGPU sorting has also found use within parallel hashing [**13**], database acceleration [**14**,**15**], data mining [**16**], and game engine AI [**17**].

### 1.1 Contributions

We present the design of our strategy for radix sorting on GPGPU stream architectures, demonstrating that our approach is significantly faster than all previously published techniques for sorting large sequences of fixed-size numerical keys. We consider the GPGPU sorting algorithms described by Satish et al. [**18**] (and implemented in the CUDPP data parallel primitives library [**19**]) to be representative of the current state-of-the-art. Our implementations demonstrate factors of speedup of at least 2x for all fully-programmable generations of NVIDIA GPUs, and up to 3.7x for current generation models. Our local radix sorting strategies exhibit up to 2.5x speedup over bitonic networks for small problem instances that can be entirely sorted within the shared memory of a single GPU core (e.g., 128-512 elements), demonstrating the applicability of our work beyond our targeted problem domain.

[1] Technical Report CS2010-03, Department of Computer Science, University of Virginia. February 2010.

Revisiting previous sorting comparisons in the literature between many-core CPU and GPU architectures [**20**], our speedups show older NVIDIA G80-based GPUs to outperform Intel Core2 quad-core processors. We also demonstrate the NVIDIA GT200-based GPUs to outperform cycle-accurate sorting simulations of the unreleased Intel 32-core Larrabee architecture by as much as 42%. Among other features, write-coherent caches and alternative styles of synchronization were supposed to provide the Larrabee architecture with a clear advantage over many-core GPGPUs for cooperative problems like sorting [**21**].

We refer to our radix sorting approach as a *strategy* [**22**] rather than as a specific algorithm because it is actually a flexible hybrid composition of several different algorithms. We use different algorithms for composing and implementing different phases of computation. For example, our approach uses a flexible meta-strategy for problem decomposition across GPU cores in order to accommodate different processor configurations. We also employ different algorithms for local computation within a GPU core for (a) thread-independent processing in registers; (b) inter-warp cooperation in shared memory; and (c) intra-warp cooperation within a SIMD lane. As hybrid compositions, our strategies are adjustable in terms of the number of steps needed for each phase. This allows our solution to be flexible in terms of support for different SIMD widths, shared memory sizes, and mated to optimal patterns of device memory transactions.

As with traditional high performance computing, GPGPU applications strive to maximally utilize both computational and I/O resources; inefficient usage or underutilization of either resource is often indicative of suboptimal problem decomposition. Our performance speedup over the CUDPP implementation is due to our improved usage of both resources: we require 38% fewer bytes to be moved through the global memory subsystem (for both key-value and key-only sorting), and a 64% reduction in the number of thread-cycles needed for computation (60% reduction for keys-only).

Two important and related factors have contributed to our ability to make such efficient use of the hardware. The first is our prior work on the development of very efficient, memory-bound parallel *prefix scan* routines [**23**]. Scanning tasks play an important role in the composition of many parallel algorithms for shared-memory architectures: scanning routines allow processing elements to dynamically and cooperatively determine the appropriate memory location(s) into which their output data can be placed. The radix sorting method is a perfect example: keys can be processed in a data-parallel fashion for a given digit-place, but cooperation is needed among processing elements so that each may determine the appropriate destinations for relocating its keys.

The second contributing factor is that we employ a distinctly new pattern of program composition for stream architectures. Our key insight is that application performance depends not only upon the *implementation* of a given primitive, but its *usage* as well. In typical design, stream primitives such as prefix scan are invoked by the host program as black-box subroutines. The stream kernel (or short sequence of kernels) appears to be a natural abstraction boundary: kernel steps in a stream pipeline are often functionally orthogonal, resulting in a low degree of coupling with other routines. Additionally, writing sequential code for orchestrating parallel steps is comparatively easier than writing parallel kernel code.

When reusable primitives such as prefix scan are extremely memory-bound, we demonstrate that a much higher degree of overall system utilization can be achieved by applying a *visitor* pattern [**22**] of task composition within the design process, complementing the usual pattern of procedural stream composition. More specifically, we advocate another pair of abstraction boundaries in which the primitive invokes (i.e., “visits”) application-specific logic for (a) input-generation and (b) post-processing behavior. For example, our radix sorting design supplements the prefix scan primitive with *binning* and *scatter* routines for flagging the particular numeral represented within a given key and digit place, and for relocating keys (and values) based upon the ordering results computed by the scan primitive.

This pattern of composition provides an elegant mechanism for increasing the arithmetic intensity of memory-bound reduction and scan primitives. In addition to improved utilization of computing resources, the overall number of memory transactions needed by the application is dramatically reduced because we obviate the need to move intermediate state (e.g., the input/output sequences for scan) through global device memory. Not only does this reduce the overall load on the memory subsystem, but the absence of load/store instructions lowers the cumulative dynamic instruction count as well.

### 1.2 Organization of the Report

Section 2 presents a brief overview of GPGPU stream architectures, highlighting the novel performance characteristics that we seek to leverage with our radix sorting design. Section 3 reviews the radix sorting method and discusses prior solutions for performing this type of numerical-key sorting on the GPGPU. Section 4 describes the design of our strategy, providing an overview of our parallel scan primitive and the details for how we extend it to provide stable digit-sorting functionality. Section 5 presents our performance evaluations and Section 6 concludes.

## 2 Parallel Computation on the GPGPU

### 2.1 Stream Machine Model

GPGPU stream processors expose a parallel abstract machine model capable of concurrently executing large quantities of ultra-fine-grained tasks with an unprecedented degree of efficiency. The model 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*. Present GPGPUs rely upon a host platform in order to orchestrate kernel invocation and data movement.

Typical GPGPU organization entails a collection of processor 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*. Modern GPU processor dies typically contain several tens of SM cores. Each SM contains only enough ALUs to actively execute one or two warps, 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 core 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.

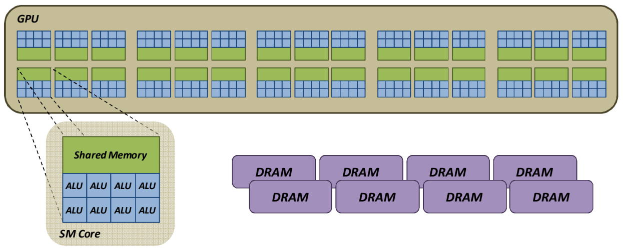

**Figure 1.** A typical GPGPU organization comprised of 30 SM cores, each having 8 SIMD ALUs and a local shared memory space. Globally-visible DRAM memory is off-chip.

Communication between processing elements is achieved by reading and writing data to various shared memory spaces. Typical implementations 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.

### 2.2 Stream Programming Paradigm

A kernel is an imperative function executed by all threads. In a typical pattern of data-parallel decomposition, a thread uses its identity to select and read corresponding input elements from global device memory, performs some finite computation (possibly involving local cooperation), and writes its result back to global device memory. The host orchestrates the global flow data by repeatedly invoking new kernel instances, each containing a grid of threads that is initially presented with a consistent view of the results from the previous kernel invocation. This sequential pipeline of kernel invocations is called a *stream*.

Language-level constructs for thread-grouping are often provided to facilitate logical problem decomposition in a way that is convenient for mapping blocks of threads onto physical SM cores. The CUDA programming framework exposes two levels of grouping: a CTA (*cooperative thread array*) of individual threads that share a memory space local to an SM core, and a *grid* of homogeneous threadblocks that encapsulates all of the threads for a given kernel.

Cooperation amongst threads is based on the *bulk-synchronous* model [**24**]: 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. An important consequence is that cooperation amongst SM cores requires the invocation of multiple kernel instances.

### 2.3 Performance Implications

The “over-threaded” style of SMT enables stream architectures to hide enormous 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. Although it can complicate analysis and modeling, this behavior is perhaps the most interesting performance feature in terms of the opportunities it presents for maximally utilizing device resources.

### 2.3.1 Resource Utilization

It is commonly considered less desirable for a given problem to be memory-bound (or, more generally, I/O-bound) than compute-bound. Historically, the trend of growing disparity between processor throughput and I/O bandwidth for successive microprocessor generations has meant that I/O-bound implementations would benefit substantially less from the simple passage of time.

