Implicit radix sorting on GPUs

Linh Ha[*], Jens Krüger[*], Cláudio T. Silva[*]

July 27, 2010

In this chapter, we present a high performance sorting function on GPUs that is able to exploit the parallel processing power and memory bandwidth of modern GPUs to sort large quantities of data at a very high speed. We revisit the traditional radix sorting framework, analyze the weaknesses, and then propose a solution based on the implicit counting data presentation and its associated operations. We also improve the bandwidth utilization with our hybrid data structure and redefine the concept of arithmetic intensity as a guidance for GPU optimization process.

1 Introduction, Problem Statement and Context

1.1 Motivation

Sorting is undeniably one of the most fundamental algorithmic building blocks and one of the most widely-studied problem in computer science literature. There are numerous algorithms in which sorting is an essential component. Improving this core algorithm can significantly improve the performance of many applications. Not only real time systems benefit from a fast sorting algorithm, but also many execution-time limited applications become feasible with its use. Hence, the results of this work are of interest for general research and development in HPC and GPGPU communities. A number of applications will directly benefit from a fast sorting framework, including data querying, exploration, classification, visualization, physical-based simulation, computer games, and more.

[*]Scientific Computing and Imaging Institute, University of Utah, Salt Lake city, UT84112

Modern GPUs offer massive parallel computational power and extreme memory bandwidth, the foundations for fast sorting algorithms. Previous GPU sorting approaches, however, were not able to exploit these computational powers. In particular, scattered write operations prevent coalesced data movement, a key component for efficient GPU programming. Consequently, GPU sorters were memory bounded and had low compute-memory efficiency. In this chapter, we analyze these issues and propose two major improvements: First, an implicit counting structure with associated operations, and second a hybrid Structure of Arrays (SoA) and Array of Structures (AoS) data presentation.

1.2 GPU sorting overview

The dramatic changes of the GPU architecture over the past decade has led to two trends in GPU sorting algorithm: GPU-based sorting implementation based on the graphical pipeline, and parallel sorting on General Processing GPUs based on GPGPU APIs.

Parallel sorting network - Comparison-based sorters. Most traditional GPU sorting implementations have been based on sorting networks, in particular, the bitonic sorting network. The main idea is that a given network configuration will sort the data in a fixed number of steps using static communication paths. This property suits the traditional GPU architectures well, because sorting algorithms can be expressed in terms of shader functions, which have very limited branching and no scattering support. The complexity of such sorting networks, however, is $0(n \log_n^2)$, which is higher than that of the optimal comparison-based sorting, $0(n \log_n)$.

The complexity drawback was tackled by Gres *et al.* [5], who employed an adaptive bitonic sorting strategy to lower the complexity to the optimal bound of $O(n \log n)$. Cache strategies were also considered to improve the performance, Govindaraju *et al.* [3] presented an improved bitonic sorting network with more cache-efficient data access and data layout to speed up GPU based sorting by about a factor of 1.5.

The introduction of general parallel processing architectures and high level GPU programming languages such as CUDA, Direct Compute and OpenCL gave developers full access to the computational power and memory bandwidth of modern GPUs. These programming features offered developers more control of the memory cache, the parallel thread execution, and the efficient branching and fine grain hierarchical memory-execution structure.

Peters *et al.* [10] implemented a fast bitonic sorting algorithm in CUDA which reached 60M pairs per second on the GTX 280. A competitive performance is achieved by the parallel merge sort of Satish *et al.* [11], which becomes part of the Thrust library. So far the fastest comparison-based sorter, however, is the GPU Sample Sort by Leischner *et al.* [9], which is about 30 percent faster than the parallel merge sort.

Despite achieving considerable improvement over CPU sorters, the log-factor of comparison-based approaches is costly, especially when dealing with a large number of inputs. Comparison-based sorters are only considered when inputs are non-integer or have variable length, and when in-placed sorting is the main concern. Otherwise, a more efficient approach is the counting-based sorting scheme with a linear bound complexity.

