Revisiting Sorting for GPGPU Stream Architectures

Duane G. Merrill
Department of Computer Science
University of Virginia
dgm4d@virginia.edu

Andrew S. Grimshaw
Department of Computer Science
University of Virginia
grimshaw@virginia.edu

ABSTRACT

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

These results motivate a different breed of parallel primitives for GPGPU stream architectures that can better exploit the memory and computational resources while maintaining the flexibility of a reusable component. Our sorting performance is derived from 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*).

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Sorting and Searching; G.4 [**Mathematical Software**]: Parallel and Vector Implementations

General Terms

Algorithms, Performance, Design

Keywords

GPU, sorting, radix sorting, prefix scan, kernel fusion

1. INTRODUCTION

Software primitives promote software flexibility via abstraction and reuse, and much effort has been spent investigating fast and efficient primitives for GPGPU stream architectures [1]. Sorting has been no exception. 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.

We present our strategy for radix sorting on GPGPU stream architectures, demonstrating that our approach is significantly faster than previously published techniques for sorting large sequences of fixed-size numerical keys. We consider the GPGPU sorting algorithms described by Satish et al. [2] (and implemented in the CUDPP data parallel primitives library [3]) to be representative of the current state-of-the-art. Our work demonstrates factors of speedup of at least 2x for all fully-programmable generations of NVIDIA GPUs, and up to 3.7x for current generation models.

Revisiting previous sorting comparisons in the literature between many-core CPU and GPU architectures [4], 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 platform by as much as 42%. The Larrabee architecture provides write-coherent caches and alternative styles of synchronization in an effort to provider higher performance for cooperative problems such as sorting [5].

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 [6,7]. Sorting routines are germane to many GPU rendering applications, including shadow and transparency modeling, volume rendering, particle rendering and animation, and ray tracing. Sorting also serves as a procedural step within the construction of KD-trees and bounding volume hierarchies, both of which are useful acceleration 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, database acceleration, data mining, and game engine AI. [8]

2. RADIX SORTING STRATEGY

The radix sorting 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. 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. 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. The asymptotic work complexity of the distribution sort is $O(n)$ because each input item needs comparing with only a fixed number of radix digits. With a fixed number of digit-places, the entire sorting process is also $O(n)$.

The work presented in this poster 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: sequences comprised (a) 32-bit keys paired with 32-bit satellite values; and (b) 32-bit keys only. Our solution strategies, however, can be generalized for keys and values of other sizes

We refer to our approach as a *strategy* [9] because it is a flexible hybrid composition of several different 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. Because the number of

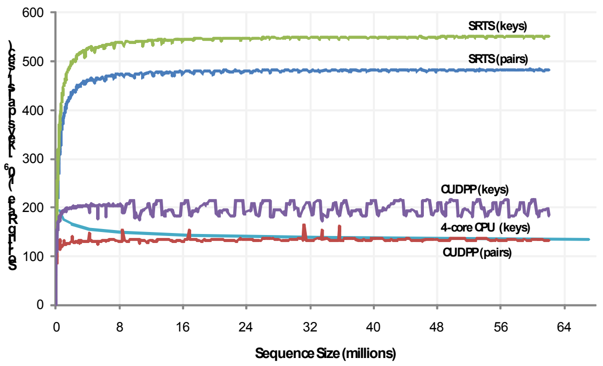

Figure 1. GTX-285 sorting rates for both CUDPP and our 4-bit SRTS-based implementations, overlaid with the Chhugani et al. [4] Intel Core-2 Q9550 key-only sorting rates.

Table 1. Saturated sorting rates for input sequences larger than 16M elements.

Device		*Key-value Rate (10⁶ pairs / sec)*		*Keys-only Rate (10⁶ 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	Q3/2007	55	**110(2.0x)**	66	**147(2.2x)**
				Merge [4]	
Intel Q9550 quad-	Q1/2008			138	
Intel Larrabee 32-	Cancell			386	

steps needed for each phase is adjustable, our solution is flexible in terms of support for different SIMD widths, shared memory sizes, and can be mated to optimal patterns of device memory transactions.

There are two contributing factors that have enabled our efficient use of the hardware. The first is our prior work on the development of efficient, memory-bound parallel prefix scan routines [10]. Prefix scan is a useful primitive for parallel shared-memory architectures: it allows 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 factor is that we apply visitor pattern [9] of task composition to our prefix-scan primitive in order to achieve kernel fusion: program steps that would typically exist in distinct kernels are coalesced using an additional pair of abstraction interfaces in which the primitive invokes (i.e., "visits") application-specific logic for (a) input-generation and (b) post-processing behavior. Our radix sorting design combines the prefix scan primitive with binning and scatter routines: binning decodes the particular numeral represented within a given key and digit place, and scatter relocates keys (and values) based upon the ordering results computed by scan.

This pattern provides an elegant mechanism for increasing the arithmetic intensity of memory-bound primitives. 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. The elimination of load/store instructions also increases the computational efficiency, further allowing our pattern to exploit those resources.

The result is a much lower aggregate memory workload. Unlike prior implementations of stable counting-sort methods, our generalizations of prefix scan only require a small, constant number of intermediate values that must be exchanged through global device memory.

3. REFERENCES

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

[2] 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.

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

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

[5] 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.

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

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

[8] Duane G. Merrill and Andrew S. Grimshaw, "Revisiting Sorting for GPGPU Stream Architectures," University of Virginia, Department of Computer Science, Charlottesville, VA, Technical Report CS2010-03, 2010.

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

[10] 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.