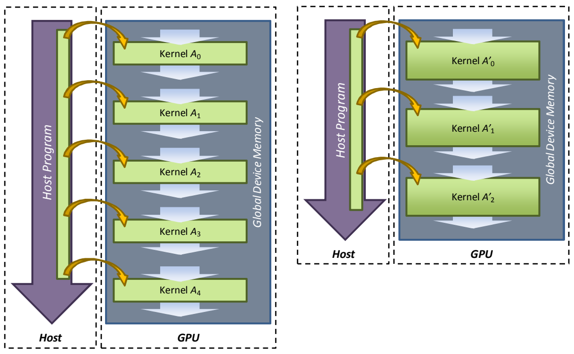

**Figure 2**. A GPGPU stream application with an adjustable computational granularity. A stream having more kernel invocations of smaller granularity is shown on the left, and a corresponding stream in which the granularity has been increased is shown on the right.

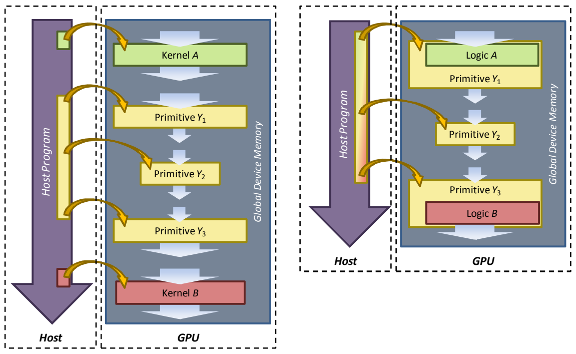

**Figure 3**. Coalescing producer and consumer logic into the kernels of a stream primitive. A stream invoking separate application-specific kernels is shown on the left, and a corresponding stream in which these tasks are visited by the stream primitive is shown on the right.

There are two approaches for rebalancing I/O-bound workloads in an effort to obtain better overall system utilization, i.e., for improving *arithmetic intensity*. The first is to increase the amount of local computation in such a way that fewer intermediate results need to be communicated off-chip, as illustrated in Figure 2. Many algorithmic strategies are capable of a variable granularity of computation, i.e., they can trade redundant computation for fewer I/O operations. For example, parallel implementations of the finite-difference method can increase the number of ghost cells surrounding local computation and perform redundant calculations in order to decrease the number of message-exchange rounds [**25**]. For radix sorting, increasing the number of bits per radix-digit decreases the total number of digit places that need iterating over. This approach has two challenges: (1) the operational details are very application-specific; and (2) linear decreases in I/O overhead often require super-linear increases in dynamic instruction counts and local storage requirements. In general, we see the latter reflected in the "power law of cache misses" [**26**], which suggests that the influence of cache size (e.g., local storage) upon miss rate (e.g., off-chip memory accesses) follows a power curve in which squaring the cache size typically results in halving the cache misses.

The second approach is to co-locate sequential steps in the stream pipeline within a single kernel, as illustrated in Figure 3. 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. For each pair of steps that can be co-located, we can save $O(n)$ global memory-references. While parallel primitives typically require global cooperation (and therefore multiple kernel launches), the producer step that generates an input sequence is often data-parallel and can be relocated inside the primitive's first kernel, replacing that kernel's original gather operation. The same can be done for a consumer step, i.e., it can replace the scatter logic within the scan primitive's last kernel. In traditional procedural algorithm composition, one would simply invoke reduction or scan functionality as routines from within application-specific logic, i.e., the application drives the primitive. However, reduction and scan primitives require multiple kernel invocations by the host: the host must actually drive the primitive's kernels which can then in turn "visit" the pre/post steps.

The over-threaded nature of the GPGPU architecture provides predictable and usable "bubbles of opportunity" for extra computation within I/O-bound primitives. If the computational overhead of the visiting logic can fit within the envelope of this bubble, this work can be performed at essentially zero-cost. If the resulting kernel is still memory-bound, we can ratchet up the application-specific arithmetic intensity, if any. This technique is particularly effective for improving the overall system utilization of streams comprised of alternating memory- and compute-bound kernels, and its degree of effectiveness is directly related to the efficiencies of the encapsulating primitives.

### 2.3.2 Analysis and Modeling

Performance models are often extended from algorithmic timestep analyses, with particular attention devoted to the practical constants involved. In an ideal network of scalar processors (i.e., one concurrent task per processor), the system becomes saturated when the number of concurrent tasks equals the number of physical processors. Overall runtime then becomes proportional to the number of tasks and processors in the system, i.e., runtime $\propto$ tasks/processors. It is common for performance models to incorporate (a) the number of processors and (b) architecture-specific task-duration constants having units time/step.

The effect of GPGPU latency-hiding, however, is that the saturation point occurs when the number of schedulable thread contexts is much greater than the number of ALUs on the GPU die. This causes the system to appear to scale as if it has many more ALUs than it does, i.e., an "effective" processor count for saturated conditions. It is problematic to reason about effective processor count and task duration, particularly in terms of virtualized notions of execution afforded by the programming model (e.g., threads, warps, CTAs, etc.). They are dynamic quantities, affected by the number of hazards experienced by the instantaneous workload. Even with an accurate model of effective scalar processor count, the SIMD nature of the SM cores complicates matters because time-slicing is per warp, not per task.

Instead we can simply scale aggregate work by the aggregate device throughputs for computation and memory. Most saturated GPU cores will exist in either a compute-bound or a memory-bound state for the duration of a stream kernel: over-threading is effective at averaging out the particular phases of behavior any given thread is experiencing. Compute-bound kernels proceed at the device's aggregate number of thread-cycles per second ($\delta_{\text{compute}}$), and memory-bound kernels proceed at the bandwidth of the memory subsystem ($\delta_{\text{mem}}$). By separately modeling the aggregate computational and memory workloads in terms of cycles and bytes, we can divide each by the respective throughput afforded by the device to obtain their corresponding temporal overheads. Because the two workloads are effectively divorced, the overall time overhead will be the larger of the two. This separation-of-concerns can be generalized for other types of workloads, e.g., to accommodate devices and streaming problems that make use of L2 caches.

Another difference between traditional massively-parallel systems and GPGPUs is that the latter has a much higher ratio of global communication cost to processor cores. The relative cost of moving data through global memory makes it undesirable for tree-based cooperation between SM cores to be comprised of more than two levels. Consider massively-parallel reduction over an ideal network of scalar processors in which the set of processors consume $O(n/p)$ time to generate per-processor sums in parallel, and then $O(\log_2 p)$ time to reduce them in a binary computation tree [**27**]. For some device-specific task-duration constants $c_1$ and $c_2$, we would model saturated runtime as:

$$T_{\text{reduce-scalar}}(n,p) = \frac{c_1 n}{p} + c_2 \log_2 p$$

The GPGPU version will only be comprised of two kernels: (1) a saturated bottom-level kernel comprised of $C$ CTAs that each reduces $n/C$ elements; and (2) an unsaturated top-level kernel in which a single CTA reduces the $C$ intermediate results [**23**]. For saturating inputs, we can safely ignore the overhead of the insignificant top-level kernel and drop the $\log_2 p$ component from the model above. Assuming conversion constants $c_3$ to model reduction operations in terms of cycles and $c_4$ to model words in terms of bytes, we would model GPGPU saturated runtime as:

$$T_{\text{reduce-gpu}}(n, \delta_{\text{compute}}, \delta_{\text{mem}}) = max\left(\frac{c_3 n}{\delta_{\text{compute}}}, \frac{c_4(n+C)}{\delta_{\text{mem}}}\right)$$

## 3 GPGPU Sorting

### 3.1 Radix Sorting

The radix sorting method is one of the oldest sorting techniques, implemented as early as the 1880s within Hollerith's machines for sorting punched cards [**28**]. The method relies upon a positional representation for keys, i.e., each key is comprised of an ordered sequence of symbols (i.e., *digits*) specified from least-significant to most-significant. For a specific total ordering of the symbolic alphabet and a given input sequence of keys, the radix sorting method produces a lexicographic ordering of those keys.