Counting sorters. Though counting-based sorters were introduced later to the GPU, they have achieved remarkably performance improvement and have proved to be the more GPU friendly and scalable approaches. In 2007, the hybrid sorting algorithm by Sintorn and Assarsson [13] based on a vectorized mergesort in combination with a bucket-sort using atomic GPU operations, was twice as fast as the previous fastest GPU-based bitonic sorting algorithm [3]. The most preferred and efficient GPU counting-based scheme, however, is the radix sorting. The GPU radix-16 by Satish *et al.* [11] is the first single-device sorter that is capable of sorting more than a hundred million key-index pairs in a second.

Radix-sorting algorithm is often referred as *radix-r*, where r is the number of radix buckets. In practice, the key is 32-bit length, hence it requires $[32/log_2(r)]$ passes, each performs a radix step on $\log_2(r)$-bit of the key from the right most bit to the left most bit (Least Significant Bits strategy - LSB). The radix sorting can be used for arbitrary number-typed inputs: float and integer, and with arbitrary key-length [6].

In a single pass, each key is placed into one of r buckets. The position of the r-sorted output element, called *global rank*, is equal to the total number of elements in lower buckets and those preceding of the same bucket. For parallel efficiency, the global rank is computed using a fine grain approach by adding a *local rank*, the rank of the element in its block with the number of the same radix value on previous blocks, then with the total number of elements in lower radix buckets. When the global ranks are computed for all input elements, the final step shuffles inputs onto locations determined by their ranks. Then the attention is moved to the next higher bit group and the process continues until all the input bits are sorted.

The performance of GPU radix-sorting depends on how fast the global ranking computation is and how friendly to the memory cache the shuffle step can be implemented. There are two main schemes to compute global rank: histogram-based methods [4, 13] and scan-based methods [6, 11].

Histogram-based methods explicitly compute a histogram for all radix buckets. Sintorn and Assarsson [13] exploited CUDA atomic functions on CUDA 1.1 hardware to count the number of elements in each bucket. Therefore, their performance depends heavily on the input distribution, and suffers from parallel resource fighting. To tackle this drawback, Le Grand [4] exploited the on-chip fast-access explicit cache, the *shared memory*, for radix counters, and divided parallel threads onto thread groups. Each thread group has different radix counters; hence, resource fighting between groups was eliminated. However, the method serializes the increment of radix counters sharing between threads of the same group.

Scan-based methods depend on prefix sum operation to implicitly compute the histogram. First presented by Harris *et al.* [8], the GPU scan operator can achieve optimal bandwidth of streaming operations on the GPUs. As a direct result, Sengupta *et al.* [12] implemented a binary-radix sorting which requires n radix passes with n being the key-length in bits. The method is bandwidth-bounded and under-utilized GPU power, resulting in similar performance as the hybrid sort but slower than Le Grand's radix-16.

To exploit the parallel processing power of the GPUs and to reduce the number of radix passes, Ha *et al.* [6] proposed a fast 4-way radix sorting that took advantages of the instructional parallelism to perform four scan counting paths at the same time. Satish *et al.* [11] exploits the simplicity and efficiency of the implicit binary-radix sorting to perform multiple radix passes on the GPU's shared memory. Both methods were based on the modified radix sorting with local pre-sorting step to handle non-coalesced pattern of the final mapping step. As a result, Satish's radix sorting is almost six-times faster than Le Grand' radix-16 with the capability to sort 140 millions input pairs per second on the NVIDIA GTX 280, the fastest published results with both GPU and CPU sorting on a single desktop.

2 Core sorting frameworks

To further improve the efficiency of ranking computation and the cache coherency of the mapping step, Satish *et al.* [11], and Ha *et al.* [6] proposed an improved framework that performs sorting in 3 main steps:

- Parallel local radix counting and pre-sorting
- Global radix ranking
- Coalesced global shuffling

The basic difference of the improved framework from the traditional one is the local pre-sorting step, which happens inside the shared memory and is incorporated into the regular local counting step. The pre-sorter divides data into radix blocks, which then move together in the final mapping. This strategy greatly increases the cache coherency of the data. To further improve the performance, a coalesced mapping step was proposed [6, 11] that assigns each thread to the data based on its output location to satisfy the coalesced mapping condition.