Radix sort (more specifically, the least-significant-digit variety) proceeds 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. The method is particularly amenable for sorting sequences of numerical keys. Given an $n$-element sequence of $k$-bit keys and a radix $r = 2^d$, a radix sort of these keys will require $k/d$ passes of a distribution sort over all $n$ keys. With a fixed digit-size of $d$-bits, the distribution sort will have an asymptotic time complexity of $O(n)$. Because the number of digit-places is unrelated to the input size, the entire process is also $O(n)$.

The distribution sort is the fundamental component of the radix sorting method. In a data-parallel decomposition for shared-memory implementations, a logical processor uses its rank to gather its key, decodes the specific digit at the given digit-place, and then must cooperate with other processors in order to determine the appropriate scatter location for its key. The relocation offset will be the key's global rank, i.e., 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. The latter ensures a stable sort in which the partial orderings established by the previous passes are not disturbed.

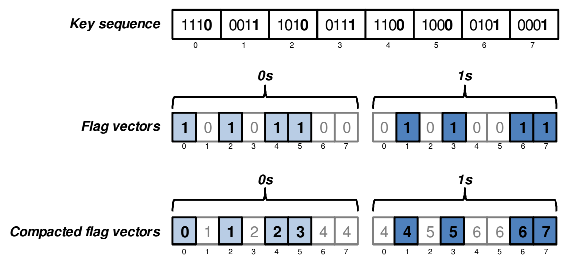

**Figure 4.** Example of a radix $r = 2$ distribution sort on the first digit-place of a sequence of eight keys in which a prefix scan of flag vectors is used to determine the scatter destinations for each key.

The ranking process can be constructed from one or more parallel prefix scan operations used to partition the keys based the digits contained at a given digit-place. In their promotion of prefix scan, Blelloch et al. demonstrated a distribution sort using a binary *split* primitive comprised of prefix scans over two $n$-element binary flag vectors: the first initialized with 1s for keys whose digit was 0, the second to 1s for keys whose digit was 1 [**29**]. The two scan operations are dependent: the scan of the 1s vector can be seeded with the number of zeros from the 0s scan, or equivalently the two vectors can be concatenated and processed by one large scan as shown in Figure 4. After the scans, the $i^{th}$ element in the appropriate compacted flag vector will indicate the relocation offset for the the $i^{th}$ key. An optimization for radix $r = 2$ is to obviate the 1s scan: the destination for a 1s key can be determined by adding the total number of 0 keys to the processor-rank and then subtracting the result from compacted 0s vector [**30**].

| Step | Kernel | Purpose | Read Workload | Write Workload |
|---|---|---|---|---|
| 1 | *binning* | Create flags | $n$ keys | $nr$ flags |
| 2 | *bottom-level reduce* | Compact flags (scan primitive) | $nr$ flags | (*insignificant constant*) |
| 3 | *top-level scan* | | (*insignificant constant*) | (*insignificant constant*) |
| 4 | *bottom-level scan* | | $nr$ flags + (*insignificant constant*) | $nr$ offsets |
| 5 | *scatter* | Distribute keys | $n$ offsets + $n$ keys (+ $n$ values) | $n$ keys (+ $n$ values) |

**Total Memory Workload:** $(k/d)(n)(r + 4)$ keys only
$(k/d)(n)(r + 6)$ with values

**Figure 5.** A naïve distribution sorting GPGPU stream constructed from a black-box parallel scan primitive and binning and scatter kernels with $d$-bit radix digits, radix $r = 2^d$, and an $n$-element input sequence of $k$-bit keys.

A simple, naïve GPGPU distribution sort implementation can be constructed from a black-box parallel scan primitive sandwiched between separate *binning* and *scatter* kernels. The binning kernel would be used to create a concatenated flag vector in global memory and the scatter kernel to redistribute the keys (and values) according to their compacted offsets. The stream's sequence of kernel invocations is shown in Figure 5, where the middle three kernels comprise the parallel scan primitive that compacts the concatenated flag vectors. The *top-level scan* kernel is unsaturated and contributes negligible performance overhead for sorting problems large enough to saturate the other kernels. The *scatter* kernel need not read in all $nr$ offsets; only those corresponding to the particular digits (re)decoded from the keys.

The naïve approach suffers from excessive use of global device memory. The number of flags that are moved through memory is $O(rn)$, i.e. it has a linear coefficient that is exponential in terms of the number of radix digit bits $d$. In this case, selecting the number of radix digit bits $d = 1$ will always minimize the total overhead for $k/d$ passes. The entire radix sort requires the memory subsystem to process $448n$ words ($320n$ if obviating the 1s scan) when sorting 32-bit keys and values. The aggregate memory workload will set a lower bound on the achievable performance.

For SPMD architectures, the number of parallel processors has historically been smaller than the input sequence size, making it natural to distribute portions of the input to processors in blocks of $b$ keys. Instead of collectively producing an $n$-element binary flag vector for each radix digit, the processors can each write out an $r$-element histogram of digit-counts. This reduces the intermediate storage requirements by a factor of $b$. Processors write their digit-histograms in column-major to global memory in a grid, i.e., a matrix where each row is comprised of the processor counts for a specific digit. After performing a parallel scan operation upon the grid of histograms in a rowmajor order, each processor can then obtain the relative digit-offsets for its block of keys from the resulting matrix. These offsets can then be applied to the local key rankings within the block in order to redistribute the keys. [**31**,**32**]

| **Step** | **Kernel** | **Purpose** | **Read Workload** | **Write Workload** |
|---|---|---|---|---|
| 1 | *local digit-sort* | Maximize coherence | *n* keys (+ *n* values) | *n* keys (+ *n* values) |
| 2 | *histogram* | Create histograms | *n* keys | *nr/b* counts |
| 3 | *bottom-level reduce* | Scan histograms (scan primitive) | *nr/b* counts | (*insignificant constant*) |
| 4 | *top-level scan* | | (*insignificant constant*) | (*insignificant constant*) |
| 5 | *bottom-level scan* | | *nr/b* counts + (*insignificant constant*) | *nr/b* offsets |
| 6 | *scatter* | Distribute keys | *nr/b* offsets + *n* keys (+ *n* values) | *n* keys (+ *n* values) |

**Total Memory Workload:** $(k/d)(n)(5r/b + 7)$ keys only
$(k/d)(n)(5r/b + 9)$ with values

**Figure 6.** GPGPU stream representative of the Satish et al. method for distribution sorting with $d$-bit radix digits, radix $r = 2^d$, local block size of $b$ keys, and an $n$-element input sequence of $k$-bit keys.

Several GPGPU radix sort implementations have followed this histogram-based approach, treating each CTA as a logical processor operating over a block of $b$ keys [**33**,**34**,**18**]. Of these, we consider the radix sort implementation described by Satish et al. to be representative of the current state-of-the-art. Their procedure is depicted in Figure 6. Although the overall memory workload still has a linear coefficient that is exponential in terms of the number of radix digit bits $d$, it will be significantly reduced by common CTA block-sizes of 128-1024 keys. For these implementations, the block-size factor elicits a bathtub effect in which an optimal $d$ exists to produce a minimal memory overhead for a given block size $b$. The Satish et al. implementation uses a block size $b = 512$ and a radix digit size $d = 4$ bits, requiring the memory subsystem to process $73.3n$ words for an entire sort of 32-bit keys and values. (If their stream kernels were all memory bound, the optimal radix digit size $d$ would be 8 bits.)

Their design incorporates a kernel that locally sorts individual blocks of keys by their digits at a specific digit-place. This helps to maximize the coherence of the writes to global memory during the scatter phase. Stream processors such as the NVIDIA GPUs obtain maximum bandwidth by *coalescing* concurrent memory accesses, i.e. the references made by a SIMD half-warp that fall within a contiguous memory segment can be combined into one memory transaction to provide higher overall utilization. Although it is relatively expensive in terms of memory and computational workloads, this localized sorting creates ordered subsequences of keys can then be contiguously and efficiently scattered to global memory in a subsequent kernel.

## 3.2 Other Sorting Approaches

The radix sorting methods that we've discussed make certain positional and symbolic assumptions regarding the bitwise representations of keys. When these assumptions do not hold for a given set of ordering rules, a comparison-based sorting method is required. Unlike radix sorting methods, comparison-based sorting methods must have work-complexity $O(n\log_2 n)$ [**1**], making them less efficient as problem size grows.

Sorting networks such as Batcher's bitonic and odd-even networks were among the first proposed methods for parallel sorting [**35**]. Because the sequence of comparisons is fixed beforehand, mapping their computation onto SIMD and SPMD architectures is often straightforward. Global sorting networks have been implemented for GPGPUs by Kipfer et al. [**36**], He et al. [**14**], Greβ et al. within GPU-ABiSort [**37**], and Govadaranju et al. within GPUTeraSort [**15**]. Although the $O(n\log_2^2 n)$ work complexity for these sorting networks causes performance to suffer for large inputs, other hybrid strategies make use of sorting networks for small, local sequences of keys [**18**,**14**]. Bitonic merging has also been used to compose blocks of keys ordered by other local sorting methods [**38**]. In this report, we demonstrate that our local radix sorting strategy is more efficient than these local sorting networks.

Sorting algorithms based upon $O(n\log_2 n)$ top-down partitioning and bottom-up merging strategies have also been adapted for the GPGPU. Cederman et al. [**39**] and He et al. [**14**] have demonstrated parallel quicksort implementations, He et al. have implemented a version of most-significant-digit radix sort [**34**], and Leischner et al. [**40**] and Dehne et al. [**41**] have adapted sample-sort (a multi-pivot variation of quicksort) as well. Satish et al. have also developed an efficient GPGPU merge-sort to compliment their radix sorting method [**18**]. 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 CTA.

There are several contributing factors that give radix sorting methods an advantage over their comparison-based counterparts. 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 comparison-based methods. In addition, 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 GPGPU cores, an issue of concern for comparison-based methods involving pivot selection.

## 4 Radix Sort Design

The primary design goal of our radix sorting strategy is to reduce the aggregate memory workload. In addition to being a lower-bound for overall performance, the size of the memory workload also greatly contributes to the size of the computational workload. For saturating problem sizes, the execution of load, store, and offset-calculation instructions for millions of intermediate results can account for a sizeable portion of the aggregate dynamic instruction count.

In this section, we describe how we generalize prefix scan to implement a distribution sort with a minimal number of intermediate values that must be exchanged through global device memory. More specifically, we do this by adding two capabilities to the stream kernels that comprise the parallel scan primitive: *visiting logic* and *multi-scan*. We use visiting logic to perform binning and scatter tasks, and multi-scan to compute the prefix sums of radix *r* flag vectors in parallel. Both features serve to increase the arithmetic intensity of our memory-bound primitive and allow the entire distribution sort to be constructed without incurring additional stream kernels or factors of memory accesses.

### 4.1 Meta-strategy

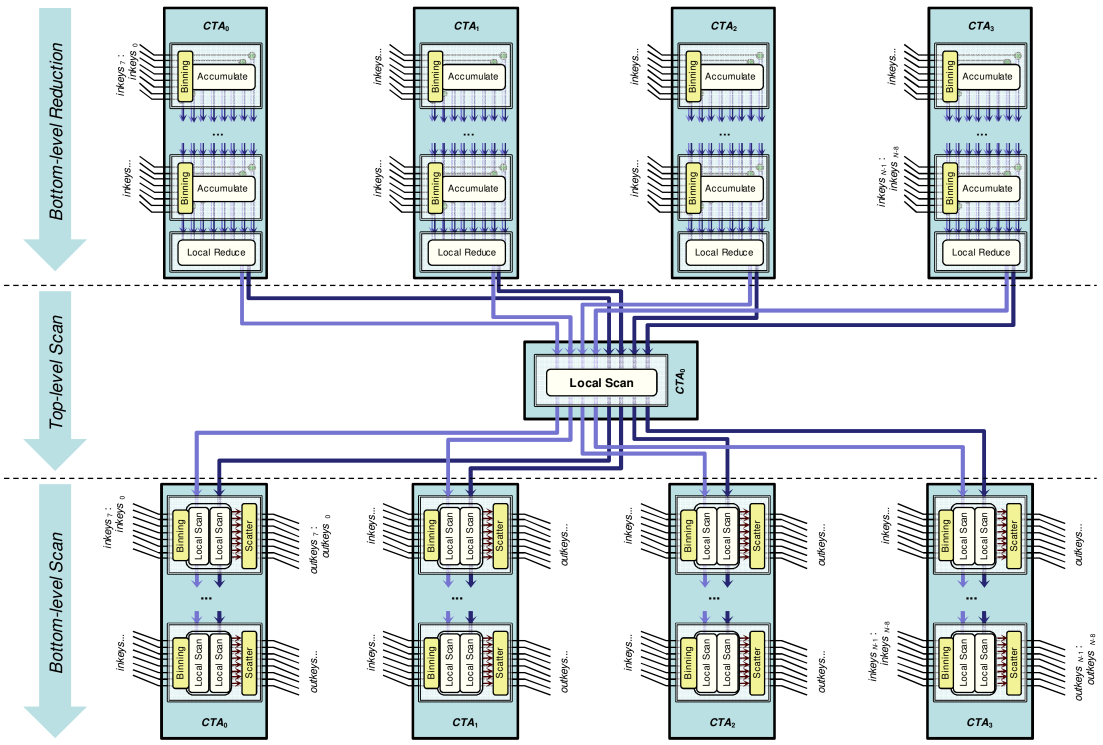

**Figure 7**. GPGPU stream sequence of kernel invocations for a distribution sort with radix $r = 2$. The bottom-level kernels are shown as being invoked in grids of $C = 4$ CTAs. Time is depicted as flowing downwards with each CTA processing blocks of $b = 8$ values (semi-transparent yellow) in serial fashion. Uncontained data-flow arrows indicate live intermediate values within global device memory, contained arrows indicate live values within local registers/shared memory. Data-flow arrows in light-blue indicate live intermediate values pertaining to the 0s scan, dark-blue for the 1s scan.

In prior work, we developed several efficient GPGPU scan strategies that use a *two-level reduce-then-scan* meta-strategy for problem decomposition across SM cores [**23**]. This meta-strategy is composed of three stream kernels: a bottom-level reduction, a top-level scan, and a bottom-level scan. Instead of allocating a unique thread for every input element, our bottom-level kernels deviate from the data-parallel programming paradigm and instead dispatch a fixed number *C* of CTAs in which threads are re-used to process the input sequence in successive blocks of *b* elements each. Reduction and scan dependencies between blocks are carried in thread-private registers or local shared memory. Figure 7 shows how these kernels have been supplemented with *visiting logic* and *multi-scan* capability.

The bottom-level reduction kernel reduces *n* inputs into *rC* partial reductions. Our reduction threads employ a *loop-raking* strategy [**42**] in which each thread accumulates values in private registers. The standard gather functionality has been replaced with binning logic that is parameterized with the current digit-place, the number of threads in the CTA, and the local block size *b*. When threads in the binning logic read in a block of keys, each decodes the digits at that digit-place for its keys and returns the corresponding digit-counts for each of the *r* possible digits to the kernel's accumulation logic. After processing their last block, the threads within each CTA perform cooperative reductions in which their accumulated values are reduced into *r* partial reductions and written out to global device memory, similar to Harris et al. [**43**].

The single-CTA, top-level scan has been generalized to scan a concatenation of *rC* partial reductions. For our purposes here, a single scan is performed over these sets of partial reductions. The top-level scan is capable of operating in a segmented-scan mode for multi-scan scenarios that produce independent sequences of input.

In the bottom-level scan kernel, CTAs enact the distribution sort for their portions of the input sequence, seeded with the appropriate prefix sums provided by the top-level scan. Each CTA serially reads consecutive blocks of *b* elements, re-bins them into *r* local flag vectors, and scans these vectors using a local parallel scan strategy. After the local scans have completed, the scatter logic is presented with the *r* prefix sums specific to each key. The scatter operation uses this information to redistribute the keys. It is also responsible for loading and similarly redistributing any satellite values. The aggregate counts for each digit are serially curried into the next *b*-sized block.

| Step | Kernel | Purpose | Read Workload | Write Workload |
|---|---|---|---|---|
| 1 | *bottom-level reduce* | Create flags, compact flags, scatter keys | *n* keys | (*insignificant constant*) |
| 2 | *top-level scan* | | (*insignificant constant*) | (*insignificant constant*) |
| 3 | *bottom-level scan* | | *n* keys (+ *n* values) + (*insignificant constant*) | *n* keys (+ *n* values) |

**Total Memory Workload:** $(k/d)(3n)$ keys only
$(k/d)(5n)$ with values

**Figure 8.** Our distribution sorting GPGPU stream constructed from a parallel multi-scan primitive and visiting binning and scatter kernels with *d*-bit radix digits, radix $r = 2^d$, and an *n*-element input sequence of *k*-bit keys.