Revision of the arithmetic intensity concept. An analysis of the computational characteristics of existing sorting algorithms shows that few arithmetic operations are involved, i.e. the counting with radix-based solutions and simple comparisons with other sorting solutions. Data movement is the most common operation. Consequently, sorting algorithms are memory-bounded, low-compute efficient, and rarely able to benefit from the huge computational power of GPUs.

Though the improved framework tried to tackle the non-coalesced effect, the memory bandwidth efficiency of the global mapping step is still a fraction of the full memory bandwidth. Together with low-compute efficiency of pre-sorting step, they are the two major performance bottlenecks. We see these issues as the problem of low arithmetic intensity of existing approaches.

There are two typical views about arithmetic intensity: Dally *et al.* [2] defined “arithmetic intensity” as “math operation per memory op”, Buck *et al.* [1] defined “computational intensity” as “time spent on computation over data transfer”. We believe that these definitions, do not reflect the actual efficiency of a kernel and insufficiently capture the goal of optimization. We rather consider efficiency as “the overall amount of work done over the data” i.e. work per time so that a more efficient kernel will do more effective work per data unit (i.e. implicit binary vs binary sorting) and spend less time to complete the same amount of work, i.e. sorting task. Our definition considers both computational and memory usage efficiency in the optimization process.

Algorithmic improvements. Use the new concept as a guidance, we proposed two major algorithmic improvements: an implicit parallel counting and a mixed-data structure. The implicit counting exploits GPU instructional parallelism to reduce the number of passes inside the shared memory in comparison with Satish's method by a factor of two. The mixed-data structure allows a more efficient mapping step which is immune to the non-ideally coalesced effect. Both strategies successfully address the efficiency issues, leading to a significant improvement over highly optimized solution of Satish *et al.*. In the next section, we will go further into the details of our sorting method.

3 Algorithms, Implementations, and Evaluations

3.1 Implicit counting - Improving compute efficiency

Two major components of this arithmetic improvement are the implicit counting number and its associated operations. An implicit counting number encodes three counters in one 32-bit register, each counter is assigned 10 bits, in particular

$$impl_{cnt} = cnt_0 + (cnt_1 \ll 10) + (cnt_2 \ll 20)$$

where cnt_0, cnt_1, cnt_2 are the counting values of radix value 0,1, and 2.

For a single radix value, the corresponding implicit counting value (Figure 1 b) is computed

$$impl_{val} = (val < 3) \ll (10 * val)$$

Note that the implicit counting value of the radix mask 3 is 0 in the example given in Figure 1b.

The *radix counting operation* for a radix value is computed implicitly by adding $impl_{val}$ to the common counter $impl_{cnt}$ (as shown on Figure 1b,c)

$$impl_{cnt} = impl_{cnt} + impl_{val}$$

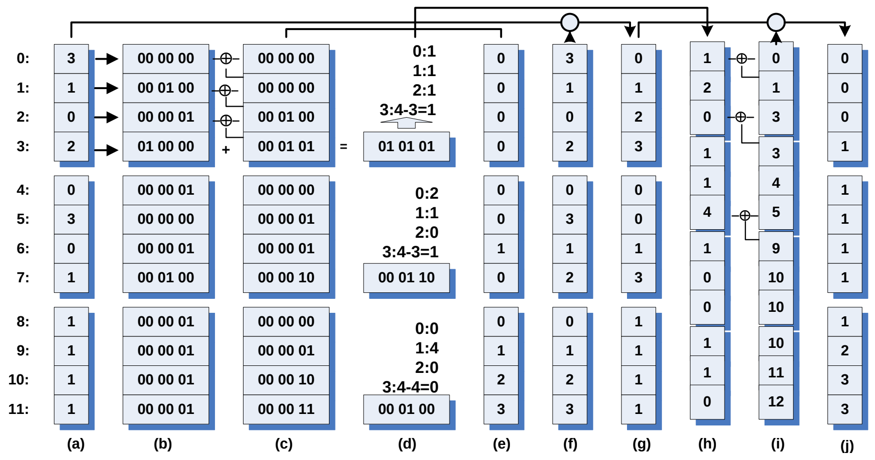
Figure 1: Illustration of our implicit radix sorting (intermediate steps) a) Inputs b) Implicit-presentation of the input c) The local-prefix sum d) Number of each radix bucket e) Number of previous same bucket elements f) local rank g) pre-sorted result h) Number of radix values in each block i) Start offset j) Sorted output