The memory workloads for our distribution-sorting scan primitive are depicted in Figure 9. Only a constant number of memory accesses are used for the storage of intermediate results, and the overall workload no longer has a linear coefficient that is exponential in terms of the number of radix digit bits *d*. This implies that there is no optimal *d* to produce a minimal memory overhead. Because memory workload monotonically decreases with increasing *d*, our strategy is positioned to advantage itself of additional computational power that may allow us to increase *d* in the future. Current NVIDIA GPUs can afford our strategy a radix digit size $d = 4$ bits before exponentially-growing demands on local storage prevent us from saturating the device. This configuration only requires the memory subsystem to process 40*n* words for an entire sort of 32-bit keys and values.

## 4.2 Local Strategy

While the details of the reduction kernel are fairly straightforward, the local operation of our scan kernels warrants some discussion. Of our designs for scan primitives, the SRTS variant is the most efficient at processing blocks of contiguous elements [**23**]. In this subsection, we briefly review the details of its operation and describe the manner in which we adapt it for distribution sorting.

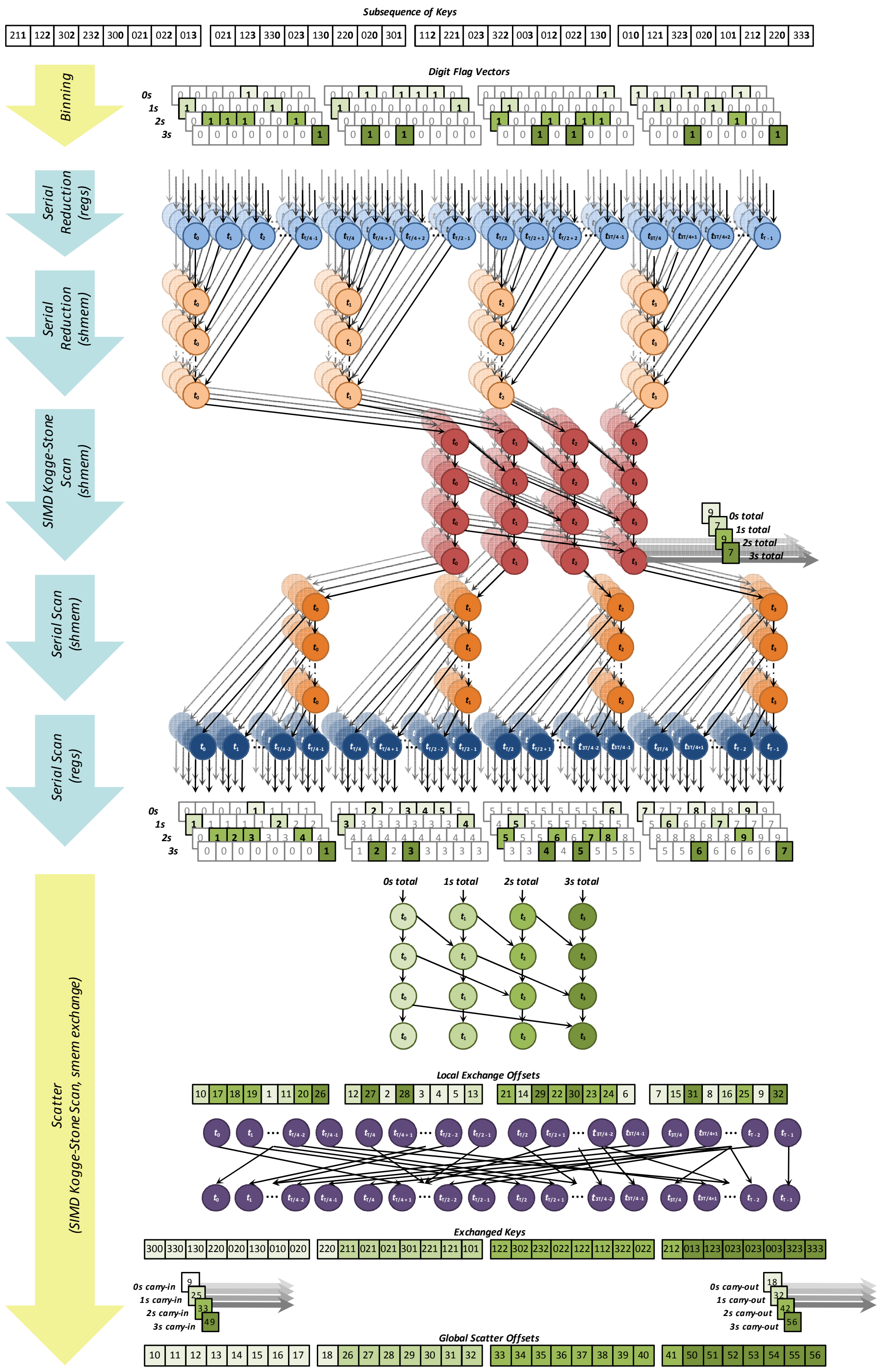

**Figure 9.** The operation of a generalized SRTS multi-scan CTA that has been supplemented with visiting logic for binning and scatter operations. This figure depicts computation and time flowing downward for an input block size of $b = 32$ keys, a radix $r = 4$ digits, and a warp-size $w = 4$ threads. The five SRTS stages are labeled in light blue, the visiting stages in yellow. Circles indicate the assignment of a given thread $t_i$ to a binary associative task. Flag vector encoding is not shown. The blue thread-independent processing phase is shown to accommodate 2-element (128B) loads/stores.

Figure 9 shows how we have augmented the bottom-level SRTS scan kernel with visiting logic to perform binning and scatter tasks, and with multi-scan to compute the local prefix sums of radix *r* flag vectors in parallel. The figure is illustrated from the point of a single CTA processing a particular block of input values. Threads within the binning 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. The scan logic is comprised of five stages, each of which is replicated *r*-times. This ultimately produces *r* vectors of *b* prefix sums each: one for each of the *r* possible digits.

The scan logic itself is a flexible hierarchy of reduce-then-scan strategies composed of three phases of upsweep/downsweep operation: (1) *thread-independent* processing in registers, shown in blue; (2) *inter-warp cooperation*, shown in orange; and (3) *intra-warp cooperation*, shown in red. The thread-independent phase serves to transition the problem from the block size *b* into a smaller version that will fit into shared memory and back again. This provides flexibility in terms of facilitating different memory transaction sizes (e.g., 1/2/4-element load/stores) without impacting the size of the shared-memory allocation. In the inter-warp cooperation phase, a single warp serially reduces and scans though the partial reductions placed in shared memory by the other warps in a raking manner similar to [**44**], transitioning the problem size into one that can be cooperatively processed by a single warp and back again. This provides flexibility in terms of facilitating shared memory allocations of different sizes, supporting alternative SIMD warp sizes, and accommodating arbitrary numbers of warps per CTA. For a given warp-size of *w* threads, the intra-warp phase implements $\log_2 w$ steps of a Kogge-Stone scan [**45**] in a synchronization-free SIMD fashion as per [**46**]. Running totals from the previous block are carried into the SIMD warpscan, incorporated into the prefix sums of the current block's elements, and new running totals are carried out for the next block, all in local shared memory.

The scatter operation is provided with the block of *b* keys, the *r* vectors of local prefix sums, the local 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 binning, doing so would result in poor write coherence. A random distribution of keys would result in each half-warp of threads making many transactions to many different memory segments instead of just one or two. Instead we use the local prefix sums to scatter them to local shared memory where consecutive threads can pick up consecutive keys and then scatter them to global device memory with a minimal number of memory transactions. In order to do this, the scatter operation computes the dependencies among prefix sum vectors with a SIMD warpscan of the local digit totals. Although our two-phase scatter procedure is fairly expensive in terms of dynamic instruction overhead and unavoidable bank conflicts, it is much more efficient than the sorting phase implemented by [**18**]. 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.