The counting values of three first radix buckets are easily restored from the common counter using shift operations (Figure 1d)

$$cnt[val] = impl_{cnt} \gg (10 * val)$$

The *fourth counting value* - radix bucket 3, can be computed based on the three others using

$$cnt[3] = id - cnt[0] - cnt[1] - cnt[2]$$

because the total number of preceding elements in the four radix buckets to an element index *id* is exactly *id*.

We apply the idea of implicit counting twice: First to compute the fourth counting value from the common counting values of the three other buckets, and second to reduce number of scan paths from four to one. The implicit counting function allows us to compute the four radix buckets with only a single sweep. This is twice as efficient as the implicit binary approach of Satish *et al.*.

3.2 Improving memory bandwidth

Hybrid data representation. To increase the memory bandwidth efficiency in the global shuffling step we proposed a hybrid data representation

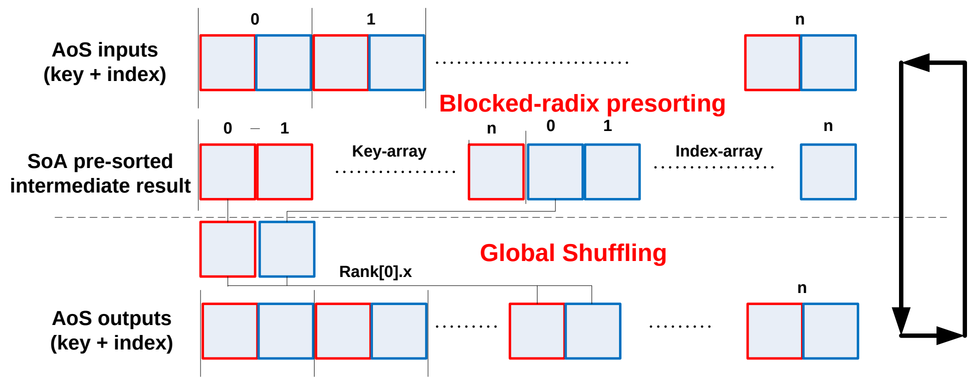

Figure 2: The flow of our hybrid-data format. The conversion occurred implicitly inside the global shuffling kernel and at the beginning of local counting kernel using texture memory.

that uses SoA as the input and AoS as the output. The conversion is illustrated in Figure 2. The key observations is that though the proposed mapping methods [6, 11] are coalesced, the input of the mapping step still come in fragments, we call this a non-ideal effect. When it happens, the longer data format (i.e. int2, int4) suffers less performance downgrade than the sorter one (int). Therefore, our AoS output data structure significantly reduces the suboptimal coalesced scattering effect in comparison to SoA. Moreover, the multi-fragments requires multiple coalesced shuffle passes which turns out to be costly, we saw the improvement by applying only one pass on the pre-sorting data. We also achieved the full memory bandwidth for the thread input element, which is $4 \times int2$ length, using the texture cache.

Shared memory bank conflict-free access. We applied a bank conflict-free access pattern that stores a long format data structure, such as *float4*, into separated arrays. This handles the bank conflict inherently to accessing long format data on GPU shared memory. We then performs the operation on each components and writes results back to the register. The bank conflict free mechanism is illustrated in Figure 3. A similar concept has been applied by Satish *et al.*, but without a deeper analysis. In contrast we propose the bank-free conflict mechanism as a general optimization technique when working with long format data.