## 4.3 Multi-scan Optimization (`popc()` versus flag-encoding)

For the purposes of performing a distribution sort, the visiting binning logic returns a set of *compaction* problems (i.e., prefix scans of binary-valued input elements) to the multi-scan component. A single-instruction, native implementation of the 32-bit `popc()` intrinsic has been promoted as a mechanism for improving the efficiency of parallel compaction [**47**]. The `popc()` intrinsic is a bitwise operation that returns the number of bits set in a given word. A local compaction strategy using `popc()` would likely require at least 4 instructions per element per radix digit: a `ballot()` instruction to cooperatively obtain a 32-bit word with the appropriate bits set, a mask to remove the higher order bits, a `popc()` to extract the prefix sum, and one or more instructions to handle prefix dependencies between multiple `popc()`-words when compacting sequences of more than 32 elements.

We have designed a better alternative for increasing the efficiency of binary-valued multi-scans. Our binning and scatter operations implement a form of flag vector encoding that takes advantage of the otherwise unused high-order bits of the flag words. By breaking a $b = 512$ element block into two sets of 256-element multi-scans, the scatter logic can encode up to four digit flags within a single 32-bit word. This allows us to effectively process four radix digits with a single composite scan. For example, a CTA configured to process a 4-bit digit place can effectively compact all 16 local digit vectors with only four scan operations.

We can decompose the computational overhead of the bottom-level scan kernel in terms of the following tasks: data-movement to/from global memory; digit inspection and encoding of flag vectors in shared memory; shared-memory scanning; decoding local rank from shared memory; and locally exchanging keys and values prior to scatter. For a given key-value pair, each task will incur a fixed cost $\alpha$ in terms of thread-instructions. The flag-encoding and scanning operations will also incur a per-pair cost of $\beta$ instructions per composite scan. We model the computational workload of the bottom-level scan kernel in terms of thread-instructions as follows:

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

For the NVIDIA GT200 architecture, we have empirically determined $instrs_{\text{scankernel}}(n,r) = n(51.4 + r)$. The fixed-costs per pair are: $\alpha_{\text{mem}} = 6.3$ instructions; $\alpha_{\text{encflags}} = 5.5$ instructions; $\alpha_{\text{scan}} = 10.7$ instructions; $\alpha_{\text{decflags}} = 13.9$ instructions; and $\alpha_{\text{exchange}} = 14.7$ instructions. The variable per-scan costs per pair per composite scan are: $\beta_{\text{encflags}} = 2.6$ instructions; and $\beta_{\text{scan}} = 1.4$ instructions. In a conservative comparison when $r \geq 8$ (two or more composite scans), the local scanning efficiency of our approach will be much lower than that of a `popc()`-based strategy. Because of the small overheads for each additional composite scan, the discrepancy between the two approaches is likely to be several factors for our GT200-based strategies in which $r = 16$.

## 5 Evaluation

This section presents the performance of our SRTS-based radix sorting strategy along with the *CudPP* v1.1 implementation as a reference for comparison. Our core evaluation topics are overall sorting throughput, overall memory and computational workloads, and individual kernel memory and computational workloads.

For primary testing, we used a Linux system with an Intel i7 quad-core CPU and an NVIDIA GTX-285 GPU. Unless otherwise noted, our analyses are derived from performance measurements taken over a suite of 1,521 input problems initialized from keys and values sampled from a uniformly random distribution. We composed our input suite from three subsets:

- 750 problems with sequence sizes sampled uniformly from the range $[2^5, 2^{26}]$
- 750 problems with sequence sizes sampled log-normally (base-2) from the range $[2^5, 2^{26}]$
- 21 problems with sequence sizes comprising the powers-of-two between $2^5$ and $2^{26}$

The data points presented are the averages of measurements taken from two iterations of each problem instance. The measurements themselves (e.g., elapsed time, dynamic instruction count, warp serializations, memory transactions, etc.) are taken from hardware performance counters located within the GPU itself. The results they provide are very deterministic, eliminating the need for averaging over large numbers of redundant tests. Our analyses is reflective of *in situ* sorting problems: it precludes the 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.

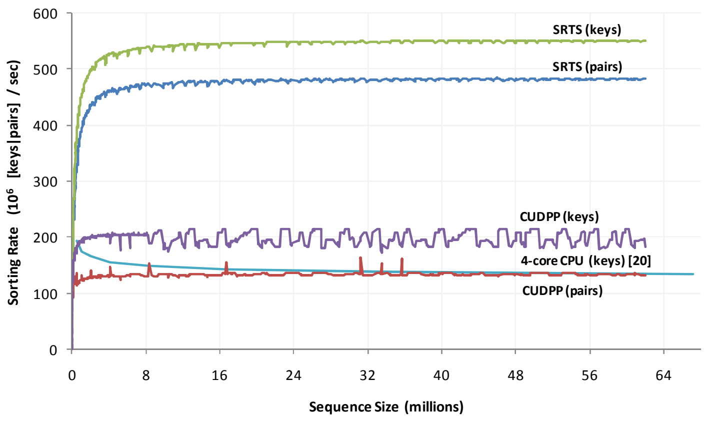

**Figure 10**. GTX-285 key-value and key-only radix sorting rates for the CUDPP and our 4-bit SRTS-based implementations, overlaid with Chhugani et al. key-only sorting rates for the Intel Core-2 Q9550 quad-core CPU.

| *Device* | | *Key-value Rate ($10^6$ pairs / sec)* | | *Keys-only Rate ($10^6$ keys / sec)* | |
|---|---|---|---|---|---|
| *Name* | *Release Date* | *CUDPP Radix* | ***SRTS Radix (speedup)*** | *CUDPP Radix* | ***SRTS Radix (speedup)*** |
| **NVIDIA GTX 285** | Q1/2009 | 134 | **482 (3.6x)** | 199 | **550 (2.8x)** |
| **NVIDIA GTX 280** | Q2/2008 | 117 | **428 (3.7x)** | 184 | **474 (2.6x)** |
| **NVIDIA Tesla C1060** | Q2/2008 | 111 | **330 (3.0x)** | 176 | **471 (2.7x)** |
| **NVIDIA 9800 GTX+** | Q3/2008 | 82 | **165 (2.0x)** | 111 | **226 (2.0x)** |
| **NVIDIA 8800 GT** | Q4/2007 | 63 | **129 (2.1x)** | 83 | **171 (2.1x)** |
| **NVIDIA 9800 GT** | Q3/2008 | 61 | **121 (2.0x)** | 82 | **165 (2.0x)** |
| **NVIDIA 8800 GTX** | Q4/2006 | 57 | **116 (2.0x)** | 72 | **153 (2.1x)** |
| **NVIDIA Quadro FX5600** | Q3/2007 | 55 | **110 (2.0x)** | 66 | **147 (2.2x)** |
| | | | | *Merge* [**20**] | |
| **Intel Q9550 quad-core** | Q1/2008 | | | 138 | |
| **Intel Larrabee 32-core** | Cancelled | | | 386 | |

**Figure 11**. Saturated sorting rates for input sequences larger than 16M elements.

Figure 10 plots the measured radix sorting rates exhibited by our implementation and the CUDPP primitive. For all NVIDIA GT200 and G80 GPUs, we best parameterize our strategy with radix $r = 16$ digits (digit size $d = 4$ bits). We have also overlaid the keys-only sorting results presented Chhugani et al. for the Intel Core-2 Q9550 quad-core CPU [**20**], which we believe to be the fastest hand-tuned numerical sorting implementation for multi-core CPUs. As expected, we observe that the radix sorting performances plateau into steady-state as the GPU's resources become saturated. In addition to exhibiting 3.6x and 2.8x speedups over the CUDPP implementation on the same device, our key-value and key-only implementations provide smoother, more consistent performance across the sampled problem sizes.

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. At the time, Chhugani et al. championed the performances of Intel's fastest consumer-grade Q9550 quad-core processor and cycle-accurate simulations of the 32-core Larrabee platform over G80-based NVIDIA GPUs [**20**]. Shortly afterward, Satish et al. presented GPGPU performance that was superior to the Q9550 from the newer NVIDIA GT200 architecture [**18**]. We extended our comparison to a superset of the devices evaluated by these publications. The saturated sorting rates on these devices for input sequences of 16M+ keys are denoted in Figure 11. We observe speedups over Satish et al. (CUDPP) of 3x+ for the GT200 architectures, and 2x+ for the G80 architectures. Using our method, all of the NVIDIA GPUs outperform the Q9550 CPU and, perhaps more strikingly, our sorting rates for GT200 GPUs exhibit up to 1.4x speedup over the cycle-accurate results for 32-core Larrabee (which would have dominated otherwise).

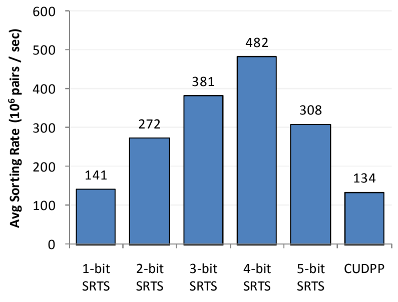

**Figure 12**. Saturated GTX-285 sorting rates ($n \geq$ 16M key-value pairs).

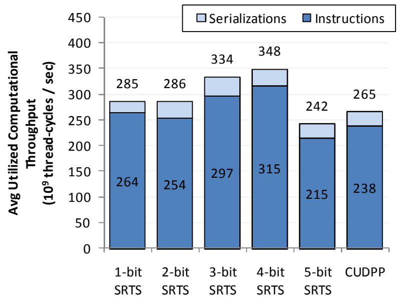

**Figure 13.** Realized GTX-285 computational throughputs ($n \geq$ 16M key-value pairs).

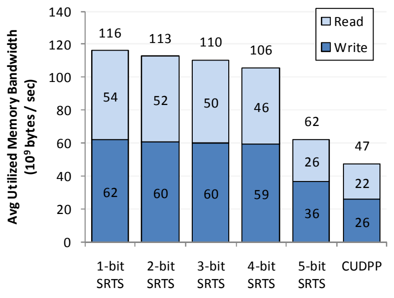

**Figure 14**. Realized GTX-285 memory bandwidths ($n \geq$ 16M key-value pairs).

After coalescing the binning, scanning, and scatter functionalities into a minimum of stream kernels, we can continue to increase the arithmetic intensity of our radix sorting strategy in an application-specific manner by increasing the number of radix digit bits $d$ (and thus decreasing the number of distribution sorting passes). Figure 12 shows our average saturated sorting rates for $1 \leq d \leq 5$. 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 radix digits $r = 2^d$ that need scanning. The first is that the cumulative computational workload is no longer decreasing with reduced passes. Because the two bottom-level kernels are compute-bound under this load, continuing to increase overall the computational workload will only result in progressively larger slowdowns. The second issue is that increasing local storage requirements (i.e., registers and shared memory) prevent SM saturation: the occupancy per GPU core is reduced from 640 to 256 threads.

Figures 13 and 14 show the computational throughputs and bandwidths realized by our SRTS variants and the CUDPP implementation. The realistic GTX-285 device maximums are $\delta_{compute} \approx 354 \times 10^9$ thread-cycles/second and $\delta_{mem} \approx 136\text{-}149 \times 10^9$ bytes/second. Because their saturating kernels are compute-bound, our 3-bit and 4-bit variants achieve 94+% utilizations of the available computational resources. 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. 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.

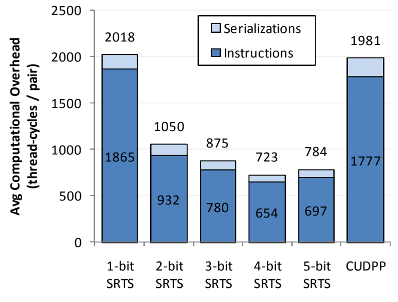

**Figure 15**. Aggregate GTX-285 computational overhead ($n \geq$ 16M key-value pairs).

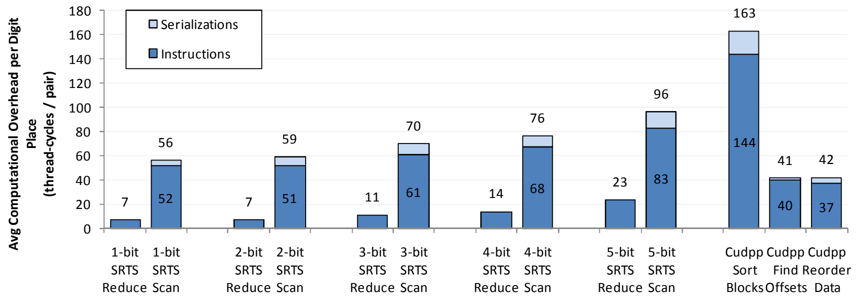

**Figure 16.** GTX-285 computational overhead per distribution sort ($n \geq$ 16M key-value pairs).

Our five SRTS columns in Figure 15 illustrate the "bathtub" curve of computational overhead versus digit size: workload decreases with the number of passes over digit-places until the cost of scanning radix digits becomes dominant at $d = 5$. This overhead is 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 element by our 4-bit implementation may seem substantial, yet efficiency is 2.7x that of the CUDPP implementation.

Figure 16 presents the computational overheads of the individual kernel invocations that comprise a single digit-place iteration, i.e., distribution sort. For $d > 2$, we observe that the workload deltas between scan kernels double as $d$ is incremented, scaling with $r$ as expected and validating our model of instruction overhead from Section 4.3. Our 1-bit and 2-bit variants don't follow this parameterized model: the optimizing compiler produces different code for them because flag vector encoding yields only one composite scan.

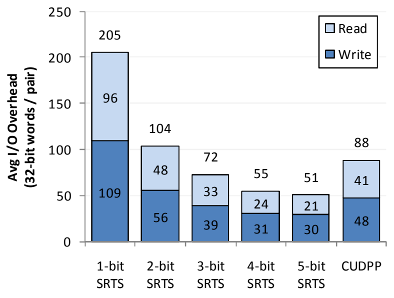

**Figure 17**. Aggregate GTX-285 memory overhead ($n \geq$ 16M key-value pairs).

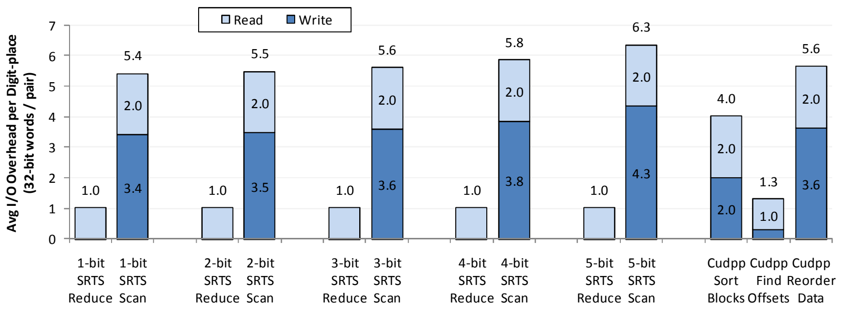

**Figure 18.** GTX-285 memory overhead per distribution sort ($n \geq$ 16M key-value pairs).

The overall memory workloads for these implementations are shown in Figure 17. We confirm our model from Section 4.1 that memory overhead monotonically decreases with increasing $d$. The memory workload of the CUDPP implementation (4-bit digits) is 1.6x that of our 4-bit variant.