3.3 Performance tuning

Range limiter. While radix sorting time scales with the number of bits

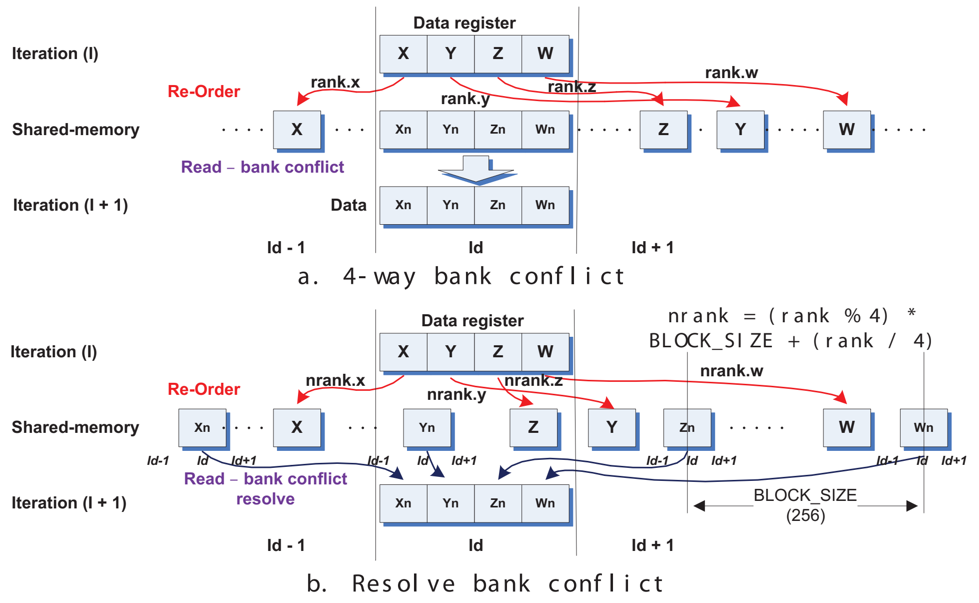

Figure 3: Resolve the 4-way memory conflict

used to represent the data, the actual number of sorting bits may substantially lower than the full length of the sorting key. For example sorting of the point-based simulation on the 256^3 grid only require 24 bits of integer.

Our methods exploits this prior knowledge about input ranges to reduce the number of radix passes. We use a simple scale and bias to map arbitrary numbers from the range $[a, b]$ to $[0, b - a]$. On the GPU we can quickly determine the range of the inputs by applying a reduce operation, which is as fast as a memory copy device operation [7].

While this works well with integers, such a simple mapping technique is not very efficient with floating point numbers as the range in its integer-converted format is likely to require as many as 32 bits, even for a small data range. However, as floating point numbers in the range of $[2^n, 2^{n+1})$ share the same leading exponential bits, we can reduce the range from full 32-bits to 24-bits of factional data using the normalized linear mapping from $[a, b]$ to $[0.5, 1)$ range. This mapping yields a 30% performance improvement. Table 4 illustrates the binary presentation of floating point numbers in the normal range $[0.5, 1)$.

While the mapping is linear, it certainly is not one-to-one due to the adaptive range of floating point presentation, hence it is possible that two numbers

Float number	Binary presentation
0.5	0 01111110 00000000000000000000000
0.7	0 01111110 01100110011001100110011
0.99999997	0 01111110 11111111111111111111111

Figure 4: Binary presentation of floating point number between 0.5 and 1

may mapped to the same number in the normal range. This sorting result is an approximate sorting of the input. For many real time applications—especially in computer graphics and visualization—this approximation is acceptable.

4 Final evaluation and validation of results, total benefits, limitations

Our method extends and improves the fastest previously published implementation of Satish *et al.* [11] (CUDPP1.1) in both the pre-sorting and global shuffling steps. In the following we take a closer look at those two improvements.

We first focus on the pre-sorting step. Please note that all timings are given in microseconds on an NVIDIA GTX 260 with 192 CUDA cores and 896MB memory. The size of the input $N(M)$ is the number of key-index input pairs in millions. To demonstrate the consistently improved behavior of our method we perform the pre-sorting step with different input sizes N:

N(M)	2	4	6	8	10	12	14	16
Satish et al	11	21	33	45	58	69	79	91
Impl radix	7	13	20	27	34	41	48	54

As can be seen in Figure 5, our pre-sorting step is about 1.5 to 1.8 times faster than the CUDPP 1.1 implementation.

Next, we take a look at the global shuffling improvements. We demonstrate our global shuffling step on 100 random radix-16 pre-sorted arrays, which are partially sorted with 16-bin radix in groups of 1024 elements, with sizes ranging from 1 to 16M key-value input pairs. The results show that by using an AoS structure instead of SoA as the output format we improve the performance by 25%. At the same time, our one-pass implementation of SoA shuffling is more efficient than CUDPP1.1 by an additional 15%. Overall,

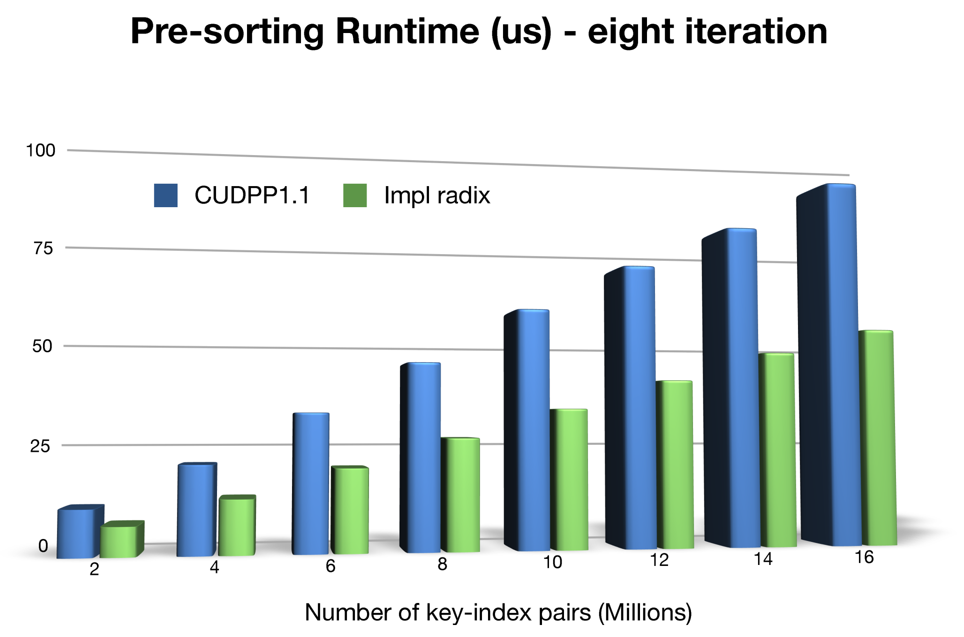

Figure 5: Total run-time of pre-sorting step (ms) with Implicit Radix and Satish CUDPP1.1 radix-16

our global shuffling is 1.4 times faster than that of CUDPP as illustrated in Figure 6. It is approximately 1.4 times more expensive than a fully-coalesced memory copy operation, the upper bound.