Figure 18 illustrates the overheads for a single distribution sort, broken down by kernel and type. While our scatter operation logically writes two 32-bit words per pair (the key and the value), we observe that the hardware issues additional write transactions when threads within the same half-warp write keys having different radix digits to different memory segments. On this architecture, these write instructions incur a ~70% I/O overhead (~28% overall with respect to the cumulative I/O model) that increases in proportion with $r$, the number of digit partitions. In comparison, the CUDPP implementation experiences a 22% overhead with respect to its cumulative I/O model. The scatter inefficiencies decrease for less-random key distributions. With zero effective random bits (uniformly identical keys), our 4-bit implementation averages a saturated sorting rate of 550 x$10^6$ pairs/second. These compute-bound kernels do not benefit from this lower memory workload: this speedup is instead gained from the elimination of warp-serialization hazards that stem from bank conflicts incurred during key exchange.

| *Device* | *Memory Bandwidth ($10^9$ bytes/s)* | *Compute Throughput ($10^9$ thread-cycles/s)* | *Memory wall (bytes/cycle)* |
|---|---|---|---|
| **NVIDIA GTX 285** | 159.0 | 354.2 | 0.449 |
| **NVIDIA GTX 280** | 141.7 | 311.0 | 0.456 |
| **NVIDIA Tesla C1060** | 102.0 | 312.0 | 0.327 |
| **NVIDIA 9800 GTX+** | 70.4 | 235.0 | 0.300 |
| **NVIDIA 8800 GT** | 57.6 | 168.0 | 0.343 |
| **NVIDIA 9800 GT** | 57.6 | 168.0 | 0.343 |
| **NVIDIA 8800 GTX** | 86.4 | 172.8 | 0.500 |
| **NVIDIA Quadro FX 5600** | 76.8 | 152.3 | 0.504 |

**Figure 19**. Device throughputs and transition boundaries between memory-bounded and compute-bounded workloads for various NVIDIA GPUs

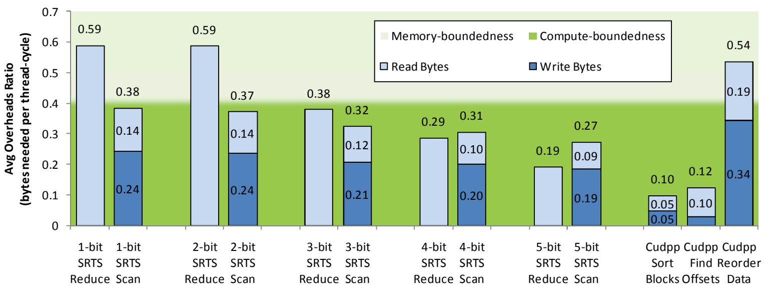

**Figure 20.** Memory-to-compute workload ratios for individual stream kernels, with the GTX-285 memory wall as a backdrop.

The boundary between compute-boundedness and memory-boundedness, or the *memory wall*, is the ratio between computational and memory throughputs. It is often expressed in terms of the number of cycles that can execute per memory-reference, but we prefer the inverse: the average number of bytes that can be serviced per cycle. Figure 19 enumerates the particular $\delta_{mem}/\delta_{compute}$ ratios for several models of NVIDIA GPUs. Figure 20 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 do not oscillate between and memory-bound and compute-bound kernels. The CUDPP implementation contains a mixture of memory-bound and extremely compute-bound kernels.

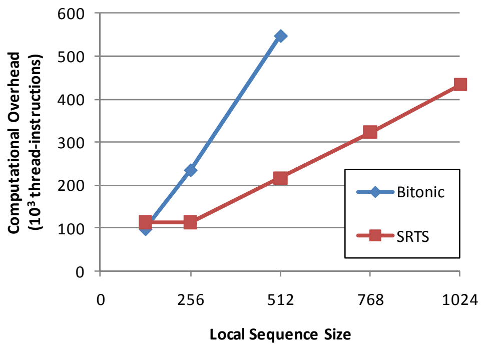

**Figure 21**. Computational workloads for local bitonic and SRTS radix sorting networks. These measurements exclude instructions pertaining

For input sequences smaller than 2,048 keys, it is more efficient to implement the entire set of *k*/*d* distribution-sorting passes as a local sorting problem, i.e., within a single kernel invocation consisting of a single CTA. Keys are read and written only once from global memory and exchanged *k*/*d* times in local shared memory. Bitonic sorting networks are often used for such local sorting problems; Figure 21 compares the computational overhead of our local radix sort versus that of the bitonic sorting implementation provided by the NVIDIA CUDA SDK [**48**]. Our 4-bit implementation incurs the same overhead for all inputs smaller than the 256-element local block-exchange size; the bitonic implementation is limited to $n \leq 512$, the maximum number of threads per CTA. The efficiency of our radix sort exceeds that of bitonic sort for sequences larger than 128 keys, and exhibits a speedup of 2.5x for 512 keys.

## 6 Conclusion

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

We have also demonstrated a method for encoding multiple binary-valued flag vectors into a single, composite representation. This allows us to execute several compaction tasks in shared memory while only incurring the cost of a single parallel scan. For small sorting problems suitable for a single GPU core, this technique allows our local sorting implementations to be several times more efficient than popular bitonic sorting methods. While this technique certainly allows us to increase the number of digit bits *d* more than we would be able to otherwise, it is not a critical ingredient for our speedup success. Without flag vector encoding, our $d = 2$ bits distribution sort would require four local scans, the same computational overhead as our 4-bit distribution sort. This four-scan distribution sort exhibits a sorting rate of 3.9 x$10^9$ pairs/second; sixteen passes would result in an overall sorting rate of 241 x$10^6$ pairs/second (and a speedup of 1.8x over the CUDPP implementation).

This work illustrates the need for a different breed of parallel primitives for GPGPU stream architectures. We obtain our sorting performance and efficiency by generalizing the parallel scan stream primitive in two ways: (1) with interfaces for producer/consumer operations (*visiting logic*), and (2) with interfaces for performing multiple related, concurrent prefix scans (*multi-scan*). These abstractions allow us to improve the overall utilization of the device's memory and computational resources while maintaining the flexibility of a reusable component. Visiting logic allows us to increase the arithmetic intensity of the memory-bound parallel prefix scan primitive in an application-neutral manner by coalescing orthogonal stream kernels; multi-scan allows us to increase the arithmetic intensity in an application-specific way by increasing the size of the radix digits.

## 7 References

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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

[10] 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.

[11] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha, "Fast BVH Construction on GPUs.," *Comput. Graph. Forum*, vol. 28, pp. 375-384, 2009.

[12] B Fabianowski and J Dingliana, "Interactive Global Photon Mapping," *Computer Graphics Forum*, vol. 28, pp. 1151-1159(9), June-July 2009.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] Larry Seiler et al., "Larrabee: a many-core x86 architecture for visual computing," in *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, Los Angeles, CA, 2008, pp. 1-15.

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

[23] 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.

[24] Leslie G Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103-111, 1990.

[25] Jiayuan Meng and Kevin Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs," in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, Yorktown Heights, NY, 2009, pp. 256-265.

[26] 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.

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

[28] Herman Hollerith, "Art of Compiling Statistics," US Patent 395,781, January 8, 1889.

[29] 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.

[30] G E Blelloch, "Scans as Primitive Parallel Operations," *IEEE Trans. Comput.*, vol. 38, pp. 1526-1538, 1989.

[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] K E Batcher, "Sorting networks and their applications," in *AFIPS '68 (Spring): Proceedings of the April 30--May 2, 1968, spring joint computer conference*, Atlantic City, NJ, 1968, pp. 307-314.

[36] Peter Kipfer and Rüdiger Westermann, "Improved GPU sorting," in *GPUGems 2*, Matt Pharr, Ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2005, ch. 46, pp. 733-746.

[37] A Greb and G Zachmann, "GPU-ABiSort: optimal parallel sorting on stream architectures," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, p. 10 pp.

[38] Mark Harris, Shubhabrata Sengupta, and John Owens, "Parallel Prefix Sum (Scan) with CUDA," in *GPU Gems 3*, Hubert Nguyen, Ed. Boston, MA: Addison-Wesley, 2007, ch. 39.

[39] 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.

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

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

[42] 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.

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

[44] 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.

[45] 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.

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

[47] Markus Billeter, Ola Olsson, and Ulf Assarsson, "Efficient stream compaction on wide SIMD many-core architectures," in *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, New Orleans, LA, 2009, pp. 159-166.

[48] NVIDIA. (2010, February) NVIDIA CUDA SDK - CUDA Basic Topics. [Online]. http://www.nvidia.com/content/cudazone/cuda_sdk/CUDA_Basic_Topics.html