N(M)	2	4	6	8	10	12	14	16
AOS	0.47	0.90	1.34	1.77	2.21	2.64	3.08	3.52
SOA	0.61	1.16	1.73	2.31	2.87	3.44	4.01	4.33
CUDPP	0.72	1.38	2.05	2.73	3.42	4.09	4.77	5.15
Memcpy	0.34	0.66	1.01	1.34	1.66	1.99	2.34	2.67

Finally, we compare the component runtime in one iteration of a 16M-pair input between our implicit sorting and the Satish *et al.* (CUDPP1.1) implementation (time is measured in milisecond)

16M pairs	Pre-sort	Glb rank	Glb Shuff	Total	Memcpy$DtoD$
Satish et al	12.25	0.15	5.15	17.55	
Impl radix 16	8.15	0.15	3.75	12.05	2.78

In Figure 7, we measure the sorting rate (million-pairs per second) for random unsigned integer input arrays with size ranging from 1M to 16M. Both

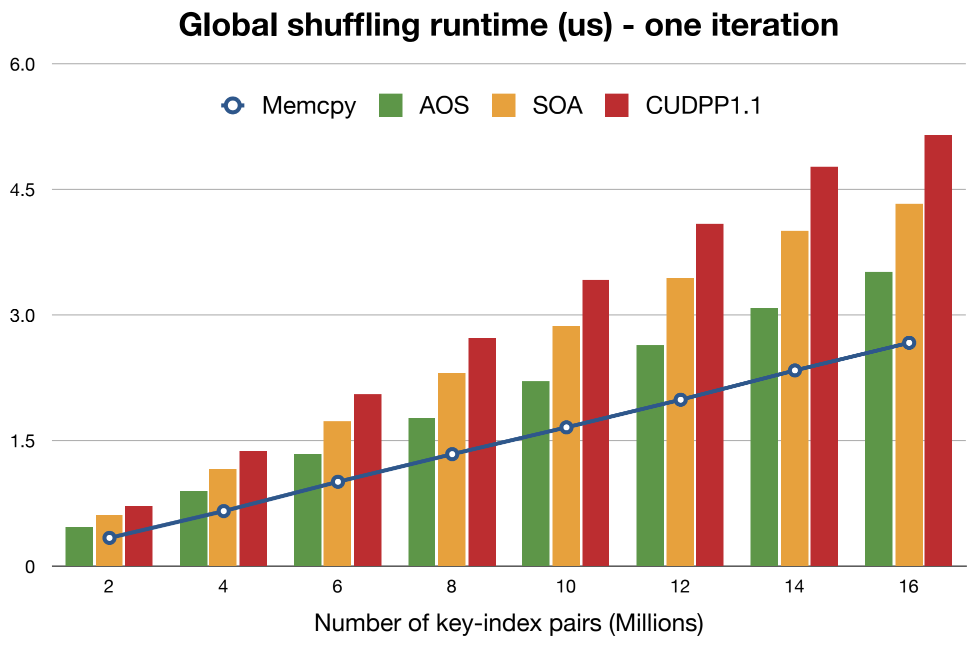

Figure 6: Global shuffling run-time comparison (ns) between our implementation of global shuffling with AoS, SoA structures, and CUDPP1.1 in reference to the device to device memcopy of the same input size:

our method and the Satish *et al.* implementation require eight iterations for the 32-bit key.

N(M)	2	4	6	8	10	12	14	16
Satish et al	116	112	116	120	119	116	115	123
Impl radix	170	160	174	177	177	177	177	178

As can be seen, our method is able to sort about 180M key/value pairs per second on the target hardware, making it a factor 1.5 times faster than the the previous radix-16 implementation on the same hardware. When using our approximate single precision floating point sorting scheme we achieve another 30% speedup as we need only sort 24 bits of the 32 bits key.

We also observe significant performance improvements with integers when the dynamic range does not cover the full 32 bit range.

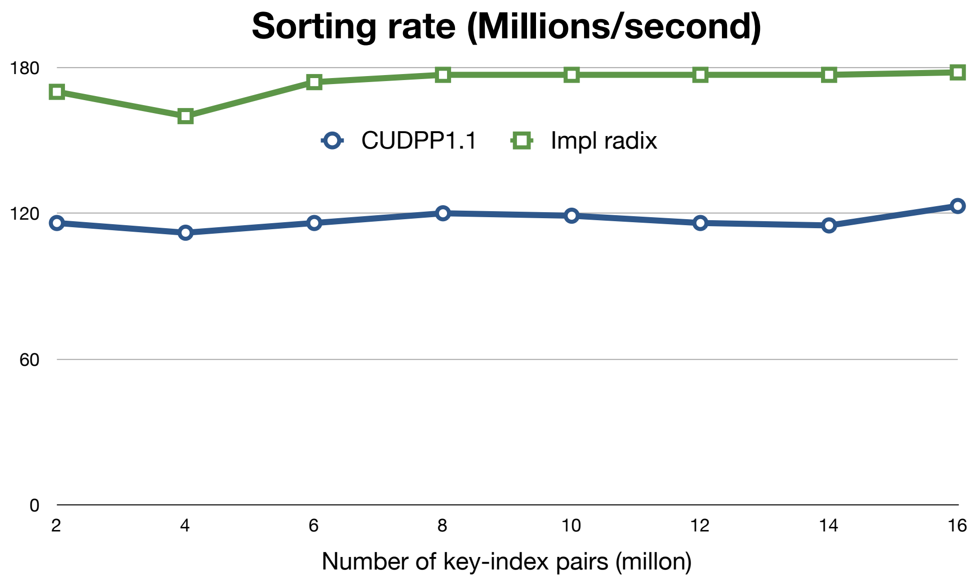

Figure 7: The sorting rate comparison of random 32-bit unsigned inputs

5 Future directions

In this chapter, we propose a new sorting algorithm to improve the performance of GPGPU implementations on modern GPU architectures including:

- A revision of the arithmetic intensity concept to evaluate the efficiency of GPU algorithms, which can be used as a guideline for optimization
- A new data structure and operations to exploit instructional parallelism, reducing significantly the amount of computation
- An adaptive data structure concept to tune performance at each algorithm stage

Our sorting framework efficiently address performance issues of existing approach and successfully exploit both the compute power and memory bandwidth of modern GPUs.

While constraint of 1024 elements block size seems to affect the scalability of the method in the future device, we believe it is not the case since the number of thread in one block 256 sufficiently hide the memory latency. Moreover, with a minor change in the algorithm, we could increase the block size to 2048 elements with one implicit counting bit to achieve 33-bits implicit counter. However, on the current architecture the 1024 elements is the optimal size.

Although our approach increases the arithmetic intensity of sorting problem, the full power of the GPU has not yet been exploited, one possible solution is to combine our implicit counting and multiple parallel scan path of Ha *et al.* [6], this method also overcomes the 1024 block size limitation.

When the input of the radix sorting is a number, the algorithm could easily be extented to segmented sorting, that sorts multiple segments of input at the same time. Segmented sorting has applications in visual sorting when fragments are sorted per rays.

For future work, we want to combine GPU and CPU sorting to exploit both memory bandwidth and processing power of GPUs and CPUs to achieve the highest performance and to handle extremely large data set on the GPU cluster.

References

[1] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[2] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2003. IEEE Computer Society.

[3] 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*, pages 325–336, New York, NY, USA, 2006. ACM.

[4] Scott Le Grand. Broad-phase collision detection with cuda. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[5] Alexander Greß and Gabriel Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, 25–29 April 2006.

[6] Linh Ha, Jens Kruger, and Claudio T. Silva. Fast 4-way parallel radix sorting on gpus. *CGF, Computter Graphic Forum*, page to be appeared, 2009.

[7] Mark Harris. Optimizing parallel reduction in cuda. `http://tinyurl.com/6dazkd/reduction.pdf`, 2007.

[8] Mark Harris, John Owens, Shubho Sengupta, Yao Zhang, and Andrew Davidson. Cudpp: Cuda data parallel primitives library. `http://www.gpgpu.org/developer/cudpp/`, 2007.

[9] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. Gpu sample sort, 2009.

[10] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast comparison-based in-place sorting with CUDA. Technical report, Christian-Albrechts-University Kiel, German, 2009.

[11] 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*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM, August 2007.

[13] Erik Sintorn and Ulf Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2007.